A Python B Implementation - PyB A Second Tool-Chain

Inaugural-Dissertation

zur
Erlangung des Doktorgrades der
Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

John Witulski

aus Chemnitz

August 2018

Aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent: Prof. Dr. Michael Leuschel
Heinrich-Heine-Universität Düsseldorf

1. Koreferent: Prof. Dr. Michael Schöttner
Tag der mündlichen Prüfung: 11.06.2018

Die hier vorgelegte Dissertation habe ich eigenständig und ohne unerlaubte Hilfe angefertigt. Die Dissertation wurde in der vorgelegten oder in ähnlicher Form noch bei keiner anderen Institution eingereicht. Ich habe bisher keine erfolglosen Promotionsversuche unternommen.

Düsseldorf, den 24. August 2018

John Witulski

Abstract

This thesis is about the topics interpreter construction, just-in-time compilation and software tools for the specification language B. There are two main topics: The second tool chain and the RPython-JIT. Both topics need an implementation of B. This implementation is PyB, an interpreter and model checker written in Python. The **contribution** of this thesis is PyB and the experiments done with it.

1. Abstract Second tool chain: The **first research question** is the development and evaluation of a second tool chain (PyB). This is a software tool which checks the computations of an other tool (ProB). Of interest are aspects of B which can easily implemented, which can not be implemented and the usefulness and evaluation of the whole approach. The **motivation** of this work is the necessity of an additional check of ProB, a tool used for the B method which is used to develop and model safety critical software for which reliability is vital. The second tool chains goal is to raise this reliability. The **method** of development was that of a clean room implementation which dictates that all computations are independent of the main tool: No ProB code is known. The main idea is that it is easy to build a simple tool to check a complex tool. In **result** this assumption was partially correct. It is wrong when check includes finding solutions. In B this can be unavoidable. The implementation was simple except of infinite sets and constrains solving. The implementation supports the whole B language and was successfully used on machines from industry. In **conclusion** the approach is useful if the tool must not find solutions. Otherwise it will also become a complex tool.

2. Abstract RPython-JIT: The **second research question** was the application of the RPython technology on a B implementation. The means the adaption of Python source code to the RPython requirements (static subset of Python) and the addition of a JIT. This was only done on dynamic languages before. The **motivation** was to examine if the performance results can be transfered to a specification language like B. A second branch was used, because the goal of a simple and a fast tool are in conflict. The performance was evaluated by the **method** of micro benchmarks and benchmarks of industrial machines. The **result** was a speed up of one magnitude (compared to ProB) on model checking of simple machines. Machines which need constraint solving suffer from a performance loss by several orders of magnitude. In **conclusion** the application of the RPython-JIT technology was a success anyway. If PyB will be improved by complex features like constraint solving better results can be expected.

Zusammenfassung

Diese Dissertation befasst sich mit den Themen Interpreterbau, Just-In-Time Kompilation und Software-Werkzeuge für die Spezifikationssprache B. Behandelt wurden zwei unterschiedliche Themen: Das Thema der zweiten Kette und das Thema des RPython-JIT. Beide Themen erfordern eine Implementierung von B. Bei dieser Implementierung handelt es sich um PyB, einen Interpreter und Modelprüfer geschrieben in Python. Der **Beitrag** dieser Arbeit ist PyB und die damit durchgeführten Experimente.

1. Zusammenfassung zweite Kette: Die **erste Problemstellung** ist die Entwicklung und Untersuchung einer zweiten Kette (PyB). Dies ist ein Software-Werkzeug, welches die Berechnungen eines anderen Werkzeuges (ProB) überprüft. Von Interesse war, wie bestimmte Aspekte der Sprache B besonders einfach implementiert werden können, für welche Aspekte dies nicht möglich ist und die Bestimmung von Nützlichkeit und Grenzen dieses Ansatzes. **Motivation** dieser Fragestellungen ist die Notwendigkeit einer zusätzlichen Überprüfung von ProB, einem Software-Werkzeug für die B-Methode, welche zur Entwicklung und Modellierung im Bereich sicherheitskritische Software eingesetzt wird, wo Zuverlässigkeit unabdingbar ist. Die zweite Kette ist ein Ansatz, der diese Zuverlässigkeit erhöhen soll. **Methode** der Entwicklung war die einer Cleanroom-Implementierung, welche fordert, dass PyB seine Berechnungen völlig unabhängig vom ersten Werkzeug (ProB) durchführt: Kein ProBCode ist bekannt. Die Grundannahme ist, dass es einfach ist, ein simpleres Werkzeug zu entwickeln um ein komplexes zu testen. Im **Ergebnis** hat sich diese Annahme nur als teilweise richtig herausgestellt. Sie ist falsch, wenn PyB zur Überprüfung einer Lösung selbst Werte finden muss. Dies ist in B in einigen Fällen unausweichlich. Simpel war die B-Implementierung mit Ausnahme von unendlichen Mengen und constraint solving. Die Implementierung unterstützt den vollen B Sprachumfang und wurde erfolgreich mit industriellen Maschinen getestet. Im **Fazit** ist dieser Ansatz für B nur nützlich, wenn die zweite Kette nicht selbst Lösungen finden muss, da diese sonst selbst zu einem komplexen fehleranfälligen Werkzeug wird.

2. Zusammenfassung RPython-JIT: Die **zweite Problemstellung** ist die Anwendung der RPython Technologie auf eine B-Implementierung. Hierbei ist die Anpassung des PyB-Quellcodes an die RPython Anforderungen (eine statische Python Untermenge) sowie das Hinzufügen eines JITs gemeint. Dieses wurde bisher nur auf Implementierungen von dynamischen Programmiersprachen angewandt. **Motivation** ist es, die Übertragbarkeit von bereits bestehenden Performanceergebnissen auf eine Spezifikationssprache wie B zu überprüfen. Da das Ziel, ein simples Werkzeug mit dem Ziel ein performantes Werkzeug zu schreiben, im Konflikt steht, wurde hierbei an zwei unterschiedlichen braches entwickelt. Überprüft wurde die Performance mit der **Methode** der Micro-Benchmarks und Benchmarks bestehend aus industriellen Beispielen. Im **Ergebnis** hat sich für das Modelprüfen von simplen Modellen ein speed up von einer Größenordnung (im Vergleich zu ProB) ergeben, während bei Beispielen welche constraint solving erfordern, die Performance um Größenordnungen schlechter sein kann. Als **Fazit** wird die Anwendung von der RPython-JIT Technologie auf B dennoch als nützlich bewertet. Wenn PyB um komplexe Features wie constraint solving erweitert wird, sind auch hier bessere Ergebnisse zu erwarten.

Contents

1 Introduction

1.1. Formal methods and tools

Reliability and safety of computer systems are an evermore important issue of modern life. Computers can be found not only inside many devices in our everyday life, but also as part of safety critical systems like trains, airplanes, space probes and many more.

In some cases computers are just an addition to improve some devices in some way. In other cases computers are critical. The signals or switches of a railway-system, displays containing critical data like the altitude and speed of a plane or the angle, position and distance of a space probe are just a few examples where reliable computers are a necessity.

As a man made machine, a computer can always fail. In some cases, like desktop PC applications, a failure is annoying, but the resulting damage is low or at least acceptable. In other cases, like the case of safety critical systems, a failure means risking a vast amount of money at the best or human lives at the worst.

Even though it is clear that a perfect machine can never be built by man, it is obvious that some applications need a failure tolerance that is as close to zero as possible. In computer science, one approach to solve this problem are formal methods.

Formal methods are a way of building computer software and hardware using mathematical notations like predicate logic or set-theory. These formalisms describe software- or

system-abstractions which are called models (or machines in the case of the B-Method). The goal of formal methods is to prove certain properties of the software and if it is possible, to find false predicates, traces or counter examples describing undesired behavior. This sort of behavior can be found and refuted or eliminated from the software. This abstraction is similar to that introduced by programming languages: *Programming languages introduce a formalism more easy to understand by a human, but typically not executed on computer*, *Formal methods introduce a formalism more easy to be analyzed, but typically not executable on a computer*. The focus of this theses is the B-Method, a *a correct by construction* approach and example of a formal method.

Finding bugs inside models, writing models, proving model properties or the analysis of models can be done using software tools: for example model-checkers, animators or automatic provers. These tools will be described in more detail later. One problem of formal methods in practice can be the lack of good development (software)-tools. A bug inside such a tool (for example the ProB tool) should never lead to bugs in the safety critical software. Such tool bugs have also to be detected, corrected and removed. This thesis is a contribution to the reliability of formal method tooling using the ProB tool as an example. The goal of this work is to find a way to efficiently detect errors in such tools (focusing on the ProB tool) using a second tool chain approach (introduced in section 1.3.1). One of two topics of thesis is how this is done, which problems have occurred, and how are they solved.

1.1.1. The B-Method

General Concept

There are a great variety of formal methods. The method used in this thesis is the B-Method [2]. It is a formal method based on set-theory and predicate logic. This thesis is concerned with the B-Method and the B language, which was invented by Jean-Raymond Abrial.

Using the B-Method, software (or hardware) is modeled at a high level of abstraction using set-theory and predicate logic which typically cannot be run but be analyzed on a computer. Analyzing means checking the model of undesired behavior, i.e failures and bugs. This can be done by software tools. A B model abstraction uses decomposition: it typically consists of some sub-models which can be seen as state machines. A state is a set of constants and variable values of all machines. B models are also called B machines.

The set of reachable states is called a state space. A state space and its transitions can be seen as a graph: the nodes are states and the edges are state transitions, which

donate how executing a statement changes the variables. Every state should satisfy safety properties stated in the properties and invariant clause of a machine. Clauses are subsections of a B machine file. Invariant- and properties- clauses are predicates. A false invariant is called invariant violation and represents undesired model behavior.

A transition between two states is performed by operations on the machine variables. Operations consist of (possibly non deterministic) substitutions which change the variable values of the machine. Examples of substitutions are assignments or if-then-else blocks. The operations can be disabled by preconditions (operation guards). Preconditions are predicates which must be true in the current state. Otherwise, an operation can not be executed. An invariant violation corresponds to undesired behavior of modeled software.

There are some aspects of the B-Method which have no further relevance in this thesis. They are only introduced with an intentionally short summary to let the reader understand the correct by construction approach and how the B method can be used for verification:

The B model can be refined stepwise to less abstract software until it is an executable implementation. This means that there are different levels of abstraction and development states. The developed software, PyB (which is introduced later) does not check refinements steps from model to model. One goal of formal methods is to find bugs or show the correctness of software in an early development state. This is done by showing that an invariant violation is not possible. One possibility is proof. A proof can be automated or aided by proving tools. An other possibility is model checking (described later). This thesis is **not** about proving but about model checking. If the machine is valid, i.e the correct system described by the requirements is modeled, and no invariant violation is present in development step, this approach leads to correct software. Code can be generated from the most concrete level in this chain of refinements. For example, this generated code can be C, Ada or Java code [62]. PyB does not do any code generation.

Another B application is data validation. In this approach data must satisfy some safety properties which can be expressed as a B model. The B-Method is not only used in an academic context but also in industry for real world software: B was used for software development of railway applications or smart card technology [29] (and many more software projects).

An overview about the B syntax can be found in Appendix B. (It only covers the syntax which is implemented by PyB)

```
1 MACHINE Lift
2 CONCRETE_VARIABLES floor
3 INVARIANT floor : 0..10 /* NAT */
4 INITIALISATION floor := 4
5 OPERATIONS
6         inc = PRE floor<10 THEN floor := floor + 1 END ;
7         dec = BEGIN floor := floor - 1 END
8 END
```

Figure 1.1.: A simple B machine example

B example

Figure 1.1 shows a simple B machine textbook example without properties- and constants clauses. The machine is faulty: The value of floor can be negative which violates the invariant in line 3.

- The first line gives a name to the machine file.
- The second line introduce a variable floor inside the variables clause.
- The third line states an invariant. The variable floor is only allowed to take an integer value from 0 to 10
- The fourth line assigns the value 4 to floor. This is the initial machine state.
- The sixth line adds the inc operation to the B machine. This operation can be executed if floor has an value below 10 and increments floor by one.
- The seventh line adds the dec operation to the B machine. This operation decrements the variable floor by one. The execution of this operation is unconditional.

No tools are needed to see the possible invariant violation in this machine. The missing precondition of the dec operation can cause a value of floor below zero which is an invariant violation. A possible fix of this B machine would be adding an additional precondition to the dec operation to prevent this behavior. This bug can be found by model checking (which is described later in more detail) because the state {floor=-1} violates the invariant and can be produced from the initial state {floor=4} by the repeated call of the dec operation. Real life examples used in industry are not that obviously faulty or correct.

1.1.2. ProB

ProB [53] [43] is a B-Method tool. Its main features and applications are model-checking, data-validation, constraint-based checking, and model-animation. The ProB-core is

written in Prolog, a logic programming language which also supports constraint solving. Constraint solving is a technique which can be used to solve predicates, i.e. finding values of bound variables satisfying a predicate. The ProB tool was originally written by Michael Leuschel and has been in development for more than 13 years [42]

Interesting features in more detail:

- **model-checking**:
  A set of B machines is seen as an abstraction for a state-machine describing real software. A state is a set of all variables and constants in the software. The machine starts in an initial state which fulfills the properties and invariant of the machine. Constants are assigned once at the set up of the machine, and can violate the safety properties expressed by the properties clause. Variables can be changed by any operation. If the machine contains operations, (a set of guarded substitutions, which may change variable values) these operations may change the state. Executing an enabled operation means switching the machine state from the current state to a following state. Typically there is more than one successor state. The set of all states is called a state-space.

  Model-checking is the process of exploring the whole state-space and checking the safety properties (B-properties for constants and B-invariant for variables and constants) for every possible state. Of course, this can be done by performing a breadth-first search from the initial states, if previously visited states (cycles in the state-space) are memorized. Another possibility is starting at a undesired false state and checking if there is a path from the initial state to this faulty one (see constraint based checking). One obvious use case of model checking is finding a sequence of operations in a model, which produce a invariant violation. The cause can be a wrong invariant[1], a false operation precondition (guard), or substitution body. Also other checks than the violation of safety properties like deadlock freeness are possible. If the set of states is very large, this approach can result in a (too) long computation. This combinatoric problem is called state space explosion. Model checkers typically used advanced techniques to speed up this naive approach.

- **constraint-based checking**:
  This technique uses constraint solving to check if a B operation can produce a state which violates the invariant. This state must not be reachable from the set of initial machine states. If such a state can not be produced by any operation, this kind of checking can be faster than model checking because it avoids the state space explosion problem.

- **data-validation**:

[1] the author of the B machine used a wrong invariant predicate which is false in at least one state

Data validation means checking some properties and assertions for a given set of data. It can be seen as special case of model checking, where only one state is checked. A use case may be checking a railway-system [52] against some undesired properties. For example ProB has been used for data validation in railway topologies or on university time sheets [34].

- **model-animation**:
  Animation of a model is a step-by-step interactive model-checking that may be used for model validation or teaching. The use case of animation may not be finding an invariant violation (except in the case of model-debugging), but understanding the model. Using animation, it is possible to evaluate if the model is modeling the correct thing corresponding to the requirements. Animation helps to answer the question "Am I building the right thing". For example a model that can not perform all desired operations may not enter a faulty[2] state but is still does not perform correctly.

To check the computation of a model checker, a re-evaluation of the safety properties (B invariant) in all states must be computed by a second tool. If all state-transitions are possible or if possible state transitions are absent[3] must be computed by a second tool too. Since most B-operations are guarded by a predicate to check if a state-transition is possible a case of predicate evaluation. Also a reevaluation of a safety property (B invariant) is a case of predicate evaluation. This second tool chain approach will be discussed in more detail in subsection 1.3.1.

Figure 1.2 shows a screen shot of ProB Version 1.3 in animation mode. The view consists of four frames: The B machine code on the top, the B machine state on the bottom left, the enabled B operations inside this state on the bottom middle and the animation history (previous executed operations) on the bottom right.

1.1.3. Other B-Method tools

There are other tools for the B-Method. For example the AtelierB provers pp and ml. It may be used for automated or interactive proof of B machines. As ProB the PyB implementation is compatible (besides some few exceptions) to the AtelierB implementation as defined in its reference manual [60]. More tools are discussed in the related work section 8.1.1.

[2] a faulty state violates the invariant
[3] e.g. found by the main tool but not by the second

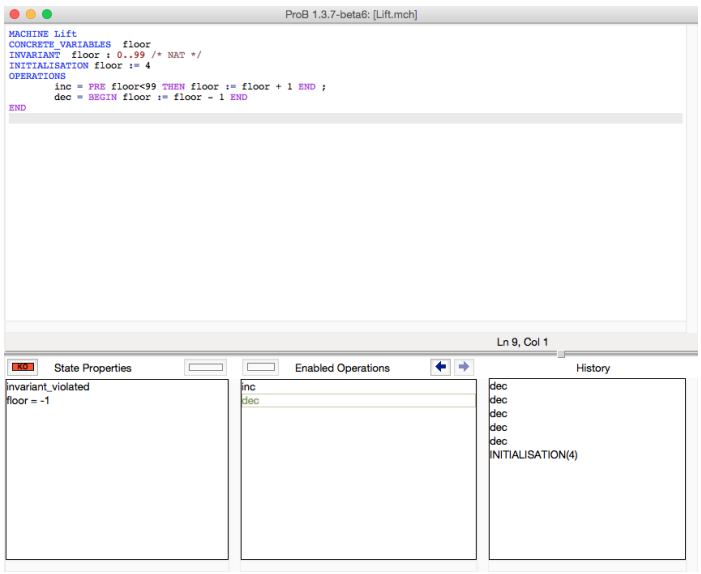


Figure 1.2.: ProB screen shot (gui)

1.1.4. Norms and external requirements in industry

There are european and international standards for safety critical software like EN 50128 [26] or IEC 61508 [13] (railroad), or ISO 26262 (cars) which define safety integrity levels from 0 to 4 (SIL4= highest safety). For example the company ClearSy presented a *Double-Core SIL4 architecture* [4] [5] in May 12th 2016, Jussieu (Paris). These documents also define a tool qualification level TCL. A tool used in a SIL project must match a tool qualification class. The qualification classes are from T1 to T3 [26].

- T1:
  Tool output does not contribute to executable code.

- T2:
  Tool tests / verifies design or executable code. It cannot introduce defects into the executable code but may fail to detect existing defects

- T3:
  Tool output contributes to executable code

A tool is certified by an external organization (e.g. SGS TUV Saar) to match that level of safety. Its intended to reach T3 for ProB, the main tool. One usual certification approach is using a second tool.

[4] http://www.atelierb.eu/en/2016/03/15/double-core-sil4-architecture-presented-during-open-source-innovation-spring-paris/
[5] http://www.clearsy.com/en/systems-and-projects/railway-systems/urbalis-evolution/

```
1  fish(sole).
2  fish(tuna).
3
4  meat(pork).
5  meat(beef).
6
7  dessert(cake).
8  dessert(fruit).
9
10 appetizer(radishes).
11 appetizer(pate).
12
13 meal(A,M,D):- appetizer(A),main(M),dessert(D).
14 main(M):-fish(M).
15 main(M):-meat(M).
```

Figure 1.3.: Simple Prolog example taken from [27]

1.2. Dynamic languages

Two languages are of interest in the context of this thesis. Prolog is the language of the main tool: ProB. Python, the language of the second tool: PyB. Both languages have an impact on the implemented tools. Some language features have to be introduced in order to understand the PyB implementation, the evaluation and the comparison between PyB/ ProB.

1.2.1. Prolog

The ProB core is implemented in Prolog [28], a logical programing language invented by Alain Colmerauer et. al. in 1973. Prolog is a declarative language which was originally developed for natural language processing. In contrast to imperative languages, Prolog programs describe no algorithms but rather problems and relationships expressed by facts and rules which are horn clauses. A Prolog [61] program can be seen as a set of predicates: a database of facts and (horn) clauses. Even though its late binding and dynamic typing are dynamic features, Prolog is no typical example of a dynamic language. A Prolog interpreter typically processes a user query via resolution to do computations and produce an answer/solution. There are different Prolog implementations for example the free SWI Prolog [67][6] or the commercial Sicstus Prolog [24][7].

Figure 1.3 shows a intentionally simple Prolog example which should help explain the

[6] http://www.swi-prolog.org
[7] https://sicstus.sics.se

previous paragraph without describing complex Prolog techniques. Line 1. to 11. show Prolog facts. One of these declares that pork is a meat. The last three lines show Prolog rules. The capital letters A, M and D are variables. The ":-" can be understood as a logical implication. For example the rule meal will only be satisfied if a solution for appetizer, main and dessert is found at which time the variables A, M and D are set to a value. One possible solution of a query meal(A,M,D) would be $\{A = radishes, M = sole, D = cake\}$. This tuple will be found by the Prolog system using SLD-resolution and unification. The Prolog system is able to find all possible solutions. More complex terms and relations can also be expressed. For example, data structures like ASTs can be used instead of atoms like pork. The example is taken from "Prolog in 10 Figures" [27].

Some important features of Prolog are:

- **Unification**.
  Unification a process similar to pattern matching on Prolog terms. This feature can be used for type checking, allowing the introduction of type variables. PyB's implementation of unification is explained in chapter 2 (2.1.2).
- **Non-determinism**.
  If there is more than one possibility to satisfy a Prolog clause, a Prolog system explores every possibility by using backtracking. This feature can be used to sequentially execute nondeterministic B-substitutions. The approach in PyB can be found in chapter 2 (2.1.6).
- **CLP(FD)**.
  SWI [65] [25] and Sicstus Prolog are equipped with constraint solving features for finite domains. This feature comes in use when dealing with the enumeration of sets defined by predicates. It has an impact on Prolog software like ProB, because it enables such tools to find solutions for quantified predicates. PyBs reimplementation of (simple and less powerful) constraint solving is introduced in chapter 3 (3.1).
- **Purity**.
  It is possible to modify a Prolog database at runtime (using assert and retract built-ins), but most Prolog predicates are pure. Data structures are immutable. For example an environment (program state) will not be directly changed by a statement predicate, but by producing another modified environment (without a side-effect).

1.2.2. Python

Python [56] is a popular [64] imperative, object-oriented, dynamic language. It was written by Guido van Rossum[8] in 1989[9]. The language is described by its developers as

[8] http://www.artima.com/intv/pythonP.html
[9] http://python-history.blogspot.de/2009/01/brief-timeline-of-python.html

```
1  class NaturalNumbers:
2      def __gt__(self, other):
3          for e in other:
4              if e<0:
5                  return False
6          return True
7
8  >>> A = frozenset([1,2,3])
9  >>> B = frozenset([-1])
10 >>> C = NaturalNumbers()
11 >>> B > A
12 False
13 >>> C > B
14 False
15 >>> C > A
16 True
```

Figure 1.4.: Python special method example and usage (simplified)

follows :
"*Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax*"[10] Python is now at version 3, but the PyB implementation is based on version 2.6 because PyPy (see next subsection) does not support all 3.0 features. Both versions were released in 2008 and are still supported.

Some important features of Python are:

- **Object oriented**
  PyB does not use Python's multiple inheritance feature. All classes have only one base class at most. PyB also does not use prototype based object creation. No class attributes/methods are added or removed during runtime (but only at import time). Only values are changed. All methods behave in the same way for all instances of any PyB ("old-style") classes. The main reason for this limitation are the RPython restrictions (discussed in chapter 5).

- **Dynamic (duck-)typing**
  Data in Python is typed. Every variable is instance of a built-in type or is an object-instance of some class. In contrast to languages like Java or C++, this type is determined at runtime. Declaration and definition happens at the same time. For example an assignment "x=1" creates an integer with the value one at runtime, without a declaration like "int x;" (in C++ or Java). This binding at runtime

[10] https://docs.python.org/2/faq/general.html#what-is-python

```
1  def generate_infinite_integer_set():
2      yield 0
3      i = 1
4      while True:
5          yield i
6          yield -i
7          i = i +1
8
9  for value in generate_infinite_integer_set():
10     result = eval_predicate(value, predicate)
11     if result==True:
12         break
```

Figure 1.5.: Python generator example (simplified)

```
1  class SymbolicSet:
2      def __init__(self):
3          self.generator = self.generator_method()
4
5      def generator_method(self):
6          for i in range(10):
7              yield i
8
9      def __iter__(self):
10         return self
11
12     def next(self):
13         return self.generator.next()
14
15 S1 = SymbolicSet()
16 for x in S1:
17     print x
18
19 S2 = SymbolicSet()
20 print S2.next()
```

Figure 1.6.: Alternative generator example. Outputs 0, .. 9 in line 17 and 0 in line 20

is called late binding. If x were the parameter of a function, its type would be known at runtime. If x implements a set of operations (for example addition) these methods will be available at runtime: x is dynamically-typed.

For example, a Python-function that adds two arguments but has no static type information about its parameters and will work properly if the types of arguments support the plus operation. The arguments may be of the type integer, float, string (plus is concatenation) or of a class instance. This dynamic typing feature is called duck typing. If all needed operations are supported by a data type, it is of no interest which type it is. This is summarized by the duck test from James Whitcomb Riley: "*If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.*". Duck typing allows a higher level of abstraction. Implementations of algorithms can be used with different data types if all needed operations are implemented on these types, for example by replacing a built-in set type like frozenset with object instances.

- **Special methods and operator overwriting**
  Of course, Python allows the definition of objects and methods, but some methods with predefined names (which all start with a double underscore[11]) are called magic methods or special methods. These methods are called (when present) if a corresponding operator is used on an object instance. The python manual describes special methods as follows:
  "*A class can implement certain operations that are invoked by special syntax [...] by defining methods with special names. This is Python's approach to operator overloading, allowing classes to define their own behavior with respect to language operators.*"[12]

  Figure 1.4 shows a Python class example and lines 8 to 14 show how it can be used in Python's interactive mode. Blocks are expressed by indentations which are mandatory in Python. In line 8, 9 and 10 three sets are created. Two use the built-in type frozenset and one is represented by an object instance of the class NaturalNumbers seen from line 1 to 6. Line 11 to 14 show a Python expression using the greater operator and its evaluation True and False. The greater operator is syntactic sugar for the superset operation and can be implemented by overwriting the special $\_\_gt\_\_$ method of a class. The greater than operation in the expression $C > B$ cause a call of this method and returns True or False. This version of NaturalNumbers is simplified, assuming that the duck typed variable *other* supports iteration (line 3) and that all elements support a comparison with zero.

- **Meta programming**

[11] https://wiki.python.org/moin/DunderAlias
[12] Quote from https://docs.python.org/2.5/ref/specialnames.html

Code which generates code. This is possible at any time (RPython at import time). This is used in the PyB parser.

- **Generators**
  A useful aspect of Python's dynamic features are generators. These are special functions which use the yield keyword instead of return to leave a function. After a generator function returns a value, they can be reentered at this point instead of the function beginning. The local variables of the function at these exit-points are restored. This feature can be used to generate large or infinite sets lazily.

  Figure 1.5 shows a code-snippet of a Python generator and its application in a simplified existence-quantor evaluation. It generates the values {0,1,-1,2,-2,...}. One value is generated at every loop iteration. The loop terminates if the predicate is True or a timeout-exception (not shown in the listing) is thrown.

  An alternative is the iteration protocol: Generators can also be added to objects by overwriting the *__iter__* and a *next* method. The *__iter__* method implicitly returns the generator object, while the next method can explicitly be called to generate an element. Figure 1.6 shows an example. This technique was used in the symbolic set implementation in chapter 3.

- **Batteries included**
  Like many modern programming languages, Python comes with a large module library. Modules containing data structures, sockets, and threading implementations are installed by default and don't have to be added afterwards. This advantage for the Python programmer becomes a burden for the Python implementor. For example a new Python implementation (like PyPy) is expected to support this large variety of standard modules. Also, Python's functionality is extended with a large third party module library (PyPi) which includes more than 50000 modules[13]. Python has no build in constraint solving features.

1.2.3. Python and Prolog as implementation language

This subsection is a small language comparison between Python and Prolog.

- Nondeterminism
  This can be more easily implemented in Prolog, because of the "automatic" backtracking while a Python implementation needs to use generators (see 2.1.6).

- Type checking
  Also typing using unification is much more easy in Prolog. But this feature is only

[13] https://pypi.python.org/pypi

introduced to PyB because of compatibility with ProB (see 2.1.2).

- Infinite sets
  Because of the clean room implementation approach, the implementation in Prolog is unknown. Anyway a symbolic representation of sets should be equally difficult in both languages (see 3.1).
- Constraint solving
  Most Prolog implementation come with built-in constraint solving capabilities [25]. This is a main disadvantage of Python (see 3.2, chapter 4 and chapter 6)

Prolog is the better choice for a formal tool implementation. Despite constraint solving Python is also a good choice. Anyway using Python was a necessity to do the RPython experiments (next subsection). A comparison of code is not presented because it would violate the clean room approach of this thesis.

1.2.4. PyPy

Originally PyPy [54] was a Python implementation (an interpreter) written in a Python subset. Today it is also the name of a tool-chain which can be used for automated interpreter generation, i.e. translating an interpreter written in Python (like PyB) to C and possibly adding a meta-tracing just-in-time compiler to this code. PyPy *is a Python VM that is fully compatible with the 'standard' C Python VM (known as 'CPython')* [17]. In more recent papers, this is referred to as the RPython tool chain instead of the PyPy tool chain.

RPython

The name PyPy is short of "Python in Python". PyPy is implemented using a subset of Python called RPython [6] (restricted Python). This RPython code can be translated to C code. " PyPy *is written as an interpreter in RPython, a statically typed subset of Python that allows translation to (efficient) C*" [17]. The restrictions and their impact on PyB are discussed in subsection 5.1

RPython Toolchain

The Python subset RPython and the RPython tool chain have to be distinguished. The tool chain is also a part of PyPy. It enables the translation of VMs and interpreters written in RPython to C. The RPython code can be viewed as a language specification of a dynamic language (or any language). One design goal of this translation is the separation of concerns: After a successful translation to C, it is possible to weave low

level aspects into the generated code. A important examples is to add a tracing just-in-time compiler (JIT). "*RPython is not a thin skin over C: it is fully garbage collected and contains several high-level data-types. More interestingly, RPython automatically generates a tracing JIT customized to the interpreter.* " [17]. Also RPython is code on a higher level of abstraction than C code which is an advantage for language implementers.

Meta-tracing JIT

There are two ways of implementing a programming language: Typically a compiler generates low level machine code while an interpreter executes a program without generating code. Interpretation supports some dynamic language features at runtime, while compiled programs are usually much faster. A hybrid between these two approaches is just in time (JIT) compilation [7]. To merge the positive aspects of interpretation and compilation, low level code is not generated ahead of time but at runtime. The decision which code is translated at a given time is the result of program analysis at runtime. One possible kind of analysis records executed operations into a trace. Recurring traces of code become hot after some time. In simple words: The interpreter search for hot loops in the source program and translates these loops to machine language. Executing machine language is faster than interpretation. Another analysis could be to record which method is called very often. In this case, only the trace approach is of interest. In these concept two levels can be identified: The source program (e.g. Python or B) on the one level and the interpreter and its JIT target code (e.g. machine language) on the other. *The basic approach of a tracing JIT is to only generate machine code for the hot code paths of commonly executed loops and to interpret the rest of the program.* [16]

In contrast to other JITs, meta tracing involves a third level. The source program of the tool chain is not the program to be executed but an interpreter executing these programs. This interpreter can be seen as language description. The goal of a meta tracing JIT is to find loops in these first level of the program (e.g. B) to speed up the program by speeding up the interpreter (e.g. PyB). The PyPy-tool chain is able to weave the aspect of a meta tracing JIT into a interpreter written in RPython. PyB is such an RPython interpreter. The cross cutting concern of adding a JIT is not the task of the interpreter implementer. He only has to add some JIT hints in the code (chapter 5). "*When a 'hot' loop is detected at run-time, RPython starts tracing the interpreter ('meta-tracing'), logging its actions. When the loop is finished, the trace is heavily optimized and converted into machine code. Subsequent executions of the loop then use the machine code version.*" [17]. A meta tracing JIT is added to the PyB implementation. It will be discussed in chapter 5 and 6. The usage of the translation from RPython to C, the JIT and its evaluation are the second contribution of this thesis.

1.3. Motivation and Research Questions

The general topic of this thesis is the implementation of B and its evaluation. **Two** questions are of main interest: The implementation of a second tool chain and the experimentation with a formal language implementation in RPython using the PyPy project.

1.3.1. A second tool chain

PyB is a B implementation in Python. It may be used in the context of data validation as a second tool chain. To do this, it validates data by comparing a computed solution against some predicates. PyB evaluates predicates[14] and expression in a given state. This state can be precomputed by an other tool. In this case ProB is the main tool, and PyB the second tool.

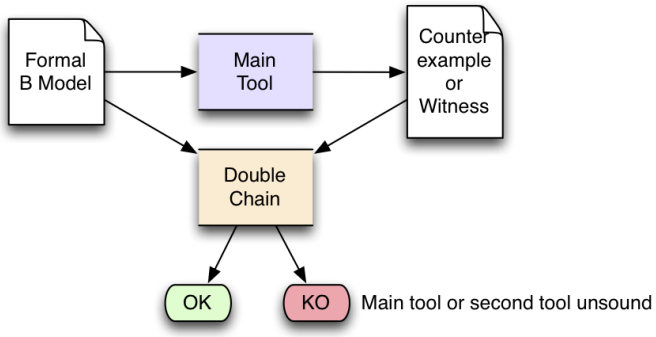


Figure 1.7.: Second tool chain concept. Picture taken from [69]

Figure 1.7 illustrates this approach. The main tool, ProB, computes solutions for a formal B model, and this output can be used by the second tool. The second tool, PyB, evaluates the formal model again using the computed solution. The second tool should not compute a new solution. Computing a solution is much more complex than using it. After the evaluation, the second tool checks its evaluation (typically True or False) against that from the main tool. If the evaluation differs, a bug has been found inside the main tool or the secondary tool. One assumption is that implementing a tool which is using a solution results in a less complex tool than implementing a tool which is used to find a solution. Notice this approach is not an alternative to a tool proof.

