Exploring the limitations of the Atelier B automated prover

Agata Borkowska, UID: 1690550, *MSc in Computer Science, University of Warwick*

***Abstract*—AtelierB is a tool for formal software development through refinement, using the B-method. It incorporates an automated prover, which has been recognized as the most thorough prover for B set theory, and has been used as a basis for many others. Nevertheless it has multiple shortcomings. Various approaches have been suggested and taken to improve its performance, including extensions to the proof rule base, plug-ins and third-party software. In this work we strive to establish the limitations of the prover without such additions, and discover at which point they become necessary. The secondary goal is to improve the robustness of the software without straying from pure B method, and taking into account the ease of use. As a metric of our success, we use the benchmarks proposed by Conchon and Iguernlala [1].**

***Index Terms*—B method, formal verification, specification, abstract machine, proof**

I. Introduction

THE aim of formal specification and verification is to ensure the correctness of software. While overall less popular than quality assurance through testing, it is primarily, but not exclusively used in safety-critical areas, where testing may not feasible.

A. Structure of a machine

In the B language, using Atelier B syntax, an abstract machine usually has the following clauses, in order:

- machine name - must match the name of the file, followed by a list of parameters without type definitions, given in brackets as a comma-separated list, for example `Bagmch(ITEMS, max_elem)`.
- `SETS` - deferred sets, i.e. those which will be specified at a later point. They can be used to define types of constants and variables.
- `CONSTANTS` - either scalar or set-valued constant. Their names are declared here.
- `PROPERTIES` - types of constants and the relations between them.
- `DEFINITIONS` - constants with a concrete value. According to the *Interactive Prover Reference Manual*, it is preferable to define such constants here, rather than declare them in the Constants clause and assign them a value in the Properties clause, because it reduces the amount of rewriting the tool does. [19]
- `VARIABLES` - declarations of variables, which the machine operates on.
- `INVARIANT` - properties of the machine that have to be maintain at every point of its operation. This clause includes type definitions of the variables, bounds on their values, and relations between them and the constants.
- `ASSERTIONS` - a series of statements which can act as intermediate steps in a proof. Note that they are necessarily ordered and every statement is proven under the assumption of those preceding it.
- `INITIALISATION` - the initial state of the machine, where the values of the variables are assigned.
- `OPERATIONS` - operations performed on the variables of the machine. Some of them may change the values of the variables, while other may only output a result of some calculation.

The first few clauses up to and including Assertions are considered the static part of the machine, and they provide definitions or describe properties that hold at all times. The last two clauses, namely Initialisation and Operations, form the dynamic part of the machine, as they either assign or change values of the variables.

Many of the clauses are optional, for example Definitions or Assertions. Others are closely linked together. For example it is imperative to declare Properties, if the machine contains a Constants clause. Similarly, if there is a Variables clause, it must be followed by Initialisation. Such dependencies are picked by static analysis in the source code editor.

Other clauses which are not analysed in this project, are structuring clauses, such as `USES`, `SEES`, or `INCLUDES`, and clauses necessary for refinement and implementation.

B. Atelier B Provers

Beyond serving as a source code editor for the B method, Atelier B also allows the users to verify the

machines they have written and check that they are indeed correct. This is done in three ways.

As could be expected, static analysis takes place while the user is writing the code, and serves to indicate primarily syntax errors. It may also come in useful when a user mistakenly calls a set operation on a scalar value or vice versa.

Once the machine is free from these errors, the user may proceed to generate proof obligations. They are statements which should be demonstrated to hold, or as it is sometimes called, discharged, or else the machine is likely to be incorrect. It should be noted that a proof obligation may be demonstrated to hold, be it by the tools included in the software or others, or manually, but it may not be disproved. The problem is known to be undecidable.[CITE] Therefore an undischarged proof obligation does not necessarily point to a mistake in the machine, but quite possibly is a result of a shortcoming of the prover tool.

Proof obligations generated by Atelier B can be divided into two types. Most proof obligations will be concerned with preservation of the Invariant by the Initialisation and the Operations which change the variables. A proof obligation of this type, which cannot be demonstrated, indicates that performing an operation may result in the machine entering an invalid state.

The second kind of proof obligations check the well-defineness property of certain expressions. A proof obligation of this kind which is impossible to discharge indicates that the user has likely not been specific enough when defining variables and their properties. For example, the expression $max(\varnothing)$ is erroneous, because there is no greatest element in the empty set. Similarly in the B language, for a set A, the expression $card(A)$ makes sense if and only if A is finite. The well-defineness conditions for all such expressions can be found in the *B Language Reference Manual* [17].

With the proof obligations generated, the user may begin the process of discharging them. Atelier B includes two provers which can achieve this. Firstly, the Automatic Prover should be used to demonstrate as many proof obligations as possible without the user's input.

If that is not sufficient, the user may employ the Interactive Prover. In it, they may view the undischarged proof obligations and guide the proof process along, for example by suggesting intermediate steps of the proof. The Interactive Prover also allows users to search through the proof rule base included in the software, as well as enables users to write their own proof rules.

II. Definitions and Conventions

A. Definitions and notations

Care shall be taken to use unambiguous notation and terms. We begin with a brief summary of the logic notation, comparing it to the B syntax, which comprises of ASCII characters or approximations thereof. We first give the unicode symbol which will be used when discussing logical expressions in this writeup, followed by the ASCII equivalent written in a monotype font, as it can be seen in code snippets.

- AND is denoted by $\wedge$ or `&`
- OR is denoted by $\vee$ or `or`
- the existential quantifier is $\exists$ or `#`
- the universal quantifier is $\forall$ or `!`
- the negation is denoted by $\neg$ or `not(x)`

This list includes only the most commonly seen symbols, some of which may not be intuitive to some users. Other symbols will be defined if or when they are needed.

Since the B method is heavily based on Zermelo-Fraenkel set theory, it is worth recalling key concepts and definitions from this area. [12]

Firstly, a **set** is a collection of distinct objects, which we will call **elements**. We say that a is an element of a set x or that it belongs to the set x. Set membership is a binary relation denoted with the symbol '$\in$' or with '`:`' in the B syntax.

By the Axiom of Extensionality, two sets are equal if and only if they have the same elements. Hence, in set axiomatic set theory $\{x\} = \{x, x\}$ - for each element of the first set is in the second, and each x in the second set is in the first one. The B method also recognizes this equality [14].

