Checking Computations of Formal Method Tools - A Secondary Toolchain for ProB

John Witulski and Michael Leuschel

Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{witulski, leuschel}@cs.uni-duesseldorf.de

Abstract. We present the implementation of pyB, a predicate- and expression-checker for the B language. The tool is to be used for a secondary tool chain for data validation and data generation, with ProB being used in the primary tool chain. Indeed, pyB is an independent cleanroom-implementation which is used to double-check solutions generated by ProB, an animator and model-checker for B specifications. One of the major goals is to use ProB together with pyB to generate reliable outputs for high-integrity safety critical applications. Although pyB is still work in progress, the ProB/pyB toolchain has already been successfully tested on various industrial B machines and data validation tasks.

1 Introduction

1.1 Motivation

The success of formal methods in practice depends on fast, scalable but also reliable tools. Indeed, a bug inside a tool can have disastrous consequences in the context of safety critical software.

One solution to this problem is to prove the correctness of the tool itself. However, many tools used in the context of formal methods consist of tens or hundreds of thousands of lines of code which haver evolved over long periods of time. Reimplementing these tools to be correct by construction or verifying these tools formally a posteriori is often impractical.

An alternate solution to increase the trust in the output of the tool by using a *double chain*, i.e. validating the output of the main tool by a second, independently developed tool. In some cases, the second tool can also be much simpler, as its purpose is just checking an output, not producing it in the first place. This is the solution we have pursued in this paper for the ProB tool [LB03,PL10], which is an animator, model checker and constraint solver for the B-method [Abr96].

More concretely, our long term goal is to enable ProB also to be used as a tool of class T3 within the norm EN 50128, i.e., moving from a tool of class T2 that *"supports the test or verification of the design or executable code, where errors in the tool can fail to reveal defects but cannot directly create errors in the executable software"* [CEN11] to a tool that *"generates outputs which can*

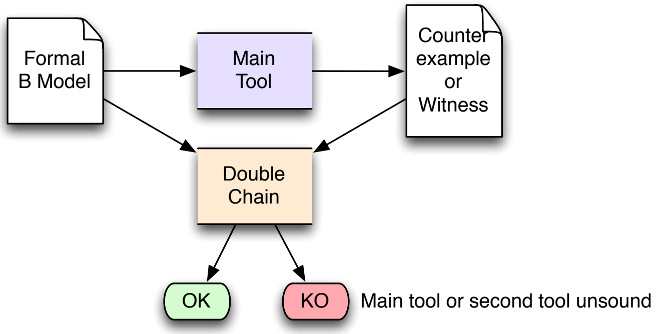


Fig. 1. Double Chain

directly or indirectly contribute to the executable code (including data) of the safety related system" [CEN11].

1.2 Approach

The idea of a double chain is to double-check every result or output (e.g. the value of a predicate) calculated by the main tool a second time using an independently developed secondary toolchain (see Figure 1). For instance, after a tool like ProB checks the invariant for some state of a B model, a tool like pyB will check it again. The main tool can provide some additional information to make the task of the secondary toolchain simpler (e.g., provide witness values for existential quantifiers).

If the results of the two toolchains disagree, then an error is raised by the secondary toolchain and the issue will have to be investigated manually (as there must be a bug either in the main or in the secondary toolchain). If the results of the two toolchains are identical we can have increased confidence in the validity of the result. The hope is that, in case the main tool (ProB) contains a bug, it is very unlikely that an independently developed tool will exhibit the same bug for the same input. As such, pyB is a clean-room implementation which only shares the parser with ProB. While the kernel and interpreter of ProB is written in Prolog, pyB is written in Python, a dynamic imperative language. Exactly like ProB, pyB covers the B language as specified by the ClearSy B language reference manual [Cle09].

1.3 Outline

In Section 2, we provide a short summery of pyB's main features, while in Section 3 we give an overview of pyB's implementation. We provide our experience in developing pyB in Section 4, which we hope will be useful for readers interested in embarking upon a similar path than ours. In Section 5 we present one case study and some empirical results of our tools. We conclude with related and future work in Section 6.

2 Main Features of PyB

PYB's role is to double check PROB's results in the context of data-validation. As such, it has to be able to

1. evaluate B expressions over sets, relations, functions and sequences. It also supports set comprehensions and lambda abstractions.
2. evaluate B predicates with universal and existential quantifiers,
3. execute B substitutions (certain data validation are encoded as sequences of operations).

It has been integrated into PROB's Tcl-Tk version, and can be called automatically to double check the results of PROB. Other features of the tool are a REPL (read-eval-print-loop) and its interactive animation mode. These features are not discussed, because they are of little relevance in the context of the double chain.

3 Architecture of PyB

The next section will introduce some implementation details of PYB. Figure 2 shows a module overview.

3.1 Parsing and Typing

PYB is an independent clean-room implementation, except for its Java parser. This Java parser was written by Fabian Fritz in 2008 and is also used by PROB. PYB uses its parser to recognise B-constructs like predicates and expressions. These constructs are translated to an intermediate representation: an abstract syntax Tree (AST) made of Java objects. These Java objects are now translated to Python objects via an AST-visitor, an addition to the Java code. This visitor is the only part of PYB which is written in Java.