Research questions:

[14] for example a B invariant

- Is the usage of a second tool chain a good way to raise reliability of a tool? When is this the case and when not?
- Does a second tool chain become as complex as the main tool?
- How much effort is needed to implement a second tool chain using an language like Python and how is it done?
- Which aspects of B can be easily implemented? Which aspects are more difficult to be implemented and how is this done?

These questions lead to the following thesis goals:

- Documentation:
  Implement a second tool chain and describe how it was implemented in Python. This includes B aspects which are especially difficult.
- Correctness:
  Use an implementation style which produces easily verified Python code or code which is obviously correct.
- Completeness:
  Implement the full B syntax. Support every B construct.
- Speed:
  Ensure a good performance. Be able to evaluate predicates and states very quickly (compared to other tools).
- Time:
  Use much less time to accomplish these goals compared with the verification with the main tool ProB.
- Independence:
  Writing a clean room implementation: not using any code or algorithms from ProB.
- Evaluation and Discussion:
  Evaluate the tool performance, complexity and correctness.

Some of these goals conflict with each other. Simplicity may be in conflict with speed. Nearly all goals are in conflict with the time goal.

The implementation of the tool can be found in chapter 2 and 3. An evaluation and discussion can be found in chapter 4. Speed is discussed in chapter 6. The development process can be found in chapter 7.

1.3.2. Using PyPy on a B implementation

The PyPy tool chain was used on implementations of languages like Smalltalk [20] or Prolog [18] but never on a formal language like B. The second goal of this thesis is to find out whether a B implementation can also benefit from choosing a high level language like Python, translation to C and adding a meta-tracing JIT [15]. In the context of this topic, B is a language which is executed by an RPython interpreter: (PyB). The implementation of features like model checking is useful to evaluate the RPython translation and JIT impact. The prior expectation was a speedup by translation and JIT.

Research question:

- What is Python's impact on B implementations in terms of performance and simplicity?
- What are the differences of using an imperative high level language like Python instead of a declarative language like Prolog for implementing B?
- Can the RPython C translation and meta tracing JIT generation be successfully used on a B implementation? What are the benefits and what are the costs? Which tool refactoring is necessary?
- Which B machines can benefit from a translation to C and a meta tracing JIT and which can not?

These questions lead to the following goals:

- C translation:
  Use PyPy to speed up PyB by translating from RPython to C.
- JIT:
  Use PyPy to speed up PyB by adding a meta tracing JIT.
- Input analysis:
  Determine which B machines or B constructs can benefit from a meta tracing JIT and which can not.

The translation and refactoring process from PyB-Python to C and the JIT extension is described in chapter 5. The result, benchmarks and an analysis of how machines performance was improved can be found in chapter 6.

1.3.3. PyB

PyB is an implementation of the B-Method in Python. It is designed to be used as a second tool chain for the ProB tool. PyB is able to evaluate predicates, enumerate set's and execute substitutions. The development was done via test-driven development. PyB was developed by myself from 2011 to 2016 and is the main contribution of this

thesis. PyB's main features are:

- Loading, parsing (using a Java parser) and type checking B machines and predicates e.g. ProB solution files
- Predicate and expression evaluation, e.g. invariant and properties clauses of B machines
- Substitution execution e.g. exploring a state space of a machine by performing its operations

Apart from the Java-written parser, PyB is a Python clean room implementation. A detailed overview of the supported B syntax can be found in Appendix B. The main reason for choosing Python as an implementation language, was to use the C-translation and JIT feature of the PyPy project on this interpreter.

Notice: PyB is not a B prover.

1.4. Outline

The thesis is organized as follows:

- Chapter 2. introduces the simple parts of the PyB implementation.
- Chapter 3. discusses the more complex parts of the PyB implementation such as infinite sets and csp enumeration problems
- Chapter 4. presents the second tool chain evaluation results
- Chapter 5. describes the translation from Python to RPython, the translation itself, and how a a JIT was added.
- Chapter 6. evaluates the RPython implementation using benchmarks.
- Chapter 7. is about the development process.
- Chapter 8. compares PyB to other tools and implementations and contains the thesis conclusion.

1.5. Summary

This chapter introduced the two topics of this thesis, their motivation and their goals. It also introduces some aspects of the second tool chain approach, B, Python, Prolog, model

checking, data validation, RPython, and JIT-compilation which are the foundation to understand the remaining thesis.

2

Tool Overview

2.1. Architecture of PyB

This section presents the PyB implementation. It also discusses implementation alternatives to justify design decisions and to present wrong approaches. Purpose of this sections is to convince the reader of the simplicity of the tool which is mostly implemented using standard techniques of compiler and interpreter construction [4]. Difficult implementation aspects are presented in chapter 3. Table 2.1 summarizes the modules in PyB.

Figure 2.2 shows the initial steps taken by PyB when processing a B model: parsing, AST generation, definition handling and type checking. These steps are presented in the next subsections. The definition handling step is optional, because the definitions clause can be omitted in a B model. The steps performed after start up (at runtime) depend on operation mode and input. PyB behaves differently when it is used as repl, simple predicate checker or when it processes whole B machines. PyB's operation modes will be discussed in subsection 2.2.1. The PyB start up should not be confused with the start up for a B machine which is something entirely different (see B state subsection).

2.1.1. Parsing and Definition Handling

The task of parsing is to generate Python data structures from B input files. The data structure is an abstract syntax tree (AST).

Figure 2.1.: Module Overview

Name	Summary
animation_clui.py	console interface for animation mode and repl
animation.py	main animation computation
ast_nodes.py	classes representing AST-nodes
bexceptions.py	custom exception objects
bmachine.py	a class representing one B machine
boperation.py	a class representing one B machine operation
bstate.py	a class representing one B machine state
btypes.py	type classes
config.py	main config file
constrainsolver.py	PyB constraint solving code
definition_handler.py	main definition handling code
enumeration.py	enumeration methods for sets, functions, relations and more
environment.py	code for managing B machine state
external_functions.py	implementation of external functions
helpers.py	miscellaneous helper functions
interp.py	main interpreter code. Predicate/expression evaluation code. Substitution execution code
parsing.py	helper functions to execute Python AST-code
pretty_printer.py	pretty printer for B predicates and expressions
pyB.py	main module
quick_eval.py	special case membership evaluation
repl.py	read-eval-print-loop code
statespace.py	implementation of the state space
symbolic_sets.py	symbolic set classes
symbolic_function_sets.py	symbolic function set classes
typing.py	main type checking code

The Java Parser

PyB is an independent clean-room implementation except for its B parser which is written in Java. This Java parser was written by Fabian Fritz in 2008 and has been maintained and extended by others and is also used by ProB. This parser is generated by a parser generator named SableCC [31]. SableCC was originally developed for the master thesis of Etienne Gagnon in 1998. SableCC automatically creates a lexer, parser, and other helper classes from a set of regular expressions, grammar definitions, and productions. The generated code is Java. The Java parser outputs an (abstract) syntax tree for every input which is a valid program defined by the grammar.

The developers describe SableCC as follows: *"SableCC is a parser generator which*

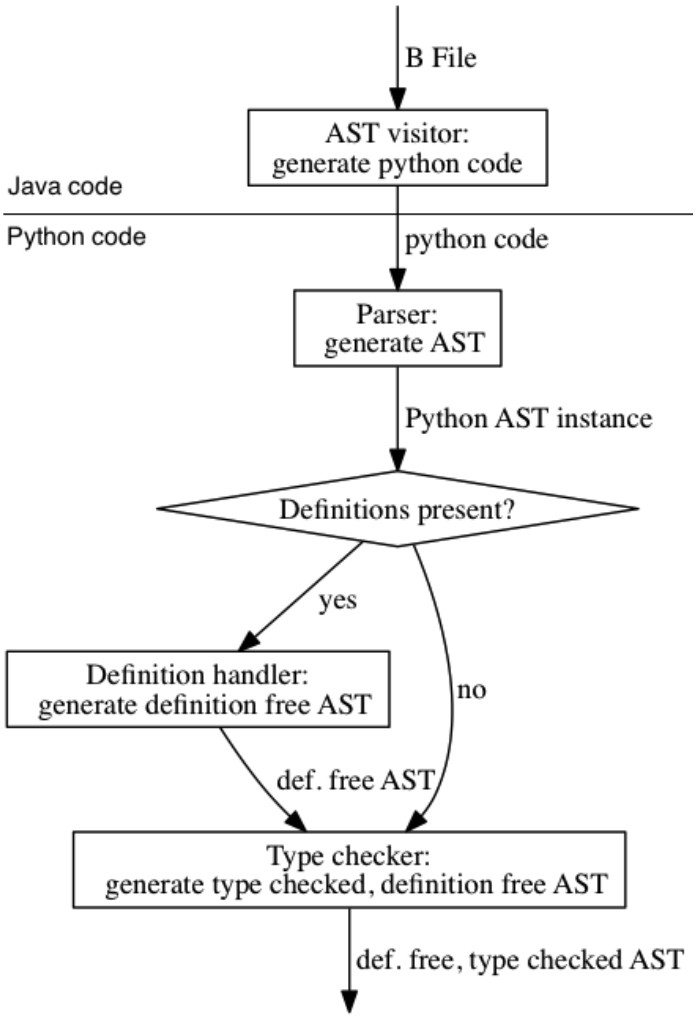


Figure 2.2.: PyB startup process

generates fully featured object-oriented frameworks for building compilers, interpreters and other text parsers."[1]

PyB uses ProB's parser to recognize B-constructs like predicates, expressions, and substitutions. These constructs are translated to an intermediate representation: an abstract syntax tree (AST) represented by Java classes. This is done by a mapping concrete productions to abstract ones (of a second abstract grammar) inside the SableCC input file. These AST Java objects are translated to Python objects via an AST-visitor, an addition to the Java code written only for PyB. The AST-visitor is a Java class which performs operation on some AST nodes without modifying any of the Java objects. It can easily be written by extending (Java inheritance) the auto generated SableCC depth-first helper classes. The visitor is the only contribution of this thesis on the Java parser level.

In "Design Patterns Elements of Reusable Object-Oriented Software" (Erich Gama et al.) a visitor is described as follows:
"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates." [32]

The AST-visitor emits a string of Python code by performing a depth first walk over the Java AST. The dynamic features of Python enable the execution of this Python code after the Java visitor execution has terminated. This is used by PyB to generate a Python AST representation from the Python string output.

The listing in Figure 2.3 shows the Python code created by the Java visitor for the simple predicate $1+1 = x$. Figure 2.4 shows the corresponding AST. This data structure is hierarchical: the expression evaluation of 1+1 and the lookup of the variable x has to be done before the check of equality. The example is very simple on purpose. More complex ASTs are generated exactly the same way.

The AST identifier objects are numbered from 0 to 5. The last one is the root of the tree. All these Python objects are derived from one node class. Code like this can be evaluated by Python and is the main input for most PyB evaluation methods.

The visitor consists of 1830 lines of code.

[1] http://sablecc.org/

Figure 2.3.: String of executable Python code of the predicate 1+1=x generated by the Java AST-visitor

```
id0=AIntegerExpression(1)
id1=AIntegerExpression(1)
id2=AAddExpression()
id2.children.append(id0)
id2.children.append(id1)
id3=AIdentifierExpression("x")
id4=AEqualPredicate()
id4.children.append(id2)
id4.children.append(id3)
id5=APredicateParseUnit()
id5.children.append(id4)
root = id5
```

Figure 2.4.: Simple AST of 1+1=x

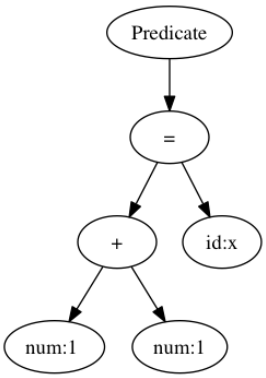


Figure 2.5.: A B machine using definitions

```
MACHINE Test
VARIABLES z, b, x
INVARIANT x:NAT & z:NAT & b:BOOL
INITIALISATION x:=2 ; Assign(x+1, z) ; Assign(TRUE, b)
DEFINITIONS Assign(Expr, VarName) == VarName := Expr;
END
```

Definitions - B macros

Figure 2.5 shows a B machine which can be processed by PyB. This B machine contains a definition clause. Definitions can be compared to C macros: They are a text replacement mechanism. All definitions are listed in the definition clause. A definition consists of a declaration of a name and parameter names on the left side and an expression on the right side of the two equal symbols. A definition can be used in predicates, expressions and substitutions. The definition's name in Figure 2.5 is Assign. The parameter names are Expr and VarName. It is used in the initialisation clause. After the replacement, the initialisation clause consists of three substitutions: **x:=2; z:=x+1; b:=TRUE**. While x and z are of type integer, b is of type boolean. Depending on the parameters in the application of a definition, the type of the generated B code may be different. This is why definitions should be evaluated before type checking. Like ProB PyB applies definitions on the AST before type checking is performed.

The Java/Python parser generates two kinds of definition nodes. The first kind is only present in subtrees of the definition clause node and represents the pattern of a B definition. The other kind represents the application of a definition pattern and may be found at any AST location where expressions, predicates or substitutions are possible. The implementation of B definitions by PyB is done in 3 steps:

1. parsing all definitions inside the definition clause. Storing definition pattern represented by ASTs.
2. whenever a definition is used construct new sub ASTs corresponding to the given parameter values using a clone of the AST definition pattern.
3. replace the definition call with the new tree.

The replacement phase is also used to replace AST nodes calling external functions. This is also a ProB feature. External functions are not written in B and need additional verification. In contrast to definition replacement, the node is replaced with a custom external function node which wrap a Python function written by the user. It is the only kind of AST node which cannot be generated by the parser.

Figure 2.6 shows a B example using an external function length which computes the length

Figure 2.6.: A B machine using an external function (simplified)

```
MACHINE LibraryStrings
CONSTANTS length
PROPERTIES
  /* compute the length of a string */
  length: STRING --> INTEGER &
  length = %x.(x:STRING|STRING_LENGTH(x))
DEFINITIONS
  STRING_LENGTH(x) == length(x);
  EXTERNAL_FUNCTION_STRING_LENGTH == STRING --> INTEGER;
END
```

of a string. The external function is introduced with a prefix EXTERNAL_FUNCTION inside the definition clause.

PyB's AST is a immutable data-structure (at runtime) and no node will be altered. The only exception is the evaluation of B definitions after Java parsing and before Python type checking. There is no change of the AST node instances after this step. This fact is important to the RPython translation (see chapter 5).

Every node of the AST is only visited once which means the algorithm is linear. The algorithm terminates because the tree consists of a finite number of nodes.

External functions

Like ProB, PyB supports external functions. These functions are not checked for correctness by PyB and are a responsibility of the tool user. Examples are printing functions or missing string operations not supported by classical B. Typically they are introduced with an external function prefix and some type information into the B machine. The implementation is done with external code. In ProB this is done using Prolog, in PyB is it done in Python (the external_functions module).

Implementation Alternatives

The obvious alternative to using the Java parser is implementing a new parser in Python. This violates the clean room implementation goal: A parsing bug inside the main tool would also be inside the second tool. However there are good reasons for this exception.

Parsers are typically not written by hand but generated from grammars and regular expressions using tools like SableCC. The temptation of using parts of these grammar is

high and writing them new from scratch would likely not lead to a more reliable version of the parser. Furthermore doing this would be very time consuming. Last of all, a parser can still be written in Python at any time. When considering limited time resources there are simply other implementation issues which are more important.

Printing executable Python code can cause performance problems on large B machines. If the Java parser is not replaced by a Python parser in the future, this can be still be improved by using a more direct communication between the Java and Python part, e.g. using sockets.

There is no good alternative in the case of definition handling. Modifying the AST after its creation is the obvious implementation. This is equivalent to generating a new AST in a second pass.

2.1.2. Type Checking

The type checking phase is performed after parsing and definition handling. It uses the ASTs (one AST per B machine) as input to produce a mapping of identifier AST-nodes to PyB's B type tree instances. A type tree is only a single node in the case of a scalar type like boolean or integer and a tree of nodes in the case of a function type or a set (see example). The motivation of type checking is error detection in B machines and to provide type informations for enumeration methods discussed in chapter 3.

To sum up, the goals for type checking:

- map every identifier node to a type tree.
- compute all type trees independent of the predicate subterm order (compatibility with ProB).
- keep the AST data structure immutable in order to separate concerns. Store type informations in a different data structure.
- keep the implementation simple

$$\boxed{f = A * B \ \& \ c = f(42) \ \& \ A = NAT \ \& \ B = BOOL}.$$

Figure 2.7.: Type checking: Example predicate.

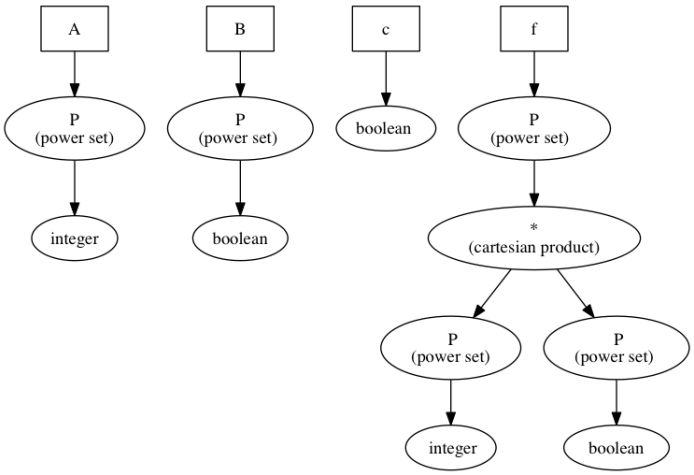


Figure 2.8.: Type trees created after successful type checking of a B predicate. Rects represent identifier nodes while the ellipses represent PyB type nodes.

B supports integer, boolean, string and sets as basic types. Data of more complex types are constructed using structs, pairs (cartesian product) or the powerset operation. From the "B LANGUAGE REFERENCE MANUAL" [60] 3.2 B Types:
A B type is either a basic type, or a type built on a type constructor. The basic types are:

- *the set of relative integers Z,*
- *the set of Boolean BOOL, defined as* $BOOL = \{FALSE, TRUE\}$*, with* $TRUE \neq FALSE$*,*
- *the set of character strings STRING,*
- *the deferred sets and enumerated sets[..],*
- *There are three type constructors: the power-set 'P', the Cartesian product '*' and the collection of records labeled 'struct'.[...]*

These types are implemented using corresponding Python classes. There are two kinds of PyB type classes inside a type tree: leaves and inner nodes/root nodes. IntegerType, BooleanType, StringType and SetType instances are leaves in a type tree which correspond to the basic types. Composed type classes are: PowerSetType, CartType and StructType which are type tree roots or inner nodes but never tree leafs. Figure 2.8 shows the constructed type trees of a example B predicate (2.7):

The predicate is processed from left to right. Considering this example, the first subpredicate encountered by the type checking algorithm will be $f = A * B$. Without processing

the whole predicate, an exact typing is not possible but the order of the subpredicates should not affect the type checking. Otherwise PyB would not be compatible with ProB. If the ordering of subpredicates is irrelevant, type checking of predicates like $x = y \wedge x = 42$ must be possible. This problem is solved by the introduction of type variables [51].

Beside the subpredicate order, another problem to be solved is the consideration of scopes. B allows nested quantified predicates like $\forall m.(m \in \mathbb{N} \implies \exists n.(n \in \mathbb{N} \rightarrow m < n))$. The variable m is bound to the universal quantifier and may appear in the outer and inner parenthesis pair while the variable n is bound to the existential quantifier and may only appear in the inner pair of parenthesis. Any other use of this variable would be a type error and will be detected by the type checker.

Other examples of scopes are bound variables within substitutions, operations or variables of other (included, seen ..etc.) B machines. The B specification does not forbid reusing identifier names in different scopes, but some B tools prohibit this not or at least give warnings(ProB)) to the user.

Implementation of Type Checking

The type checking implementation consists of two functions operating on two kinds of trees which should not be confused: The AST (handled by a type checking function) and the type trees (handled by a unification function). Some AST nodes are used to construct a mapping from identifiers to type trees. For example equality (x=y) or membership (x:S) will cause the type checking function to call the unification function. Initially, every identifier node is mapped to a type variable. The AST is visited by a depth first algorithm while propagating type informations from the leaves to the root. Leaves may return concrete B types or other type variables. Different unification steps are performed depending on the visited node.

PyB uses a simple unification algorithm [51] to implement type checking. PyB's unification is a pattern matching algorithm performed on two type trees. Both trees are traversed in the same order while comparing each node. Step by step a third type tree (the output) is constructed, which contains fewer or an equal number of type variable nodes than each unified tree. That means the new tree is more or equally concrete to both trees. This procedure can be summarized by these three steps:

- (TYPE 1) If a type variable is unified with a concrete type, the variable is set to this type.
- (TYPE 2) If a type variable X is unified with an other type variable Y, then a pointer is set from X to Y to save the information that both variables point to the same type. In the case of a chain of type variables (starting from X), the reference

to a concrete type is set by the last type variable. The type of the AST node of type variable X is the type of the last node in the chain (see example in next subsection). This step can also be used in the case of a membership relation by passing a subtype argument to the unification step.

- (TYPE 3) If a concrete type is unified with another concrete type (or type constructor), the unification continues if the types are equal or if a type error has been found (unequal types) .

Inner nodes of type trees are only concrete types (corresponding to cartesian product or powerset) and not type variables (except the A-B and A*B special case of the next paragraph). Before any unification is performed, a closure function walks the chain of type variables finding the last variable. Every unification is performed only at the last type variable (which is implicitly stated in step TYPE 2). Because of this cycles (type variable pointing at each other) are not possible (see example in figure 2.10).

After the ASTs have been fully visited, a resolve phase walks over every type tree again. If it is not possible to find a path from every type tree root to a concrete type (which means that a type tree still contains a unresolved type variables), a type error has been found.

There are two special cases of type variable nodes created by expressions A*B and A-B. The operations can be multiplication or cartesian product and integer subtraction or set subtraction. In both cases A and B can be of type integer or set. This can also be resolved after the whole AST has been visited.

PyB uses a standard approach to solve the scope problem: a stack. Every time a new scope is entered or left, a new hash map (identifier nodes to type trees) is pushed or popped. A id lookup method always traverses down this stack from the top and throws an exception if no entry is found. Variable names are known before their use. Quantified variables are introduced explicitly and global variables can be found in the SETS or VARIABLES clause of the B machine.

The operator precedence is encoded inside the AST by the parser. If the parser is correct, PyB automatically uses the right precedence.

The type checking algorithm visits every AST node only once. Because of the finite number of nodes, the algorithm will terminate. Of course the unification step adds extra computation time to the type checking computations. It terminates because cycles inside type trees will not be produced and the number of variables is finite. Termination of type checking may also be caused by throwing a type error.

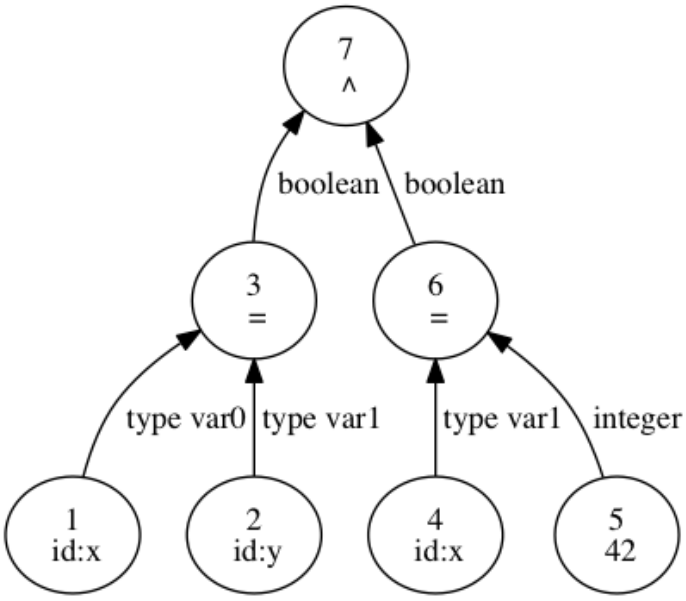


Figure 2.9.: AST of $x = y \wedge x = 42$ Order of type checking traversal from 1 to 7, returned types or type variables

Type Checking Example

Figure 2.9 shows an AST and figure 2.10 shows a stepwise computation of a type tree. The numbers in 2.9 indicate the order in which nodes are visited. The edges show the type of the lower nodes which are returned to the upper node. Unification computations are performed in the equal node 3 (TYPE 2), equal node 6 (TYPE 1) and conjunct node 7 (TYPE 3). Figure 2.10 shows the stepwise computation of the type trees on initialization of the map (step A), after unification in equal node 3 (step B) and after unification in equal node 6 (step C). The resolve phase is omitted in figure 2.10. Both variables x and y are of type integer.

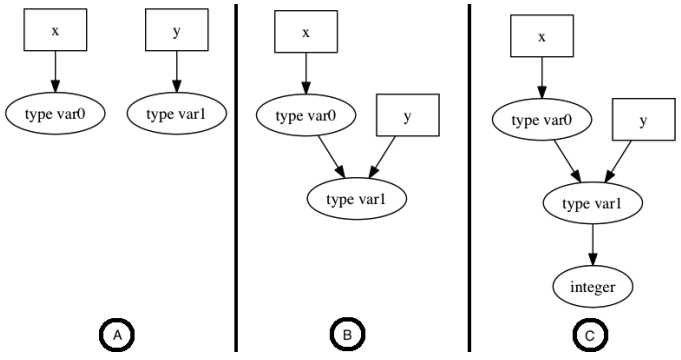


Figure 2.10.: Stepwise computation of type trees of example from Figure 2.9 $\sigma := \{typevar0 \rightarrow integer, typevar1 \rightarrow integer\}$

The algorithm does not produce any type variable cycles. For example in $x = y \ \& \ y = x \ \& \ x = 42$ the second subpredicate $y = x$ has no effect on the type tree because both x and y already point to the same type variable. When the type checking functions encounters $x = y$ the type variables (argument to the unification function) are different. At the equal node of $y = x$ the type variables are already equal.

Implementation Alternatives

Variable renaming (e.g. in quantified predicates) was unnecessary because of the stack usage inside the type environment and the mapping from identifier nodes to type trees instead of the mapping of strings (id names). The implemented solution uses a 1:1 relation between nodes and type trees. A type ambiguity (e.g. of different bound variables in a nested quantified predicate) is impossible. Also, a stack is needed in the renaming solution too.

Instead of creating a map from id nodes to type trees an annotation of the tree could be an alternative. This violates the immutability of AST and is not really a different approach. There is no advantage in using the standard approach of an attributed ASTs (nodes contain type information). The only difference is where informations are stored.

The unification could have been avoided at the price of lower compatibility to ProB. This would make PyB a more simple tool and would be consistent to the Atelier B specification. Atelier B can not type predicates of arbitrary order without explicit type informations. If compatibility is mandatory, the alternative to unification would be a reordering of terms by PyB (at type checking phase). For example, $x = y \wedge x = 42$ can be reordered to $x = 42 \wedge x = y$. A reordered term with explicit type information would be $x \in Integer \wedge y \in Integer \wedge x = 42 \wedge x = y$ This reordering approach is more error prone than the unification solution and violates the intended immutability of the ASTs. The explicit typed version would also needs a rewrite of the predicate by the B machine author.

2.1.3. Implementation of B's data types

B's basic and composed data types are easily implemented by Pythons built-in data types. There is no distinction between primitive and others types in Python. Every value is an instance of a subclass of the class object. Every operation can be expressed using a method call. For example x + y is equivalent to x.__add__(y).

Furthermore, mapping infinite data objects (even a set of integers) to finite ones is a

problem even if Python supports data types which are only limited by machine memory. This aspect is addressed in chapter 3.

Explicit implementation of B's integer types

B's integer type is represented by Pythons integer type. It supports numbers from -2^{31} *to* $2^{31}-1$ and even larger numbers at the expense of more memory usage:
Plain integers:These represent numbers in the range -2147483648 through 2147483647. [...] When the result of an operation would fall outside this range, the result is normally returned as a long integer. [...] Long integers. These represent numbers in an unlimited range, subject to available (virtual) memory only.[2]

Explicit implementation of B's set types

The most important type in B is the set. While there are B built-in sets for boolean, natural and integer numbers, there are also ways to declare user defined sets using the sets clause or simply set operations like intersection, union etc.

Python offers two built-in set types: *set* and *frozenset*. The explicit B set representation is done using Pythons built-in type *frozenset*. The name explicit is used in this thesis to distinguish the set representation introduced by this subsection from that of chapter 3. Instances of the built-in *set* type can be modified after creation (adding and removing elements at runtime), while *frozensets* are immutable. The *set* type is an unhashable type, i.e. it is not possible to use the *set* type to build a set of sets. The *frozenset* type is hashable because of its immutability.

Both the *set* and the *frozenset* type implement most of the needed set operations. Figure 2.11 shows the built-in methods and a short description. The table is a modified copy from the Python documentation[3]. S and T are sets and x is an element.

Sets of sets or sets of tuples are created using a combination of frozensets and Python's built-in tuple type. More complex sets can be created implementing the cartesian product or the powerset operation. For example the poweset of the set $\{1,2\}$ is represented by frozenset([frozenset([]),frozenset([1]), frozenset([2]), frozenset([1,2])]). Relations are represented as a set of tuples. Functions and sequences are a special case of relations. For example, a finite B-function f which maps the numbers 1 to 3 to its square numbers "$f = \%x.(x > 0 \ \& \ x < 4|x * x)$" is represented on the Python level as

[2]https://docs.python.org/2/reference/datamodel.html
[3]https://docs.python.org/2.4/lib/types-set.html

Operation	Description
len(S)	cardinality of set S
x in S	test x for membership in S
x not in S	test x for non-membership in S
S.issubset(T)	test whether every element in S is in T
S.issuperset(T)	test whether every element in T is in S
S.union(T)	new set instance with elements from both T and S
S.intersection(T)	new set instance with elements common to T and S
S.difference(T)	new set instance with elements in S but not in T
S.symmetric_difference(T)	new set instance with elements in either S or T but not both
S.copy()	new set with shallow copy of s

Figure 2.11.: Taken from Python documentation 2.3.7 Set Types: Possible operations of set and frozenset instances

frozenset([(1,1),(2,4),(3,9)]). Instances like this can be created at runtime during the interpretation of B.

Explicit implementation of B's record types

Structs are also implemented using Python's frozenzet type. Every record is a set of 2-tuples, containing a string key and a B value. Python dictionaries (hash maps) are not chosen for the implementation because they are not hashable. Every set content must be hashable to enable sets of sets.

Implementation Alternatives

Since any B implementation needs to represent set of sets, Pythons mutable set type is not suitable. The Python documentation about set types:
2.3.7 Set Types [...] The set type is mutable [...] it has no hash value and cannot be used as either a dictionary key or as an element of another set.
An object is hashable if it has a hash value which never changes during its lifetime [...] Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally. All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. [4]
As a frozenset is immutable and hashable, it can be used again as an element of another set, or as a dictionary key. [5]

[4]https://docs.python.org/2/glossary.html#term-hashable
[5]https://docs.python.org/3/reference/datamodel.html

2.1.4. B state representation

A B state is a set of variable (or constant) name-value pairs. Every variable change can create a new B state. The set of all states is the state space. Figure 2.12 shows all reachable states of a fixed B Lift example from Figure 1.1 (last chapter) after init and one inc operation execution.

B states are implemented using a stack of dictionaries (Python hashmaps). Each dictionary maps an identifier key (represented as string) to an explicit or symbolic value. Symbolic values will be introduced in chapter 3. Explicit values are those of the last subsection. The dictionary is the obvious implementation for a state. The usage of a stack is the result of the B scoping (variable visibility) rules. If a new scope is entered, a new dictionary is pushed on the stack containing only the keys and a None value. Values are set using an PROPERTIES/INITIALISATION-clause in the (special-)case of the initial state with possible parameter values using OPERATION preconditions or assignments. If a scope is left, the dictionary is popped from the stack. If a value should be written or read, a lookup algorithm searches from the top of the stack down to the stack bottom, until it finds an entry. This allows identical variable names on different scope levels. If no entry is found an exception is raised. This means an unknown variable is accessed which must be a bug inside PyB. This solution is common in programming language implementations.

Each B machine instance keeps its own state. Which state should be used is managed by the environment. The environment changes and resets the current machine scope. An alternative implementation would be a static analysis of duplicate variable names. The disadvantage would be the constraint prohibiting identical identifier names, which would be a problem for bound variables. This issue was already discussed in the last subsection.

This implementation is wrapped by a environment class which is used for example by the interpreter evaluation function.

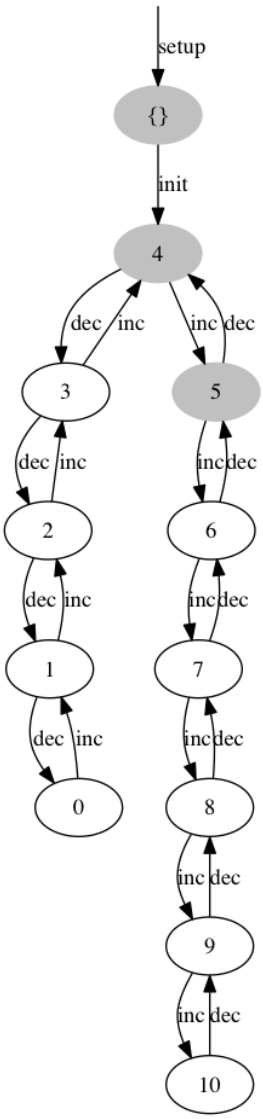


Figure 2.12.: Simple state space, gray states are visited

```
1  MACHINE Scope /*modified Schnieder book page 115*/
2  SETS S
3  PROPERTIES card(S)=3
4  VARIABLES f
5  INVARIANT f:S --> 0..6
6  INITIALISATION f:=S*{0}
7  OPERATIONS
8  op1(rr, nn) = PRE rr:S & nn:1..6 & f(rr)=0
9                 THEN f(rr):= nn
10                END;
11 nn <-- op2 = nn:= SIGMA(zz).(zz:S | f(zz))
12 END
```

Figure 2.13.: B scoping example

The scoping rules of one single B machine is as follows (Atelier B reference [60] pages are shown in parentheses):

- (abstract) Constants and Sets are visible inside the INVARIANT-, PROPERTIES-, ASSERTIONS-, INITIALISATION-, OPERATIONS-, INCLUDES- and EXTENDS-clause. (7.15)
- Variables are visible inside the INVARIANT-, ASSERTIONS and OPERATIONS-clause. (7.19)
- Bound variables of set comprehensions or existential or universal quantified are visible in their corresponding predicates. (5.7, 4.2)
- Bound variables in a lambda expression, generalized summation or production and quantified unions or intersections are visible in their corresponding predicate and expression. (5.16, 5.8, 5.4)
- Local variables introduced in a ANY, VAR or LET substitution are only visible in the substitution itself. (6.10, 6.14, 6.11)
- Operation parameters are visible in the operation body. (7.23)
- Operation return values are visible for write mode in the operation body.