Note that a set can be an element of another set, however a set must not belong to itself - i.e. there does not exist a set x such that $x \in x$. This is known as Russell's Paradox.

By the Null Set Axiom, there exists a set having no elements - i.e. the **empty set**. It is denoted $\varnothing$ or $\{\}$ in the B syntax. The empty set is unique.

A **subset** y of a set x is a set such that each element of y is also in x, but not necessarily the other way round. It is denoted $y \subseteq x$ or `y <: x` in the B syntax. Note that the empty set is a subset of every set, and that every set is a subset of itself.

The **powerset** of a set x, denoted $\mathbb{P}(x)$ or `POW(x)` in the B syntax is the set containing all subsets of the set x. For example if $x = \{a, b\}$, then $\mathbb{P}(x) = \{\varnothing, \{a\}, \{b\}, \{a, b\}\}$. Thus the expression $y \subseteq x$ is semantically equivalent to $y \in \mathbb{P}(x)$.

The **union** of sets x and y is defined as the set of elements which belong to either x or y, i.e. $x \cup y =$

$\{z|z \in x \vee z \in y\}$ and in the B syntax it is written as `x\/y`. The **intersection** of sets x and y is defined to be the set of all elements which belong to both x and y, i.e. $x \cap y = \{z|z \in x \wedge z \in y\}$, which in the B syntax is `x/\y`.

Natural numbers will be often seen in this project. Firstly, and a little informally, the set of natural numbers, denoted $\mathbb{N}$ or `NAT` in the B syntax, is the set $\{0, 1, 2, ...\}$. The B syntax also has an abbreviation for the natural numbers excluding 0, i.e. `NAT1` shall be used to mean $\mathbb{N} - \{0\}$.

The natural numbers give the first discrepancy between the B method and the mathematical theories which it is based on, which we encounter throughout this work. Formally, in axiomatic set theory, the natural numbers are defined as follows:

Let x^+ denote the successor set of the set x, which is $x^+ = x \cup x$. Then $\mathbb{N}$ is the smallest (with respect to the number of elements) set such that $\varnothing \in \mathbb{N}$ and if $x \in \mathbb{N}$ then $x^+ \in \mathbb{N}$.

For simplicity, it is often defined that $\varnothing = 0$, $1 = \{\varnothing\}$, $2 = \{\varnothing, \{\varnothing\}\} = \{0, 1\}$, and so on. In set theory, every element of the natural numbers is a set itself. This is not the case in B, and an attempt to talk about an element of an $n \in \mathbb{N}$ gives a type error in Atelier B during static analysis.

To make the distinction clear between integers and sets of integers as defined above, we will use the notation $[n]$ for some $n \in \mathbb{N}$ to indicate the set of all integers smaller than n, which is: $[n] = \{0, 1, ..., n-1\}$. The B syntax has an abbreviation for it, and $[n]$ can be denoted as `0..n-1`. More generally, this B notation means a segment of natural numbers - for $m, n \in \mathbb{N}$, `m..n` $= \{x|x \in \mathbb{N} \wedge m \leq x \wedge x \leq n\}$

A **Cartesian product** of two sets X and Y is the set of all pairs $\{(a, b)|a \in X \wedge b \in Y\}$. It is denoted $X \times Y$ or `x*y`. The pairs, also called **maplets** are written as `a |-> b` in the B syntax. They are the elements of relations, functions, and sequences in the B language, and while we will not go into the details of the notation here.

With the help of those definitions we can finally define a finite set. A **finite** set is a set x such that there exists a bijection between x and $[n]$ for some $n \in \mathbb{N}$. The B syntax also provides an abbreviation for the set of all finite subsets of a set x. `FIN(x)` is hence defined to be $\{y|y \in \mathbb{P}(x) \wedge y$ is finite$\}$.

Then the **cardinality** of x is the number of elements in x, in this case n, and is denoted $|x|$ or `card(x)` in the B syntax. Note that in the B language, the expression `card(x)` is well-defined only for finite sets.

B. Naming Conventions

Throughout this work we will keep to the following naming conventions.

Variables and constants in the B language are named according to the following rules:

- names of sets, including deferred sets and those given as machine parameters, are written in all capitals. In this text, they will also be written in a monotype font, for example `ITEMS`.
- scalar constants' names shall be written in lower case, in a monotype font, for example `max_elem`.
- names of variables shall be written in lower case, in a monotype font, and must be at least two characters in length, for example `aa`.

Single-character variable names are not allowed in the B-syntax, and so we will avoid them in all contexts. They are reserved for wildcards in user-written proof rules.

Furthermore, file names will be written in italic. In the case of Atelier B machine files, the extensions will be omitted. The files will be named according to the following pattern: *[name of the reference machine]_[abbreviated description of the variation]*. Machine clause names will be written with the first letter upper case, unless they are part of a code snippet.

III. Literature review

A. Industrial examples

Although this work is aimed at students and researchers, it is necessary to put it into perspective of how formal methods can be used in industry. Looking at the applications of formal methods will serve to demonstrate that this work indeed serves a purpose and that there are users who may want to refer to the discoveries done in this project in order to facilitate their work.

As mentioned in the Introduction, formal methods are primarily used in safety-critical areas, where testing is difficult or insufficient as an evidence of the correctness of the software, while failure can result in injury or loss of life.

In particular, the B method is popular in the railway industry. Their use is supported by standards and regulations which describe the requirements for software controlling signalling, communication and processing. [2].

A prime example of an application of the B-method is the project launched by Siemens Transportation System in collaboration with ClearSy, to develop a driverless, automated shuttle for the Charles de Gaulle Airport [3]. They have used specifically the B method as a high-level language, because of it being well-suited for proving properties.

ClearSy

Less directly applicable, although very interesting is the effort of Reichl *et al.* to model a train station in Event-B, using the Rodin tool [4]. The software which could be built from this model would route trains and control signalling between tracks and intersections, so that collisions are avoided and a train efficiently progresses towards its destination. The aim of this work is to demonstrate what can be achieved in terms of modelling complex interlocking systems with Event-B, and what are the limitations of such models. Of the latter, the key observation is that all stations are different and require careful changes to the model, the source code for which is available online. Furthermore the authors observe that the tool they have used is not yet industry-ready, and there are some promising alternatives under development. This was a very large-scale model in comparison and the authors may have set the bar too high, given the currently available tools. As we have seen before, direct applications of B method or Event-B tend to focus on smaller systems, such as automated doors on a platform.