Figure 2.13 shows a simple (simplified) textbook example, to demonstrate scoping. The CONSTANTS- and ASSERTIONS-clause is omitted in the example. The state consists of a deferred set S of three elements and a variable f which is a function from S to integer. The machine offers two operations: changing the functions mapping or performing a summation on the functions image.

Figure 2.14 shows a possible state change of this machine at the Python level. It can

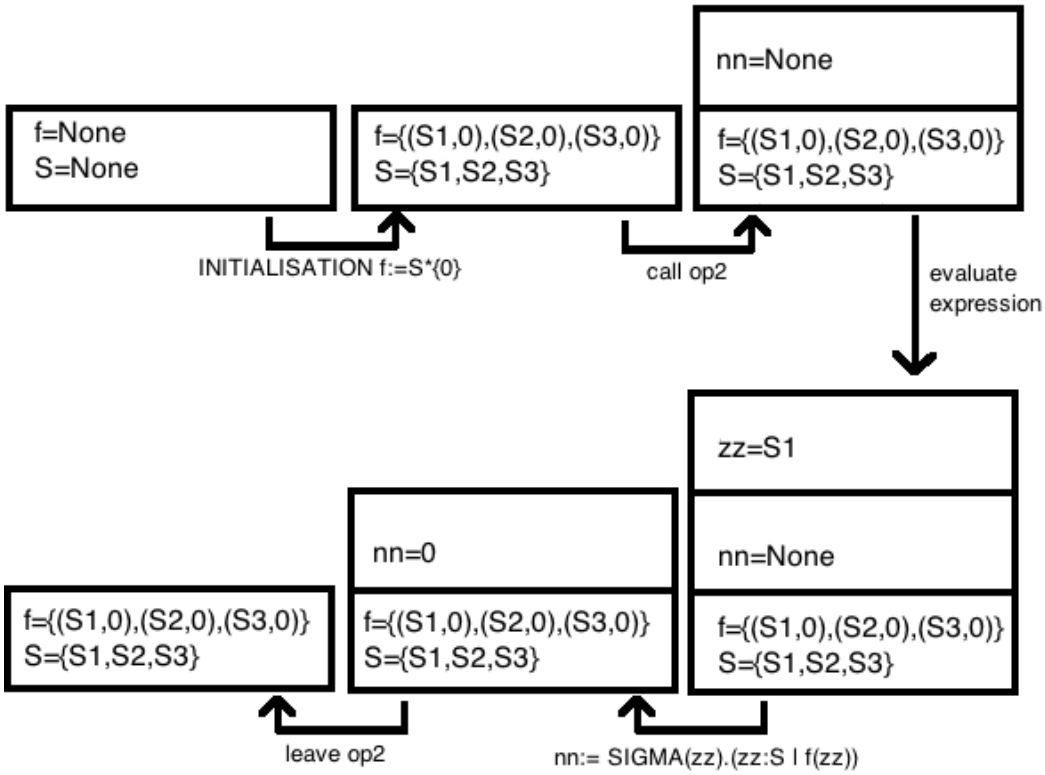


Figure 2.14.: PyB state representation and state change

occur during machine animation or model-checking. Every square represents a state or an intermediary state. The arrows represent a computation step. The call of op2 is split into more steps. Only the aspects relevant for scoping are described:
First a mapping from identifier to None (no value) is added to the lowest level of the stack. After the execution of the INITIALISATION-clause, the variables are bound to a possible value. Deferred sets are filled with a number of dummy elements: S1, S2, S3 (or more), defined in a PyB config file. Now the machine startup is done. After operation op2 has been called, the parameters (if present) and return values are added to a new frame. The evaluation of a predicate with a bound variable (zz) adds a new frame for every nested predicate. The expression f(zz) is evaluated for every (this is omitted from the figure) possible value of zz e.g. S1. After the evaluation of the predicate, the frame of the bound variables is dropped. Operation parameters and return values are dropped when the operation execution finishes. The return value will be used by the caller.

2.1.5. Interpretation

Explicit evaluation of B expressions

The explicit interpretation of B formulas is done using a depth first algorithm on the AST. This algorithm is recursive: every node is evaluated using the evaluated expressions of its subtrees. The leaves of the AST may be identifier-, number-, set- nodes etc. The value of the whole expression represented by an AST is generated this way. It is the usual approach of writing interpreters. This subsection does not describe the evaluation of symbolic values (introduced in chapter 3) but rather B data represented using Python types like string, integers, frozensets or tuples introduces in the previous subsection

PyB's evaluation method uses two parameters: an AST node and an environment. The AST node may also be the root of an AST. The environment delivers a link to the current B state. The evaluation method is pure: it does not change the B state which means it has no side effects. This is an advantage for the independent evaluation of subexpressions (chapter 3) and an important property for the correctness of the PyB code. Substitutions are evaluated with a different method.

Figure 2.15 shows the evaluation of the predicate 5+3<card(S) represented by an AST. The nodes are visited using a depth first search. The edges of the tree are labeled with the expression returned by the node evaluation. The evaluation of the identifier S is done by a lookup inside the environment. The RPython version of the "interpret" method uses wrapped types and is discussed in chapter 6.

Figure 2.16 shows an excerpt of the "interpret" method. This method is basically a big switch over all node cases. Inner nodes like add or card use the evaluation of their child nodes to compute their value. Leaf nodes like integer or identifier just return a value. Because ASTs are finite, the algorithm obviously terminates if every implemented case

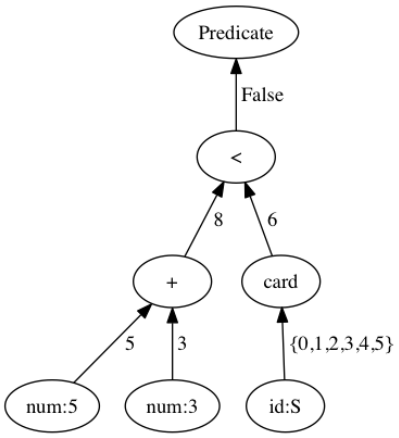


Figure 2.15.: AST evaluation example of the predicate 5+3<card(S)

Figure 2.16.: PyB interpreter excerpt. omitted code: [...]

```
1     elif isinstance(node, AAddExpression):
2         expr1 = interpret(node.children[0], env)
3         expr2 = interpret(node.children[1], env)
4         return expr1 + expr2
5 [...]
6     elif isinstance(node, ACardExpression):
7         aSet = interpret(node.children[0], env)
8         return len(aSet)
9 [...]
10    elif isinstance(node, AGreaterPredicate):
11        expr1 = interpret(node.children[0], env)
12        expr2 = interpret(node.children[1], env)
13        return expr1 > expr2
14 [...]
15    elif isinstance(node, AIntegerExpression):
16        return node.intValue
17 [...]
18    elif isinstance(node, AIdentifierExpression):
19        return env.get_value(node.idName)
```

Figure 2.17.: An (artificial) B machine containing a nondeterministic operation

```
1 MACHINE NonDet
2 VARIABLES x, y
3 INVARIANT x:NAT & y:NAT1
4 INITIALISATION x:=3 ; y:=3
5 OPERATIONS
6         op = BEGIN x::{0,1,2} ; y::{x+1}-{1} END
7 END
```

terminates. Of course other node evaluation cases are more complicated than the nodes in this example.

Implementation Alternatives

It would be more object oriented to add the evaluation methods like type checking or interpretation to the corresponding node class. This approach was not implemented because of readability of PyB code. It is used in the RPython branch of PyB for performance reasons (see chapter 5)

2.1.6. Substitution execution

B's substitution statements can be compared to statements used by many programming languages. The difference to other languages is the high level of abstraction of some substitutions. While usual substitutions like direct assignments or if-then-else statements exist, there are other substitutions which allow nondeterminism. One challenge when implementing B is to implement these nondeterministic substitutions and their possible parallel execution[6]. Also, the B implementor must keep in mind that substitutions will be used to implement[7] animation and explicit state model checking into PyB, which is a technique which computes every reachable state (section 2.1.9).

Figure 2.17 shows an artificial B machine example. This machine is purposely simple to demonstrate implementation issues introduced by B's nondeterminism. The machine's initial state sets the variable x and y to a valid state (3,3) which does not violate the invariant (the maximal Nat set value is greater than 3). The value both of x and y must be greater than or equal to zero. The operation op consists of a sequence of two nondeterministic substitutions: x will be assigned to the value 0, 1 or 2, and y will be assigned after x has been set to a value. Y will be set to the value x+1 unless x's value

[6]Parallel at B level (modeling), not at interpreter level

[7]The PyB code of substitutions is used by the model checking loop code to compute the transition from one state to another

is 0. In this case the whole operation is impossible because one of its substitutions is impossible: The set difference $\{0 + 1\} - \{1\}$ will produce an empty set.

This machine shows how values of variables can be constrained by the values of other variables which are chosen nondeterministically. Possible value pairs for x and y are: (1,2), (2,3) and (3,3). The implementation obviously needs some kind of backtracking algorithm. The effect of nondeterminism and the constraints between variables expressible by B can be much less obvious than in this simple example. The expression or predicate evaluation needed to compute a value can be much more complicated and time consuming than in this example. Not only the assignment of values to variables, but also which execution branch is chosen can be nondeterministic in some cases (not shown in example).

Table 2.18 shows all implemented substitution names, an example, when they are disabled (execution is not possible), and if they are deterministic. A substitution is only enabled when its body is enabled (recursion) and when its condition (if present) is true.

A substitution can be **indirectly** nondeterministic when is body contains at least one nondeterministic substitution. For example when one (or more) non deterministic substitutions are a part of substitution sequence. An other example would be an if then else substitution (which is deterministic), which will become potentially nondeterministic if one branch contains nondeterminism. The table only marks substitutions with a yes in the last column when they are always nondeterministic.

Some notes have to be made on this table:

- *Becomes Element of* and *Becomes such that* are nondeterministic if the set contains more than one element
- precondition and assertion conditions should not be false.
- a loop is not possible if no iteration is possible i.e the body is not enabled. This is different from a loop which is not entered because its condition is false.

Implementation of B's Substitution Statements

Figure 2.19 shows a code snippet of PyB's substitution implementation. The implementation uses Python's generator feature to implement the nondeterminism. Generators and the yield keyword were introduced in section 1.2.2. The generator uses two arguments: an AST node sub and an environment env. The generator returns true if a possible substitution branch has been found. In most cases, the environment was changed if one or more variables have been set to a value. A state change may be the side-effect of a substitution execution. The implementation of three nodes is shown in these examples:

- The first example ranges from line 2 to 3. The skip substitution implementation is always True, even if no value was changed.

Figure 2.18.: B substitutions and its PyB implementation [60]

name	example	disabled / not feasible	non deterministic
Block	BEGIN x:=42 END	if body is disabled	-
Skip	skip	never	-
Becomes Equal	x:=42	never	-
Precondition	PRE x:NAT THEN x:=42 END	if precondition is false (error) or body disabled	-
Assertion	ASSERT x:NAT THEN x:=42 END	if assertion is false (error)[8] or body disabled	-
Bounded choice	CHOICE x:=1 OR x:=2 END	if every choice body is disabled	yes
IF conditional	IF x:1,2,3 THEN x:=42 END	if every body of every enabled branch (condition True) is disabled and the ELSE branch is present but its body is disabled[9]	-
Conditional Bounded choice	SELECT $x > 0$ THEN y:=x-1 WHEN $x < 0$ THEN y= x+1 END	if every body of every enabled branch (condition True) is disabled	-
Case Conditional	CASE x-10 OF EITHER 2 THEN y:= 2 OR 7,13 THEN y:= 3 OR 8,11 THEN y:= 4 ELSE y:= 5 END END	if every body of every enabled branch (condition True) is disabled and the ELSE branch is present but its body is disabled[10]	-
Unbounded choice	ANY x BE $x = 42$ IN y=x+2 END	if precondition is false or body disabled for all values	yes
Local definition	LET x WHERE $x = 4$ THEN y=x+2 END	if body disabled	-
Becomes Element of	x:: 4 .. 10	if set of values is empty	yes
Becomes such that	$x : (x : Nat \wedge x < 42)$	if set of values is empty	yes
Local Variable	VAR v IN v:=y+1; y:=v*2 END	if body is disabled	-
Sequencing	x:=4; y:=x+1	if one substitution is disabled	-
Operation Call	op(42,TRUE)	if operation is disabled	-
While Loop	WHILE cpt<5 DO varLoc := varLoc + 1 ; cpt := cpt+1 INVARIANT cpt : NAT & cpt<= 5 & varLoc : NAT & varLoc = var1 + cpt VARIANT 5 - cpt END	if no iteration possible	-
Simultaneous	x:=y || y:=x	if one body is disabled	-

Figure 2.19.: PyBs substitution implementation (excerpt). omitted code: [...]

```
1  def exec_substitution(sub, env):
2      if isinstance(sub, ASkipSubstitution):
3          yield True # always possible
4      [...]
5      elif isinstance(sub, ABecomesElementOfSubstitution):
6          values = interpret(sub.children[-1], env)
7          if values==frozenset([]): #empty set has no elements ->
               ↪ subst. impossible
8              yield False
9          else:
10             for value in values:
11                 for child in sub.children[:-1]:
12                     assert isinstance(child,
                           ↪ AIdentifierExpression)
13                     env.set_value(child.idName, value)
14                 yield True # assign was successful
15     [...]
16     elif isinstance(sub, ABlockSubstitution):
17         ex_generator = exec_substitution(sub.children[-1], env)
18         for possible in ex_generator:
19             yield possible
```

- The second example ranges from line 5 to 14. This is an example of nondeterminism. This substitution implementation computes the set of possible values. If it is empty, the substitution is disabled and False is returned. Otherwise, all variables (possibly only one) are set to a value in line 13 and True is returned. Because the for loop successively ses every possible value when the generator is invoked again, every possible solution will be generated.
- the last example ranges from line 16 to 19. It is an example of recursion inside the generator. The block substitution is enabled when the generator for the body substitution returns true, so the result of this generator is returned. Of course, paths may be enabled and disabled if the substitution in the body is nondeterministic. For that reason, every result is successively checked in the for loop.

The substitution implementation uses side-effects to change states. A new state has been created for every choice point introduced by a non deterministic substitution. This happens also in the animation method.

Implementation Alternatives

Nondeterministic statements can of course **not** be implemented using random numbers. If the substitution implementation is used to enable model checking which needs to explore every possible state the substitution implementation must be able to systematically follow every possible execution path. Also, some kind of back tracking is needed to implement the nondeterminism. This back-tracking is automatically possible with generators. This back-tracking is efficient: For example, if a sequence of substitutions starts with complex computation (before a choice point) followed by a non deterministic substitution, the complex computation has only been computed once.

2.1.7. Structuring B machines

B offers a variety of methods to structure, reference and link B machines. Keywords are USES, SEES, INCLUDES, IMPORTS, EXTENDS. which express different visibility rules between machine components. This mechanism is complicated. Because this feature is not fully supported by PyB it will not be described any further.

2.1.8. Animation of B

Animation is the interactive exploration of a state space by a user to explore the B model's behavior. In this case, the transition from one state to an other is selected by the user. Animation is implemented by operation execution. After analyzing which operations are enabled and the execution of substitutions is added to PyB, an implementation of the interactive animation is simple. Animation may not be used to check a state, but

```
1  compute initial states
2  add initial states to states # e.g a stack
3  while states is not empty:
4      get next state from states
5      compute invariant
6      if invariant is false:
7          print history
8          quit with error message
9      compute next states
10     if next states is empty:
11         deadlock found
12         quit with error message
13     remove current state from states
14     for state in set of next states:
15         if state is new state:
16             add state to states
17 print number of checked states, and success message
```

Figure 2.20.: Model checking algorithm (pseudo code)

to check if a transition between states exists and behaved correct. For example, this includes (manually) verifying that a faulty state (violation of the invariant) is really reachable from the initial states or validating the model's behavior.

2.1.9. Model Checking Algorithm

The goal of model checking is to check the invariant for every reachable state.

Figure 2.20 shows PyB's model checking algorithm. To avoid confusion with the implementation details, the algorithm is presented in pseudo code. The algorithm is a naive search of all reachable states of the model. If all states are checked against the invariant without finding any violation, the model checking was successful. If the state space is very large or infinite, this algorithm will not terminate.

The algorithm will be presented in more detail:
First, all initial states are computed. This can differ from machine to machine depending on whether the model has a properties clause, parameters or deferred sets. For example, constants must be checked against the properties clause. The main algorithm is the a loop from line 3 to 16. If the state space is finite, it will terminate because only new states are added to the states data structure and a state is removed in every iteration. Of course checking if a state is new depends on the number of visited states because every visible state is compared to the new state.

The term "state" has to be clarified in this context: If more than one machine is involved, every machine maintains its own B machine state (for example a main B machine which includes an other B machine). Of course in this algorithm a state refers to the a tuple of states of all involved B machines. This implementation detail is only revealed to show that there is no inconsistency with the B state representation subsection (2.1.4) i.e. algorithm also works on more than one machine.

Optionally, a history of visited states and predecessor states can be recorded during state computation in line 9. This history is printed after a faulty state has been found. This information can be useful to understand which operation sequence led to the invariant violation.

The exploration of the state space can be done using a depth-first search or a breadth-first search depending on a PyB flag. The flag is modified by the PyB user. If depth-first search is performed, the states data structure is a stack. Otherwise it is a queue. New states are always added at the end of the states data structure data structure.

States are compared using a hashing algorithm. Of course, a hash collision is always possible. If two states have the same hash, a comparison by value is performed.

Considering data validation the model checking feature is not necessary for a second tool chain because this can be done by the main tool. However, it is useful for PyPy experiments, to investigate if a language like B can benefit from tracing JITs.

This algorithm is quadratic in the number of states. Every state has to be compared against all seen states. Hashing can only speed up the comparison computation.

2.2. Using PyB

2.2.1. PyB operation modes

There are various features within PyB. Some of them are only useful as second tool chain. Others are only justified as part of the PyPy experiments and some of them are useful for both topics of this thesis.

- Check precomputed states:
  Checking a precomputed state is PyBs primary function as a second tool chain. PyB reads in the state, starts the machine to be checked and evaluates its properties and invariant clause. PyB outputs which values were successfully used from the solution file. If no violation occurred with the used values, PyB returns True. This feature can be used at the command line for a single state and extended (e.g. by a script) to a check the state space of a machine (example in the next subsection). An example call and output:

```
$ python pyB.py -c Cruise_finite1.mch
   ↪ Cruise_finite1_state.txt
reading solution file examples/
   ↪ Cruise_finite1_values.txt ...
learned from solution-file (constants and variables
   ↪ ): ['CruiseSpeedAtMax', 'CruiseActive', '
   ↪ NumberOfSetCruise', 'ObstacleRelativeSpeed',
   ↪ 'VehicleTryKeepTimeGap', '
   ↪ CruiseSpeedChangeInProgress', 'CruiseAllowed'
   ↪ , 'VehicleTryKeepSpeed', '
   ↪ VehicleAtCruiseSpeed', 'VehicleCanKeepSpeed',
   ↪  'SpeedAboveMax', 'ObstacleStatusJustChanged'
   ↪ , 'ObstaclePresent', 'ObstacleDistance', '
   ↪ CCInitialisationInProgress']
checking properties (with prob solutions) now..
...no vialation found
Invariant: True
```

- Model checking mode: The model checking mode explores the whole state space as described in the previous section and figure 2.20. It does not use an external solution file and was added to PyB to evaluate the PyPy goals in this thesis (see)1.3.2). An example call and output:

```
$ python pyB.py -mc examples/Lift2.mch
checked 100 states.No invariant violation found.
$
```

- Interactive animation: The animation mode is inspired by ProB which also offers this feature. Every transition from one state to the next can be chosen by the user. The state and the status of the invariant is printed after every action. The user can explore the behavior of the model, e.g. to check if it matches the requirements or to debug it. The feature was included because of the small effort needed after model checking was added to PyB. In the context of a second tool chain it can be used (to some extent) to check which operations are enabled in a state and which transitions between states are possible. It can be used to validate a model. PyB offers no graphical interface but only text mode. Here is an example call, user input, and tool output:

```
$ python pyB.py examples/scheduler.mch
[0]: INITIALISATION(active={} ready={} waiting={} )
[1]: leave PyB

Input (0-1):0
active={} waiting={} process1='p1' process3='p3'
   ↪ process2='p2' ready={} PID={'p1', 'p3', 'p2'}
[0]: new(pp='process1' )
[1]: new(pp='process3' )
[2]: new(pp='process2' )
[3]: undo
[4]: leave PyB

Input (0-4):1
```

- REPL: PyB offers a read–eval–print loop (REPL). This interactive operation mode can be used to evaluate B predicates to quickly compute values or test the tool. Because this mode is of little interest in the context of a second tool chain or the intended PyPy experiments, it is not fully implemented. For example most computation of bound variables are done brute force. An example:

```
$ python pyB.py -repl
 >>{x|x>5 & x<10 & x mod 2=0}
{8, 6}
>>
```

Some of these operation modes can be modified by a config file. For example the checking of assertion can be enabled.

2.2.2. Linking PyB with ProB

Below we use an example of a complex B machine (cruise control model 4.1.3) with a simple state. Figure 2.21 shows ProB in its animation-mode for this example. At some point the user can save the B-state to a file (Figure 2.22).
This solution-file contains a list of constant and variable values. The right side of each equation may be any B-formula: a number, a set, a relation, a function or even a lambda-expression. The state does not contain informations about constants like MAX INT values or about bound variables, e.g. when dealing with existential quantified predicates. Eventually PyB reads this file, generates a B-state, evaluates the Properties- and Invariant clause of the B-file and outputs if a safety property was violated in this state. Currently this process is automated via a Python script, but this will be fully included into the official ProB release in the future.

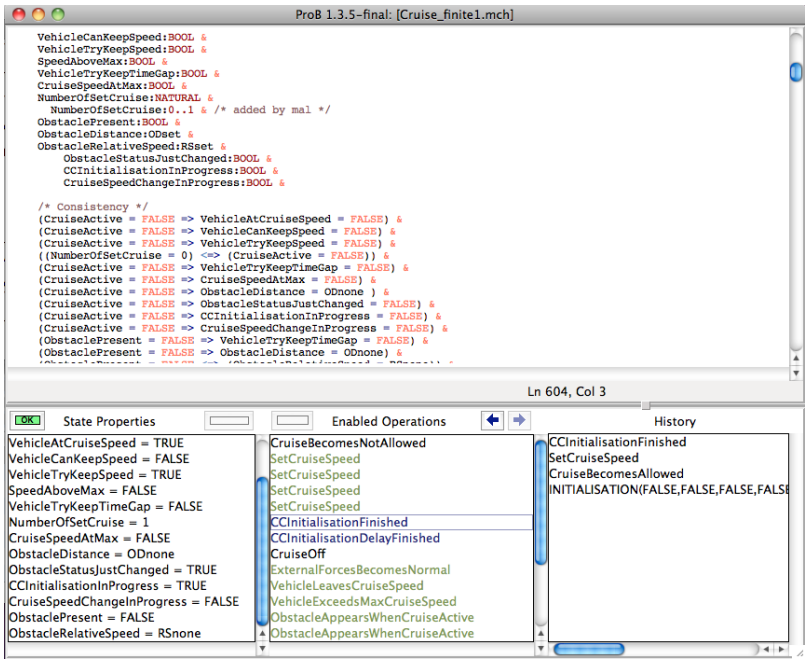


Figure 2.21.: ProB animating a B machine: The B machine code, the B-state (values and constants), enabled operations, history

```
/* Variables */
 #PREDICATE
    CruiseAllowed = FALSE
 & CruiseActive = FALSE
 & VehicleAtCruiseSpeed = FALSE
 & VehicleCanKeepSpeed = FALSE
 & VehicleTryKeepSpeed = FALSE
 & SpeedAboveMax = FALSE
 & VehicleTryKeepTimeGap = FALSE
 & NumberOfSetCruise = 0
 & CruiseSpeedAtMax = FALSE
 & ObstacleDistance = ODnone
 & ObstacleStatusJustChanged = FALSE
 & CCInitialisationInProgress = FALSE
 & CruiseSpeedChangeInProgress = FALSE
 & ObstaclePresent = TRUE
 & ObstacleRelativeSpeed = RSequal
```

Figure 2.22.: A simple input example of PyB: This file contains all values and constants of a B-state. the first line #PREDICATE was added for parsing reasons

Possible Linking Modes:

- A script drives the process: Every state is created and double checked via a ProB/PyB cli call.
- ProB drives the process: PyB is called by ProB after every state creation/ transition. The result from PyB can be used by ProB to create error messages although this is not implemented yet
- Manual use: The input is manually checked using both tools separately. For example, the animation mode or a complete model checking of both tools can be used without the exchange of state informations.

More detailed examples can be found in the case studies subsection of this thesis. This example is taken my paper at FIDE [69]

2.3. Summary

This chapter described the tool implementation. This included why the clean room approach was violated for parsing, how typing was solved using unification, how data is represented, how predicates can be evaluated, how nondeterminism was implemented using Python generators, the operation modes available in PyB, how the main tool was linked and how model checking was implemented. Apart from constraint solving and infinity (discussed in the next chapter), the tool is a simple, straight forward implementation which uses standard approaches.

3

Implementation Challenges for B

As can be seen in the last chapter, most of the implementation of B described by this thesis was straightforward: PyB is a simple implementation, easy to test and maintain. Despite its simplicity, there are always some aspects in language implementations which are less obvious. In this case, the concept of infinite set possible in B, and solving of quantified predicates were difficult. This was more complex than simple interpreter construction which only requires receiving input, program and producing the correct output. These two aspects are described below by showing 'difficult' B expressions and how they are evaluated by PyB. The third subsection is about timeouts and data representation which was not particularly difficult but is related to the other two topics.

3.1. Infinite and large sets

Because of the set-based nature of the B-Language, a good set implementation is the key to an efficient tool. PyB uses two types of set representation: explicit set representations (in chapter 2) and symbolic set representations (this chapter). The next two subsections introduce and compare representations. The reasons for introducing the symbolic set representation will also be described.

3.1.1. Explicit Set Representation

The notation *explicit set* refers to the data representation discussed in the previous chapter. As a summary: Explicit sets are constructed using built-in Python data types.

Figure 3.1.: PyB's explicit set representation (selective)

no.	B expression	Name	Explicit PyB (Python) Representation	Mathematical equivalent
1	$\{1, 2, 3\}$	Set Enumeration	frozenset([1,2,3])	{1,2,3}
2	$\{x \mid x : NATURAL \ \& \ x < 5\}$	Set Comprehension	frozenset([0,1,2,3,4])	$\{x \mid x \in \mathbb{N} \wedge x < 5\}$
3	$\%(x).(x : NATURAL \ \& \ x < 5 \mid x * x)$	Lambda Abstraction	frozenset([(0,0),(1,1), (2,4),(3,9),(4,16)])	$\lambda x.x \in 0..4 \mid x^2$
4	$\{x, y \mid x : NATURAL \ \& \ x < 5 \ y = x * x\}$	Set Comprehension	frozenset([(0,0),(1,1), (2,4),(3,9),(4,16)])	$\{x, y \mid x \in \mathbb{N} \wedge x < 5 \wedge y = x^2\}$

These types are frozenset[1], tuple, integer, boolean and string. It is possible to represent every finite (and small) B-set with these basic types i.e. with a combination of frozenset, tuples and primitive types (see section 2.1.3). For example, a function can be expressed as a set of tuples of other types (like integers).

The advantage of this representation is simplicity. The simplicity of the set representation affects the entire PyB implementation. Because a lot of B predicates and expressions use set expressions, most of B is easily implemented whenever sets are finite. The code that is produced is maintainable. One of the main goals of a second tool chain: "writing a simple tool to check a complex tool", can be achieved this way.

Sadly this approach has limitations. B sets can be very large (e.g 2^{32} elements) or infinite. Even if it was possible to hold many large sets in main memory, the computation of these sets may take too much time if an explicit representation is used. Large sets are an issue because time and memory-resources are limited. Particularly in the case of infinite sets, an alternative representation is mandatory. An explicit set must consist of a limited number of elements. A symbolic set can be either finite or infinite.

The table in Figure 3.1 shows some examples of B-sets (Name and Syntax), their chosen explicit Python implementation and a mathematical formula representing this set. Column 3 shows Python code: Frozenset is the constructor call of the build-in frozenset type. Square brackets and round brackets are syntactic sugar used for the list constructor and the tuple constructor.

3.1.2. Symbolic Set Representation

Motivation

An explicit set representation limits the number of possible B machines which can be processed. In the context of data validation, it will limit the number of solution files which can be used. Even if the amount of solutions to be checked is small (less then 1000 elements per set), finite sets, large and infinite sets require another implementation,

[1] https://docs.python.org/2.4/lib/types-set.html

because large or infinite sets may not occur in the solution file but in the machine code to be checked. This problem is not rare or theoretical, but occurs in machines from industry. And if a solution of a B property or B invariant is an infinite set, this set will also **appear in a solution** (file) and must be handled by PyB. A solution may contain variables bound to a (infinite) set comprehension or a lambda expression, i.e. a predicate with bound variables. In this case, the value of such a variable will always be a set defined by a predicate if no explicit finite set representation exists.

There is no alternative to an infinite set representation: Sometimes the explicit representation is not practical for performance reasons. In other cases, when PyB checks a solution using finite sets in the bounds of the minimum and maximum integer ranges, may simply not be correct. Checking the computation of a complex tool like ProB only in the bounds of finite integer ranges may not be sufficient for a convincing double check. Of course it can still be useful if the main tool behaves similar but the evaluation of large or infinite sets must be possible. PyB solves this problem by **not** using explicit set representation whenever possible. This is realized using a symbolic set representation and (simple) "constraint based" enumeration[2] (next subsection).

An example of an infinite set is **f=x.(x:INTEGER|-x)**, which is the set of tuples mapping any integer number (except zero) to its negative (f={(1,-1), (-1,1), (2,-2)... }). In a solution file (compute by ProB), the variable f will be assigned to that set. This set can **not** be represented using a finite set and will always appear as this expression inside a solution file.

Even if a set is not infinite, a symbolic representation is used if its very large [3] . Figure 3.2 shows the performance problems which occur when explicit sets are used inside a counter B machine (like the lift from Figure 1.1). The Figure shows the time to check the simple invariant $x \in 0..n$ with values of n from 100 to 100000. In this case, there are n states. If the interval (set of numbers from 0 to n) is created in every check, the tool becomes unable to check this property in reasonable time. Caching can also not solve this problem at some set size. So even if this set is always finite, a symbolic representation is needed to avoid a timeout. The symbolic version in contrast performs very well. Not every large set can be identified as easily as in this example. A large set can be the product of set operations like powerset or the cartesian product.

Implementation

A symbolic set is implemented by a Python class. The term symbolic was chosen to imply an implementation at a higher level of abstraction, which does not use concrete

[2] set enumeration should not be confused with Bs enumerated sets, which are something completely different

[3] e.g 2^{32} elements

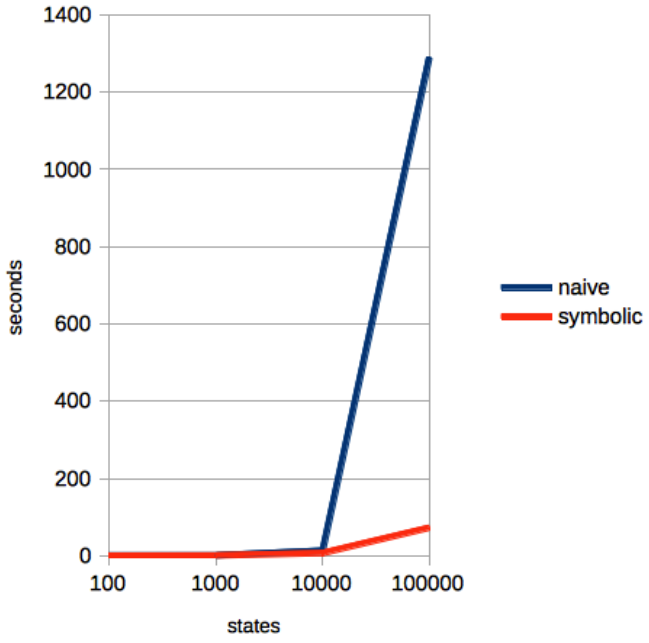


Figure 3.2.: Checking a invariant x:0..n using explicit or symbolic sets

data to represent set elements. Examples of symbolic sets are the infinite INTEGER set, STRING set, an interval set, composed symbolic sets like the powerset of a symbolic set, the union of two symbolic sets, the set of relations or a set defined by a predicate like a lambda expression. Forty symbolic set classes are implemented in PyB(which are listed in appendix E).

Symbolic sets can be categorized in four kinds of sets:

- Sets representing infinite B sets, for example the NATURAL set (natural numbers).
- Sets representing large B sets, for example the NAT1 set (which is bound by the MAX_INT value)
- Set operations between at least one symbolic set (potentially infinite) and any other set. For example the union of NATURAL and some set enumeration or the power set of a power set of a symbolic set. These sets will be referenced in this and the next subsection as **composed symbolic sets**.
- Set expressions using bound variables and quantifiers. Those expressions can be finite and small, finite and large or infinite. Without any analysis, this is unknown (see constraint solving in the next subsection).

Depending on the kind of symbolic set, certain properties for the set instance may be:

set name, number of bound variables, the predicate defining the set, an expression (for lambda expressions) and other (explicit or symbolic) sets defining the symbolic set. All symbolic sets inherit from a base class, which implements all default set operations using brute force (conversion to explicit sets). To achieve better performance, these operations are overwritten by the subclasses. Membership and enumeration methods are nearly always overwritten. A set generator method must be implemented. A membership test can be evaluated without enumerating the whole set which is important for the common case of using membership in B to express typing informations of a set like $S : NAT * NAT$

When an explicit set representation is used, the AST interpreter can treat predicate and expression nodes in the same way in one recursive evaluation method (see 2.1.5). This approach treats predicates like functions which return the value True or False. For example, a membership test is implemented by recursively walking the AST, generating the set and element and returning True or False by using the built-in frozenset method, e.g. *1 in frozenset([1,2,3])*. Symbolic sets must behave like explicit sets if both implementations should be interchangeable.

Using symbolic sets, every set expression (AST node) has a symbolic counterpart. A class represents the set expression, and the class methods implements boolean functions (e.g. membership) and basic set operations like union or subset (see table 3.1 and 3.2.). Every symbolic set implements a method for every possible B set operation. Set operations are those operations which return a set (e.g. intersection). Non-set operations such as conjunction, disjunction, implication, equivalence, negation, and quantified predicates have no method counterparts, with the exception of equality, inequality and function application (which returns data of some B type).

The set operations are implemented via the Python special methods[4]. This feature is comparable to operator overwriting in C/C++. For example, a membership test like $x : S$ calls the __contains__ method of S. In this example, the argument will be x and the return value will be True or False. A overview of important special methods is shown in table 3.1 and 3.2. If this method is implemented (or overwritten by a subclass), S can be replaced by a symbolic set instead of a frozenset. In some cases, default implementations of the symbolic set base class may be sufficient. In other cases, an adjusted implementation for the class is a necessity for performance reasons.

Every symbolic set class inherits from a base class: SymbolicSet. This class implements (brute-force) default implementations for the operations: union, intersection, subset, superset, equality, inequality, cartesian product and set difference. Some methods of the base class are delegating to other methods. For example, __le__ is delegated to issubset (which is a frozenset method) and __rand__ (operand on the right side) switches operands

[4]https://docs.python.org/2/reference/datamodel.html#special-method-names

Table 3.1.: Set predicates and their special method counterparts

name	example	special method
membership	$x : S$	__contains__
equality	P = Q	__eq__
inequality	P /= Q	__ne__
superset	$T <: S$	__ge__
subset	$S <: T$	__le__

Table 3.2.: Basic set operations and their special method counterparts

name	example	special method
union	$S \backslash / T$	__or__
intersection	$S / \backslash T$	__and__
set difference	$S - T$	__sub__
cartesian product	$S * T$	__mul__
function application	$f(x)$	__getitem__
relational image	r[S]	__getitem__

and calls __and__. Every subclass must implement a membership method, a generator method for the set, and overwrite many other operations. A generator method must satisfy the following properties:

- (1) It yields only one element of the set at every call
- (2) No element is generated twice
- (3) It guarantees the same enumeration order on multiple enumerations

Points 2-3 are about correctness and determinism. For example, a quantified sum expression will produce a false sum value if a solution is generated twice, while a random enumeration order would result in different tool behaviors (produce timeout or evaluation result), e.g. when evaluating a existential quantified predicate over an infinite set. Point 1 is about performance, because sometimes only a subset of the whole set needs to be enumerated. For example, if a symbolic set is used in a quantified expression, then the enumeration loop will terminate if a False (universally quantified) or a True (existentially quantified) element is found because of a break in the implementation method of quantified predicates inside the interpreter. Enumeration is introduced in detail in the next subsection.

PyB's predicate evaluation uses procedural programming when dealing with explicit finite sets, but it is an object oriented interpreter when dealing with symbolic sets. In the case of explicit sets the computation is done in one interpretation function which consists of a case for every kind of operation. In the case of symbolic sets the methods of the symbolic set classes are called. This is transparent because of the use of special

```
1  class SymbolicIntervalSet(LargeSet):
2
3    def __init__(self, l, r):
4      SymbolicSet.__init__(self)
5      self.l = l
6      self.r = r
7
8    def __contains__(self, element):
9      if not isinstance(element, int):
10       raise PyBException("Fail: membership test with non-
           ↪ integer")
11     if element<=self.r and element>=self.l:
12       return True
13     else:
14       return False
15
16   def return_generator(self):
17     for i in range(self.l, self.r+1):
18       yield i
19
20   def __eq__(self, other):
21     if self.__class__ == other.__class__:
22       return other.l==self.l and other.r==self.r
23     return SymbolicSet.__eq__(self,other)
```

Figure 3.3.: Python symbolic set example

methods. The reason for this design decision is the re-usability of the AST interpreter which was written only for explicit sets. This way the simplicity of the interpreter and evaluation method is preserved.

Figure 3.3 shows a simple example of a symbolic set implementation. Most functions like set union or subset are inherited from the LargeSet class (which is a subclass of SymbolicSet class) and are not shown. The default implementation of set membership (__contains__) and equality (__eq__) are overwritten for performance reasons. Computing the equality of two interval sets is very quick, but every other comparison would cause a call of the default implementation (line 23), for example the expression {1,2,3,4,5}=1..5. All other operations are implemented by the SymbolicSet class and may be overwritten for better performance (omitted). Anyway, this small example would be already enough to work with the implementation of the last chapter. It can transparently replace a frozenset. The code introduced in interpretation subsection (Figure 2.16) is also used on symbolic sets, but the set operations are dispatched to the dedicated symbolic set

methods instead to the frozenset built-in implementation.

All implementations are recursive methods. If a symbolic set is defined with other symbolic sets, usually their methods are called as well because such symbolic sets are tree like structures. For example, consider the set of all functions F between two sets $F = S \leftrightarrow T$. A membership test of a function f ($f \in F$) is done by checking the membership of fs elements in S and T. If they are symbolic sets, their membership method is called. This is done for all these methods, e.g. for set enumeration.

The equality method (__eq__) checks if the other instance is also an interval set. Otherwise, the default implementation is called, for example in the case $\{1, 2, 3\} = 1..3$. Instances of this class behave like the built-in frozenset type. In a language like C++ this would be solved by operator overwriting.

3.1.3. Limitations and Alternatives

Limitations

The symbolic approach is limited. All sorts of comparisons of sets (S=T) are an issue. In general, it is not possible to efficiently check if two symbolic sets are equal. If the sets are instances of the same class, a AST comparison is done. For example, the expression $x.(x : NAT \mid x + 1) = y.(y : NAT \mid y + 1)$ can be successfully checked. Also some special cases like $(x, y).(x : N \ \& \ y : N \mid x) = prj1(N, N)$ are implemented because they are common in ProB solution files. In general, the comparison of a symbolic set with an other symbolic set leads to enumeration of the symbolic sets. For example, checking $\{x, y | x : \{1, 2, 3\} \ \& \ y = x + 1\} = x.(x : \{1, 2, 3\} | x + 1)$ is true can only be done by PyB after enumeration (conversion to explicit set) has been done. PyB will compare two explicit sets in this case. Usually PyB is unaware if a quicker enumeration (than brute force) of a symbolic set is possible. The enumeration is also unavoidable in the case of a comparison with an explicit set. A membership check of all explicit elements is of course not correct, because it has to be assured that the symbolic set does not contain any other elements. Every operation which needs an equality check (like strict inclusion of a set) suffers from this limitation. Also, it is not possible to check function properties like injectivity, surjectivity or if a function is total if the underlying sets are infinite.

The enumeration of infinite sets leads to a timeout. When PyB is checking solutions generated by ProB, the enumeration is always be possible in theory if the ProB result is correct. Unfortunately, PyB has limited constraint solving capabilities which are presented in the next subsection.

There is one exception to the reusage of the evaluation method of the AST interpreter for symbolic sets: the usage of symbolic sets which represent set operations. For example

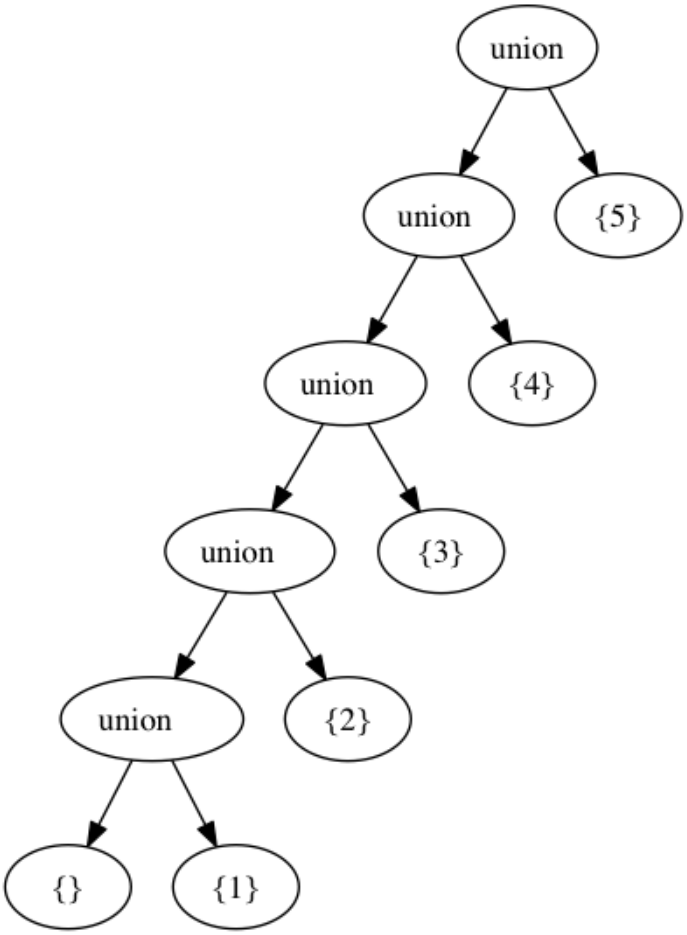


Figure 3.4.: Tree of symbolic union sets (which is avoided by checks inside PyB)

the test if an element is member of a infinite sets $x \in S \cup NATURAL$ (e.g. S=1..3) will cause an infinite enumeration of NATURAL if the union is performed. This is prevented by a symbolic union set. A symbolic union set delegates a membership tests to NATURAL and the set S. Creating this symbolic union set by default inside the AST interpreter would cause problems for other examples. An example is a while loop substitution which adds an integer element at every iteration by starting from the empty set. Using an explicit set in this while-loop case would be fine, using the symbolic union set will create a linked list of symbolic union sets. Figure 3.4 illustrates this problem, leading to inefficient membership checks. This is solved by analyzing the operands of every set operation inside the evaluation method: operations on explicit sets never create an union set instance but are computed using the built-in frozenset union. So the tool needs both set representations.

The symbolic set implementation indirectly uses a blacklist of excluded elements. For example, the expression NAT-{0} causes the creation of a symbolic difference set instance and two pointers to a symbolic NAT set and an explicit set containing the integer zero (frozenset([0])). This is a tree data structure which can only be collapsed by PyB using explicit enumeration. If the infinite or large set is assigned to a variable and elements or finite subsets are removed or added in a loop, this representation becomes larger and larger, because the enumeration to an explicit representation is neither possible nor desired. This is not considered a big problem because the occurrence of this scenario is unlikely to be found in machines from industry and not relevant for double checking of only one state.

Alternatives

An alternative to symbolic set classes is the usage of modified AST nodes: Instead of introducing symbolic set classes, the AST nodse of sets and set expressions could have been extended by methods which are now part of the symbolic set classes. This approach was discarded because the immutability of the AST is desired for the RPython translation (next chapter) and to separate of concerns. The AST is the input for the interpreter and not data to be mutated. Also, one single function (a big switch) for every membership, enumeration etc. of every AST node would be an alternative to symbolic sets, but it would be less readable than the object oriented approach of using a symbolic set class for every set AST node. Every AST node has its symbolic instance counterpart. This avoids enumeration problems if an expression contains more than one set of the same kind. Also, symbolic sets which represent set operations cannot be expressed this way.

A check if an enumeration must lead to a timeout (for example by checking if the domains of lambda expressions are infinite) could be future work. This could be useful but would need additional time and effort.

The symbolic set base class was not inherited from the built-in frozenset type and the frozenset type was not modified. This would not have worked because RPython doesn't

support subclassing built-in types. Also, deriving a potentially infinite set from a class which represents finite sets is bad design because this could cause bugs inside PyB. Historically, a common base class was removed from Python's two set types frozenset (immutable and hashable) and set (mutable). Also, the Python coercion exceptions are avoided (3.4.9 Python manual).

3.2. Set enumeration

3.2.1. Motivation and Introduction

Set enumeration is the process of generating all elements of a set. In some cases, this process can be time consuming. Symbolic set representations can delay or avoid the enumeration of a set depending on the expression. For example, a member check ($x \in S$) does not cause enumeration. However, in some cases the enumeration is unavoidable. When two sets are in relation to each other, a partial enumeration of one set (e.g. in case of subset predicates) or a full enumeration of both (e.g. in case of equality) may be necessary. In most cases, evaluating a predicate between sets causes the evaluation of set equality. Evaluating equality is very important in the context of ProB-PyB data validation. Of course, in some cases an enumeration is not needed to evaluate equality:

Enumeration **not** needed to check equality, when:

1. two explicit frozensets are compared: They are already enumerated.
2. the equality of a finite and an infinite set is evaluated: it is obviously False. (e.g. a frozenset instance an a smybolic NaturalSet instance)
3. sets of different types are compared (e.g NAT=S*T). This was done by type checking.

Not every possible B expression can be checked by PyB: If two sets are infinite or very large, the question of equality remains unknown for PyB in most cases. In practice (e.g. data validation), a symbolic set is usually compared with a finite explicit frozenset or itself (one in the B file the other in the solution file). Two instances of the same set (apart from variable renaming) can be compared using AST comparison.

Composed symbolic sets which are **not** defined by a predicate (not one of the predicates in the enumeration below) can be enumerated by (lazy and recursive) brute force enumerators as described in the last section (e.g. figure 3.3 and 3.6). But the enumeration implementation of sets defined by a predicate is not obvious or trivial. Figure 3.5 is a simplified set example from a industrial data validation file which can be computed by PyB:

The example 3.5 illustrates the complexity of the class of problems that has to be solved.

$$\{x|x \in (\mathbb{Z} * \mathbb{Z}) * \mathbb{Z} \wedge (x \notin \{((1 \mapsto 2) \mapsto 3), ((4 \mapsto 5) \mapsto 6)\} \vee (prj1(\mathbb{Z} * \mathbb{Z}, \mathbb{Z})(x) \notin dom(\{((1 \mapsto 2) \mapsto 83), ((1 \mapsto 15) \mapsto 83)\}) \wedge x \in ((0..209) * (0..209)) * \{-1\}))\}$$

Figure 3.5.: A B set from a ProB solution file of a industrial machine. $prj1(X, Y) = \{x, y, z|x, y, z \in X \times Y \times X \wedge z = x\}$

The set is defined by one predicated over x. It can be split into several subpredicates (conjunctions).

Many B constructs can be defined by a predicate. Here is a list of all B predicates, expressions, and substitutions which can cause an enumeration:

1. Universal quantification $\forall\ (x_0 \dots x_n) \cdot P(x_0 \dots x_n)$
2. Existential quantification $\exists\ (x_0 \dots x_n) \cdot P(x_0 \dots x_n)$
3. General sum $\sum\ (x_0 \dots x_n) \cdot (P(x_0 \dots x_n)|E(x_0 \dots x_n))$
4. General product $\Pi\ (x_0 \dots x_n) \cdot (P(x_0 \dots x_n)|E(x_0 \dots x_n))$
5. Lambda expressions $\lambda\ (x_0 \dots x_n) \cdot (P(x_0 \dots x_n)|E(x_0 \dots x_n))$
6. Set comprehensions $\{(x_0 \dots x_n) \cdot P(x_0 \dots x_n)\}$
7. Quantified intersections $\bigcap\ (x_0 \dots x_n) \cdot P(x_0 \dots x_n)$
8. Quantified unions $\bigcup(x_0 \dots x_n).P(x_0 \dots x_n)$
9. Any-, Let- and Becomes substitutions
10. Parameter values of operations

The enumeration of those constructs are the topic of the next paragraphs.

PyBs enumeration of symbolic sets which are defined by a predicate can be summarized as follows: The input is a set of bound variables $(x_1 \dots x_n)$ and a predicate. For each variable there is a set of values determined by the type of the variable which is called a domain. The predicate, which can be a conjunction $(c_1 \dots c_n)$ of subpredicates must be fulfilled. These subpredicates are constraints for possible elements of the sets. PyB has to compute variable value combinations from their domains which fulfill all the predicates. A solution is a tuple of values corresponding to every bound variable which fulfills the predicate. This problem is called the constraint satisfaction problem. In "Artificial Intelligence: A Modern Approach" Stuart Russell and Peter Norvig [57] define this problem as follows:

A constraint satisfaction problem (or CSP) is defined by a set of variables, $X_1, X_2, \dots, X_n$, and a set of constraints, $C_1, C_2, \dots, C_m$. Each variable X_i has a nonempty domain D_i of possible values. Each constraint C_i involves some subset of the variables and specifies

the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all of the variables, $\{X_i = v_i, X_j = V_j, \dots\}$. *An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a CSP is a complete assignment that satisfies all the constraints.*

If the domains are finite, this search problem can be solved by a brute force algorithm which checks all possible value combinations for all (sub-)predicates. If the domains contain only a few elements and the number of variables is small, this strategy will quickly compute a B set. Otherwise it will run into a timeout (next subsection). Not all domains in B are finite and discrete[5]. An algorithm must find not one but all solutions to this problem. The set of **all solutions** is desired (every solution is a complete assignment). This aspect is important to understand the chosen implementation.

For example in a set comprehension $\{x \in NAT \wedge x < 3 \wedge x > 0\}$ got the solution x=1 and x=2. Both solutions are needed to enumerate the set. So the set of all solutions would be $\{(x, 1), (x, 2)\}$ and the set itself would be $\{1, 2\}$.

The constraints of this problem are the B predicates. Only constraints which involve one bound variable are implemented in the domain restriction computation (next subsection). Constraints involving only one variable are called unary constraints.

The simplest type is the unary constraint, which restricts the value of a single variable. [...] A binary constraint relates two variables.[...] Higher-order constraints involve three or more variables. [57]

Binary and higher-order constraints (more than 2 variables) are used by PyB by enumerating the domains of all involved variables. They are only used to alter the domain of a single variable if all other depended variables have already reduced to a finite domain (see example in 3.2.2). For example the constraint $w = x + y + z$ will only be used by PyB to constrain w if finite domains of x, y and z have already been computed. A heuristic and variable ordering is used to improve the performance of this labeling approach.

A CSP can be represented as a directed graph. The nodes are the variables and their current domains, and the edges (also called arcs) are the constraints which affect these variables. There are different kinds of consistency for the CSP. This definition is from A. K. Mackworth [47]:

1. (A) Node consistency
   Node i is node consistent iff for any value $x \in D_i, P_i(x)$ holds.
2. (B) Arc consistency
   Arc (i,j) is arc consistent iff for any value $x \in D_i$ such that $P_i(x)$, there is a value $y \in D_j$ such that $P_j(y)$ and $P_{ij}(x, y)$.
3. (C) Path consistency
   A path of length m through the nodes $(i_0, i_1, \dots, i_m)$ is path consistent iff for any

[5]The approach in this chapter will only work on finite discrete domains

value $x \in D_{i_0}$ and $y \in D_{i_m}$ such that $P_{i_0}(x)$ and $P_{i_m}(y)$ and $P_{i_0 I_m}(x, y)$, there is a sequence of values $z_1 \in D_i, \ldots, z_{m-1} \in D_{i_{m-1}}$ such that

a) $P_{i_1}(z_1)$ and ... and $P_{i_{m-1}}(z_{m-1})$,

b) $P_{i_0 i_1}(x, z_1)$ and $P_{i_1 i_2}(z_1, z_2)$ and ... and $P_{i_{m-1} i_m}(z_{m-1}, y)$

In this implementation D_i are the possible variable values and P_i are the subpredicates constraining those variables. The next subsection will describe how PyB archives all those kinds of consistencies with a labeling phase after domain reduction (constraint usage).

3.2.2. Implementation

There are different kinds of enumerations implemented in PyB.

1. enumeration of a type, e.g integer
2. enumeration of symbolic sets.
   a) defined by a predicate or not defined by a predicate
   b) lazy or non lazy (full enumeration)

An enumeration is triggered by calling a symbolic set's enumeration method. Every symbolic set implements its own enumeration method. Whether this enumeration is a simple or a complex one depends on the set type. This means that whether or not the constraint solver is called depends on the kind of set. If it is a full enumeration of all elements, the result is cached and reused on further enumeration calls.

A simple enumeration of a composed set (see last section and appendix E) without any predicate can easily be implemented by recursively calling generators. This can be seen in figure 3.6. For example, the composed symbolic set of all relation between S and T ($S \leftrightarrow T$) can be enumerated by calling the generator for S and T and combining the result to a set element. The recursion base case is a non composed set like an explicit frozenset or a symbolic set like NAT1. It can be implemented by using Python generators. This can be seen in Figure 1.5(chapter 1). If a symbolic set is not defined by a predicate, it is not composed of more than two other sets. This simple algorithm is used to solve the more complex enumeration problem of sets defined by predicates.

This implementation is lazy: A caller of this method does not have to generate all elements of this set. The implementation generates one element after another and does not generate an element twice. If all involved sets are finite, it can be used to enumerate all elements.

The following text is about the enumeration of sets defined by predicates:
This problem is solved by a naive constraint solver implementation (contribution of this

```
1 enumerate_set(astNode):
2     if composed set:
3         for every lvalue in enumerate_set(astNode.leftChildNode
              ↪ ):
4             for every rvalue in enumerate_set(astNode.
                  ↪ rightChildNode):
5                 element = compute_tuple(rvalue, lvalue)
6                 yield element
7     else:
8         for element in lazy_enumerate_generator(astNode):
9             yield element
```

Figure 3.6.: Naive enumeration of set defined by two other sets and no predicate (pseudo code)

thesis) which uses unary constraints to check node consistency of finite domains. Also, unary constraints are used for domain reduction. **PyB-Constraint** restricts seemingly infinite domains to finite ones if possible. In other words: it recognizes whether only a finite subset of an infinite domain is needed to enumerate all elements of a set (see section below). Also binary and n-ary constraints are used in combination with variable labeling. Finally, a consistency check (labeling) is performed using a depth first search strategy.

The predicate to be checked and the conjunction of constrains (or subpredicates) will be used synonymously in this subsection.

The motivation of constraint solving in general:
Of course, a brute force algorithm which computes every combination of values, checks them against the predicate (constraints), and returns the correct results will solve the constraint satisfaction problem if domains are finite. Such an algorithm will be easy, correct, and in most cases useless. It will be useless because of combinatorial explosion. B does not only allow boolean and integer arithmetic but also sets of sets. The set of all variable combinations will be too large: even computing less complex set definitions will take too much time. So the goal of the PyB constraint solver is improving performance (compared to the naive brute force approach). Every computation which is done by the constraint solver must result in a reduced amount of computations in the labeling phase. In most cases, computing a superset of the desired domain will be good enough. Some few wrong combinations can be quickly eliminated by checking them against the predicates. Thus, an over approximation of the desired domain and removing the false elements in the last phase can be less time consuming than an exact domain computation before labeling.

Recognizing finite domains is a necessity. e.g. in $x : INTEGER \ \& \ x = 42$ the domain of x is obviously not infinite. In this case constraint solving is not only about performance but about enabling PyB to check a larger class of predicates. Without constraint solving, such simple examples would not work.

Figure 3.7 shows the implemented constraint solving algorithm in pseudo code. Brackets show the parameters used to compute a step. The algorithm consists of three phases: An analysis phase, a domain reduction (solving) phase and a labeling (checking) phase. The analysis phase collects information about the problem while the domain reduction phase uses this informations to constrain the variables domains. Finally, the labeling phase (lazily) computes one solution after another.

Analysis phase

The first five lines show the analysis phase. This predicate is split into its conjuncts. Every subpredicate is a constraint. At least one constraint is always present.

Every constraint is evaluated by a (domain specific) heuristic function which outputs an estimation of the expected computation time. Constraints which involve a (potentially) infinite computation (e.g. x:INTEGER) will not be used for domain restriction and are only used to check a solution in the checking phase in the last four lines of the algorithm after the variable domains have successfully been constrained. The estimation of the computation time is hard coded into PyB. For example a constraint like $x = 42$ is considered as fast while $x <: S$ (x subset of S) is considered slow if S is a large set (see example in the next subsection).

A second function analyses all involved constraints if these are implemented[6] cases which can be used by PyB in the second phase. Which variable domains can be restricted by the constraints and what other domains have to be known to use a (n-ary) constraint is saved. For example, the domain of a variable on the left side of a membership predicate ($x \in S$) or a single variable on either side of equation can be directly computed if no other bound variables are involved. This information is used to find unary constraints which can be used in less time to constrain the domain of a variable without the knowledge of other bound variables at the evaluation time. Binary and higher order constraints are used with much more effort.

In the last step an ordering of bound variables is computed. It is done by a topological sort using the information obtained as explained in this paragraph. After those steps, the constraint graph can be constructed and used in the domain reduction phase. A failure in this phase means that PyB was unable to find at least on contraint to compute a finite domain of at least one bound variable or was not able to compute a possible ordering. A correct enumeration cannot be done without at least one usable (implemented) constraint present

[6] not all constrains are implemented because of the big time effort need

```
1  // analysis
2  for every constraint:
3      estimate computation time
4      determine involved variables
5  compute variable ordering(constraints, variables)
6  compute constraint ordering(constraints, variables)
7
8  // domain reduction (solve)
9  for every variable:
10     domain = None
11     for every constraint of variable:
12         if domain is not None and constraint computation time
               ↪ is high:
13             break
14
15         if constraint is not unary:
16             label dependent domains
17         values = revise(constraint, variable)
18
19         if domain is None:
20             domain = values
21         else:
22             domain = domain intersection with values
23     if domain is empty set or None:
24         fail
25     save domain of variable
26
27 // labeling (checking)
28 for every variable:
29      for every value in domain:
30          if check predicate is True:
31              yield value combination
```

Figure 3.7.: Constraint solving algorithm (pseudocode)

for every variable in the domain reduction phase. If a variable cannot be constrained, PyB assumes its domain is infinite which means a full enumeration is impossible for PyB.

Domain reduction phase

The remaining lines (8 to 25) of code describe the domain reduction phase. Another possible name for this phase, would be the solving phase. The first line iterates over every node (bound variables) in the constraint graph. The inner loop iterates over every arc / edge adjacent to this node, which is every (usable) constraint that makes a statement about this variable. The inner loop distinguishes two cases: unary constraints and all other types of constraints. If the unary constraint can be used with low computation time, a constraint domain is computed. Otherwise, the domain of dependent variables is enumerated and the n-ary (higher order) constraint is used for all combinations (This is similar to labeling at the end). Because a variable ordering has been found in the analysis phase, the domain of the depended variables can always be enumerated in finite time. After every loop, the actual domain of the variable is computed by the intersection of the partial result computed by previous iterations and this iteration.

If the intersection results in an empty set, an inconsistency (e.g. $x = 1$ & $x = 2$) has been found and an exception is thrown. This exception will be handled by the caller and can have different effects: For example, in the case of nested quantified expressions, this can be caused by a wrong value of an outer bound variable, which may lead to a retry with a different value (e.g. $x = 1$ & $x = y$ with y:{1,2,3}) In the case of a set comprehension, this will result in an empty set expression.

If a constraint domain of a variable has been found, constraints with a long computation time (found by the heuristic) are skipped. This means a possible further domain restriction is not used. This is a trade-off between solving and checking computation time.

Labeling phase

In the last loop, a combination of all possible values is computed using the restricted domains and **all** constraints. The domains can still include some wrong results. If the constraint solving was successful, wrong results can now be removed very quickly. A combinatoric explosion is still possible but does usually not occur (see next chapter). As always, the tuples of values are returned lazily using Python's generators.

The labeling is a breadth first search [57]. No solution is generated twice because at every level of the search tree only one variable has been set to a value. For example, a solution to the predicate $x = 1$ & $y = x + 1$ can be found by setting x to a value first, and then computing y (x at the root of the search tree) or setting y first and then x.

Both would result in the solution {x=1,y=2}. The second case is avoided because of the fix variable ordering (in this case x before y).

Solving/Checking time trade-off

The analysis phase is linear in the number of constraints (constraint computation time estimation) and variables. The topological sorting can be quadratic. The real performance problem is caused by the labeling in the solving phase and the checking phase which scales with the number of combinations of the domain values of each variable to be computed and checked. If the domains are large, a timeout is likely. The algorithm must find a good balance between solving (domain restriction) and checking (labeling) computation time.

Concerning this trade-off: Using a constraint to restrict a domain can take different amounts of time. In practice, this matter is relevant. For example, consider the constraint $x : NAT \ \& \ x = 42$. The constraint $x : NAT$ will not be used in the solving phase if the number of elements of NAT is very large and a faster constraint has already been used. In this case, this would obviously an useless computation because x has already been constrained to 42. In other cases, this second computation can still be a good decision, for instance if the constraint domain of x would include more than one element and NAT is small. For example in the case $x : NAT \ \& \ x : \{-1, 0, 1\}$. A intersection of three elements and a small NAT set can be computed fast and really reduce the computation time in the labeling phase (for example a combinatoric blow up if other variables are involved).

On the one hand, allowing many wrong values to not to be detected until the last phase, can lead to a long computation for combinatoric reasons because the combination of all domain values of all bound variables is checked. On the other hand, performing an useless computation in the solving phase with little hope of discovering many new wrong values will also cause performance issues. If the analysis phase estimates the computation time of a constraint correctly as large (thresholds can be modified by the PyB user), and the solving phase uses this information to make the right decisions, this will cause a speedup.

Full example

Consider the example predicate $x = 42 \ \& \ x : INTEGER \ \& \ y : S \ \& \ y = x + 1$ with the bound variables x and y. S is some finite set (free variable). Figure 3.8 shows the graph. The nodes are the bound variables and the edges are the constraints. The heuristic will estimate the computation time (in the solving phase) of the four constraints as follows:

1. x=42 : fast
2. x:INTEGER : infinite/useless

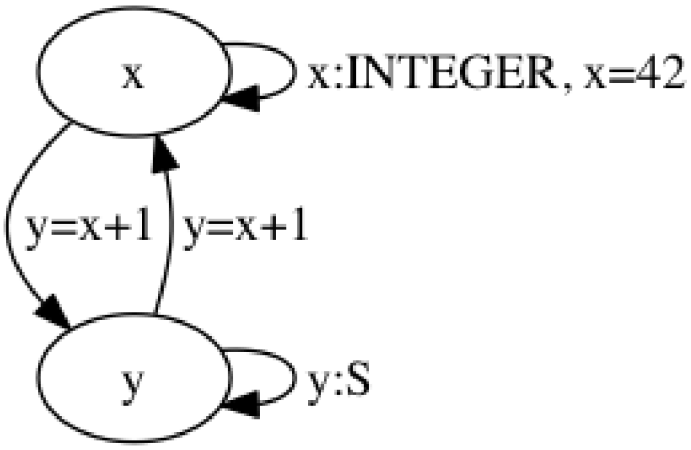


Figure 3.8.: Simple constraint graph of $x = 42$ & $x : INTEGER$ & $y : S$ & $y = x + 1$

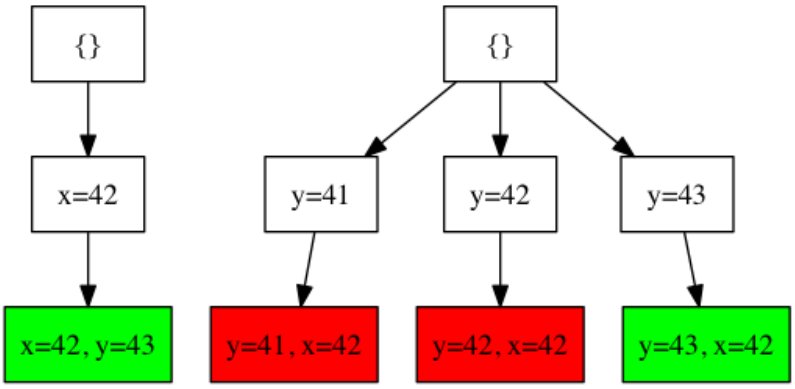


Figure 3.9.: Two search trees of $x = 42$ & $x : INTEGER$ & $y : S$ & $y = x + 1$ with $S = \{41, 42, 43\}$ and different variable ordering created at the checking (last labeling) phase

3. y:S : fast or slow (depending on the size of S)

4. y=x+1 : fast (if the domain of x is computed before y)

The constraint y=x+1 will cause the variable x to proceed y (example of variable ordering). Because a finite unary constraint can be found for every variable, both orderings (x,y or y,x) would be possible in this case, but x,y is preferable. While the height of the search tree is 2, the number of branches is infinite if no constraint is successfully used. An infinite amount of branches in the checking phase is impossible, because this would have already caused a failure in an earlier phase. If S is not infinite but very large (e.g S=NAT with MAXINT=2^{32}) setting y before x causes a failure in all but one case, while setting x first and then using the constraint $y = x + 1$ results in only one computation. Figure 3.9 shows the effect of variable ordering on the search trees for the case $S = \{41, 42, 43\}$.

PyB can **not** transform y=x+1 to x=y-1. Only when the computation of the domain of x was successful, can the binary constraint about y be propagated. This means that the variable ordering leads to a fail early behavior in the solving phase. After the constraint $x = 42$ is used, the constraint propagation causes a limitation of the domain of x to only one value (42) and the constraint $y = x + 1$ limits the domain of y in the same way after a labeling of x. After that, the checking phase has only to check one combination, which fulfills the whole predicate.