Other industrial examples of formal methods in use include Amazon Web Services, which use highly complex systems. [5] The engineers there have observed that human intuition is prone to errors and that formal methods help find subtle bugs, which can be easily overlooked by other means of checking the correctness of the software. They use TLA+, a formal specification language developed by Lamport [6]. The Amazon Web Services engineers also acknowledge that formal methods in industry have the reputation for requiring long and expensive training before they can be used with confidence, however they have found this opinion to be false. This may be specific to the method they have chosen, but nevertheless it is encouraging that people are questioning such opinions, and are open to trying out this approach despite them.

IV. Project overview

A. The aims of the project

The key aim of this project is to discover the limitations of the Atelier B software. In the previous section, we have mentioned multiple plugins and extensions to the Atelier B software, which were created in order to improve the functionality. They highlight what other users have found lacking in Atelier B and what they have considered in need of improvement. However none of them demonstrate what can be achieved with the original software alone. Hence we strive to assess at what point Atelier B alone becomes insufficient, and the extensions are necessary to successfully verify a project.

We intend to explore various ways of expressing a specification in the B language, paying attention to how seemingly equivalent expressions result in different proof obligations being generated by the automated prover, and follow it up by seeking an explanation of the differences using the information contained in the source texts. [CITE] illustrates the scale of the proofs in industrial projects and serves to show that reducing the number of proof obligations which require user input to discharge, can greatly save man-hours.

This approach is often taken by students and academics beginning their work with the B-method and the Atelier B software. They are unlikely to have a thorough knowledge of the intricacies of the B-method as implemented in the Atelier B software, and would attempt to write a machine first, then search the manuals to explain unexpected behaviours of the software. The software itself has been found to be unintuitive and not user-friendly, as noted by [CITE].

From our experience and observations, new users, especially students, tend to blindly follow methods and patterns which their colleagues have found to be effective, without giving much thought to why this works and if its applicable in their particular situation. This phenomenon has been sometimes described as 'cargo cult' in the programming community [13], and is disapproved off, as it often leads to unnecessary obfuscation of the code and decrease of performance in a program. One of such tips we have come across, which served as an inspiration for this work, was to deal with undischarged proof obligations by putting their goal in the Invariant clause of a machine. While this method was found to be working, there are more effective means of reducing the number of proof obligations generated, and thus the time taken to complete the proof.

We hope to provide the new users with an explanation for the most commonly encountered proof obligations which are not automatically discharged, and persistent patterns among them. Both understanding the information given by the automated prover and avoiding undischarged proof obligations in the first place is of interest.

An extension to this aim is answering the question: at what point is it necessary to add user-created rules in order to facilitate the proofs in the automated prover? It needs to be stressed that adding user-created rules is not advised unless it is absolutely necessary - and every manual as well as the works discussing methods of validating such rules stress this point. The rules then have to be thoroughly verified by means other than Atelier B software, to ensure that they are in fact correct. An error in an added rule would invalidate the entire proof

process. Hence we will strive to circumvent obstacles by all means other than adding proof rules, before resorting to it. We hope to gather such advice and guidelines and make it accessible to others, so that they may be dissuaded from unnecessarily adding proof rules in their work. It is possible that adding proof rules to the Atelier B's rule base may be avoided entirely in the scenarios we have chosen, as they are by no means exhaustive; otherwise we will present the rules and their proofs for the use of others who wish to study the B-method.

The observations arising from our work can be separated into two groups. Firstly there will be points highlighting the intricacies of the B method, which can be reasoned about and explained easily using source texts, primarily Abrial's *The B-book* [15] and Schneider's *The B-Method* [14].

Secondly, there will be disparities between the B method and its implementation in the Atelier B software. Not all of them can be classified as bugs, and some are clearly conscious choices which diverge from the pure theory of the B method. The manuals provided with the software will aid us with comparing the implementation to the theory. They are:

- *Atelier B 4.0 User Manual* [16]
- *B Language Reference Manual* [17]
- *Proof Obligation Manual* [18]
- *Interactive Prover Reference Manual* [19]

B. The scope of the project

This work focused on abstract machines - they are the first step towards a formally verified implementation of a specification, thus making them the foundation of any project developed using the B method. An implication of this choice is focusing on set-related structures, including relations which are understood as sets of maplets, and operations such as set comprehension, union, and intersection. These abstract constructs are not allowed in the later stages of the development, where all data structures have to be concrete, and operations deterministic. We will also not discuss proof obligations related to loops, since those arise in the implementation stage and are not permitted in the abstract machine, because of their sequential nature which requires temporal logic to reason about them. Similarly, we will not analyse proof obligations related to structuring of machines. We wish to focus on a narrower scope and thoroughly explore it, rather than spread our resources too thinly and overlook some details.

Another reason for this choice is the distinct lack of literature focusing on this stage of development, in comparison to the refinement and implementation stages.[CITE] The latter stages undeniably generate more proof obligations, since on top of the proof obligations which can appear in the abstract machines, we need to consider those related to connection between a refinement machine and the machine being refined.

Similarly, as can be seen in the literature review, there has been significant amount of work done on verifying user-created proof rules, despite commonly seen advice to use this option as a last resort to prove correctness. We have found little discussion on how to avoid writing user's own proof rules.

We shall approach this project from an academic rather than industrial point of view, focusing on smaller yet more illustrative examples. The main reason for it is to limit the number of factors affecting the number of generated proof obligations and be able to control them more precisely. An industrial-scale project with hundreds of proof obligations would be rather unwieldy for our purposes. Secondly a person new to formal development and the B-method is more likely to be able to follow clear, exemplar scenarios.

Nevertheless we hope that not only students and researchers will find our work helpful. Being able to minimise the number of proof obligations to be manually discharged has the potential to reduce the time required to complete any project. The constructs and expressions we discuss are the same ones as those used in industry. In fact, we found that the more complex structures in the B language, such as sequences, are rarely used in large-scale projects, as exemplified by [CITE].

V. Methodology

The *Interactive Prover User Manual* [19] outlines the part of the proof process concerning the abstract machine as follows:

1) write the abstract machine
2) check that the specification has been formalized correctly
3) launch the Automatic Prover on this abstract machine
4) if not all proof obligations are demonstrated automatically, check that they are true and indeed should be demonstrated. If they are not, the machine is erroneous.

It then encourages the users to write the implementation, and if undischarged proof obligations remain after that, only then the users are told to use the Interactive Prover. We acknowledge this advice. However the steps listed above are nevertheless very general, and in particular the users are not advised on how to establish whether an undischarged proof obligation is beyond the

capabilities of the Automatic Prover or simply false. hence, we intend to use the Interactive Prover at the abstract machine stage of the development in order to assess this.

It should be accentuated that quite often undischarged proof obligations do not immediately appear false. From our experience, this is in particular true for the well-definess

VI. Project Management

A. Choice of scenarios

This work has focused on two generic scenarios and explored various ways of fulfilling a specification for each of them. The specifications were kept deliberately imprecise for two reasons. Firstly, it allowed for an in-depth exploration of the theme. More precise specification would narrow down the options significantly, limiting our findings. Secondly in an industry setting it is not unheard of to have specification documents which leave details up to interpretation or are open-ended.

After exhausting the ways each expressing the specification in the B-method, we created a few more machines for each scenario, which illustrate other variations on the theme, although they diverge further from the original intentions. They served to analyse constructs which are more particular or less suited to the chosen scenarios, but nevertheless not unheard of.

It is important to observe that in large-scale industry projects which apply the B method, the constructs are kept simple and straightforward to avoid obfuscation. Thus in the sections below, dedicated to the chosen scenarios, the examples created were ordered by complexity. The later ones, for example in the case of the Bag Machine those involving sequences or relations, were analysed to compare the number of proof obligations generated next to their simpler variants.

VII. The Bag Machine - various approaches to describing sets and collections

A. Specification

Given a set of items, we want to describe a bag containing some of them. Initially, the bag is empty. We can perform the following operations on the bag:

- add an item to the bag
- remove an item from the bag
- find out the number of items in the bag
- find out which items are in the bag
- query whether a given item is in the bag

B. Discussion of the specification

We take the metaphorical bag to be as generic as possible and attempt to interpret the specification in any reasonable way. The details are deliberately left up to interpretation - for example it is not specified whether multiple copies of an item can be included or not, thus making the content of a bag into a set following the Zermelo-Fraenkel definition, or a multiset. Both possibilities will be explored.

An image of a bag of items was chosen due to its simplicity, although we recognize that it is potentially an unhelpful deviation from the most general description of this scenario, which can be achieved solely in set-theoretical terminology. We argue that this scenario is applicable in many circumstances. For example, one may be asked to develop a system that controls the barriers to a private car park. Then the system would maintain a set of registration numbers of the vehicles permitted to park there, which is a subset of all possible registration numbers. Another example is a library system, where `ITEMS` is the set of books in the library, and `content` are the books a person has on loan. We may want to impose a limit on the total number of items in the subset - bound by the number of spaces in the car park or the maximum number of books permitted to have on loan at the same time. This variation, although not explicitly required by the specification, was analysed in depth among others listed below.

The aim of this scenario is to explore different ways of expressing sets and operations on them. The two sets we are working on are `ITEMS`, consisting of all possible items that can be put in the bag, and `content`, the items contained in the bag at a given point. There are various ways of describing each of these sets.

We take the relation between them firstly to be that of subset, namely $\texttt{content} \subseteq \texttt{ITEMS}$ or equivalently $\texttt{content} \in \mathbb{P}(\texttt{ITEMS})$. This already shows two different ways of expressing a simple relation like this. Furthermore we may want to impose the limitation that the content of the bag is a finite set, thus arriving at $\texttt{content} \in \texttt{FIN}(\texttt{ITEMS})$ in B notation, where `FIN(ITEMS)` denotes all finite subsets of the set of `ITEMS`.

There are other ways of describing the relation between the `ITEMS` and the `content` of the bag. For example latter can be a mapping from a subset of `ITEMS` to the number of times a given item appears in the bag.

At the level of abstract machines, it is permitted to use set comprehension and other operations and properties of sets, according to the Zermelo-Fraenkel set theory. The B language offers abbreviations of some more common

expressions, such as domain or range restrictions. Another thing for us to explore is how using these shorthand expressions rather than writing them out fully affects the proof process.

C. Variants of the Bag Machine

There are various ways of expressing the requirements given in the specification as an abstract machine. The following files are contained in the *Test_scenarios* archive for reference of the reader. We begin by having ITEMS as a deferred set - a set which will be defined at some later point of the development process, and content as simply a subset of ITEMS, then inspect different ways of including the set of ITEMS in the machine, then we focus on the relation between the two sets as expressed in the Invariant clause. We then move onto different ways of expressing the set of ITEMS, for example as an enumerated set or one of a basic type. We finally explore various ways of describing the content of the bag, such as using sequences or relations. The reason for this order of tasks is to begin with the most intuitive implementation of the specification, before discussing less obvious changes to it.

1) Bagmch: is the reference machine which we take to be the core of this scenario. It implements exactly the specification without imposing any non-required conditions, such as the limit on the number of items in the bag. At the same time it includes one condition not explicitly mentioned in the specification, namely:

```
INVARIANT
    content : FIN(ITEMS)
```

The specification requires only that content $\subseteq$ ITEMS, however in any implementation it is infeasible to have truly infinite sets, thus the software considers them to be erroneous. In the *B Language Reference Manual* [17] we find the following definition for the set of natural numbers: NAT $=$ 0..MAXINT, where MAXINT can be set by the user for a given project, although it is usually understood to be $2^{31} - 1$. This definition is not supported by the *B-book*, thus demonstrating a small disparity between the theory of the B-method and its implementation in Atelier B. Nevertheless it is very reasonable for practical purposes.

This machine generates only four proof obligations and all of them are discharged automatically. The first three check that the INVARIANT is preserved in the initialisation and by the operations to add and remove items - the only three actions affecting the state of the bag. The last one is as follows:

```
"Well definedness"
    =>
        content: FIN(content)
```

It is concerned with the well-defineness of the operation howmany, which returns the number of items in the bag. The well-defineness proof obligations arise when an expression is used which requires certain conditions to be met in order to be well-defined. In this case, card(content) is well-defined only if content is a finite set. The well-definess conditions for all such expressions can be found in *B Language Reference Manual* [17]. The following machine illustrates the proof obligations generated when the well-definess conditions are not met.