3.2.3. Limitations and Alternatives

Limitations

The implemented constraint solving algorithm can only use unary constraints to restrict domains. All constraints which involve more than one bound variable can only be solved via brute force. PyB cannot break down higher order constraints to binary constraints or solve binary constraints at all without any labeling. Therefore, arc- and path consistency are not checked without enumerating at least one bound variable. Table 3.3 shows some examples of unary constraints implemented by PyB. In general, every expression which only involves one bound variable can be used directly. PyB is not able to use any constraint which needs to be transformed into the supported form. Example are polynomial or exponential equations, inequation or indirect variable domain definitions like $f(x) \notin g(y)$, or mutual definitions like $x = y + 1 \wedge y = x - 1$. Implementing all cases is a big task, because B allows not only boolean and integer arithmetic but also sets of sets. Also infinite domains cannot be handled if they can not be reduced to finite ones using unary constraints. Even if there is a solution, it may not be found in reasonable time.

Boolean CSPs include as special cases some NP-complete problems, such as 3SAT. [...] In the worst case, therefore, we cannot expect to solve finite-domain CSPs in less than exponential time. [57]

Table 3.3.: Examples of unary constraints used by PyB

name	example
Match tuple of identifiers	$a \mapsto b \mapsto c = expression$
Membership of finite set	$x \in expression$
Equality and inequality	$x = expression$

If a bound variable is defined by a useless constraint like $x : INTEGER$ and other constraints which are not implemented in PyB, the computation can only be done if the B machine author rewrites its predicate by finding a finite domain for the variables and adding this information to the predicate, e.g $x : NAT$ (or what ever is correct in the concrete case). **Thus the constraint solver works if the domain of every bound variable can be reduced to a small finite domain using a unary constraint.** If PyB-compatible constraint cannot be found by a machine author then PyB can not check the predicate an will likely run into a timeout. This case is unlikely in data validation applications for the second tool chain. The comparison of infinite sets is not implemented except from a few special cases.

Alternatives

The enumeration is a problem where several implementation goals are in conflict (see 1.3.1). Goals that are in conflict are the goals of completeness of the tool, the implementation time constraints of the tool, and the goal of tool simplicity. In theory, the tool is also complete without a constraint solver. A brute force algorithm is able to compute all finite expressions. In practice, a tool which uses this approach will be useless because some simple and all complex expressions would cause a timeout and return a "dont know" result instead of a true or false value. Also, the constraint solver will only be useful if it can handle n-ary constraints more efficiently. This would be useful but would need months of effort without investigating the real questions and goals of the thesis. Also, this would be to unlikely lead to "obviously correct" code. Also, this simple constraint solver needs some explanation to be understood.

There are many constraint solving algorithm alternatives. One alternative is the AC-3 [47] implementation which is similar to the PyB constraint solving algorithm. It is used in simple constraint solvers and operates on finite domains using unary and binary constraints. Figure 3.10 shows the implementation introduced by A. K. Mackworth. The example of AC-3 is used to express the similarities and differences between PyB's constraint solving implementation and that of other simple algorithms.

1. infinite sets:
   AC-3 does not operate on infinite sets. Actually, it assumes a superset of the searched domain is present from the beginning. As a result, an analysis phase which drops infinite constrains is unnecessary. Also, a final check of all computed

```
1  begin
2      for i := 1 until n do NC(i);
3          Q := {(i.j)|(i,j)∈ arcs(G), i ≠ j}
4          while Q not empty do
5              begin
6                  select and delete any arc (k,m) from Q;
7                  if REVISE ((k,m)) then
8                      Q := Q ∪ {(i.k)|(i,k) ∈arcs(G), i ≠ k , i ≠ m}
9              end
10 end
11
12 procedure NC(i):
13 D_i:= D_i ∩ {x|P_i(x)}
14 begin
15     for i := 1 until n do NC(i)
16 end
17
18 procedure REVISE((i,j)):
19 begin
20     DELETE := false
21     for each x ∈ D_i do
22         if there is no y∈ D_jsuch that P_ij(x,y) then
23             begin
24                 delete x from D_i;
25                 DELETE := true
26             end;
27      return DELETE;
28 end
```

Figure 3.10.: AC-3 , NC and REVISE procedure [47]. An alternative to the PyB constraint solver

domains is unavoidable.

2. unary constraints and NC procedure:
   AC-3 handles unary constraints in a single pass to assure node consistency. PyB uses all kinds of constraints in one loop and decides by a heuristic which constraint should be used first. Not the arity of the constraint but its expected impact and computation time are relevant. This is because a unary constraint may not always be the best choice to constrain a domain, and an exact result is not a necessity because of the checking phase.

3. binary and n-ary constraints:
   AC-3 only uses binary constraints, while PyB can uses any constraint as long as all domains are known at the time revise is called, and the constraint is implemented.

4. revise procedure:
   AC-3 assumes that every binary constraint can be used. PyB cannot use every kind of constraint (not all types implemented)in its reduction phase. Also PyB searches a value $y \in D_j$ by enumerating D_j and trying every element, while AC-3 is less specific.

5. arc and path consistency:
   AC-3 assures consistency by using every constraint. This is done by visiting every edge (arc) of the constraint graph for all n nodes and revisiting all edges of a modified nodes (domain of the node) again to check effects on a third variable. Because PyB assures consistency by the checking phase at the end, a fixpoint loop over all edges (AC-1) or a revisit of other edges is not implemented. This results in a less exact solving result but spares implementation time.

6. predicate evaluation
   AC-3 assumes a finite domain is present from the beginning. Because of that it is possible to use every constraint by systematically generating and testing of every domain value. Only an interpreter is needed to do that. But if a domain is potentially infinite, a finite domain is not always present. Because of this, PyB needs much more implementation effort to evaluate a predicate and revise a domain.

Obviously writing and extending a constraint solver from scratch is not the best approach. It is a trade-off between invested implementation time and completeness. Verifying a simple constraint solver is possible [21]. A better solution would be to stop the development of the PyB constraint solver and instead writing a B interface which is able to compute B expressions by using an external (and possibly verified) constraint solver. Python bindings for some popular constraint solvers already exist[7]. Of course, some time would also need to be invested for the integration.

[7] https://github.com/pysmt/pysmt

3.3. Timeout Implementation

3.3.1. Motivation and Introduction

Some computations may take more time than a user is willing to accept. The cause of a long computation may be a missing optimization inside PyB (e.g. the constraint solver) or it is simply inherent to the problem for combinatoric reasons. This problem can be solved by adding a timeout feature to PyB. Every computation has a limited time to be completed which is defined by the user. After that, the tool will output a warning. So two threads are involved: a computation thread(worker) and a timer(master). The result of a computation with a timeout is neither true or false.

The implementation of a timeout is related to the data representation of PyBs output. In case of a timeout, users may be interested in similar information as with a failure or violation: In the case of data validation, the user is interested which parts of a safety property caused a timeout with which data. In the case of model checking, the user is also interested in which operation sequence lead to the timeout. A partial result/information will be useful in this case: showing which parts of a safety property are true, false or not computed because of a timeout.

In this context, the term "computation" needs to be defined. Implementing the timeout on the state level (computation = computing a full state) will not result in useful tool outputs. When model checking is performed only the information about which operation caused the timeout will be present. When performing data validation, no information at all will be present because there is only one state.

Implementing a timeout on the AST node level (computation = evaluation one node) may result in a complicated tool which creates and kills threads constantly. An implementation must therefore find some compromise between a simple maintainable implementation, useful precise data output, and acceptable performance.

It is also worth noting that Python implementations use a "global interpreter lock". The Python manual defines the *global interpreter lock* as: *The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. [...] Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.*[8] Although the timeout is implemented using concurrency, this is not done for performance reasons.

[8] https://docs.python.org/2/glossary.html#term-global-interpreter-lock

3.3.2. Implementation

The timeout was implemented with a simple master-worker pattern. The master thread starts a worker which computes a subpredicate and will be killed after the defined time has passed. The master accumulates the results and computes the overall result. Again this implementation is not about parallelism. The Workers are executed not concurrently but sequentially one after another. Only one worker exists at a time. Any parallelism is future work.

The implementation uses the Python multiprocessing module instead of the threading module. The multiprocessing module starts processes instead of threads. The threading module does not support killing a thread. It works on the invariant clause subpredicate level and the properties subpredicate level. Only the invariant level is explained, the properties clause level is implemented in the same way.

The safety properties introduced by an invariant clause are a conjunction of predicates which are broken into subpredicates. These subpredicates are evaluated separately in a thread. If one of them causes a timeout, the truth value of the whole invariant is unknown but the evaluation results of the other subpredicates can still be delivered to the user. Also, only two code positions were modified to create the thread implement the, timeout and termination of the thread. This is still easy and maintainable.

An inconsistent state is not possible:

1. PyBs evaluation function is pure. It has no side-effects, and only reads variables but does not write them. The only exception is the scope of variables (explained below).
2. The evaluation function: no threads are created, no unsharable recourses are used
3. Subpredicates can be evaluated separately. They do not affect each other.

The only implementation detail which can cause corruption is a killed thread which does not pop something from the value stack (see subsection about state representation in chapter 2). This may be caused by entering a new scope but not leaving it properly because of the kill. For example, this can happen if the computation was canceled during the evaluation of a quantified predicate. This can easily be handled by saving the stack height of each value stack of the current state and restoring the correct height. Cloning the environment is not necessary. It can simply be passed to the next worker.

Figure 3.11 shows the simplified version of the algorithm. Line 4 starts a process p by calling the interpret method (predicate evaluation) with a node n and the environment (the state) env. The result of the computation is placed inside a queue. If the queue is empty (line 11) the computation was canceled early and a timeout message is given to the

```
1  for n in lst:
2          import multiprocessing
3          que = multiprocessing.Queue()
4          p=multiprocessing.Process(target=lambda q,n,env:q.put(
               ↪ interpret(n,env)),args=(que, n, env))
5
6          # code to safe state/stack hight omitted
7          p.start()
8          p.join(PROPERTIES_TIMEOUT)
9          if not que.empty():
10                 value = que.get()
11         else:
12                 p.terminate()
13                 print "TIMEOUT: ("+pretty_print(n)+")"
14
15                 # code to restore state/stack hight omitted
16                 timeout = timeout +1
17                 continue
```

Figure 3.11.: PyB timeout implementation. One iteration of a evaluation loop

user. The value PROPERTIES_TIMEOUT (line 8) is a user defined value inside config.py

No timeout is added at the substitution level because this is not relevant in the case of data validation. The timeout feature is not implemented in the RPython version of PyB. Generally, multithreading and software transactional memory is supported by RPython [50][9]. Timeouts of five seconds shown to be a good choice in most case studies (see next chapter).

3.3.3. Limitations and Alternatives

In general killing a thread is bad programming style. If the function executed by a thread has any side-effects, a corrupted program state is possible. Thinking of all possibilities which can cause such a state is error prone and may result in source code which is simply not correct. The alternative implementation would be a thread which terminated itself, for example using notification mechanism or setting a stop flag.

In this particular case this implementation alternative would be the inferior choice because is has some disadvantages. The problem is checking the flag at the right position. To

[9] https://morepypy.blogspot.de/2015/03/pypy-stm-251-released.html

do this a tool implementer must find a good source code position to add this check. An implementor must assume which computation likely would consume much time, and if there is more than one, then any of the time consuming code candidates must be extended by a termination check. Doing that would violate the design goal to write a simple maintainable tool, because the check must be added at many source code positions and it would not be possible to be sure that all possible positions have been considered. Also the main argument against the killing solution (the corrupted state) can be invalidated by the fact that a corrupted state is not possible when dealing with simple evaluation because the interpret function (evaluates predicates and expressions) is without any side-effect and in the case of operation execution the state is already created by cloning an uncorrupted copy state. Corruption is therefore not a problem in either case.

3.4. Summary

This chapter described the implementation for some complicated aspects of B: infinite sets, enumeration using constraint solving and timeouts. PyB is limited by those aspects. Further implementation is in conflict with the simplicity goal of the tool.

4

Second Tool Chain Case Studies

This section evaluates the second tool chain approach on machines from industry instead of constructed artificial or text book examples. It is of particular interest what PyB's limitations are, i.e which expressions can not be computed. This chapter is not about performance, which will be discussed separately in chapter 6. This section is also not concerned with the discussion and conclusion of the thesis goals. That can be found in chapter 8.

4.1. B Case Studies

Some B machines are fully compatible with the implemented second tool chain, others are not. B expressions that can be easily checked and those which cannot are described. The details of the machines cannot be shown or discussed for reasons of confidentiality. However, the complexity and relevant aspects are presented. This chapter presents test results using industrial machines: i.e. machines which are used in real live and not only for academic purposes or teaching.

The input files were checked using the standard second tool chain approach from chapter 1(subsection)1.7). The main tool, ProB computes a state by setting machine constants to values which are constrained by predicates in the properties clause and computes the initialization clause or an operation. Those state values are written to a file. Every solution value is double checked by PyB by reevaluating the predicates inside the properties and invariant clause. PyBs output is a list in which the predicates from the properties and invariant clauses are True, False or causing a timeout.

4.1.1. Alstom Case Study

Alstom is a company in the railroad sector.[1][2]. The Alstom case study consists of 3 industrial size B machines. The following machines were double checked using PyB.

- Rule_DB_Route_0001ori:
  The first machine consists of 35 constants defined by properties about sequences and 30 variables. The invariant mostly consists of typing information expressed via membership expressions. The operations use lambda and other quantified expressions to assign values to variables. Model checking explores 2076 states. Some states need more than 2500 msec to be computed by ProB. The machine is model checked by ProB in approximately 2.5 seconds. Some states are deadlocks. PyB checks all B states in about 156 minutes[3]. No false state is found.
- Rule_DB_Route_0001ori_modified:
  The second machine is a modified version of the first and shows similar behavior when processed by ProB. PyBalso checks all B states in about 156 minutes. No false state is found.
- Rule_DB_SIGAREA_0024_ori:
  The third machine has only 36 states. The number of variables and constants is of the same order of magnitude as that of the other machines. Also, the operations construct sets via quantified expressions. PyBchecks all B states in about 3 minutes and 22 seconds. No false state is found.

The Alstom examples work very well with PyB. The invariants which are mostly typing predicates can be computed using symbolic sets. PyBs constraint solving can compute all quantified expression because of their finite domains. However, in some cases the performance was not very good.

4.1.2. Systerel Case Study

Systerel is a company *specialized in critical software and RAMS(Reliability /Availability /Maintainability /Safety)*[4]. The Systerel Case Study consists of 134 B machines. These B machines contain no invariants, but a CONSTANTS, PROPERTIES, and ASSERTIONS-clause. The value of these constants are defined by predicates in the PROPERTIES-clause. The number of constants range from 10 to 200. Some constants are defined by simple expressions, but most of them by quantified predicates. Without the features described in chapter 3, even simple expressions would cause a timeout. Other simple expressions

[1] Alstom describes itself as follows: *As a promoter of sustainable mobility, Alstom develops and markets systems, equipment and services for the railway sector.*

[2] http://www.alstom.com/about-us/

[3] PyB opens and parses a state file for every state. The double checking performance can be improved by using an other kind of ProB-PyB communication which is currently not implemented

[4] http://www.systerel.fr/en/company/

Figure 4.1.: Example of predicates successfully double checked by the ProB-PyB tool chain. [...] indicates omitted ProB solution code.

no.	Predicate	ProB Solution
1.	f=INTEGER * NATURAL1	INTEGER*NATURAL1
2.	f:INTEGER ↔ INTEGER	$\{(0 \mapsto 1), (1 \mapsto 2), [...], (253 \mapsto -1), (254 \mapsto -1)\}$
3.	$f = \%x.(x : INTEGER \mid \{FALSE \mapsto x, TRUE \mapsto -x\}(bool(x < 0)))$	$\%x.(x : INTEGER \mid \{FALSE \mapsto x, TRUE \mapsto -x\}(bool(x < 0)))$
4.	$f = \%(x,y).(x : STRING \& y : STRING \mid STRING\_APPEND(x,y))$	$\%(x,y).(x : STRING \& y : STRING \mid STRING\_APPEND(x,y))$
5.	$f=(bf/\backslash bg \sim \mid\mid ah \mapsto ah)(bf/\backslash bf \sim \mid\mid \{ah \mapsto ag\})(bg/\backslash bf \sim \mid\mid ag \mapsto ag)(bg/\backslash bg \sim \mid\mid \{ag \mapsto ah\})$	$\{((0 \mapsto 1) \mapsto (2 \mapsto 1)), ((1 \mapsto 0) \mapsto (3 \mapsto 0)), ((2 \mapsto 0) \mapsto (0 \mapsto 0)), [...]((233 \mapsto 1) \mapsto (227 \mapsto 1)), ((234 \mapsto 0) \mapsto (228 \mapsto 1))\}$

like $x \in NAT \leftrightarrow NAT$ need a symbolic representation to be computed. PyB operates in min and max integer ranges of -2^{32} *to* 2^{32}. These machines only generated one state. The case study is more concerned with data validation, than the other examples. Solution values (the constants) are checked against one (large) property.[5][6]

Table 4.1 shows five representative examples of predicates and ProB Solutions, which can be successfully checked by the ProB-PyB tool chain. Table 4.3, 4.3, 4.4 list the detailed results of every machine.

- 1. The first expression is the cartesian product of two infinite sets. The ProB solution file contains the same B expression as predicate solution. Because both sets are infinite, a explicit enumeration is not possible. PyB needs its symbolic features to check machines containing such expressions.
- 2. The second example also requires symbolic features to be evaluated. The predicate is a check if a set of tuples is a relation between INTEGER and INTEGER. Here the check must be possible without generating the (infinite) set of INTEGER relations.
- 3. The third example is an infinite set represented with a lambda expression. Members of this set are tuples like (1,1), (0,0) or (-1,1). The function calculates the absolute value of an integer. The domain of this function is infinite, so again a finite enumeration of tuples is not possible. PyB gets the same function as result

[5] A solution file can be computed by *probcli BFile.mch -init -p MAXINT 2147483648 -p MININT -2147483648 -p TIME_OUT 1000000 -sptxt Solution.txt* and checked by *python pyB.py -c BFile.mch Solution.txt*

[6] The ProB column can be reproduced by *probcli -init -p SYMBOLIC TRUE -p MAXINT 2147483648 -p MININT -2147483648 -p TIME_OUT 500000 BFile.mch*

Figure 4.2.: Systerel case study details

name	total	ok	failures	timeouts	comment	PyB time	ProB time
151_001.mch	41	41	0	0		11sec	0,220 sec
verdi1.mch					PyB Time-out	7m 53sec	0,350 sec
verdi2.mch					PyB Time-out	7m 50sec	0,350 sec
623_001.mch	52	50	0	2		131sec	2620 sec
m-PROP_REG_VM_04_002.mch	150	136	0	14		190sec	2,120 sec
m-PROP_SCL_VTT_0304_001.mch	109	109	0	0		7sec	0,140 sec
590_004.mch	63	60	0	3		83sec	2,650 sec
m-PROP_SCL_VTT_S_0316_001.mch	109	109	0	0		27sec	0,160 sec
machines2/0670_003.mch	34	30	0	4		345 sec	0,450 sec
machines2/0664_001.mch	75	72	0	3		385 sec	2,080 sec
machines2/0682_001.mch	20	15	0	5		460sec	0,190 sec
machines2/0682_002.mch	20	15	0	5		425sec	0,200 sec
CF_CV_1.mch	83	70	0	13		69sec	3,640 sec
590_004.mch	63	60	0	3		49sec	2,440 sec
670_005.mch	24	8	0	16		163sec	3,090 sec
670_006.mch					PyB Time-out	22m10sec	51,020 sec
590_005.mch	37	35	0	2		297sec	2,480 sec
CC_ZAUM_1.mch	179	168	0	11		71sec	87,150 sec
CF_ZSM_CBTC_7.mch	160	137	0	2		147sec	9,270 sec
CF_SEGMENT_9.mch	288	262	0	26		161sec	12,070 sec
590_007.mch	41	39	0	2		303sec	2,670 sec
CF_ZSM_SIG_3.mch	130	114	0	16		87sec	3,400 sec
CF_ZSM_SIG_4.mch	181	162	0	19		107sec	3,640 sec
CF_ZSM_SIG_6.mch	171	154	0	17		94sec	3,470 sec
CC_PLACE_MAINTENANCE_1.mch	207	188	0	19		126sec	104,290 sec
580_001_adapted.mch	72	71	0	1		20sec	0,370 sec
600_001.mch	60	58	0	2		306sec	2,540 sec
CF_ZAUM_12.mch	211	191	0	20		124sec	90,980 sec
062_101.mch	104	94	2	8		67sec	1,190 sec
CF_TVD_4.mch	334	311	0	23		153sec	6,850 sec
360_002.mch	72	70	0	2		18sec	0,250 sec

of the ProB computation and checks its equality using an AST comparison of these two expressions. It is a check using syntactically equality.

- 4. The fourth predicate is a call to an external function which concatenates strings. The external function was reimplemented in Python and is hooked into the call as described in chapter 2.

- 5. Finally, the fifth example is a set which can be represented with a finite enumeration of tuples. The set is generated by PyB and successfully compared to the ProB solution.

The case study consists of 134 machines. The machines consist of 20 to 512 subpredicates. 6 machines are correctly double checked in an average time of 16 seconds. A solutions is not found by ProB in the case of 21 machines in a time less than 60 seconds. 17 machines can not be checked. The remaining machines contain at least one subpredicate which cannot be checked by PyB in less than 2.5 seconds. This means that that there is room for improvement for PyB in this case study.

Figure 4.3.: Systerel case study details

name	total	ok	failures	timeouts	comment	PyB time	ProB time
049_001.mch					PyB Time-out	9m32.694	0,180 sec
CF_ZACP_8.mch	318	287	0	31		182sec	17,440 sec
CF_TVD_8.mch	117	105	0	12		60sec	3,140 sec
R_ZAUHT_2.mch	117	106	0	11		57sec	39,230 sec
680_001.mch	74	49	1	24		409sec	47,410 sec
CF_ZAUM_1.mch	181	166	0	15		95sec	88,030 sec
019_100_adapted.mch	56	52	0	4		42sec	0,200 sec
330_001.mch	27	26	0	1		84 sec	0,230 sec
610_001.mch	67	64	0	3		335sec	2,530 sec
612_001.mch	67	64	0	3		311sec	2,510 sec
614_001.mch	69	66	0	3		323sec	2,510 sec
651_001.mch	80	78	0	2		305sec	2,470 sec
652_001.mch	67	65	0	2		339sec	2,490 sec
CF_ZMS_AUM_2.mch	351	331	0	20		140sec	143,940 sec
CF_ZMS_AUM_3.mch	216	196	0	20		121sec	9,5610 sec
CF_ZMA_PRUD_1.mch	104	97	0	7		60sec	1,570 sec
CF_CBTC_TER_1.mch	177	166	0	11		48sec	1,160 sec
CF_ZMA_PRUD_7.mch	104	97	0	7		55sec	1,570 sec
019_100_corrected.mch	56	52	0	4		41sec	0,200 sec
CF_ZAUHT_2.mch	236	216	0	20		111sec	54,730 sec
005_100.mch					PyB Time-out	8m15	0,200 sec
CF_CBTC_TER_9.mch	334	310	0	24		146sec	4,930 sec
R_PLACE_MAINTENANCE_1.mch	408	388	0	20		233sec	345,840 sec
620_001.mch	37	13	0	24		-killed	2,370 sec
CF_LD_1.mch	130	118	0	12		71sec	3030 sec
R_PLACE_MAINTENANCE_2.mch	183	172	0	11		84sec	103,030 sec
CF_AIG_2.mch	368	352	0	16		132sec	4,040 sec
623_001.mch	52	50	0	2		364sec	2,530 sec
CF_CORR_LOC_1.mch	187	172	0	15		106sec	107,920 sec
CF_CORR_LOC_3-1.mch	351	323	0	28		216sec	108,790 sec
380_002.mch	66	64	0	2		23sec	0,260 sec
CF_CORR_LOC_2.mch	182	170	0	12		91sec	101,000 sec
CF_CORR_LOC_3-2.mch	292	271	0	21		193sec	112,650 sec
580_001.mch	72	71	0	1		32sec	0,360 sec
CF_ZGAR_2.mch	146	131	0	15		99sec	36,910 sec
CF_CORR_LOC_4.mch	198	181	0	17		144sec	106,620 sec
662_001.mch	65	62	0	3		370sec	2,500 sec
CF_CORR_LOC_5.mch	270	249	0	21		138sec	136,620 sec
664_001.mch	75	72	0	3		329sec	2,470 sec
CF_ZTR_2.mch	503	485	0	18		122sec	4,000 sec
CF_ZCH_1.mch	181	166	0	15		78sec	23,100 sec
CF_ZACQ_FU_MR_1.mch	196	185	0	11		66sec	7,500 sec
CF_ZCH_4.mch	211	188	0	23		115sec	22,920 sec
CC_ZSUIVI_2.mch	393	376	0	17		110sec	4,030 sec
CC_LD_1.mch	143	131	0	12		80sec	3,900 sec

Figure 4.4.: Systerel case study details

name	total	ok	failures	timeouts	comment	PyB time	ProB time
019_100.mch	56	52	0	4		42sec	0,200 ms
435_002.mch	118	113	0	5		39sec	0,360 ms
670_002.mch					PyB Time-out	2m46 sec	4,600 ms
CF_ZSM_CBTC_4.mch	93	78	0	15		104sec	6,440 ms
670_004.mch					PyB Time-out	4m46sec	55,630 ms
bug72ano.mch					PyB Time-out	4m35 sec	4,140 ms
0050_001.mch	70	64	0	6		58sec	0,290 ms
0670_004.mch					PyB Time-out	4m53 sec	195,590 ms
0590_004.mch	63	60	0	3		70sec	3,080 ms
0670_005.mch	24	8	0	16		179sec	2,360 ms
0670_006.mch					PyB Time-out	4m50 sec	213,160 ms
03_001.mch	61	53	0	8		59 sec	0,440 ms
0622_001.mch	55	53	0	2		74 sec	3,130 ms
SIM_11_001.mch	87	82	0	5		26 sec	0,280 ms
PL_01_001.mch	64	59	0	5		28 sec	0,290 ms
410_002.mch	58	57	0	1		21 sec	0,230 ms
861_001_corrected.mch	15	14	0	1		11sec	0,140 ms
Z_01_001_modified.mch	103	90	0	13		63sec	3,660 ms
Z_01_001.mch	96	84	0	12		115sec	0,650 ms
440_004.mch	78	77	0	1		25sec	-
435_002.mch	119	114	0	5		31sec	0,310 ms
440_006.mch	95	91	0	4		38sec	0,330 ms
PS_00611_006.mch	42	41	0	1		12 sec	0,150 ms
410_002_simple.mch	3	3	0	0		4 sec	0,050 ms
861_001.mch	15	14	0	1		13 sec	0,140 ms
670_004.mch					PyB Time-out	4m49 sec	157,890 ms
10_001.mch	85	74	0	11		40sec	0,280 ms
590_004.mch	63	60	0	3		14 sec	3,140 ms
02_001.mch	496	490	0	6		207sec	1,450 ms
670_006.mch					PyB Time-out	4m52 sec	140,780 ms
03_001.mch	61	53	0	8		55sec	0,450 ms
04_002.mch	152	138	0	14		322sec	2,290 ms
590_004_simplified	63	60	0	3			3,150 ms
I_01_001.mch	72	63	1	8		40 sec	0,520 ms
PB_00611_005.mch	62	61	0	1		12 sec	0,150 ms
0021_002.mch	44	44	0	0		17 sec	0,230 ms
R_02_002.mch	522	516	0	6		219 sec	1,590 ms
R_03_001.mch	75	67	0	8		195sec	0,360 ms
0050_001.mch	69	63	0	6		61sec	0,330 ms
R_04_001.mch	23	23	0	0		8sec	0,160 ms
R_07_001.mch	512	506	0	6		224sec	1,590 ms

Some machines only have 2 to 5 subpredicates which run into a timeout. Others have 20 or more. Examples of predicates in which a timeout occurs are predicates which contain expressions like closure or predicates which contain a lot of bound variables (up to 23). This causes timeouts for combinatoric reasons and because of the simple CSP implementation of PyB.

One assumption of a the second tool chain approach to this thesis is that it is more simple to check solutions than to compute one. One observation from this case study is, that a language like B is able to express predicates which are hard to check in reasonable time even if the solution is already present. This is not only the case if a solution is complicated but can also occur if the checking causes a (re-)computation of a set.

Checking if every element of a solution set can be generated by a predicate is easy, but it is not enough. The tool must (of course) compute the full set to exclude the possibility that elements are missing or more elements are present. This can be very time consuming using brute force enumeration. Implementing more advanced constraint solving features to avoid brute force enumeration is needed to check all examples. This is in conflict with the development goal to write an easy tool in reasonable time.

4.1.3. Volvo Case Study

The Volvo case study is a model of a cruise control system. It is a good example of a model which works very well with PyB. Double checking states with ProB and stand alone model checking without ProB was successful. Stand alone model checking is only relevant in the next chapters. The B machine uses non deterministic substitutions and initializations. The integer min. and max values range are -2^{31} and 2^{31} respectively, but the states of the model are very simple. The states are consist of variables of type boolean and integer, and two are of a custom set type.

PyB explored the same 1360 states as ProB in 1.7 seconds without any invariant violation (stand alone mode). When PyB is used as a second tool chain the checking of 1360 state files (generated by ProB) takes 32 minutes and no error was found.

4.2. Summary

This chapter examines the second tool chain approach by using PyB and the main tool ProB on industrial machines. PyB is less useful at “real life” examples if advanced constraint solving is need.

5

RPython - C Translation and JIT

This chapter describes the adaptations which have to be made on PyB to transform its Python code to RPython code and automatically add a just in time compiler (JIT) when translating to C[1]. PyPy, RPython and the RPython toolchain were introduced in subsection 1.2.4

5.1. RPython and translation to C

Programs which are written in RPyhon can be translated to C with the RPython toolchain. It is also possible to add a JIT into the translation process (see figure 5.1). The contribution of this chapter is the RPython-PyB code. The translation procedure was used and developed by others [19].

RPython is a statically typed subset of Python. PyB was originally written in Python. This section describes the differences between Python and RPython and shows adaptations which have made to PyB to enable the translation process. An unmodified branch of the interpreter was developed in parallel to the RPython version to compare the modified and unmodified version. This is found in chapter 6.

[1] run *PYTHONPATH=<PYPYDIR>:. python <PYPYDIR>/rpython/translator/goal/translate.py –batch pyB_RPython.py* to build the c executable

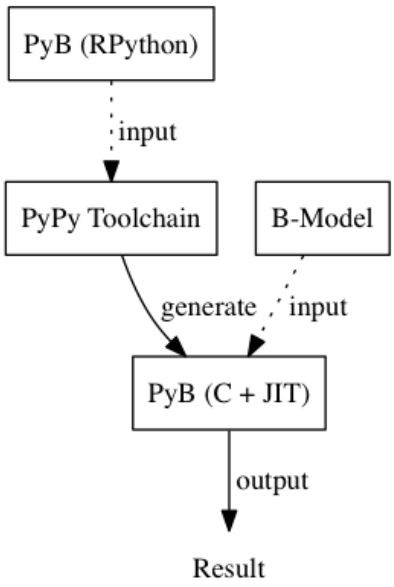


Figure 5.1.: PyB translation from RPython to C

5.1.1. Unsupported Python built-in types

To enable a translation of RPython to C using the translation tool chain, all built-in types used by a RPython application (like PyB) must be reimplemented. Most of this task has already been done by the PyPy developers. However there are some differences and exceptions:

- Python's big integers which are only limited by the platform's memory size are mapped to primitive integers.
- Booleans are represented with the values 0 and 1.
- Frozensets are not implemented and must be reimplemented by the interpreter implementor (myself).
- RPython does not support mixed type data structures like maps with different types of keys, mixed tuples or mixed lists which contain elements of different types. This is solved by using wrapper classes for list, map or tuple instances for every type.

5.1.2. Unsupported Python syntax

Some standard Python syntax is not supported by RPython. Normally these cases can be easily solved with reimplementation. Some examples which occurred in the RPython PyB-implementation:

- generator expressions
  - problem:
  RPython supports generators to some extent. Generator expressions like *frozenset((x,y) for x in S for y in T)* are not supported.
  - solution:
  generator expressions can easily implemented using two (or more) nested loops

- exponents of expression
  - problem:
  RPython does not support exponent expressions like $basis**exp$.
  - solution:
  a for-loop can be used for the reimplementation

- built-in string functions
  problem:
  string functions like replace() only work on characters, the empty string is not supported at all.
  solution:
  comparing each character one by one when dealing with replace(). Obviously the performance of this solution is not very good.

- None check
  problem: Tests like *X==None* or *dic=={}* are not implemented
  solution:
  those were replaced by *X is None* and *len(dic)==0*

- problem: Slicing is not fully implemented.
  For example in classical Python the expression L[1:5] donates a sublist from indexes 1 to 4. The expression L[-1] returns the last element
  solution:
  compute explicit indexes

5.1.3. Exec and meta programming at import time

The PyB Python version uses the exec command to generate AST nodes from the Java parser output. Exec executes a string if the string contains valid Python code. For example, *exec("id2=AAddExpression()")* will create an instance of an add AST (class must be known) node and assign it to id2. This dynamic feature is not supported by RPython. The AST generation was discussed in chapter 2. An output of the Java visitor can be seen in Figure 2.3.

The PyB RPython version implements all possible AST inputs of the exec function

```
1 def interp(node):
2     if node =="Int_Node":
3         return 42
4     else:
5         return  frozenset([1,2,3])
6
7 interp(''Int_Node")
8 interp(''Set_Node")
```

Figure 5.2.: Incompatible with RPython: A function without a static return type

with simple string parsing. All node names are known and node numbers can also be easily parsed from the string. This exec replacement is a big switch over all cases and is generated by meta programming at import time.

5.1.4. Object model modification

RPython code must be statically typeable: RPython does not support functions with different parameters or return types. Writing an interpreter method like the function shown in Figure 5.2 is not possible. Calling this function with different arguments would cause a RPython UnionError between the type frozenset and integer (last two lines) during the translation process. Using the same object model introduced in chapter 2 (for example primitives like 'True', '42' or built-ins like 'tuple') is not possible.