This machine is shown in Appendix A for ease of access.

2) Bagmch_unbounded: illustrates the necessity to impose finiteness on the set of items contained in the bag. The sole difference between this and the reference Bag Machine is the statement:

```
INVARIANT
    content <: ITEMS
```

This machine generates the same four proof obligations as the reference machine, however the last one remains undischarged by the automated prover and correctly points the user to a problem in the abstract machine.

3) Bagmch_pre: shows a workaround the well-defineness proof obligation in the previous example.

4) Bagmch_params: differs from the reference machine in the way the set of ITEMS is included in it. *Bagmch* is given the set of ITEMS as a deferred set, which is required to be explicitly defined at a later stage of the development. Here, it is given as a set-valued parameter, which must be instantiated any time the machine is used. [14] However at the stage of abstract machine which is not used by any other machine, it does not make a difference in the prover.

Similar behaviour can be observed later in the machines *Bagmch_limited* and *Bagmch_limited_params*, with a scalar-valued parameter.

5) Bagmch_long_inv: demonstrates that a provably equivalent way of writing an expression may lead to a different behaviour of the prover.

There are multiple ways to denote subsets and finite subsets in these two machines. For unbounded subsets content $\subseteq$ ITEMS is equivalent to content $\in$ POW(ITEMS). We include the latter form in the Invariant of this machine

As we have discovered earlier, we need to limit the content of the bag to only finite subsets of ITEMS in the Invariant clause. A phrasing has been suggested by the well-defineness proof obligation generated by the

reference machine. Thus, another way of expressing it to the one seen in the *Bagmch* is:

```
INVARIANT
    content : POW(ITEMS) &
    content : FIN(content)
```

This time there are six proof obligations generated, all discharged automatically, and they are concerned only with the Invariant being preserved. There is a proof obligation for each clause of the Invariant, for each one of the initialisation and the two operation affecting the content of the bag.

Firstly, comparing the proof obligation related to the first clause of the Invariant, we see that it is not as simple as the one in *Bagmch_unbounded*. For example in the initialisation we see:

```
"Invariant is preserved" &
"Check invariant ((content):(POW(ITEMS)))"
=>
    {} <: ITEMS
```

Where we previously had:

```
"Invariant is preserved" => {} <: ITEMS
```

It strongly suggests that the notation $a \subseteq b$ is preferable to $a \in \mathbb{P}(b)$, for any two sets a and b. While the result is still the same, the former seems to require less work from the prover, since fewer clauses are generated. Thus especially for large projects may save some computation time. Again, on a small scale like this we have no means of measuring it due to the overhead.

Next we compare the proof obligations regarding finiteness of `content`. The one generated now are of the form:

```
"Invariant is preserved" &
"Check invariant ((content):(POW(ITEMS)))"
=>
    {} : FIN({})
```

We can compare it to the finiteness POs in the reference *Bagmch*:

```
"Invariant is preserved" => {}: FIN(ITEMS)
```

This change in expression does not appear to make a difference for the proof obligations regarding the Invariant being preserved. However the key distinction between this machine and the reference is that now there is now no proof obligation concerning the well-defineness of the expression `card(content)`. In this case, by adding the goal of a proof obligation regarding well-defineness, as it was seen before, to the Invariant, we have avoided the proof obligation altogether. A similar behaviour will be observed in later examples.

We conclude that these two ways of expressing the relation '`content` is a finite subset of `ITEMS`', while equivalent in theory, will affect the performance and the time taken to complete computations by the prover in different ways. One will result in more proof obligations being generated, however they will be of a simpler nature and discharged automatically, which is faster than discharging even a single proof obligation manually, such as the well-defineness one in the latter case. We recognize that it is something that may be noticeable at large-scale projects, while in a small scenario the overhead will severely affect any measurements of performance, and therefore we have no means of exploring it further in this work, but we highlight it as something users should be aware of.

[CHECK - trace system]It is regrettable that the prover does not allow us to inspect the proof rules applied when proof obligations are automatically discharged, leaving us with speculations about its inner workings.

6) Bagmch_finite_items: is an attempt to impose finiteness earlier on, on the set of `ITEMS`. We do it by adding the expression '`ITEMS` $\in$ `FIN(ITEMS)`' to the Properties clause. Otherwise the machine is identical to *Bagmch_unbounded*. We rely on the fact that any subset of a finite set is necessarily finite. Appendix B contains a proof of this claim.

We see the same four proof obligations as in *Bagmch_unbounded*, and again the one regarding well-defineness is not discharged automatically. We can of course add the clause '`content` $\in$ `FIN(content)`' or otherwise explicitly impose the finiteness property on the contents of the bag, but it should not be necessary. The machine is correct, and as the aforementioned proof suggests, it can be verified manually.

Thus we move on to the next stage in our process, as described in the Methodology section. The interactive prover allows users to search rules by the goal, as well as browse the file containing the integrated proof rule base. However none of the rules contains the goal required. No other manipulation in the interactive prover leads to discharging the proof obligation either.

The claim 'a subset of a finite set is finite' can be expressed as a B rule as follows:

```
THEORY userInFINXY IS
    band(binhyp(b: FIN(b)),
    binhyp(a <: b))
    =>
    a: FIN(a)
END
```

The notation means that the conjuncts (AND) of the two hypotheses: 'b is a finite set' and '$a \subseteq b$' implies the goal, that a is a finite set as well.

This rule can be added to the *.pmm* file for this component, so that it is not visible during other proofs, or to the Patch Prover, which contains user-added rules available to all components in a project. Either way it has to be then called explicitly in the interactive prover, with the command to apply rule 'ar(<theory name>)', to discharge the obligations with this goal. The project archive contains the *Bagmch_finite_items.pmm* file with this rule to illustrate how it works.

An alternative to including this rule in the PatchProver or the *.pmm* file is to add the goal of the proof obligation to the Invariant, as can be seen in *Bagmch_long_inv*. This is a safer alternative, because it will certainly not permit incorrect statements to be accepted by the prover, although it may restrict the number of legal states of the machine. At the same time it leads to more proof obligations and it can be considered the user performing part of the proof manually.