This problem can be solved by changing the object model in the interpreter: All possible types (parameters and return values) must be wrapped and inherit from the same base class. Figure 5.3 shows a wrapping example for booleans and integers. In this case, the function in Figure 5.2 would return a W_Object.

The object model of the RPython version of PyB contains 7 concrete classes: W_Tuple, W_Integer, W_Boolean, W_None, W_Set_Element, W_String and frozenset. All these classes and the SymbolicSet class (chapter 3) are subclasses of W_Object. Operations are implemented by overwriting special methods. Using these wrapped objects, a interpreter function can still return values of any (wrapped) type.

5.1.5. RPython generator implementation

A more subtle version of the typing problem from the last subsection occurs with the use of generators. Figure 5.4 shows a code snippet which is not RPython. It shows two (very simplified) versions of a symbolic NAT and Relation-set. Both sets can be lazily enumerated with their enumerate_set method. The caller of the method should

```
1  class W_Object:
2      pass
3
4  class W_Integer(W_Object):
5      def __init__(self, i):
6          self.ivalue = i
7
8  class W_Boolean(W_Object):
9      def __init__(self, b):
10         self.bvalue = b
11
12 class frozenset(W_Object):
13 [...]
```

Figure 5.3.: Wrapped object attributes

not need any knowledge about which set implementation he is calling. The return types are wrapped. All classes are a sub class of W_Object. The code can **not** be translated to C.

The program entry point is the main function which creates two instances and iterates over all elements in the symbolic set. This is enabled by the implementation of the __iter__ and next method which are indirectly called by the loop. Of course a real computation method would be more complicated. The translation fails with a UnionError in line 14. At this line two generator objects of different type are returned. This is not statically typed.

Figure 5.5 shows the solution to this problem. If an __iter__ method is added to every implementation and the next method contains explicit type information (by moving it up to the concrete sets), the UnionError cannot occur because the enumerate_set method then called is unambiguous. Figure 5.6 shows the output for the correct implementation (not all Nat values are shown).

5.1.6. Unit testing

PyB already has hundreds of tests (see chapter 7). However, it is also of some interest to see if the refactoring from Python to RPython or the translation to C introduces new bugs.

It is not possible to write good unit tests to check the translation process for small RPython code snippets. When more and more features are added to the RPython version of PyB, tests which passed at the beginning will fail later because the PyPy annotator is not able to infer all information from a small example. More informations mean more

```
1  MAX_INT = 2**32
2
3  class W_Object:
4  #  omitted
5
6  class W_Tuple(W_Object):
7  #  omitted
8
9  class W_Integer(W_Object):
10 #  omitted
11
12 class Set(W_Object):
13     def __iter__(self):
14         self.generator = self.enumerate_set()
15         return self
16
17     def next(self):
18         self.generator.next()
19
20 class NATSet(Set):
21      def enumerate_set(self):
22         for i in range(MAX_INT+1):
23             yield W_Integer(i)
24
25 class Relation(Set):
26     def enumerate_set(self):
27         S = [1,2,3]
28         T = [4,5,6]
29         for x in S:
30             for y in T:
31                 yield W_Tuple((x,y))
32
33 def compute(S):
34     for e in S:
35         print e.__repr__()
36
37 def main(argv):
38     S = NATSet()
39     compute(S)
40     S = Relation()
41     compute(S)
42     return 0
43
44 def target(*args):
45     return main, None # returns the entry point
```

Figure 5.4.: Not RPython: Can not be translated because of UnionError in line 14

```
1  # .... like before
2
3  class NATSet(Set):
4      def enumerate_set(self):
5          for i in range(MAX_INT+1):
6              yield W_Integer(i)
7
8      def __iter__(self):
9          self.generator = self.enumerate_set()
10         return self
11
12     def next(self):
13         return self.generator.next()
14
15 class Relation(Set):
16     def enumerate_set(self):
17         S = [1,2,3]
18         T = [4,5,6]
19         for x in S:
20             for y in T:
21                 yield W_Tuple((x,y))
22
23     def __iter__(self):
24         self.generator = self.enumerate_set()
25         return self
26
27     def next(self):
28         return self.generator.next()
29
30 def compute(S):
31  # .... like before
32
33 def main(argv):
34  # .... like before
35
36 def target(*args):
37  # .... like before
```

Figure 5.5.: Correct RPython generator implementation

```
0
1
2
3
[...]
4294967296
(1, 4)
(1, 5)
(1, 6)
(2, 4)
(2, 5)
(2, 6)
(3, 4)
(3, 5)
(3, 6)
```

Figure 5.6.: Output of generator implementation

possible conflicts.

For example PyPy (RPython toolchain) can not assure the existence of an object's field if this object is never created (or if its constructor has not been reached) by the PyPy analysis. Also, an argument of a function may only identified as not statically typed when the function is called a second time with an argument of a different type. The PyPy annotator must therefore see the whole picture. If a small test checks only one aspect of PyB like the environment value storage, AST parsing, set enumeration, etc.. then a failure may not indicate that this part of PyB was the reason for this failure.

Tests which only cover a limited part of the PyB-RPython code will not be reliable to test the translation process. Of course, it is possible to translate the whole tool and check it with complete machines after the translation has succeed. These test machines can contain every aspect of B which is implemented by the Python PyB version.

5.2. Adding a JIT

This section is not about the modifications needed to enable a PyB translation to C but rather those needed to add a JIT to the generated C code and improve its performance. The concept of a meta tracing jit was introduced in 1.2.4. A JIT can be added to a RPython interpreter when being translated to C by creating a JitDriver (a Python object) instance and annotating possible loops inside the RPython interpreter. The arguments of the JIT driver are green and red variables for the loop. Green variables are those

```
1 jitdriver = jit.JitDriver(greens=['inv', 'mch'], reds=['s_space
    ↪ ', 'env', ], get_printable_location=
    ↪ get_printable_location, name="_run_model_checking_mode")
2 def _run_model_checking_mode(env, mch):
3     inv = mch.aInvariantMachineClause
4     s_space = env.state_space
5     while not env.state_space.empty():
6         jitdriver.jit_merge_point(inv=inv, s_space=s_space, env
              ↪ =env, mch=mch)
7
8         # state checking code
```

Figure 5.7.: Jit merge point of the model checking loop

which do not change during the loop, for example an AST. Red variables are those which change on every loop iteration, for example the B state. The JitDriver annotation is not a contribution but only its usage by PyB

Figure 5.7 shows the most important jit merge point in PyB at the model checking loop. Relevant lines are line 1-3 and 8. The JIT driver is created and called inside the loop. Green variables are the invariant AST node and machine object. Red variables are the state space and environment. The function get_printable_location is used for debugging and will be used in a trace output. Other merge point are found in quantified expression like sum or inside the while loop substitution.

The normal work flow of jitting (annotation+translation) an RPython program is adding JIT merge points at potential loop bodies and examining the recorded traces. The traces are the main source of optimizations hints for the interpreter implementor and can be printed by PyPy after the C version of PyB has run in a loop for some time. A trace is a very largely piece of code and can be seen in appendix D.

5.2.1. Annotations

There are some Python decorators and hints which can improve the JIT results. Decorators are function annotations (with a @ symbol) which wrap the annotated function into another function. One main goal is to remove slow function calls inside the JIT. Some hints are not added with decorators but by adding attributes to classes. These decorators are a part of PyPy [15][2] and are used by PyB. Also it is important to keep in mind that the traces of the JIT will be optimized and translated to machine code to understand their purpose.

[2]see chapter 5

- _settled_:
  Attributes of an subclass are not copied to its base class. This hint avoids big base class objects. An example is the W_Object hierarchy from the last section.

- _immutable_fields_ and _attrs_:
  Attributes of an object which are constant can be marked as such. For example, the string of an identifier node is set only once at node creation.

- promote(x):
  Indicates that a variable may be constant most of the time. Notice: it must only be constant in the same trace position. For example the return value of the get children method of a AST node is variable but not at a specific code position because ASTs are immutable.

- @jit.elidable:
  A function is elidable if the arguments of the function never change at the trace position. That means that a trace always puts the same arguments at the same code position into that function. This does not indicate that these functions are constant or pure. Only the concrete trace position is relevant and not the function behavior over the whole program. An example is a dictionary lookup. The argument of a lookup function is variable, but not at a concrete trace position. @jit.elidable can be used for constant folding inside a trace.

- @jit.unroll_safe:
  A loop can be unrolled (replaced by a sequence of equivalent instructions) if the arguments of a loop (e.g. elements of a list) never change at this trace position. An example is a scoping lookup loop.

5.2.2. Fragmentation of eval and exec functions

A profiling of the PyB Python version revealed that most of the execution time (25% of the time) was consumed by isinstance (a Python built-in) checks in the evaluation of an execution switch. The switch tested which AST node has to be processed and called the corresponding code. This overhead could be removed by adding an exec (substitutions) and eval (expressions and predicates) method to every AST node using meta programming at import time instead of using a big switch. This resulted in a slight speedup (the interpretation overhead can not be avoided). The AST nodes are still immutable at runtime.

Another reason for the splitting of the substitution execution function into separate functions is to get better JIT traces when dealing with nondeterminism. If done correctly a speedup in the deterministic case (only one path of execution) and no effect in the nondeterministic case is achieved.

5.2.3. Removing dictionary lookups

Dictionary (hashmap) lookups inside a trace are one source of a bad JIT performance. These lookups can be replaced by array lookups. These lookups are faster and still constant (O(1)) if the index of the array is always known and no linear search is needed. This is done by adding an dictionary class implementation similar of that proposed by C. F. Bolz [15][3].
The implementation uses a map structure for every kind of map. The structure adds every key to a index dictionary which maps the key to an index inside the array. These contain the values. If the keys of the map are unlikely to change in a trace, this implementation leads to a better trace.

5.2.4. State hashing

The Python version of PyB was already able to hash B states to improve the model checking performance. States are transformed to a simple string representation and processed by a RPython hash function. Hash collisions are possible. States of the same hash are compared using an value by value comparison during model checking. This comparison is unlikely and costly. The hashing turned out to be one performance bottle-neck and was improved (e.g. by adding hash functions to W_Objects) which caused a speedup of some examples by one order of magnitude.

5.3. Summary

This chapter describes the differences between Python and RPython and which adaptations had to be made to PyB to generate a C-JIT version and improving its performance. Adaptations were wrapped types, built-in reimplementations, general code optimizations, state hashing and JIT annotations.

[3] figure chapter 14

6

RPython - Benchmarks

There are many reasons for insufficient tool performance. Inefficient algorithms, internal data structures or B data representation, the overhead introduced by the programming language (for example the CPython byte code interpreter) or the absence of features like constraint solving.

This chapter evaluates the performance of PyB. Three different versions of PyB are used to investigate the benefits (or drawbacks) of a C translation of PyB and the addition of a JIT. The versions are:

- PyB-Python:
  The version of PyB which was discussed in chapter 2 and 3. It is an unmodified non RPython version which is executed by the CPython interpreter.
- PyB-RPython:
  This is a branch of PyB which is modified respective to the last chapter. This version is also executed by the CPython interpreter. It is not translated to C.
- PyB-C-JIT:
  This is the translated C version of PyB. This version contains a JIT.

All tables in this chapter contain the performance results of these three versions. All benchmarks are compared to a reference B implementation: ProB.

The set of benchmarks also contains of small B machines which investigate the performance loses or benefits of one specific B feature. Machines from industry are used in the last subsection to investigate if performance boosts which can be observed on large machines

and to avoid a methodical error by only using artificial micro benchmarks. The number of states or iterations needed to start the JIT is stated in the subsections[1].

6.1. Benchmarks

A performance advantage of PyB-C-JIT is expected whenever the interpreter is entering a loop (meta or interpreter level). Therefore B machines which contain loops are subject of the benchmarks, because their evaluation cause a loop inside the PyB interpreter too. There are two kinds of loops:

- Meta loops
  which are loops of the interpreted B machine.
- Interpreter level loops:
  which are loops inside the RPython interpreter itself.

Interesting loops are: Quantified predicates[2], model checking loop, while substitution loops and constraint solving loops of quantified predicates. Different kind of operation also examined inside the loops: For example a loop containing integer arithmetic or set operations.

This section consider these examples because speedups and or performance losses (expected to be caused by constraint solving) may be most likely discovered by them. It is expected to get better JIT results if the number of states increase: the recording of traces, code generation and optimization takes some time. This time investment pays off when the much faster machine code is executed a large number of times. The time measurement accuracy can vary by one second. So results in this range are unreliable. The benchmarks were executed on a MacBookPro with 3 GHz dual Core processor and 8 GB main memory.

6.1.1. Metaloop: model checking (integer arithmetic)

Model checking can be seen as a meta loop, because the B model performs an operation execution loop: Possibly executing the same operation multiple times.

Table 6.1 shows the benchmark results of the model checking example shown in figure 6.1. The number of states was increased by setting the variable n. The JIT generates code when 633 or more states have been computed. PyB-C-JIT's performance is much

[1] execute sh run_benchmarks.sh to reproduce the results

[2] quantified predicates are implemented by a loop which checks every possible value of the bound variables

```
1 MACHINE Lift
2 CONCRETE_VARIABLES floor, n
3 INVARIANT n:NAT & floor : 0..n
4 INITIALISATION n:=99 ; floor := 4
5 OPERATIONS
6         inc = PRE floor<n THEN floor := floor + 1 END ;
7         dec = PRE floor>0 THEN floor := floor - 1 END
8 END
```

Figure 6.1.: A simple model checking example

Table 6.1.: model checking benchmark results of *MACHINE Lift*. Time in seconds

States	ProB	PyB-Python	PyB-RPython	PyB-C-JIT
10	1.9	0.8	1.5	0.7
100	1.9	0.9	1.7	0.7
633	1.9	1.2	2.5	0.7
1000	2.1	1.8	2.9	0.7
10000	3.7	7.5	14.9	0.8
100000	23.9	68.7	167.6	1.9
1000000	213.8	740.3	>18min	14.0

better than ProB's at this benchmark. Notice that the benchmark does not contain a complex invariant and only performs simple integer arithmetic. A JIT C implementation was expected to be faster than a Prolog implementation because C performs better at integer arithmetic than prolog in general.

```
1 MACHINE SetUnion
2 VARIABLES n, x, S
3 INVARIANT n:NATURAL & x:NATURAL & S<:NATURAL
4 INITIALISATION n:=1000; x:=0; S:={}
5 OPERATIONS
6 add = PRE x+1<n & x/:S THEN S:= S\/{x}; x:=x+1 END;
7 op = skip
8 END
```

Figure 6.2.: A set union loop

Table 6.2.: Union of sets. *MACHINE SetUnion*. Time in seconds

n	ProB	PyB-Python	PyB-RPython	PyB-C-JIT
500	2.1	1.2	10.1	0.8
1000	2.8	1.9	37.6	0.9
2000	5.5	3.6	144.8	2.0
4000	15.7	8.9	597.8	5.5
8000	54.6	26.7	>20Min	24.8
16000	221.4	110.2	-	162.4

Table 6.3.: while loop benchmark results. *MACHINE WhileLoop*. Time in seconds

States	ProB	PyB-Python	PyB-RPython	PyB-C-JIT
10000	2.2	6.6	4.3	0.6
100000	7.8	56.9	38.2	0.9
1000000	57.6	553.9	322.1	3.6
10000000	516.2	>10 min	>10 min	28.3

6.1.2. Metaloop: model checking (sets)

Table 6.2 shows the benchmark results of a slightly more complex example found in figure 6.2. Instead of increasing an integer variable floor, a set S is extended by an element x at every new state. PyB is also faster than ProB, but not by much. The RPython and PyB-C-JIT differences are a result of a more efficient hashing implementation which only works on the C level. This benchmarks shows that PyB's union operation and set representation does not as much benefit from translation and JIT than integer arithmetic. This is no surprise. A big surprise is the PyB-Python result, which is faster than the C version. This is caused by a less efficient reimplementation of the built-in frozenset type in PyB (chapter 5).

Table 6.4.: while loop benchmark results.*modified MACHINE WhileLoop*. **Loop inside operation.** Time in seconds

States	ProB	PyB-Python	PyB-RPython	PyB-C-JIT
10000	2.2	8.3	5.7	0.6
100000	7.0	76.9	41,4	1.0
1000000	61.1	750.4	451.7	4.0
10000000	532.0	-	>21 min	34.5

```
1  MACHINE WhileLoop
2  VARIABLES sum, i, n
3  INVARIANT
4     sum:NATURAL & i:NATURAL & n:NATURAL
5  INITIALISATION
6     BEGIN
7           BEGIN
8                 n := 10 ;
9                 sum := 0 ;
10                i := 0
11          END;
12          WHILE i<n DO
13                sum := sum + i;
14                      i := i+1
15          INVARIANT
16                i:NATURAL & sum:NATURAL & sum = ((i-1) * (i))/2
17          VARIANT
18                n - i
19          END
20    END
21
22 OPERATIONS
23    rr <-- op = rr:=sum /* avoid deadlock */
24 END
```

Figure 6.3.: A simple while substitution example

```
1 MACHINE SigmaLoop
2 VARIABLES n, sum
3 INVARIANT n:NAT & sum=((n+1+1) * (n+1))/2
4 INITIALISATION n:=10 ; sum:=(SIGMA i. (i:0..n | i+1))
5 OPERATIONS op = skip /* avoid deadlock */
6 END
```

Figure 6.4.: A simple sigma expression example

Table 6.5.: sigma loop benchmark results. *MACHINE SigmaLoop*. Time in seconds

n	ProB	PyB-Python	PyB-RPython	PyB-C-JIT
1000	1.7	1.7	1.6	0.7
1038	1.7	1.7	1.6	0.7
10000	1.8	4.0	2.9	0.7
100000	1.9	32.5	15.8	0.8
1000000	4.2	326.6	144.1	3.0
10000000	28.3	-	>13 min	25.5

6.1.3. Metaloop: While substitutions

While loops are not allowed in abstract B models. Anyway they are used in industry examples and they are a good way to check the performance of PyB-C-JIT. The benchmark code can be found in figure 6.3. Table 6.3 shows the results. Table 6.4 shows the results of a modified version of figure 6.3 where the loop has been moved from the initialization to the operation. The JITs starts at n=1039, but its performance impact compared to the no JIT C version is low. PyB-C-JIT handles long iterations much better than ProB. Its not important if the loop is inside an initialization or inside a B operation. The performance results are the same. Also this result can be explained by a better performance of machine code and C of integer arithmetic compared to that of Prolog.

6.1.4. Interpreter level loop: quantified predicates

Table 6.5 shows the benchmark results of a sigma expression(sum of integers) example shown in figure 6.4. The domain of the bound variable i can be used by PyB. The machine consists of only one state. The number of iterations was increased by setting the variable n. The JIT generates code when 1038 or more states have been computed. PyB-C-JIT and ProB show the same performance. The second Table 6.6 shows the results after line 4 was modified to SIGMA i. (i:0..n & $i > 0$ & $i < 2$| i+1) and sum=2 which can not be handled efficiently by PyB. The limitations of the PyB constraint solver and their performance implications have been introduced in chapter 3.

Table 6.6.: modified ($i > 0$ & $i < 2$) sigma loop benchmark results. Time in seconds

n	ProB	PyB-Python	PyB-RPython	PyB-C-JIT
1000	1.7	1.0	1.6	0.6
1038	1.7	1.1	1.6	0.6
10000	1.8	3.3	2.6	0.7
100000	1.8	26.7	13.9	0.8
1000000	1.8	262.3	-	3.0
10000000	1.9	-	-	26.1

Table 6.7.: B models from textbooks and industry. Time in seconds

Name	States	ProB	PyB-Python	PyB-RPython	PyB-C-JIT
Cruise_finite1.mch	1360	4.9	54.5	93.3	1.9
scheduler.mch	66	1.3	1.0	2.8	0.8
Doors.mch	8	1.8	0.8	1.5	0.6
Sets2.mch	1	1.8	0.8	1.9	0.6
spec.mch	1352	5.5	79.3	-	-

6.1.5. Machines from industry and publications

This subsection contains benchmarks not using artificial B machines, but examples from textbooks and industry. Table 6.7 shows the results. Table 6.8 and 6.9[3] show the results of a sorting benchmark.

Table 6.7 shows some B benchmarks also used by ProB. Most of these benchmarks do not run long enough to get an impression of PyB's performance in real world applications. The cruise control example turns out to be faster than PyB. The runtime is to small to evaluate if the model can benefit from a JIT. The sorting algorithm consists of nested loops which perform better an PyB-C-JIT for the same reasons as the while loop example.

[3]to reproduce the result: ./probcli sort_m2_data2000.mch -mc 1000000 -noass -noinv

Table 6.8.: Quadratic sorting algorithm. Time in seconds

n	states	ProB	PyB-Python	PyB-RPython	PyB-C-JIT
100	5050	7.2	33.9	1143.4	7.6
200	20100	27.5	225.8	-	75.9
500	125250	322.8	-	-	>11min
1000	500500	-	-	-	>31min

Table 6.9.: Quadratic sorting algorithm. Invariant check disabled. Time in seconds

n	states	ProB	PyB-Python	PyB-RPython	PyB-C-JIT
100	5050	5.3	11.0	80.7	1.5
200	20100	21.3	46.7	529.1	2.6
500	125250	263.5	331.8	-	17.6
1000	500500	1888.0	-	-	109.1
2000	2001000	-	-	-	813. 0

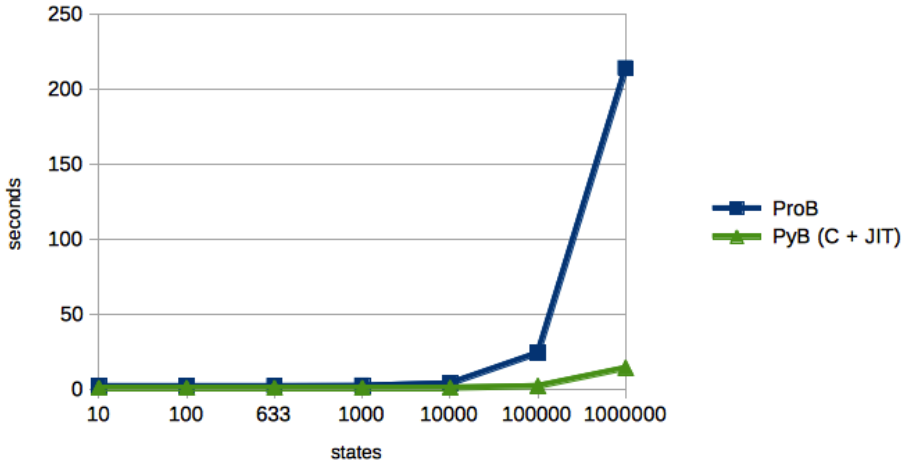


Figure 6.5.: Graph of table 6.1. No constraint solving. PyB shows good performance results

6.2. Summary

This chapter evaluates the performance results of the C-JIT version of PyB. PyB-C-JIT shows a better performance on arithmetic loops compared to ProB. This is a result of the C translation and the JIT. Its is slow on set operations because of the PyB data representation. PyB shows good results by one magnitude compared to ProB when ever constraint solving is **not** involved (see figure 6.5 and 6.6). A language like B can also benefit from the application of the PyPy translation tool chain and generation of a JIT.

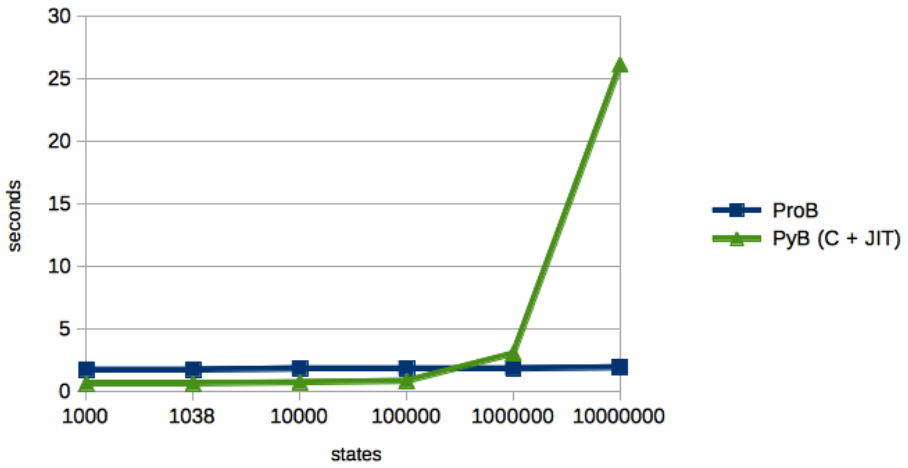


Figure 6.6.: Graph of table 6.6. Needs constraint solving. PyB shows bad performance results

7
Development Process

7.1. Timeline

PyB consists of 15510 lines of Python code, 16271 lines of test code and 1880 lines of Java code. Approximately 71 000 lines of code were committed and 39 000 lines of code where deleted in the four years of tool development. The first year (Aug. 2011-2012) was characterized by a rapid growth of tool features. The second year (Aug. 2012-2013) was used to include many B details and the first successful applications of the second tool chain. The third year (Aug. 2013-2014) was used to support more complex B features by implementing symbolic representation and naive constraint solving. The last year (Aug. 2014-2015) was used for the RPython adoptions and PyPy experiments. In retrospective much of the work of the third year was unnecessary to answer the questions of this thesis. Anyway, if the tool will be used after this thesis is published, this effort was not in vein. Tables 7.1 7.2 7.3 7.4 show the time line of the project in more detail. The tables do not cover 12 months, but logical project sets.

Table 7.1.: Project time line (first year): simple B implementation

Date	milestone
August 2011	project start
September 2011	evaluation of simple arithmetic, set-predicates, functions and relations
October 2011	type-checking of simple arithmetic, set-predicates, functions and relations
November 2011	type-checking with simple unifications and replaced interpreter state by an more complex environment
December 2011	typing and evaluation of more complex constructs. Added a simple (brute force) enumerator. First parsing of whole B machines
January 2012	first evaluation of simple B machine assertions
February 2012	implementation of more complex functions like closure and UNION. First evaluation of simple B machine PROPERTIES-, CONSTANT- and DEFINITION-clauses
March 2012	implementation of IF-THEN, CHOICE and SELECT-substitutions
April 2012	implementation of lookup of SEEN or INCLUDE B machines

Table 7.2.: Project time line (second year): advanced B features and completion of second tool chain prototype

Date	milestone
September 2012	implemented quick eval functions to speed up tool performance
October 2012	first successful usage of an extern constraint solver
November 2012	first introduction of state-space.
December 2012	successful animation of simple B machines.
January 2013	implementation of a small B-REPL
February 2013	successful usage of ProB solutions
March 2013	successful run of Alstom case-study
April 2013	complex animation-refactoring to enable nondeterministic substitutions
May 2013	animation of SEEN or INCLUDED B machines/operations
June 2013	documentation of tool-features and implementation details
July 2013	implementation of complex nondeterministic substitutions
August 2013	implementation of nondeterministic set up constants and init phase. Added pretty printer for predicates.
September 2013	typing and execution of external functions
October 2013	usage of more complex ProB solutions

Table 7.3.: Project time line (third year): difficult aspects, symbolic sets and constraints

Date	milestone
November 2013	added symbolic representation for large and infinite sets
December 2013	some Systerel (industrial B machines) successfully checked with PyB/ProB
April 2014	refactorings, tests and debugging. e.g. purity of interpreter and removal of dynamic code
May 2014	simple membership and equality constraints added e.g. $\{x\|x : NAT+ -> NAT \ \& \ x = \{(1,1),(2,2),(3,3)\}\}$ or $\{x\|x : NAT \ \& \ x : \{1,2,3\}\}$. First estimation of expression evaluation time.
June 2014	using more complex constraints like disjunctions, function domains. Added more complex symbolic sets e.g. symbolic relations, functions and lambda expressions
July 2014	operations on symbolic sets e.g. union. constraint usage of tuples. lazy enumerators of symbolic sets
August 2014	symbolic sequences and set comprehension. functions app. constraints e.g. $\{x, y\|x : S \& \ y = f(x)\}$
November 2014	more symbolic set implementations
December 2014 January 2015	timeout and detailed (sub-)predicate output

Table 7.4.: Project time line (forth year): RPython adaptions and translation to C

Date	milestone
February 2015	RPython: translation of AST nodes. Basic data type wrappers.
March 2015	RPython: Usage of parsing module
April 2015	Added model checking to PyB
May 2015	RPython: parsing of B machines e.g. Lift example
June 2015	RPython: arithmetic and simple predicate evaluation
July 2015	RPython: evaluation of simple B invariants
August 2015	RPython: execution of simple B machines e.g. Lift example
September 2015	RPython: relations, functions, sequences
October 2015	RPython: most B constructs supported

7.2. Clean room approach, workflow and Testing

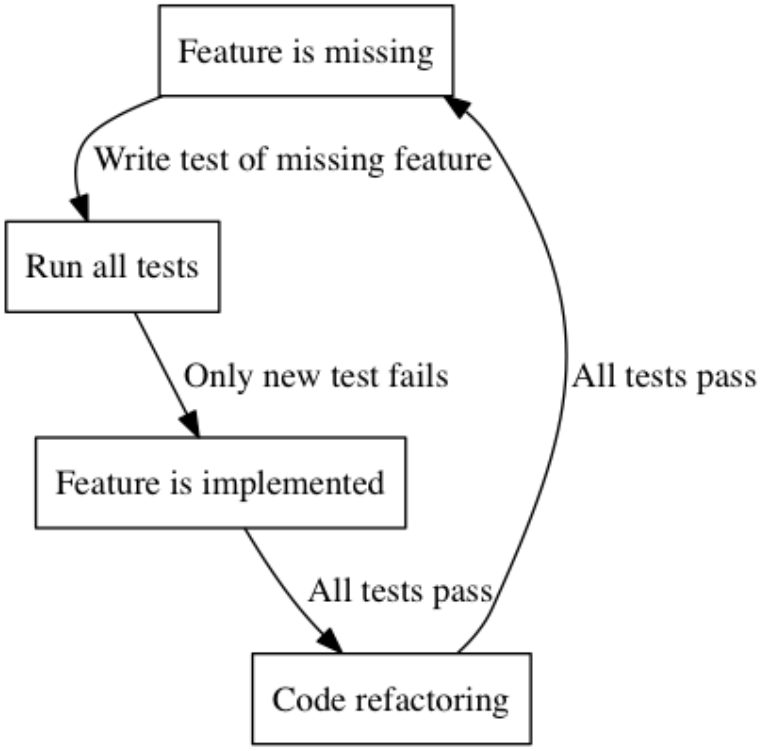
PyB is a clean room implementation, a technique originally used to avoid copyright issues. It was implemented without any knowledge or usage of the ProB code or the code of any other B implementation. A clean room implementation is a form of software design diversity [9]. In the special case of a ProB-PyB-chain it is a N-version implementation of B with N=2. The hope of a reliability increase by N-version programming is based on an independence assumption of the versions. This assumption was investigated by Knight at al [37]. The assumption holds not in all cases. Difficult implementation issues often lead to bugs in different versions. Anyway the authors conclude that N-version programming is still to be believed to lead to a greater reliability, but may not be as high like in theory.

Of course when a second tool has to check the computations of another tool, some design decisions of the main tool affect the second tool. For example the setup constants process when constants are searched which fulfill the machines property clause (which is a single step in ProB) or the typing of B expressions using unification. Also this clean room technique should not be mistaken with clean room software engineering [12]

PyB was developed using a test driven development (TDD) approach [11]. This is a software development processes which is often used along with dynamic languages like Python. The basic procedure can be seen in the flow graph of figure 7.1

- Missing feature:
  The first step is the identification of a missing tool feature. The correct specification of the desired behavior is be found in a B language manual. Finally a test is written to document the missing feature. In this step, the programmer is forced not only to think about the problem, but also to write code (the test) to specify what to be added to the tool.
- Testing
  All present test (not written in this cycle) are automatically run by a testing framework. Only the new test should not pass in this run. This assures that the new feature is not already included into the software. After this phase the new test should not be modified.
- Implementation
  In the last step, the feature is implemented, when the test passes.
- Refactoring
  The code is refactored to a more maintainable and efficient code base, without adding any new features. The test driven process starts again at the beginning. All tests are run together to assure the implementation does not introduces a bug which violates other tests.

Figure 7.1.: Test driven development in PyB



In contrast to a classical software development processes, the testing- and implementation phase are switched permanently and the tool implementor and tester are usually the same person. The advantage of this process may be a better tested tool. The drawback may be a longer development process. But both is not certain. Spending time in testing while development may shorten the debugging phase compared to a classical approach where testing is done after the implementation, so the overall time can be shorter.

PyB is covers by approximately 700 test cases. A few cases test helper functions or the cl interface, but most of them are white box unit test which cover only one aspect of B and do not only test the tool output of an evaluation but also partial results in the computation process. For example the validity of an expression, parsing, typing, the build up of frames, the state and other aspects are tested. Interpreter construction works well using TDD because an interpreter has a clear defined input/output behavior[1].

7.3. Summary

This chapter described the implementation approach and the time effort put into implementing the tool. It is a clean room implementation done in over four years.

[1]The tests can be run by typing *py.test* in the PyB home directory

8 Conclusion

8.1. Related Work

This section summarizes related work to PyB considering B implementations, second tool chains and B, and RPython interpreters.

The first subsection will answer the question if a B implementation is something unique and new contribution. The section compares how similar these tools are to PyB i.e how meaningful the comparison is. There are some implementations but ProB is the implementation most similar to PyB. None of them can be used as a complete alternative to the PyB implementation. Basically for two reasons:

- Not all of them implement the full B language. Also some features are not present. (e.g. substitution execution or model checking)
- None of them is implemented in the RPython subset, which makes RPython JIT experiments impossible.

PyB is **not** compared to proving tools like Click'n'Prove [3] or parts of Rodin, because this approach is too different from PyB. The ProB tool, which is important related work, was presented in chapter 1.1.2

The second subsection lists up second tool chains using B. It will give an overview if this approach is new to B tools. The section compares how the second tool chains have been used and what are the results of the usage. Are errors in the main tool detected?