7) Bagmch_constant_set: is very similar to the previous machines. This time the superset of `content` is a set described in the Constants clause, with the following properties:

```
CONSTANTS
    items
PROPERTIES
    items : FIN(ITEMS)
```

Thus we arrive at a sequence: `content` $\subseteq$ `items` $\subseteq$ `ITEMS`.

The behaviour of this constant is identical to that of the deferred set of `ITEMS`, as seen in *Bagmch_finite_items*.

8) Bagmch_constant_set2: is a combination of the previous two machines. The finiteness is imposed on the set of `ITEMS`, but the sets are related as in *Bagmch_constant_set*. This is captured in the Properties clause:

```
PROPERTIES
    ITEMS : FIN(ITEMS)
    items <: ITEMS
```

The `howmany` operation generates the well-defineness proof obligation as expected, and it does not get discharged automatically. This time it is really necessary to guide the interactive prover along. The prover does not deal well with the sequence of sets, and the user is required to take the interactive prover through it step by step. First it is necessary add a hypothesis that `items` is a finite set - using the command `ar(items : FIN(items))`, then apply the rule which was discussed earlier, twice.

The whole process, including the user input, is recorded in the user pass file and the part regarding the troublesome proof obligation looks as follows:

```
Operation(WellDefinedness_howmany) &
ff(0) & pr & ah(items : FIN(items)) &
ar(userInFINXY) & pr & ar(userInFINXY)
```

Where `ff(0)` denotes automatic proof with force parameter equal to 0 - i.e. with the shortest timeout, the default being 10 seconds, since in a simple scenario like this the prover is not expected to process proof obligations for long. The command `pr` is simply a call to the automatic prover, and `ar(<theory name>)` is applying the user-created rule as before. Adding a hypothesis means that, if the goal is G, the current hypotheses are $h_1, ..., h_n$, and the new one P, then the prover will first attempt to demonstrate P under the hypotheses $h_1, ..., h_n$, and then prove G under the hypotheses $h_1, ..., h_n, P$. [18] Thus the additional hypothesis has to be demonstrated first, which is the reason for applying the added rule twice.

This is a clear inconvenience when a proof requires a longer sequence of sets to have a certain property proven one by one. Fortunately, user passes as the one listed above can be added to the component's *.pmm* file and called to perform the sequence of commands with a single input from the user, as described in Chapter 5 of the *Proof Obligations Reference Manual*.

9) Bagmch_limited: imposes a limit on the number of items that can be included in the bag at once. We define this limit as `max_elem` in the `CONSTANTS` clause of the static part of the machine and give its type in the `PROPERTIES` clause:

```
CONSTANTS
    max_elem
PROPERTIES
    max_elem : NAT
```

Here, we explicitly restrict `max_elem` to be non-negative. Changing this expression to '`max_elem` $\in \mathbb{Z}$' results in an unprovable proof obligation in the Initialisation, the goal of which is $card(\{\}) \leq$ `max_elem` for all possible values of `max_elem`. It clearly does not hold for negative values of the constant, hence it is limited to natural numbers.

Furthermore, there is no difference between defining just the type of `max_elem` or its value, forgoing the type declaration, i.e. having only the clause '`max_elem` $= x$' for some chosen $x \in \mathbb{N}$ in the Properties clause. The type of it is defined implicitly and it does not affect the proof process.

This restriction impacts the `additem` operation, as we now need to check that adding an item to the bag will not exceed the limit. It can be done in the preconditions section of the operation, as is shown in the example included in the archive, or in an if-else statement. We

found these ways to be equivalent for our purposes, as they do not generate different proof obligations. It is worth noting that they will impact the refinement stages of the development. We have elected to stick to the former, since it is less restricting for potential refinement.

There are nine proof obligations generated. The initialisation and each of the operations to add and remove items results in two: one being a type check, the other making sure that the cardinality of `content` does not exceed `max_elem`. The last three proof obligations are concerned with the well-definess of the expression `card(content)` in the invariant and the operation `additem` and `howmany`, that is anywhere where the expression appears.

If the expression `content ∈ FIN(ITEMS)` is replaced with `content ⊆ ITEMS ∧ content ∈ FIN(content)` there are again nine proof obligations. However this time there are three proof obligations for each of the Initialisation and the adding and removing operations - one proof obligation for each clause of the Invariant. At the same time the well-defineness proof obligations are not generated. This is in line with the observations regarding *Bagmch_long_inv*.

10) Bagmch_limited_params: differs from the previous version by passing `max_elem` as a scalar-valued parameter instead. The software requires defining at least the type of `max_elem` in the Constraints clause, while the Constants and Properties clauses can now be removed. As in the case of *Bagmch_params* with a set-valued parameter, it does not affect the proof process.

11) Bagmch_wrong_order: is of particular interest, as it highlights a discrepancy between the theory of the B-method and its implementation in Atelier B. The Invariant clause contains a series of conjuncts which by the rules of logic are commutative. However they are not such in the software.

Firstly, recall that in the *Bagmch_long_inv*, the Invariant was as follows:

```
INVARIANT
    content <: ITEMS &
    content : FIN(content)
```

Switching these two statements around results in an error message generated as static analysis is taking place, demanding that the type of `content` is defined before applying the built-in operator `FIN` to it. The software does not allow the user to proceed with the proof as long as such errors exist.

With the addition of the limit on the number of items in the bag, this problem becomes much less obvious. For example, if the Invariant is formulated like this:

```
INVARIANT
    content <: ITEMS &
    card(content) <= max_elem &
    content : FIN(content)
```

The static analysis does not give any errors, however an additional proof obligation is generated, on top of the nine we get in the alteration to the *Bagmch_limited* mentioned above. The proof obligation is:

```
    content <: ITEMS &
    "Well definedness"
=>
    content: FIN(content)
```

It does not get discharged by any means available to the user within the interactive prover.

Based on these observations we reach the conclusion that the conjuncts in the Invariant are applied sequentially, in the order in which they are written. It is very much like the Assertion clause, which is defined to be checked sequentially in the B method. It is also understandable from the implementation point of view, since checking all permutations of the conjuncts would be computationally hard.