The last subsection lists up RPython implementations. It will show if a RPython implementation and translation of some language are something new. The completeness of the implementation is examined and its JIT performance results. Are good results are only reached for a prototype or for a full language implementation like PyB does?

8.1.1. B Implementations

Most B Tools listed below have been developed in an academic context and have been replaced by other tools or the development has simply stopped. The most comparable tool is ProB. How issues of PyB have been solved by ProB is unknown to me because of the clean room implementation approach.

Atelier-B

Atelier-B [60][1] is a set of B and Event-B tools developed by the company ClearSy. It is one of the main B tools still in development and maintenance. It contains an automatic refinement tool (BART), syntax analysers (e.g. a type checker), proofing tools, automatic translators and a project management tool. It does not contain a model checker.

Older versions of Atelier B run out of memory in the context of data validation. *Atelier B quite often runs out of memory. For example, for the San Juan development, 80 properties (out of the 300) could not be checked by Atelier B.* [45]

B-Toolkit

The B-Toolkit was developed at B-Core by Ib Sorensen and David Neilson from 1992.. The B-Toolkit [8][2] was developed by B-Core UK. The development has stopped.*"The B Toolkit is a configuration tool that manages developments under the B Method, generating proof obligations and supporting tools for the discharge of those proof obligations. There is also support for the generation of documentation, and for the browsing of developments."* [55]

B4Free

B4Free were a set of B tools who are not in development anymore. The authors of the tool recommend to use AtelierB in the future.[3]. B4Free is described as *"Academic tool enabling the operational use of formal Method B for proven software development."* B4Free was used together with the emacs editor and contained proving tools and a project

[1] Atelier-B download: http://www.atelierb.eu/en/
[2] https://github.com/edwardcrichton/BToolkit
[3] http://www.b4free.com/index-en.php

managing tool[4] B4Free has been in uses for many years and is mentioned here in the sense of completeness.

Brilliant and UML-B

Brilliant [58] is an open B tool written in OCaml[5] . It consists of a type checker and can generate B POs. Also Brilliant can be used together with UML-B[6], a Tool to generate B specifications from UML [41].

B2RODIN + Rodin, AnimB

The Rodin platform [23] is an extensible eclipse based toolset, used to develop software using Event-B. Event-B is a successor of B. Rodin design goals were ease of use and extensibility. Rodin does not directly support classical B, but it is possible to translate some B models to event-B using the B2RODIN[7] tool. The number of machines which can be translated by B2RODIN are limited. For example the guards of a lift example using preconditions must be manually added after translation.

Event-B machines can be animated by AnimB[8], an animator written in Java which is based on the predicate evaluator PredicateB. AnimB[9] was considered to be used as second tool but not used because of some limitations: Parameter or constant values can not be found automatically and must be set by the user while variable values can be found at runtime. Also AnimB lacks of constraint solving features. Like PyB it can not handle complicated predicates efficiently and suffers from performance problems when dealing with large sets [44].

Also it is possible to generate code using Rodin plug-ins like EventB2Java[10], but code generation was not in scope of PyB.

Bcomp Atelier B Parser

Bcomp is the parser of Atelier B. It is written in C++[11] Eventually this parser will be used by PyB to make it a full clean room implementation. The main task will be a translation of the C AST to Python.

[4] http://www.b4free.com/tutoriels/sampleB4free.php
[5] Brilliant download: https://gna.org/projects/brillant/
[6] more information about UML-B: http://wiki.event-b.org/index.php/UML-B
[7] http://www.methode-b.com/en/tools/rodin/b2rodin/
[8] B2RODIN is only supported by Rodin 2.8 while AnimB needs 3.0 or better. This makes a comparison to PyB complicated.
[9] http://wiki.event-b.org/index.php/AnimB and http://www.animb.org
[10] http://poporo.uma.pt/EventB2Java/EventB2Java.html
[11] https://sourceforge.net/projects/bcomp/

JeB

JEB [70] [71][12][13] is a Event-B implementation in JavaScript. The name JEB is a short form for JavaScript simulation framework for Event-B. Important features [49] are predicate evaluation and the animation of nondeterministic operation on all abstraction levels and deadlock checks of models. JEB translates Event-B machines to a JavaScript representation (executable model) and simulates the machines for validation purposes. Quantified predicates are evaluated only in bounds of min and max values by the JavaScript library. Unary membership constraints are used to reduce the domain of bound variables.

JEB supports a graphical animation of B machines which can be compated to the flash animation of AnimB or Brama [59] or the graphical animation of BMotion Studio [38] (which is used by ProB). Also JEB can not be directly compared to PyB because PyB implements B instead of Event-B. JEB has been integrated into the Rodin platform.

8.1.2. Second Tool Chains and the B-Method

Redundancy i.e. double checking is no new concept in formal methods or computing [9]. It was used in the Ovado [39] [1] [40] project which performed formal data validation in the railroad sector using the B method. Ovado uses ProB as its main tool and Predicate B (later PredicateB++) as second tool. PredicateB++ it the counterpart to PyB in this second tool chain. PredicateB suffers from the same limitations of PyB of finding solution by itself. Also PyB and PredicateB are not implementing exactly the same features, because PyB also pursued other goals. The double chain ProB/PredicateB was successful used on large data in industrial size projects. The clean room approach and research results of N-version programming [37] were presented in 7.2.

8.1.3. RPython Interpreter Implementations

There are other RPython language implementations like Python (PyPy), Ruby, Smaltalk [20] and Prolog. A formal language like B was never implemented and examined in RPython. This subsection contains an overview about other RPython implementations, their development status and performance results, to set the RPython results of this thesis into relation: Most implementations are only reached a prototype development stage. Most of them showed good tracing JIT results when loops are involved. So PyBs JIT results meet the expectations by previous work.

[12] https://hal.inria.fr/tel-00951922/document

[13] JEB download: http://dedale.loria.fr/tiki-list_file_gallery.php?galleryId=13 or http://dedale.loria.fr/?q=en/JeB

PyPy RPython Python Implementation

The most important RPython example is PyPy [54], a Python implementation in RPython. The implementation contains a JIT and is faster than C Python on specific benchmarks [16].

Spy: RPython SmallTalk Implementation

Spy is a RPython Smalltalk [20] implementation. The implementation is not complete but was faster on some benchmarks compared to Slang.

Also an other small subset of Smalltalk, SOM [48]. was implemented in RPython by Marr et. al. to compare the performance, weaknesses, commonalities and potential improve of the meta tracing- and the ast self-optimization approach, concluding that both make "valuable contributions".

Topaz: RPython Ruby Implementation

Topaz[14] is an RPython Ruby implementation[15] created by Alex Gaynor. Ruby is a dynamic imperative language. Topaz does not fully support the full Ruby language. For example the standard library is missing and threading is not supported [5]. *At present, Topaz is not considered stable or tested in the real world, and is extremely incomplete. We do not yet consider Topaz to be production ready (but it gets closer every day!). All that said, if Topaz does run your program correctly, it is likely to continue to work.* The Topaz results have not been published.

PyJS: RPython JavaScript Implementation

PyJs[16] ia JavaScript implementation which was written by Stephan Zalewski [72] in 2012. Java Script [66] is a popular object-oriented dynamic language used in web development. PyJS does not fully implement the ECMA specification, for example regular expressions are missing. The performance of PyJS was bad except on loop examples. In most cases the reference implementation spidermonkey were faster to a factor of 100 or more.

[14]Topaz download: https://github.com/topazproject/topaz
[15]http://topazruby.com
[16]PyJS download: https://bitbucket.org/pypy/lang-js

RPython Java Implementation

A JVM Python[17] implementation was written by myself in 2009 [68]. It contained a parser, bytecode interpreter, JNI-Interface, threading and parts of the java standard library. It was not fully RPython compatible and no JIT was added successfully. Compared to state of the art Java implementations, no speedup was created.

Rapydo: RPython R Implementation

Rapydo is a RPython implementation of R which was written by Sven Hager [33][18] in 2012. R is a language used for statistical data analysis [36] developed by Ross Ihaka and Robert Gentleman. The implementation only reached prototype status. A speedup up to the factor 57 compared to the reference implementation was reached on loop execution. Benchmarks without loops produced worse results.

Pyrolog: RPython Prolog Implementation

Pyrolog[19] is a Prolog implementation written by C.F. Bolz [14] [18]. Prolog is a declarative logical programming language. A Jit was sucessfully added by the PyPy JIT generator. On some benchmarks (arithmetic) the implementation achieved a speedup up to the factor 10 compared to state of the art Sicstus Prolog.

PyHaskell: RPython Haskell Implementation

PyHaskell [63] is a RPython Haskell implementation[20] written by E. Thomassen in 2013. Haskell is a general purpose, purely functional programming language [35]. PyHaskell is only a prototype. A JIT speedup was successfully created, but Py Haskell does not beat the reference implementation GHC. Anyway the author concludes that *the RPython translation tool chain is suitable for purely functional, lazy languages.*

Pixie: RPython Lisp Implementation

Pixie[21] is a RPython Lisp implementation done by Timothy Baldridge in 2015. The implementation is *in a "pre-alpha" phase* . There are no published articles about pixie. The author describes the performance as *"good-enough"*[22]

[17]JVM donwload: https://github.com/hhu-stups/python_jvm
[18]Rapydo download: https://bitbucket.org/cfbolz/rapydo/
[19]Pyrolog download: https://bitbucket.org/cfbolz/pyrolog/
[20]PyHaskell download: https://bitbucket.org/cfbolz/haskell-python/
[21]Pixie download: https://github.com/pixie-lang/pixie
[22]http://www.thestrangeloop.com/2015/pixie—a-lightweight-lisp-with-magical-powers.html

Pycket: RPython Racket (Scheme) Implementation

Pycket [10][23] is a RPython implementation of Racket. Racket is a Lisp dialect [30], which is a functional programming language. The implementation was advanced: *Pycket supports a wide variety of the sophisticated features in Racket such as contracts, continuations, classes, structures, dynamic binding, and more.* Pycket was faster on average compared to the Racket reference implementation.

Pydgin: RPython ISS

Pydgin [46] is a fast instruction set simulator[24] written in RPython. *An instruction set simulator (ISS) is a special kind of functional-level model that simulates the behavior of a processor or system-on-chip (SOC)*[25]. The implementation turned out to be fast.

PyGirl: RPython hardeware emulation

An other RPython implementation of a hardware simulator is PyGirl[26].Pygirl [22] is a whole system virtual machine (WSVM) written by Verwaest Toon in 2009. The performance was comparable to an other java vm implementation.

8.2. Future Work

While the goals of this thesis have been reached, there is still interesting work to be done in the future:

One possibility would be the extension of PyB by and external constraint solver or theorem prover: Then an efficient check of quantified safety properties would also be possible. This would extend PyBs set of compatible machines. The second tool chain can then be used on more models from industry and it would be possible to investigate if a formal language implementation can also gain a speedup from the translation process if more complex examples are processed.

8.3. Conclusion

This thesis examined and discussed two different issues, which both needed an implementation of B. This implementation is PyB, the main contribution of this thesis. The

[23]Pycket download: https://github.com/samth/pycket
[24]Pydgin download: https://github.com/cornell-brg/pydgin
[25]http://morepypy.blogspot.de/2015/03/pydgin-using-rpython-to-generate-fast.html
[26]PyGirl download: https://github.com/camillobruni/pygirl

first issue was the implementation of a second tool chain. The second topic was the application of the RPython tool chain on a B implementation. Both topics pursued different goals. While correctness of the implementation was an issue of both topics, simplicity was important for the second tool chain, while performance was vital to the RPython goal.

8.3.1. Second Tool Chain

PyB, the second tool was implemented and successfully used with the main tool, ProB. Except the parser, it is a clean room implementation written in Python. PyB implements all important B constructs[27]. It handles infinite sets, nondeterministic substitutions, simple constraints and performs model animation, single state checking and model checking. The implementation time was four years. While simple machines without quantified predicates perform very good, constraint solving and infinite sets have to emerge as the bottleneck of this simple tool: In this case PyB has also to find solutions itself instead of just checking solutions of the main tool. By computing constraint sets, the second tool becomes equally complex as the main tool. If quantified predicates are extended by finite domain informations, the tool may still handle complex B machines in reasonable time. The tool was used on textbook examples and on machines from industry. Bugs inside the main tool where not discovered.

8.3.2. RPython Translation and JIT

The RPython tool chain was successfully used on dynamic language implementations in the past. The contribution of this thesis was its usage on B, which is different from dynamic languages. Even if Python lacks some useful build-in Prolog features like nondeterminism and unification, Python was shown to be a good language to implement B. The one exception is the absence of constraint solving features.

The C translation of PyB was successfully after some RPython adaptions. Performance improvements where accomplished after JIT specific refactorings of the tool. The tool is faster than ProB by a factor of 10 on artificially constructed arithmetic examples and faster on some industry machines by the factor of 5. In general a speedup can be expected form every type of loop which contains only arithmetic operations. Machines using constraint solving are still computed slower then by ProB. This is not a result of the approach but of the limited constraint solving capabilities of PyB. The bottom line is, a language like B can profit from a Python implementation, C translation and just in time compilation.

[27] B Trees are not implemented, but usually not used in industry

A

Individual Contribution to Articles

Author : John Witulski and Michael Leuschel Booktitle: Proceedings of the 1st Workshop on Formal-IDE Year 2014 Series EPTCS 149, 2014 Publisher: Electronic Proceedings in Theoretical Computer Science Parts of chapter one and chapter two are based on a workshop paper "Checking Computations of Formal Method Tools - A Secondary Toolchain for ProB" written by John Witulski and Michael Leuschel.

B

Implemented B Syntax and Constructs

B.1. Implemented B Syntax and Constructs

This appendix summarizes the B syntax mainly to address readers who are not familiar with B. The implementation is explained in more detail in chapter 2 and 3. Its only intended to give a short overview about B machines and the capabilities of PyB. A full B specification can be found in the reference manual[60].

- Components: Clauses.
  - The machine HEADER: defining the machine name and (non-empty) set- and scalar-parameters
  - The CONSTRAINTS clause: A predicate constraining parameters. Scalar parameters are typed by this clause. The type of set parameters can not be modified.
  - The SETS clause: A list of sets used by this machine. There are two types of sets: Sets can be explicit by enumerating every element or be deffered by just giving them a name. Be convention, set-identifiers are written in capital letters.
  - The CONSTANTS clause: A list of machine constants.
  - The PROPERTIES clause: A predicate constraining the machines sets and constants.
  - The VARIABLES clause: A list of machine variables.

  - The INVARIANT clause: A predicate constraining the machines variables, sets and constants.

- Simple expressions.
  - Set enumeration, set of tuples and the empty set.
  - Set comprehensions: sets defined by a predicate.
  - Predefined sets: natural/integer numbers and their subsets from a min_int up to a max_int value. boolean and stringsets.
  - Common set operations: union, intersection, difference, cartesian product, powerset and cardinality
  - Integer expressions: addition, subtraction, multiplication, division, modulo, minimum and maximum.
  - Special cases of unions and intersections, defined on sets of sets or by a predicate and an expression.
  - Set summation and product, defined on sets of integers.
- Predicates
  - Set predicates: membership, subset, superset, equality
  - Predicate operations: conjunction, disjunction, implication, equivalence, negation, equality and inequality
  - quantified predicates: universal- and existential quantified predicates.
  - number predicates: greater, less, equality and inequality
  - boolean operator. Returing the value True or False of a predicate.
- Relations.
  - Relation expressions. Returning a set of tuples, with defined domain and image.
  - Relation operations: dom and range expressions. Returing domain set or image set.
  - composition expression concatenating two relations.
  - Id expression creating the identical relation of a set: domain and image set are equal.
  - domain and range restrictions (or subtraction) returning a a subset of the relation which only use the subset of the domain set or image set.
  - Inverse relation: switching image and domain.

  - Overwriting: change mapping of some tuple elements, defined by an other relation
  - Relational Image: returning the set of elements mapped from a specified domain set.
  - direct and parallel product.
  - first and second projection.
  - iteration: n-th composition of the relation with itself
  - closure: (possible infinite) composition of the relation and with itself until a fixpoint is reached.
- Functions.
  - a special case of relations. constraint by the properties of totality, injectivity, surjectivity or a combination of these.
  - function application. Like relational image, but only defined for elements instead of sets.
  - lambda functions. A function defined by a predicate and an expression.
- Sequences.
  - a special case of functions: Ordered set of tuples mapping natural numbers (except zero) to a set (like arrays)
  - sequence construction: returning the set of all finite sequences mapping to a specified set. Also the set of injective, bijective and non empty sequences is possible.
  - : sequence operations: concatenation, element append and prepend, sequence size, the reverse sequence, element drop and take, last, tail, front and first operations returning specified sequence parts and generalized concatenation defined on sequences of sequences.
  - Strings: sequences on the set of possible characters.
- Substitutions. Substitutions are abstract statements and assignments. A list can be found in table 2.18.

Some clauses are optional. Some constructs are only allowed in abstract B machines. Others are only allowed in concrete implementations. Also the reference suggests implementation conventions. For example that a PROPERTIES clause is follow by a conjunct of predicates (page 140 [60]).

Figure B.1 shows a list of PyB's implemented features. Not all features have been implemented in the C version. The yellow partial entry means that the implementation does not cover all possible cases but the most. The entry naive usually means that a

complex feature is not implemented using symbolic representations but by a brute force enumeration loop. All features can be implemented in the C version. A partial, naive or missing implementation is just caused by limited time.

	PyB-C	PyB-Python
Predicate Unit	partial	yes
Expression Unit	no	partial
contraint clause	yes	yes
properties clause	no	yes
invariant clause	yes	yes
assertion clause	no	yes
conjunct predicate	yes	yes
disjunct predicate	yes	yes
impl. Predicate	yes	yes
equivalence predicate	yes	yes
negation predicate	yes	yes
for all predicate	naive	using constraints
exists predicate	naive	using constraints
equal predicate	yes	yes
not equal predicate	yes	yes
set extention	yes	yes
empty set	yes	yes
comprehension set	partial**	using constraints
intersection	symbolic	symbolic
union	symbolic	symbolic
couple	naive	naive
powerset	symbolic	symbolic
powerset1	symbolic	symbolic
card expression	yes	yes
general union	naive	naive
general intersection	naive	naive
qu. intersection	partial	symbolic
qu. union	partial	symbolic
member predicate	partial	yes
not member predicate	partial	yes
subset predicate	partial	yes
not subset predicate	partial	yes
strict subset	partial	yes
not strict subset	partial	yes

	PyB-C	PyB-Python
NAT	symbolic	symbolic
NAT1	symbolic	symbolic
INT	symbolic	symbolic
NATURAL	symbolic	symbolic
NATUARAL1	symbolic	symbolic
INTEGER	symbolic	symbolic
min	yes	yes
max	yes	yes
add	yes	yes
Minus / set sub	yes	yes
Mult / cart	yes	yes
div	yes	yes
modulo	yes	yes
power of	yes	yes
interval	symbolic	symbolic
gen. Sum	naive	using constraints
gen prod	naive	using constraints
greater	yes	yes
less	yes	yes
greater equal	yes	yes
less equal	yes	yes
relation	symbolic	symbolic
domain	partial	yes
range	partial	yes
composition	partial	yes
identity expression	symbolic	symbolic
domain restriction	partial	partial
domain subtraction	partial	partial
range restriction	partial	partial
range subtraction	partial	partial
invserse	symbolic	symbolic
image	partial	yes
overwrite	partial	yes
direct prod	partial	yes
parallel prod	partial	yes
iteration	naive	naive
closure	naive	naive
ref. Closure	naive	naive
first proj	partial	yes
second proj	partial	yes

	PyB-C	PyB-Python
part function	symbolic	symbolic
total function	symbolic	symbolic
part. Inj function	symbolic	symbolic
total inj. Function	symbolic	symbolic
part. Surj. Function	symbolic	symbolic
total surj. Function	symbolic	symbolic
total bij. Function	symbolic	symbolic
part. Bij. Function	no	symbolic
lambda expression	partial**	symbolic
function expression	yes	yes
empty sequ	yes	yes
seq expression	symbolic	symbolic
Seq 1 expression	symbolic	symbolic
iseq	symbolic	symbolic
iseq1	symbolic	symbolic
perm	symbolic	symbolic
concat	yes	yes
insert	yes	yes
Instert tail	yes	yes
seq seq expression	yes	yes
size	yes	yes
rev	yes	yes
restrict	yes	yes
restrict tail	yes	yes
first	yes	yes
last	yes	yes
tail	yes	yes
front	yes	yes
gen conc	no	yes

	PyB-C	PyB-Python
string	yes	yes
bool	yes	yes
uni minus	yes	yes
int	yes	yes
min_int	yes	yes
max_int	yes	yes
id	yes	yes
prim id	no	partial
bool set	yes	yes
true	yes	yes
false	yes	yes
struct	yes	yes
rec	yes	yes
record	yes	yes
string set	yes	yes
trans	yes	yes
func	yes	yes
external func	no	yes
skip	yes	yes
:=	partial	yes
:{P}	yes	yes
::	yes	yes
S1 || S2	yes	yes
BEGIN s END	yes	yes
S1 ; S2	yes	yes
WHILE	yes	yes
PRE	yes	yes
ASSERT	yes	yes
IF	yes	yes
CHOICE	yes	yes
SELECT	yes	yes
CASE	yes	yes
VAR	yes	yes
ANY	yes	yes
OP	yes	yes

Figure B.1.: Implemented B features of PyB-Python and PyB-C-JIT

C

PyB Short User Manual

C.1. Build PyB

You need Python 2.x, PyPy (source checkout), Py.test, Java, SableCC, gcc. Tested on MacOs and Ubuntu Linux.

C.2. PyB Short User Manual

This section is a short summery how to uses PyBs features. It can be used to reproduce the results of this thesis. It can be downloaded at https://github.com/hhu-stups/pyB
PyBs features are:

1. animation of a B machine (python PyB.py MCH):

2. double checking of a state (python PyB.py -c MCH STATE):

3. stand alone model checking of a machine (python PyB.py -mc MCH):

4. interactive eval mode (python PyB.py -repl):

PyB can be configured by changing the flags in config.py

C.3. Other tools

A ProB solution file can be generated by executing **probcli MCH -sptxt Solution.txt** The tool can be translated using pypy by executing **python pypy/rpython/translator/goal/translate.py pyB_RPython.py** The PYTHONPATH enivorment variable has to be set to the pypy path and the pyB_RPython.py path.

D

JIT Trace Example

Figure D.2, D.3, D.4, D.5, D.6 and D.7 shows a trace of a while loop substitution during model checking (100000 interations). It can be seen in Figure D.1. The output is generated by **PYPYLOG=jit-log-opt:- ./RPython_pyB-c -mc** $< Machine >$. The purpose of this section is to give the reader a better impression of what is a trace. The trace consists of a long list of assignments and guard instructions. For example guard_nonnull_class, guard_no_exception, guard_false and guard_true. Guard instructions are used to check if a trace condition holds or the trace has to be left. For example last iteration of the loop.

```
1  MACHINE WhileLoop
2  VARIABLES sum, i, n
3  INVARIANT
4    sum:NATURAL & i:NATURAL & n:NATURAL & i:{0,n}
5  INITIALISATION
6    BEGIN
7          BEGIN
8               n := 100000 ;
9               sum := 0 ;
10              i := 0
11         END;
12         WHILE i<n DO
13              sum := sum + i;
14              i := i+1
15         INVARIANT
16              i:NATURAL & sum:NATURAL & sum = ((i-1) * (i))/2
17         VARIANT
18              n - i
19         END /* ;i:=-1 */
20     END
21
22 OPERATIONS
23     rr <-- op = rr:=sum /* avoid deadlock */
24 END
```

Figure D.1.: B machine containing a while loop

```
[947c566951d] {jit-log-opt-loop
# Loop 0 ((jitdriver: no get_printable_location)) : loop with 282 ops
[p0, p1, p2, p3, p4, p5, p6, p7, p8]
+121: label(p0, p1, p2, p3, p4, p5, p6, p7, p8, descr=TargetToken(4310106432))
debug_merge_point(0, 0, '(jitdriver:␣no␣get_printable_location)')
+128: i9 = getfield_gc(p2, descr=<FieldS list.length 8>)
+139: i11 = int_sub(i9, 1)
+143: p12 = getfield_gc(p2, descr=<FieldP list.items 16>)
+147: p13 = getarrayitem_gc(p12, i11, descr=<ArrayP 8>)
+152: setarrayitem_gc(p12, i11, ConstPtr(null), descr=<ArrayP 8>)
+161: i15 = arraylen_gc(p12, descr=<ArrayP 8>)
+165: i17 = int_rshift(i15, 1)
+168: i19 = int_sub(i17, 5)
+172: i20 = int_lt(i11, i19)
+183: cond_call(i20, ConstClass(
    ↪ _ll_list_resize_hint_really_look_inside_iff__listPtr_Signed_Bool), p2, i11, 0,
    ↪ descr=<Callv 0 rii EF=5>)
+247: guard_no_exception(descr=<Guard0x100e82758>) [p8, p7, p13, p3, p2, p1, p0, i11]
+267: p23 = getfield_gc(p0, descr=<FieldP environment.Environment.inst_state_space
    ↪ 128>)
+281: p24 = getfield_gc(p23, descr=<FieldP statespace.StateSpace.inst_stack 16>)
+285: setfield_gc(p2, i11, descr=<FieldS list.length 8>)
+289: i25 = getfield_gc(p24, descr=<FieldS list.length 8>)
+293: i27 = int_sub(i25, 1)
+297: p28 = getfield_gc(p24, descr=<FieldP list.items 16>)
+301: p29 = getarrayitem_gc(p28, i27, descr=<ArrayP 8>)
+301: setarrayitem_gc(p28, i27, ConstPtr(null), descr=<ArrayP 8>)
+310: i31 = arraylen_gc(p28, descr=<ArrayP 8>)
+314: i33 = int_rshift(i31, 1)
+317: i35 = int_sub(i33, 5)
+321: i36 = int_lt(i27, i35)
+332: cond_call(i36, ConstClass(
    ↪ _ll_list_resize_hint_really_look_inside_iff__listPtr_Signed_Bool), p24, i27, 0,
    ↪ descr=<Callv 0 rii EF=5>)
+401: guard_no_exception(descr=<Guard0x100e83ca8>) [p8, p7, p13, p3, p2, p1, p0, i27,
    ↪ p24]
+421: setfield_gc(p24, i27, descr=<FieldS list.length 8>)
+425: p40 = call(ConstClass(BState.clone), p13, descr=<Callr 8 r EF=5>)
+520: guard_no_exception(descr=<Guard0x100e83bf8>) [p23, p40, p8, p7, p13, p3, p2, p1,
    ↪ p0]
+540: i41 = getfield_gc(p24, descr=<FieldS list.length 8>)
+551: i43 = int_add(i41, 1)
+555: p44 = getfield_gc(p24, descr=<FieldP list.items 16>)
+559: i45 = arraylen_gc(p44, descr=<ArrayP 8>)
+563: i46 = int_lt(i45, i43)
+574: cond_call(i46, ConstClass(
    ↪ _ll_list_resize_hint_really_look_inside_iff__listPtr_Signed_Bool), p24, i43, 1,
    ↪ descr=<Callv 0 rii EF=5>)
+639: guard_no_exception(descr=<Guard0x100e83ba0>) [i41, p24, p40, p8, p7, p13, p3, p2,
    ↪  p1, p0, i43]
+659: p49 = getfield_gc(p24, descr=<FieldP list.items 16>)
+702: setarrayitem_gc(p49, i41, p40, descr=<ArrayP 8>)
+707: p50 = getarrayitem_gc(p49, i41, descr=<ArrayP 8>)
+712: p51 = getfield_gc(p0, descr=<FieldP environment.Environment.inst_root_mch 96>)
+723: setfield_gc(p24, i43, descr=<FieldS list.length 8>)
+727: guard_nonnull_class(p51, ConstClass(BMachine), descr=<Guard0x100e83af0>) [p8, p7,
    ↪  p13, p3, p2, p1, p0, p51, p50]
```

Figure D.2.: Rpython trace of a B While loop

```
+746: p53 = getfield_gc(p50, descr=<FieldP bstate.BState.inst_stack_list 40>)
+750: i54 = getfield_gc(p51, descr=<FieldS bmachine.BMachine.inst_index 208>)
+757: p55 = getfield_gc(p53, descr=<FieldP list.items 16>)
+761: p56 = getarrayitem_gc(p55, i54, descr=<ArrayP 8>)
+766: i57 = getfield_gc(p56, descr=<FieldS list.length 8>)
+770: i59 = int_sub(i57, 1)
+774: i61 = int_le(i59, -1)
guard_false(i61, descr=<Guard0x100e83a98>) [p8, p7, p13, p3, p2, p1, p0, p51, p50, i59,
    ↪ p56]
+784: i63 = int_add(i59, -1)
+788: p64 = getfield_gc(p56, descr=<FieldP list.items 16>)
+792: p65 = getarrayitem_gc(p64, i59, descr=<ArrayP 8>)
+797: p66 = getfield_gc(p65, descr=<FieldP bstate.RPythonMap.inst_structure 16>)
+801: guard_value(p66, ConstPtr(ptr67), descr=<Guard0x100e83a40>) [p8, p7, p13, p3, p2,
    ↪ p1, p0, p51, p50, i63, p56, p66, p65]
+820: p68 = getfield_gc(p65, descr=<FieldP bstate.RPythonMap.inst_storage 8>)
+824: p70 = getarrayitem_gc(p68, 2, descr=<ArrayP 8>)
+828: p71 = getarrayitem_gc(p49, i41, descr=<ArrayP 8>)
+833: p72 = getfield_gc(p71, descr=<FieldP bstate.BState.inst_stack_list 40>)
+837: p73 = getfield_gc(p72, descr=<FieldP list.items 16>)
+841: p74 = getarrayitem_gc(p73, i54, descr=<ArrayP 8>)
+846: i75 = getfield_gc(p74, descr=<FieldS list.length 8>)
+850: i77 = int_sub(i75, 1)
+854: i79 = int_le(i77, -1)
guard_false(i79, descr=<Guard0x100e839e8>) [p8, p7, p13, p3, p2, p1, p0, p70, p51, p71,
    ↪ i77, p74]
+864: i81 = int_add(i77, -1)
+868: p82 = getfield_gc(p74, descr=<FieldP list.items 16>)
+872: p83 = getarrayitem_gc(p82, i77, descr=<ArrayP 8>)
+877: p84 = getfield_gc(p83, descr=<FieldP bstate.RPythonMap.inst_structure 16>)
+881: guard_value(p84, ConstPtr(ptr85), descr=<Guard0x100e83990>) [p8, p7, p13, p3, p2,
    ↪ p1, p0, p70, p51, p71, i81, p74, p84, p83]
+900: p86 = getfield_gc(p83, descr=<FieldP bstate.RPythonMap.inst_storage 8>)
+904: p88 = getarrayitem_gc(p86, 1, descr=<ArrayP 8>)
+908: guard_nonnull_class(p70, ConstClass(W_Integer), descr=<Guard0x100e838e0>) [p8, p7
    ↪ , p13, p3, p2, p1, p0, p88, p70]
+928: guard_nonnull_class(p88, ConstClass(W_Integer), descr=<Guard0x100e83830>) [p8, p7
    ↪ , p13, p3, p2, p1, p0, p88, p70]
+947: i91 = getfield_gc(p70, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue 24>)
+951: i92 = getfield_gc(p88, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue 24>)
+955: i93 = int_sub(i91, i92)
+958: p94 = getarrayitem_gc(p49, i41, descr=<ArrayP 8>)
+963: p95 = getfield_gc(p94, descr=<FieldP bstate.BState.inst_stack_list 40>)
+967: p96 = getfield_gc(p95, descr=<FieldP list.items 16>)
+971: p97 = getarrayitem_gc(p96, i54, descr=<ArrayP 8>)
+976: i98 = getfield_gc(p97, descr=<FieldS list.length 8>)
+980: i100 = int_sub(i98, 1)
+984: i102 = int_le(i100, -1)
guard_false(i102, descr=<Guard0x100e837d8>) [p8, p7, p13, p3, p2, p1, p0, p51, p94,
    ↪ i100, p97, i93]
+994: i104 = int_add(i100, -1)
+998: p105 = getfield_gc(p97, descr=<FieldP list.items 16>)
+1002: p106 = getarrayitem_gc(p105, i100, descr=<ArrayP 8>)
+1007: p107 = getfield_gc(p106, descr=<FieldP bstate.RPythonMap.inst_structure 16>)
+1012: guard_value(p107, ConstPtr(ptr108), descr=<Guard0x100e83780>) [p8, p7, p13, p3,
    ↪ p2, p1, p0, p51, p94, i104, p97, p107, p106, i93]
+1031: p109 = getfield_gc(p106, descr=<FieldP bstate.RPythonMap.inst_storage 8>)
+1036: p111 = getarrayitem_gc(p109, 1, descr=<ArrayP 8>)
+1040: i112 = getfield_gc(p111, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue
    ↪ 24>)
```