12) Bagmch_redundant...: are four machines with redundant clauses in the Invariant, written in a different order or in a slightly different manner, but essentially nigh identical. Note that it is a very bad practice to do something like this intentionally, however as this work is aimed at people beginning their work with the B method, it is expected that a similar mistake, albeit a less obvious one, can happen.

They borrow the concept of imposing the maximum number of items that can fit in the bag from the previous machines. This time we add redundant clauses in the Invariant to explore the relation between their number and the number of proof obligations generated, as well as observe if there is any evidence of optimisation in the prover.

We begin with *Bagmch_redundant*. Just like in *Bagmch_limited* it has a constant `max_elem` which is then described in the Properties clause as `max_elem : NAT`.

The Invariant now looks as follows:

```
INVARIANT
    content : POW(ITEMS) &
    content : FIN(content) &
    card(content) <= max_elem + 4 &
    card(content) <= max_elem + 3 &
    card(content) <= max_elem + 2 &
    card(content) <= max_elem + 1 &
    card(content) <= max_elem
```

Firstly note that the third, fourth, fifth, and sixth clauses are redundant - it is sufficient that the last clause holds

for these four to also hold. At the same time it is a very simple way of creating any given number of Invariant clauses. Also, observe that we have used the more explicit way of describing the type and finiteness of `content` - as a separate clause for each of those requirement, as seen in *Bagmch_long_inv*. This way we avoid the well-defineness proof obligations.

There are 21 proof obligations generated in this case - 7 for each of the two operations affecting `content` and 7 more for the Initialisation. The Initialisation and the operation to add an item have all their proof obligations discharged automatically. Surprisingly, out of the five cardinality-related proof obligations for the operation to remove an item, only the simplest on, namely the one regarding the clause `card(content) <= max_elem`, gets discharged automatically. The other ones require user input in the interactive prover.

They can still be discharged without creating additional rules. For each one of them, the same steps work, since their structure is identical. Each one has two goals - one for the case when the item the operation is removing is an element of `content`, the other when it is not. Their initial goal is of the same form as the simple proof obligation:

```
card(content-{aa})<=max_elem+1
```

If we run the 'prove' command (denoted by 'pr' in the prover), the goal is rewritten into:

```
0<=1+max_elem-card(content-{aa})
```

We notice that the simplest proof obligation was discharged, so we add a hypothesis consisting of it rewritten in the way appearing in the goal. It is done with the command:

```
ah(0<=max_elem-card(content-{aa}))
```

Where 'ah' stands for the 'add hypothesis' command. We then prove this additional hypothesis with 'prove' and the goal turns into:

```
0<=max_elem-card(content-{aa})
=> 0<=1+max_elem-card(content-{aa})
```

Running 'prove' again satisfies this goal and moves onto the next one:

```
not(aa: content)
=> 0<=1+max_elem-card(content)
```

This one is satisfied with just the 'prove' command. Thus the User Pass for the operation 'removeitem' is recorded as follows:

```
Operation(removeitem) & ff(0) & pr &
ah(0<=max_elem-card(content-{aa})) &
pr & pr & pr
```

We have found a way of discharging these proof obligations, by comparing the goal to one which the prover has successfully dealt with before. We have essentially given the prover a simpler hypothesis, stripped of the additional scalar constants.

We do not have an explanation for why these proof obligations have been problematic. The prover has not timed out while processing them, and indeed attempting to discharge them with a greater force parameter does not change the outcome.

Bagmch_redundant2 uses a Definition clause as follows:

```
DEFINITIONS
    max_elem == 3
```

It is equivalent to defining `max_elem` as a constant in the Constants clause and giving it a value in the Properties clause. This method is advised by the *Interactive Prover User Manual* [19], which claims that it prevents the prover from performing avoidable replacements. *Bagmch_redundant3* differs by replacing the expressions '`max_elem + n`' by concrete natural numbers. In both cases all of the proof obligations are discharged automatically.

Given the previous findings about the conjuncts in the Invariant not being commutative, we have also explored ordering the clauses from the most to the least restrictive - i.e. reversing the order of the conjuncts from the Invariant shown above. We found that it did not change the outcome in this case. In *Bagmch_redundant_reverse* `max_elem : NAT` is used just like in *Bagmch_redundant* and the same proof obligations are generated and not discharged. In *Bagmch_redundant_reverse2* we use numerical values instead of a constant in each conjunct, and all proof obligations are observed to discharge automatically.

We move on to discuss the number of proof obligation generated in relation to the number of clauses in the Invariant. Recall that the machine has two operations which change the `content` variable. Each one of them and the Initialisation generates one proof obligation for each clause in the Invariant. With the Invariant as given above, it comes to three sets of seven proof obligations. Adding clauses following the pattern '`card(content) <= max_elem + n`' and checking the number of generated proof obligations demonstrates the following correlation: the number of proof obligations regarding the preservation of the Invariant is equal to the number of operations affecting the state of the machine plus one for the Initialisation multiplied by the number of clauses in the Invariant. However at this point all the clauses are related to the variable affected by

the operations. We will see if having clauses concerning other variables, unaffected by these operations, changes the pattern, in *Bagmch_2sets*.

VIII. Results

A. Summary of the work done

B. Findings and observations

Number of proof obligations

Equivalence of expressions

Ordering of the Invariant

Proof obligations in card(aa) - finiteness implied - vs xx = 4 - type implied

Sequences of sets

C. Additional rules

Even though at the very onset of this project we have anticipated the necessity to add multiple rules to ensure a smooth proof process, we have found that none of them have been truly mandatory. All of the situations where one might be tempted to write an additional rule could be circumvented by slight rephrasing of the machine, as outlined in the previous part of this section.

The only rule that was written and verified, and which is considered to be of some potential use to new users, is the rule capturing the claim that every subset of a finite set is finite. The rule is:

```
THEORY userInFINXY IS
    band(binhyp(b: FIN(b)),
    binhyp(a <: b))
    =>
    a: FIN(a)
END
```

It can be found in the *Bagmch_finite_items.pmm* file included in the project archive.

IX. Concluding Remarks

It was an initial goal of this project, to create a collection of proof rules which simplify the verification process in the automated prover. This collection was meant to include especially the rules that were found to be needed for the proofs of various scenarios. However as the work has progressed, we have found that it was not necessary to create additional rules, and instead any obstacle could be circumvented with a deeper understanding of the inner workings of the prover.