Figure D.3.: Rpython trace of a B While loop

```
+1045: i114 = int_ge(i112, 0)
+1057: guard_true(i114, descr=<Guard0x100e836d0>) [p8, p7, p13, p3, p2, p1, p0, i114,
    ↪ i93]
+1066: p115 = getarrayitem_gc(p49, i41, descr=<ArrayP 8>)
+1071: p116 = getfield_gc(p115, descr=<FieldP bstate.BState.inst_stack_list 40>)
+1076: p117 = getfield_gc(p116, descr=<FieldP list.items 16>)
+1080: p118 = getarrayitem_gc(p117, i54, descr=<ArrayP 8>)
+1085: i119 = getfield_gc(p118, descr=<FieldS list.length 8>)
+1089: i121 = int_sub(i119, 1)
+1093: i123 = int_le(i121, -1)
guard_false(i123, descr=<Guard0x100e83678>) [p8, p7, p13, p3, p2, p1, p0, p51, p115,
    ↪ i121, p118, None, i93]
+1103: i126 = int_add(i121, -1)
+1107: p127 = getfield_gc(p118, descr=<FieldP list.items 16>)
+1111: p128 = getarrayitem_gc(p127, i121, descr=<ArrayP 8>)
+1116: p129 = getfield_gc(p128, descr=<FieldP bstate.RPythonMap.inst_structure 16>)
+1120: guard_value(p129, ConstPtr(ptr130), descr=<Guard0x100e83620>) [p8, p7, p13, p3,
    ↪ p2, p1, p0, p51, p115, i126, p118, p129, p128, None, i93]
+1139: p131 = getfield_gc(p128, descr=<FieldP bstate.RPythonMap.inst_storage 8>)
+1143: p133 = getarrayitem_gc(p131, 0, descr=<ArrayP 8>)
+1147: i134 = getfield_gc(p133, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue
    ↪ 24>)
+1151: i136 = int_ge(i134, 0)
+1163: guard_true(i136, descr=<Guard0x100e835c8>) [p8, p7, p13, p3, p2, p1, p0, i136,
    ↪ None, i93]
+1172: p137 = getarrayitem_gc(p49, i41, descr=<ArrayP 8>)
+1177: p138 = getfield_gc(p137, descr=<FieldP bstate.BState.inst_stack_list 40>)
+1181: p139 = getfield_gc(p138, descr=<FieldP list.items 16>)
+1185: p140 = getarrayitem_gc(p139, i54, descr=<ArrayP 8>)
+1190: i141 = getfield_gc(p140, descr=<FieldS list.length 8>)
+1194: i143 = int_sub(i141, 1)
+1198: i145 = int_le(i143, -1)
guard_false(i145, descr=<Guard0x100e83570>) [p8, p7, p13, p3, p2, p1, p0, p51, p137,
    ↪ i143, p140, None, None, i93]
+1208: i147 = int_add(i143, -1)
+1212: p148 = getfield_gc(p140, descr=<FieldP list.items 16>)
+1216: p149 = getarrayitem_gc(p148, i143, descr=<ArrayP 8>)
+1221: p150 = getfield_gc(p149, descr=<FieldP bstate.RPythonMap.inst_structure 16>)
+1225: guard_value(p150, ConstPtr(ptr151), descr=<Guard0x100e83518>) [p8, p7, p13, p3,
    ↪ p2, p1, p0, p51, p137, i147, p140, p150, p149, None, None, i93]
+1244: p152 = getfield_gc(p149, descr=<FieldP bstate.RPythonMap.inst_storage 8>)
+1248: p154 = getarrayitem_gc(p152, 0, descr=<ArrayP 8>)
+1252: p157 = call_may_force(4295756688, ConstPtr(ptr156), p0, descr=<Callr 8 rr EF=7>)
guard_not_forced(descr=<Guard0x100e98320>) [p8, p7, p13, p3, p2, p1, p0, p154, p157,
    ↪ None, None, i93]
+1378: guard_no_exception(descr=<Guard0x100e834c0>) [p8, p7, p13, p3, p2, p1, p0, p154,
    ↪  p157, None, None, i93]
+1398: i158 = getfield_gc(p157, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue
    ↪ 24>)
+1402: i160 = int_floordiv(i158, 2)
+1422: i162 = int_lshift(i160, 1)
+1428: i163 = int_sub(i158, i162)
+1438: i165 = int_rshift(i163, 63)
+1442: i166 = int_add(i160, i165)
+1445: guard_class(p154, ConstClass(W_Integer), descr=<Guard0x100e83468>) [p8, p7, p13,
    ↪  p3, p2, p1, p0, p154, i166, None, None, i93]
+1464: i168 = getfield_gc(p154, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue
    ↪ 24>)
+1468: i169 = int_eq(i168, i166)
+1479: guard_true(i169, descr=<Guard0x100e83410>) [p8, p7, p13, p3, p2, p1, p0, i169,
    ↪ None, None, None, i93]
p171 = new_with_vtable(4299529960)
+1567: setfield_gc(p171, ConstPtr(ptr172), descr=<FieldP rpython.flowspace.generator.
    ↪ Entry.inst_g_self 16>)
```

Figure D.4.: Rpython trace of a B While loop

```
+1581: setfield_gc(p171, p0, descr=<FieldP rpython.flowspace.generator.Entry.inst_g_env
    ↪ 8>)
+1592: p174 = call_may_force(ConstClass(ASequenceSubstitution.
    ↪ exec_ASequenceSubstitution__next), p171, descr=<Callr 8 r EF=7>)
guard_not_forced(descr=<Guard0x100e982c0>) [p8, p7, p13, p3, p2, p1, p0, p174, i93]
+1691: guard_no_exception(descr=<Guard0x100e83308>) [p8, p7, p13, p3, p2, p1, p0, p174,
    ↪ i93]
+1711: p175 = getfield_gc_pure(p174, descr=<FieldP tuple2.item0 8>)
+1715: i176 = getfield_gc_pure(p174, descr=<FieldU tuple2.item1 16>)
+1720: guard_true(i176, descr=<Guard0x100e83258>) [p8, p7, p13, p3, p2, p1, p0, p175,
    ↪ i93]
+1729: p177 = getfield_gc(p0, descr=<FieldP environment.Environment.inst_state_space
    ↪ 128>)
+1743: p178 = getfield_gc(p177, descr=<FieldP statespace.StateSpace.inst_stack 16>)
+1747: i179 = getfield_gc(p178, descr=<FieldS list.length 8>)
+1751: i181 = int_add(i179, -1)
+1755: p182 = getfield_gc(p178, descr=<FieldP list.items 16>)
+1759: p183 = getarrayitem_gc(p182, i181, descr=<ArrayP 8>)
+1764: p184 = getfield_gc(p0, descr=<FieldP environment.Environment.inst_root_mch 96>)
+1768: guard_nonnull_class(p184, ConstClass(BMachine), descr=<Guard0x100e83200>) [p13,
    ↪ p2, p1, p0, p184, p183, p175, i93]
+1787: p186 = getfield_gc(p183, descr=<FieldP bstate.BState.inst_stack_list 40>)
+1791: i187 = getfield_gc(p184, descr=<FieldS bmachine.BMachine.inst_index 208>)
+1798: p188 = getfield_gc(p186, descr=<FieldP list.items 16>)
+1802: p189 = getarrayitem_gc(p188, i187, descr=<ArrayP 8>)
+1807: i190 = getfield_gc(p189, descr=<FieldS list.length 8>)
+1811: i192 = int_sub(i190, 1)
+1816: i194 = int_le(i192, -1)
guard_false(i194, descr=<Guard0x100e831a8>) [p13, p2, p1, p0, p184, p183, i192, p189,
    ↪ p175, i93]
+1826: i196 = int_add(i192, -1)
+1830: p197 = getfield_gc(p189, descr=<FieldP list.items 16>)
+1834: p198 = getarrayitem_gc(p197, i192, descr=<ArrayP 8>)
+1839: p199 = getfield_gc(p198, descr=<FieldP bstate.RPythonMap.inst_structure 16>)
+1843: guard_value(p199, ConstPtr(ptr200), descr=<Guard0x100e83150>) [p13, p2, p1, p0,
    ↪ p184, p183, i196, p189, p199, p198, p175, i93]
+1862: p201 = getfield_gc(p198, descr=<FieldP bstate.RPythonMap.inst_storage 8>)
+1866: p203 = getarrayitem_gc(p201, 2, descr=<ArrayP 8>)
+1870: p204 = getarrayitem_gc(p182, i181, descr=<ArrayP 8>)
+1875: p205 = getfield_gc(p204, descr=<FieldP bstate.BState.inst_stack_list 40>)
+1879: p206 = getfield_gc(p205, descr=<FieldP list.items 16>)
+1883: p207 = getarrayitem_gc(p206, i187, descr=<ArrayP 8>)
+1888: i208 = getfield_gc(p207, descr=<FieldS list.length 8>)
+1892: i210 = int_sub(i208, 1)
+1897: i212 = int_le(i210, -1)
guard_false(i212, descr=<Guard0x100e830f8>) [p13, p2, p1, p0, p203, p184, p204, i210,
    ↪ p207, p175, i93]
+1907: i214 = int_add(i210, -1)
+1911: p215 = getfield_gc(p207, descr=<FieldP list.items 16>)
+1915: p216 = getarrayitem_gc(p215, i210, descr=<ArrayP 8>)
+1927: p217 = getfield_gc(p216, descr=<FieldP bstate.RPythonMap.inst_structure 16>)
+1931: guard_value(p217, ConstPtr(ptr218), descr=<Guard0x100e830a0>) [p13, p2, p1, p0,
    ↪ p203, p184, p204, i214, p207, p217, p216, p175, i93]
+1950: p219 = getfield_gc(p216, descr=<FieldP bstate.RPythonMap.inst_storage 8>)
+1954: p221 = getarrayitem_gc(p219, 1, descr=<ArrayP 8>)
+1958: guard_nonnull_class(p203, ConstClass(W_Integer), descr=<Guard0x100e83048>) [p13,
    ↪ p2, p1, p0, p221, p203, p175, i93]
+1976: guard_nonnull_class(p221, ConstClass(W_Integer), descr=<Guard0x100e82ff0>) [p13,
    ↪ p2, p1, p0, p221, p203, p175, i93]
+1994: i224 = getfield_gc(p203, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue
    ↪ 24>)
+1998: i225 = getfield_gc(p221, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue
    ↪ 24>)
+2002: i226 = int_sub(i224, i225)
+2005: i227 = int_lt(i226, i93)
```

Figure D.5.: Rpython trace of a B While loop

```
guard_true(i227, descr=<Guard0x100e82f98>) [p13, p2, p1, p0, i226, p175, i93]
+2018: p228 = getarrayitem_gc(p182, i181, descr=<ArrayP 8>)
+2023: i229 = getfield_gc(p2, descr=<FieldS list.length 8>)
+2034: i231 = int_add(i229, 1)
+2038: p232 = getfield_gc(p2, descr=<FieldP list.items 16>)
+2042: i233 = arraylen_gc(p232, descr=<ArrayP 8>)
+2046: i234 = int_lt(i233, i231)
+2057: cond_call(i234, ConstClass(
    ↪ _ll_list_resize_hint_really_look_inside_iff__listPtr_Signed_Bool), p2, i231, 1,
    ↪ descr=<Callv 0 rii EF=5>)
+2132: guard_no_exception(descr=<Guard0x100e82f40>) [i229, p228, p13, p2, p1, p0, i231,
    ↪  i226, p175, i93]
+2152: p237 = getfield_gc(p2, descr=<FieldP list.items 16>)
+2195: setarrayitem_gc(p237, i229, p228, descr=<ArrayP 8>)
+2200: setfield_gc(p2, i231, descr=<FieldS list.length 8>)
+2204: i238 = getfield_gc(p178, descr=<FieldS list.length 8>)
+2208: i240 = int_add(i238, -1)
+2212: p241 = getfield_gc(p178, descr=<FieldP list.items 16>)
+2216: p242 = getarrayitem_gc(p241, i240, descr=<ArrayP 8>)
+2221: p243 = getfield_gc(p242, descr=<FieldP bstate.BState.inst_stack_list 40>)
+2225: p244 = getfield_gc(p243, descr=<FieldP list.items 16>)
+2230: p245 = getarrayitem_gc(p244, i187, descr=<ArrayP 8>)
+2235: i246 = getfield_gc(p245, descr=<FieldS list.length 8>)
+2240: i248 = int_sub(i246, 1)
+2244: i250 = int_le(i248, -1)
guard_false(i250, descr=<Guard0x100e82ee8>) [p228, p13, p2, p1, p0, p184, p242, i248,
    ↪ p245, i226, p175, i93]
+2254: i252 = int_add(i248, -1)
+2258: p253 = getfield_gc(p245, descr=<FieldP list.items 16>)
+2270: p254 = getarrayitem_gc(p253, i248, descr=<ArrayP 8>)
+2275: p255 = getfield_gc(p254, descr=<FieldP bstate.RPythonMap.inst_structure 16>)
+2279: guard_value(p255, ConstPtr(ptr256), descr=<Guard0x100e82e90>) [p228, p13, p2, p1
    ↪ , p0, p184, p242, i252, p245, p255, p254, i226, p175, i93]
+2298: p257 = getfield_gc(p254, descr=<FieldP bstate.RPythonMap.inst_storage 8>)
+2302: p259 = getarrayitem_gc(p257, 1, descr=<ArrayP 8>)
+2306: p260 = getarrayitem_gc(p241, i240, descr=<ArrayP 8>)
+2311: p261 = getfield_gc(p260, descr=<FieldP bstate.BState.inst_stack_list 40>)
+2315: p262 = getfield_gc(p261, descr=<FieldP list.items 16>)
+2320: p263 = getarrayitem_gc(p262, i187, descr=<ArrayP 8>)
+2325: i264 = getfield_gc(p263, descr=<FieldS list.length 8>)
+2330: i266 = int_sub(i264, 1)
+2334: i268 = int_le(i266, -1)
guard_false(i268, descr=<Guard0x100e82e38>) [p228, p13, p2, p1, p0, p259, p184, p260,
    ↪ i266, p263, i226, p175, i93]
+2344: i270 = int_add(i266, -1)
+2348: p271 = getfield_gc(p263, descr=<FieldP list.items 16>)
+2353: p272 = getarrayitem_gc(p271, i266, descr=<ArrayP 8>)
+2358: p273 = getfield_gc(p272, descr=<FieldP bstate.RPythonMap.inst_structure 16>)
+2362: guard_value(p273, ConstPtr(ptr274), descr=<Guard0x100e82de0>) [p228, p13, p2, p1
    ↪ , p0, p259, p184, p260, i270, p263, p273, p272, i226, p175, i93]
+2381: p275 = getfield_gc(p272, descr=<FieldP bstate.RPythonMap.inst_storage 8>)
+2385: p277 = getarrayitem_gc(p275, 2, descr=<ArrayP 8>)
+2389: guard_class(p259, ConstClass(W_Integer), descr=<Guard0x100e82d88>) [p228, p13,
    ↪ p2, p1, p0, p259, p277, i226, p175, i93]
+2401: i279 = getfield_gc(p259, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue
    ↪ 24>)
+2405: i280 = getfield_gc(p277, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue
    ↪ 24>)
+2409: i281 = int_lt(i279, i280)
+2420: guard_true(i281, descr=<Guard0x100e82c80>) [p228, p13, p2, p1, p0, i281, i226,
    ↪ p175, i93]
+2429: p282 = getarrayitem_gc(p241, i240, descr=<ArrayP 8>)
+2429: setarrayitem_gc(p241, i240, ConstPtr(null), descr=<ArrayP 8>)
+2438: i284 = arraylen_gc(p241, descr=<ArrayP 8>)
+2442: i286 = int_rshift(i284, 1)
```

Figure D.6.: Rpython trace of a B While loop

```
+2445: i288 = int_sub(i286, 5)
+2449: i289 = int_lt(i240, i288)
+2460: cond_call(i289, ConstClass(
    ↪ _ll_list_resize_hint_really_look_inside_iff__listPtr_Signed_Bool), p178, i240, 0,
    ↪  descr=<Callv 0 rii EF=5>)
+2524: guard_no_exception(descr=<Guard0x100e82bd0>) [p228, p13, p2, p1, p0, i240, p178,
    ↪  None, i226, p175, i93]
+2544: setfield_gc(p178, i240, descr=<FieldS list.length 8>)
+2548: p293 = call(ConstClass(BState.clone), p13, descr=<Callr 8 r EF=5>)
+2647: guard_no_exception(descr=<Guard0x100e82b20>) [p293, p177, p228, p13, p2, p1, p0,
    ↪  None, i226, p175, i93]
+2667: i294 = getfield_gc(p178, descr=<FieldS list.length 8>)
+2678: i296 = int_add(i294, 1)
+2682: p297 = getfield_gc(p178, descr=<FieldP list.items 16>)
+2686: i298 = arraylen_gc(p297, descr=<ArrayP 8>)
+2690: i299 = int_lt(i298, i296)
+2701: cond_call(i299, ConstClass(
    ↪ _ll_list_resize_hint_really_look_inside_iff__listPtr_Signed_Bool), p178, i296, 1,
    ↪  descr=<Callv 0 rii EF=5>)
+2765: guard_no_exception(descr=<Guard0x100e82a70>) [i294, p293, p178, p228, p13, p2,
    ↪ p1, p0, i296, None, i226, p175, i93]
+2785: p302 = getfield_gc(p178, descr=<FieldP list.items 16>)
+2828: setarrayitem_gc(p302, i294, p293, descr=<ArrayP 8>)
+2833: setfield_gc(p178, i296, descr=<FieldS list.length 8>)
+2837: p304 = call_may_force(ConstClass(ASequenceSubstitution.
    ↪ exec_ASequenceSubstitution__next), p175, descr=<Callr 8 r EF=7>)
guard_not_forced(descr=<Guard0x100e980e0>) [p228, p13, p2, p1, p0, p304, None, i226,
    ↪ None, i93]
+2940: p306 = guard_exception(ConstClass(StopIteration), descr=<Guard0x100e82a18>) [
    ↪ p228, p13, p2, p1, p0, p304, None, i226, None, i93]
+3006: i307 = getfield_gc(p2, descr=<FieldS list.length 8>)
+3017: i309 = int_ne(i307, 0)
guard_true(i309, descr=<Guard0x100e82968>) [p228, p13, p2, p1, p0, i307, None, i226,
    ↪ None, i93]
debug_merge_point(0, 0, '(jitdriver:_no_get_printable_location)')
p311 = new_with_vtable(ConstClass(W_Boolean))
p313 = new_with_vtable(ConstClass(W_Integer))
p315 = new_with_vtable(ConstClass(W_Boolean))
+3131: setfield_gc(p311, 1, descr=<FieldU rpython_b_objmodel.W_Object.inst_bvalue 104>)
p318 = new_with_vtable(ConstClass(W_Integer))
+3147: setfield_gc(p313, i93, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue
    ↪ 24>)
+3158: setfield_gc(p315, 1, descr=<FieldU rpython_b_objmodel.W_Object.inst_bvalue 104>)
+3163: setfield_gc(p318, i226, descr=<FieldS rpython_b_objmodel.W_Object.inst_ivalue
    ↪ 24>)
+3526: jump(p0, p1, p2, p311, p13, p313, p315, p318, p228, descr=TargetToken
    ↪ (4310106432))
+3568: --end of the loop--
[947c5cc5697] jit-log-opt-loop}
checked 2 states. No invariant violation found.
```

Figure D.7.: Rpython trace of a B While loop

E

Symbolic Sets

The symbolic set classes can be found in the modules: symbolic_sets.py, symbolic_functions.py and symbolic_functions_with_predicate.py. Composed sets are symbolic set classes which need to reference other sets (explicit frozenset or symbolic)

Class name	Description or B name
SymbolicSet	Base class (contains default implementations)
LargeSet	Abstract class
InfiniteSet	Abstract class
NaturalSet	Natural
Natural1Set	Natural1
IntegerSet	Integer
NatSet	Nat
Nat1Set	Nat1
IntSet	Int
StringSet	String
SymbolicIntervalSet	A ... B

Figure E.1.: Symbolic set classes of PyB(not composed)

Class name	Description or B name
SymbolicUnionSet	S ∪ T
SymbolicIntersectionSet	S ∩ T
SymbolicDifferenceSet	S - T
SymbolicCartSet	S*T
SymbolicPowerSet	Pow(S)
SymbolicPower1Set	Pow1(S)
SymbolicIntervalSet	A ... B
SymbolicStructSet	structs
SymbolicRelationSet	$S <-> T$
SymbolicPartialFunctionSet	$S +-> T$
SymbolicTotalFunctionSet	$S --> T$
SymbolicPartialInjectionSet	$S >+> T$
SymbolicTotalInjectionSet	$S >-> T$
SymbolicPartialSurjectionSet	$S +->> T$
SymbolicTotalSurjectionSet	$S -->> T$
SymbolicTotalBijectionSet	$S >->> T$
SymbolicPartialBijectionSet	$S >+>> T$
SymbolicFirstProj	prj1(S,T)
SymbolicSecondProj	prj2(S,T)
SymbolicIdentitySet	id(S)
SymbolicCompositionSet	f ; g
SymbolicTransRelation	rel(f)
SymbolicTransFunction	fnc(r)
SymbolicInverseRelation	r^{-1}
SymbolicLambda	lambda functions
SymbolicComprehensionSet	set comprehensions
SymbolicQuantifiedIntersection	$\bigcap x.(P(x)\|E)$
SymbolicQuantifiedUnion	$\bigcup x.(P(x)\|E)$
AbstractSymbolicSequence	Abstract class
SymbolicSequenceSet	seq(S)
SymbolicSequence1Set	seq1(S)
SymbolicISequenceSet	iseq(S)
SymbolicISequence1Set	iseq1(S)
SymbolicPermutationSet	perm(S)

Figure E.2.: Symbolic set classes of PyB(composed)

Class name	Description or B name
SymbolicLambda	lambda functions
SymbolicComprehensionSet	set comprehensions
SymbolicQuantifiedIntersection	$\bigcap x.(P(x)\|E)$
SymbolicQuantifiedUnion	$\bigcup x.(P(x)\|E)$

Figure E.3.: Symbolic set classes of PyB(defined by predicate)

Bibliography

[1] Robert Abo and Laurent Voisin. "Formal Implementation of Data Validation for Railway Safety-Related Systems with OVADO". In: *SEFM 2013 Collocated Workshops, FM-RAIL-Bok* (2013).

[2] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996. ISBN: 0-521-49619-5.

[3] Jean-Raymond Abrial and Dominique Cansell. "Click'n Prove: Interactive Proofs within Set Theory". In: *Theorem Proving in Higher Order Logics: 16th International Conference, TPHOLs 2003, Rome, Italy, September 8-12, 2003. Proceedings*. Ed. by David Basin and Burkhart Wolff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–24. ISBN: 978-3-540-45130-3.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[5] Alex. Gaynor et al. *Topaz*. 2016. URL: `http://docs.topazruby.com/` (visited on 06/07/2016).

[6] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. "RPython: a step towards reconciling dynamically and statically typed OO languages". In: *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*. Montreal, Quebec, Canada: ACM, 2007, pp. 53–64. ISBN: 978-1-59593-868-8.

[7] John Aycock. "A Brief History of Just-in-time". In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 97–113. ISSN: 0360-0300.

[8] Oxon B-Core (UK) Limited. "UK. B-Toolkit, On-line manual." In: *In Proceedings of TOOLS*. 1999.

[9] Benoit Baudry and Martin Monperrus. "The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond". In: *ACM Comput. Surv.* 48.1 (Sept. 2015), 16:1–16:26. ISSN: 0360-0300.

[10] Spenser Bauman et al. "Pycket: A Tracing JIT for a Functional Language". In: *SIGPLAN Not.* 50.9 (Aug. 2015), pp. 22–34. ISSN: 0362-1340.

[11] Beck. *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321146530.

[12] Shirley A. Becker and James A. Whittaker. *Cleanroom Software Engineering Practices*. Hershey, PA, USA: IGI Global, 1997. ISBN: 1878289349.

[13] Ron Bell. "Introduction to IEC 61508". In: *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*. SCS '05. Sydney, Australia: Australian Computer Society, Inc., 2006, pp. 3–12. ISBN: 1-920-68237-6.

[14] Carl F. Bolz. "A Prolog Interpreter in Python". Bachelorsthesis. HHU Düsseldorf, 2007.

[15] Carl Friedrich Bolz. "Meta-Tracing Just-in-Time Compilation for RPython". PhD thesis. Heinrich Heine University Duesseldorf, Sept. 2012.

[16] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. "Tracing the Meta-level: PyPy's Tracing JIT Compiler". In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ICOOOLPS '09. Genova, Italy: ACM, 2009, pp. 18–25. ISBN: 978-1-60558-541-3.

[17] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. "Storage strategies for collections in dynamically typed languages". In: *Proc. OOPSLA, to appear*. ACM, 2013.

[18] Carl Friedrich Bolz, Michael Leuschel, and David Schneider. "Towards a Jitting VM for Prolog Execution". In: *PPDP '10 - Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. Hagenberg, Austria: ACM, 2010.

[19] Carl Friedrich Bolz and Armin Rigo. "How to not write Virtual Machines for Dynamic Languages". In: *Proceeding of Dyla 2007*. 2007.

[20] Carl Friedrich Bolz et al. "Back to the Future in One Week – Implementing a Smalltalk VM in PyPy". In: *Self-Sustaining Systems*. Lecture Notes in Computer Science. Springer, 2008, pp. 123–139.

[21] F. Bouquet, B. Legeard, and F. Peureux. "CLPS-B—A Constraint Solver for B". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by J. P. Katoen and P. Stevens. Vol. 2280. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 188–204.

[22] Camillo Bruni, Toon Verwaest, and Marcus Denker. *PyGirl: Generating Whole-System VMs from high-level models using PyPy*. Technical Report. Technical Report -9, University of Bern, Institute of Applied Mathematics and Computer Sciences, 2009. 2009.

[23] Michael Butler and Stefan Hallerstede. "The Rodin Formal Modelling Tool". 2007.

[24] Mats Carlsson and Thom Fruehwirth. *Sicstus PROLOG User's Manual 4.3*. Books On Demand - Proquest, 2014. ISBN: 3735737447, 9783735737441.

[25] Mats Carlsson, Greger Ottosson, and Björn Carlson. "An open-ended finite domain constraint solver". In: *Programming Languages: Implementations, Logics, and Programs: 9th International Symposium, PLILP '97 Including a Special Track on Declarative Programming Languages in Education Southampton, UK, September 3–5, 1997 Proceedings*. Ed. by Hugh Glaser, Pieter Hartel, and Herbert Kuchen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 191–206. ISBN: 978-3-540-69537-0.

[26] CENELEC. *Railway Applications – Communication, signalling and processing systems – Software for railway control and protection systems*. Tech. rep. EN50128. European Standard, 2011.

[27] Alain Colmerauer. "Prolog in 10 Figures". In: *Commun. ACM* 28.12 (Dec. 1985), pp. 1296–1310. ISSN: 0001-0782.

[28] Alain Colmerauer and Philippe Roussel. "History of Programming languages—II". In: ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. New York, NY, USA: ACM, 1996. Chap. The Birth of Prolog, pp. 331–367. ISBN: 0-201-89502-1.

[29] David Déharbe, Bruno Gomes, and Anamaria Moreira. "BSmart: A Tool for the Development of Java Card Applications with the B Method". In: *Abstract State Machines, B and Z: First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*. Ed. by Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 351–352. ISBN: 978-3-540-87603-8.

[30] Matthew Flatt and PLT. *Reference: Racket*. Tech. rep. PLT-TR-2010-1. `https://racket-lang.org/tr1/`. PLT Design Inc., 2010.

[31] Étienne Gagnon and Etienne Gagnon. "Sablecc, An Object-Oriented Compiler Framework". In: *In Proceedings of TOOLS*. 1998, pp. 140–154.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612.

[33] Sven Hager. "Implementing the R Language Using RPython". Masterthesis. HHU Düsseldorf, 2012.

[34] Dominik Hansen, David Schneider, and Michael Leuschel. "Using B and ProB for Data Validation Projects". In: *Proceedings ABZ 2016*. Vol. 9675. LNCS. Springer-Verlag, 2016.

[35] *Haskell 98 Language and Libraries The Revised Report*. December 2002. URL: `https://www.haskell.org/onlinereport/intro.html` (visited on 08/07/2016).

[36] Ross Ihaka and Robert Gentleman. "R: A Language for Data Analysis and Graphics". In: *Journal of Computational and Graphical Statistics* 5.3 (1996), pp. 299–314.

[37] J. C. Knight and N. G. Leveson. "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming". In: *IEEE Trans. Softw. Eng.* 12.1 (Jan. 1986), pp. 96–109. ISSN: 0098-5589.

[38] Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. "Visualising Event-B Models with B-Motion Studio". In: *Formal Methods for Industrial Critical Systems: 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings*. Ed. by María Alpuente, Byron Cook, and Christophe Joubert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 202–204. ISBN: 978-3-642-04570-7.

[39] Thierry Lecomte, Lilian Burdy, and Michael Leuschel. "Formally Checking Large Data Sets in the Railways". In: *CoRR* abs/1210.6815 (2012). Proceedings of DS-Event-B 2012, Kyoto.

[40] Thierry Lecomte and Erwan Mottin. "Formal Data Validation in the Railways". In: *Conference: Safety-critical Systems Symposium 2016* (2016).

[41] Hung Ledang and Jeanine Souquières. "Formalizing UML Behavioral Diagrams with B". In: *Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics*. Colloque avec actes et comité de lecture. internationale. Tampa Bay, Florida, USA, Oct. 2001, 12 p.

[42] Michael Leuschel and Michael Butler. "ProB: A Model Checker for B". In: *FME*. Ed. by Araki Keijiro, Stefania Gnesi, and Mandrio Dino. Vol. 2805. Lecture Notes in Computer Science. Springer-Verlag, 2003, pp. 855–874. ISBN: 3-540-40828-2.

[43] Michael Leuschel and Michael Butler. "ProB: An Automated Analysis Toolset for the B Method". In: *Software Tools for Technology Transfer (STTT)* 10.2 (2008), pp. 185–203.

[44] Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. "Automated property verification for large scale B models with ProB". In: *Formal Aspects of Computing* 23.6 (2011), pp. 683–709. ISSN: 1433-299X.

[45] Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. "Automated Property Verification for Large Scale B Models". In: *Proceedings FM 2009*. Vol. 5850. Lecture Notes in Computer Science. Springer-Verlag, 2009, pp. 708–723.

[46] Derek Lockhart, Berkin Ilbeyi, and Christopher Batten. "Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers". In: *2015 IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*. 2015.

[47] A. K. Mackworth. "Consistency in Networks of Relations". In: *Artificial Intelligence* 8.1 (1977), pp. 99–118.

[48] S. Marr, T. Pape, and W. De Meuter. "Are We There Yet?: Simple Language Implementation Techniques for the 21st Century". In: *IEEE Software* 31.5 (2014), pp. 60–67. ISSN: 0740-7459.

[49] Atif Mashkoor, Faqing Yang, and Jean-Pierre Jacquot. "Refinement-based Validation of Event-B Specifications". In: *Software & Systems Modeling* (2016), pp. 1–20. ISSN: 1619-1374.

[50] Remigius Meier and Armin Rigo. "A Way Forward in Parallelising Dynamic Languages". In: *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*. ICOOOLPS '14. Uppsala, Sweden: ACM, 2014, 4:1–4:4. ISBN: 978-1-4503-2914-9.

[51] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375.

[52] Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, and Helen Treharne. "Defining and Model Checking Abstractions of Complex Railway Models using CSP——B". In: *HVC'2012*. 2012.

[53] Daniel Plagge and Michael Leuschel. "Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more". In: *Software Tools for Technology Transfer (STTT)* 12.1 (2010), pp. 9–21. ISSN: 1433-2779.

[54] Armin Rigo and Samuele Pedroni. "PyPy's approach to virtual machine construction". In: *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. Portland, Oregon, USA: ACM, 2006, pp. 944–953. ISBN: 1-59593-491-X.

[55] Ken Robinson. "The B method and the B toolkit". In: *Algebraic Methodology and Software Technology: 6th International Conference, AMAST'97 Sydney, Australia, December13–17, 1997 Proceedings*. Ed. by Michael Johnson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 576–580. ISBN: 978-3-540-69661-2.

[56] Guido Rossum. *Python Reference Manual*. Tech. rep. Amsterdam, The Netherlands, The Netherlands, 1995.

[57] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2nd ed. Pearson Education, 2003. ISBN: 0137903952.

[58] Georges Mariano Samuel Colin Dorian Petit. "BRILLANT: an open source platform for B." In: *Workshop on Tool Building in Formal Methods, Orford, Canada. 2010.* Feb. 2010.

[59] Thierry Servat. "BRAMA: A New Graphic Animation Tool for B Models". In: *B 2007: Formal Specification and Development in B: 7th International Conference of B Users, Besançon, France, January 17-19, 2007. Proceedings*. Ed. by Jacques Julliand and Olga Kouchnarenko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 274–276. ISBN: 978-3-540-68761-0.

[60] France Steria Aix-en-Provence. *Atelier B, User and Reference Manuals*. Available at `http://www.atelierb.societe.com`. 1996.

[61] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[62] Bruno Tatibouët, Antoine Requet, Jean-Christophe Voisinet, and Ahmed Hammad. "Java Card Code Generation from B Specifications". In: *Formal Methods and Software Engineering: 5th International Conference on Formal Engineering Methods, ICFEM 2003, Singapore, November 5-7, 2003. Proceedings*. Ed. by Jin Song Dong and Jim Woodcock. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 306–318. ISBN: 978-3-540-39893-6.

[63] Even Wiik Thomassen. "Trace-based just-in-time compiler for Haskell with RPython". Masterthesis. NTNU Trondheim, 2013.

[64] *Tiobe programming community index*. `http://tiobe.com/index.php/content/paperinfo/tpci/index.html`. Accessed: 2016-04-20.

[65] Markus Triska. "The Finite Domain Constraint Solver of SWI-Prolog". In: *FLOPS*. Vol. 7294. LNCS. 2012, pp. 307–316.

[66] *What is JavaScript*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript` (visited on 06/07/2016).

[67] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. "SWI-Prolog". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 67–96. ISSN: 1471-0684.

[68] John Witulski. "Eine Java - VM in Python (A Java-VM in Python)". Masterthesis. HHU Düsseldorf, 2009.

[69] John Witulski and Michael Leuschel. "Checking Computations of Formal Method Tools - A Secondary Toolchain for ProB". In: *Proceedings of the 1st Workshop on Formal-IDE*. EPTCS 149, 2014. Electronic Proceedings in Theoretical Computer Science, 2014.

[70] Faqing Yang. "A Simulation Framework for the Validation of Event-B Specifications". Theses. Université de Lorraine, Nov. 2013.

[71] Faqing Yang, Jean-Pierre Jacquot, and Jeanine Souquières. "The Case for Using Simulation to Validate Event-B Specifications". In: *APSEC*. Ed. by Karl R. P. H. Leung and Pornsiri Muenchaisri. IEEE, 2012, pp. 85–90. ISBN: 978-0-7695-4922-4.

[72] Stephan Zalewski. "A Javascript Interpreter in RPython". Bachelorsthesis. HHU Düsseldorf, 2012.

List of Figures

List of Tables