Instead we

X. Acknowledgements

References

[1] S. Conchon and M. Iguernlala, "Increasing Proofs Automation Rate of Atelier-B Thanks to Alt-Ergo" in *Proc. 1st Int. Conf. Reliability, Safety and Security of Railway Systems*. Paris, France, 2016, pp. 243-253

[2] *Railway applications - Communication, signalling and processing systems*, EN 50128, 2011

[3] F. Badeau and A. Amelot, "Using B as a High Level Programming Language in an Industrial Project: Roissy VAL" in *Proc. 4th Int. Conf. Z and B Users*, Guildford, UK, 2005, pp. 334-353

[4] K. Reichl et al., "Using Formal Methods for Verification and Validation in Railway" in *Proc. 10th Int. Conf. Tests and Proofs*, Vienna, Austria, 2016, pp. 3-13

[5] C. Newcombe et al., "How Amazon web services uses formal methods" in *Commun. of the ACM*, vol. 58, New York, 2015, pp. 66-73

[6] L. Lamport, "Specifying Concurrent Systems with TLA+" in *Calculational System Design*, Amsterdam, IOS Press, 1999, pp. 183-247

[7] M. Jacquel et al., "Verifying B Proof Rules Using Deep Embedding and Automated Theorem Proving" in *Proc. 9th Int. Conf. Software Eng. Formal Methods*, Montevideo, Urugway, Nov. 2011, pp. 253-268

[8] D. Déharbe, "Integration of SMT-solvers in B and Event-B development environments" in *Sci. Comp. Prog.* vol. 78, Elsevier, March 2013, pp. 310-326

[9] M. Leuschel et al., "Automated property verification for large scale B models with ProB" in *Proc. 2nd Int. Symp. Formal Methods*, Eindhoven, The Netherlands. 2009, pp. 708-723

[10] V. Medeiros Jr. and D. Déharbe, "BEval: A Plug-in to Extend Atelier B with Current Verification Technologies" in *Proc. 1st Latin Amer. Workshop Formal Methods*, Buenos Aires, Argentina, 20014, pp. 53-58

[11] *Atelier B version 4.2 release notes*, ClearSy Sys. Eng., Aix-en-Provence, France, 2014

[12] D. Goldrei, *Classic set theory*, Chapman and Hall/CRC, Boca Raton, 1998

[13] N. Bezroukov, "Cargo Cult Programming" in *Softpanorama* [Online], Open Source Software Education Society. Retrieved 21 August 2017. Available: http://www.softpanorama.org/Skeptics/cargo_cult_programming.shtml

[14] S. Schneider, *The B-Method: an Introduction*, Basinstoke, Palgrave, 2001

[15] J.-R. Abrial, *The B-book: assigning programs to meanings*, Cambridge, Cambridge Univ. Press, 1996

[16] *Atelier B User Manual*, v. 4.0, ClearSy Sys. Eng., Aix-en-Provence, France

[17] *B Language Reference Manual*, v. 1.8.7, ClearSy Sys. Eng., Aix-en-Provence, France

[18] *Proof Obligations Reference Manual*, v. 3.7, ClearSy Sys. Eng., Aix-en-Provence, France

[19] *Interactive Prover User Manual*, v. 3.7, ClearSy Sys. Eng., Aix-en-Provence, France

[20] *Redaction guide for mathematical rules* v. 1.1, ClearSy Sys.ng., Aix-en-Provence, France, accessible: http://www.atelierb.eu/wp-content/uploads/sites/3/manuels/guide-de-redaction-de-regles-mathematiques-fr.pdf
Translated by D. Déharbe, accessible: http://www.math.pku.edu.cn/teachers/qiuzy/fm_B/Atelier_B/Writing_mathematical_rules.pdf

APPENDIX A
THE REFERENCE BAG MACHINE

```
MACHINE
    Bagmch
SETS
    // possible items we can put in the bag
    ITEMS
VARIABLES
    // contents of the bag
    content
INVARIANT
    content : FIN(ITEMS) // content is a *finite* subset of ITEMS
INITIALISATION
    content := {} // we start with an empty bag
OPERATIONS
    /* Adds item aa to the bag*/
    additem(aa) =
    PRE
        aa : ITEMS
    THEN
        content := content \/ {aa}
    END;

    /* removes aa from the bag (does nothing if aa not in the bag) */
    removeitem(aa) =
    PRE
        aa : ITEMS
    THEN
        content := content - {aa}
    END;

    /* getter for the content*/
    items <-- getcontents = items := content;

    /* query how many items are in the bag */
    nn <-- howmany = nn := card(content);

    /* checks if the item aa is in the bag */
    check <-- isin(aa) =
    PRE
        aa : ITEMS
    THEN
        IF
            aa : content
        THEN
            check := TRUE
        ELSE
            check := FALSE
        END
    END
END
```

APPENDIX B
'A SUBSET OF A FINITE SET IS FINITE' - PROOF

Let A and B be sets, with $A \subseteq B$ and B finite. Let us define $[n]$ to be the set of all elements of $\mathbb{N}$ less than n, i.e. $[n] = \{0, 1, ..., n-1\}$.

Since B is finite, by the definition of finiteness there is $n \in \mathbb{N}$ such that there exists a bijection between B and $[n]$. Hence it suffices to prove that any subset of $[n]$ for $n \in \mathbb{N}$ is finite. We proceed by induction.

When $n = 0$, $[n] = \varnothing$, and trivially all subsets of the empty set are finite.

Let $n > 0$ and assume that all subsets of $[n-1]$ are finite.

Note that $[n] = \{0, 1, ..., n-1\} = \{0, 1, ..., n-2\} \cup \{n-1\} = [n-1] \cup \{n-1\}$. Let $x \subseteq [n]$. Then either $n-1 \notin x$ or $n-1 \in x$. In the first case, $x \subseteq [n-1]$, and thus it is finite.

Otherwise, $x \backslash \{n-1\} \subseteq [n-1]$ and is finite. Therefore there exists a $k \in \mathbb{N}$ such that there is a bijection $f : x \backslash \{n-1\} \to [k]$. Then $f' = f \cup \{(n-1, k)\}$ is a bijection $f' : x \to [k+1]$ and by the inductive property of the natural numbers, $k+1 \in \mathbb{N}$.

Hence, any $x \subseteq n$ is finite.

□