The AST-visitor emits a string of Python code. The dynamic features of Python enable the execution of this Python code emitted by the Java visitor.

Listing one (Figure 3) shows the Python code created by the Java-visitor for the simple predicate $1+1=x$. The AST 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 the interpreter and is the main input for most PYB methods.

Before the interpretation starts, the AST must pass a type checker, which uses a Hindley-Milner style unification algorithm. After a successful pass of the type

Fig. 2. Module Overview

Name	**Summery**
animation_clui.py	console interface for animation mode
animation.py	main animation calculation, together with the substitution method
ast_nodes.py	Python 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 opertation
bstate.py	a class representing one B-machine state
btypes.py	type objects
config.py	main config file
constrainsolver.py	B-wrapper to use a third-pary constraint solving code
definition_handler.py	main definition handling code
enumeration.py	enumeration methods for sets, functions, relations and more
environment.py	B-state managing code
external_functions.py	implementation of external functions
fake_sets.py	implementation of large and infinite sets
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	helper functions to enable evaluation without enumeration
repl.py	read-eval-print-loop code
statespace.py	implementation of the state space
typing.py	main type checking code

checking code, a type is added to every identifier node (in this example x). Type information is of course also important for enumeration of values, e.g., during the evaluation of quantified predicates. The unification based algorithm makes pyB compatible with ProB, and it is more powerful than the type checking of other tools such as Atelier-B (because the order of the predicates is less important). For example, the predicate `x=y+1` is well typed for pyB and ProB whereas Atelier-B requires the addition of typing information: `x:INTEGER & y:INTEGER & x=y+1`.

Between the parsing and type checking phase may be a phase where B definitions (macros) need to be substituted. Possible external function calls — a particular feature of ProB which allows linking external code with B specifications — are also resolved at this time.

3.2 Implementation of B's data types

The most important data type of B is the set. In B there exist built-in sets for boolean, natural or integer numbers. Relations are sets of tuples. Functions and

Fig. 3. python code for the predicate 1+1=x

```
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
```

sequences are just special cases of relations. All B-operations like power set or the cartesian product are only producing more complex sets.

While booleans and integers are represented by their python built-in counterparts, most data types are represented in PYB by the python built-in type: frozenset[1]. Frozensets are immutable objects which already implement all basic set operations like union, intersection, inclusion, membership etc.

Relations are implemented as frozensets of tuple objects, which are also python built-ins. For example a B-function 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 **frozenset([(1,1),(2,4),(3,9)])**. Objects like this can be created during the interpretation of B.

3.3 Interpretation of B

Expressions and predicates are evaluated by the interpreter module. The main evaluation method has two parameters: the AST root or the root of a subtree and an environment. The method call returns a value, if the tree represents a predicate, it returns true or false.

The interpreter recursively performs a depth-first walk on the tree while evaluating different code for every object class. The environment holds the state-space of the B-machine. Every state is a stack of hash maps holding the current values of the identifiers. Every new scope (e.g. a quantified predicate) creates a new frame on this stack. This is a standard approach of interpreter implementation.

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

Fig. 4. excerpt of the interpret method (indentation of last case changed for readability)

```
elif isinstance(node, AAddExpression):
    expr1 = interpret(node.children[0], env)
    expr2 = interpret(node.children[1], env)
    return expr1 + expr2
elif isinstance(node, AMinusOrSetSubtractExpression)
      or isinstance(node, ASetSubtractionExpression):
    expr1 = interpret(node.children[0], env)
    expr2 = interpret(node.children[1], env)
    return expr1 - expr2
```

Figure 4 shows two of over hundred cases inside the “interpret” method. The evaluation depends on the type of the visited AST-node. The first case is that of a simple addition. The interpreter recursively visits its subtrees and adds the values of the calculated subexpressions. The second case is similar to the first one with only one exception: if the expressions expr1 and expr2 are numbers a simple integer subtraction will calculated, but if they are frozensets then a set subtraction will take place. The overladed minus-operator, is natively defined on frozensets in Python. That means a method of the built-in frozenset object is executed.

3.4 Animation of B

Substitutions are handled similar to the evaluation of predicates. The main difference are two aspects: First, the evaluation does not return a value, but true or false if a sequence of substitutions was successfully executed. Second, the evaluation produces a new B state. This state is derived from a copy of the current state and will be added to the state-space. Later it can become the current state (in case of interactive animation if the user chooses this operation).

Sometimes a sequence of substitutions inside a B-operation consists of nondeterministic substitutions. These substitutions can be seen as choice point with different execution branches. PYB explores every branch by backtracking to this choice point. A new state is returned for every possible execution branch.

All visited states are saved. This enables PYB to backtrack on the state level. This is an important feature needed for interactive animation and possible model checking by PYB in the future.

3.5 Difficult Aspects of B

Checking a ProB solution can be very complex. The B formulas may contain quantified predicates, set comprehensions or lambda expressions. The solutions computed for variables or parameters can be very large sets.

pyB evaluates the B constructs by a brute force approach. It generates all possible values and checks if they fulfil the constraints. For example, pyB checks an existentially quantified predicate by checking all values of the type of the quantified variables. If no value is found, the predicate is false.

Of course this approach has to fail if the set of possible values is very large or infinite. Then a symbolic representation or constraint solving can be the solution to this dilemma. pyB generates special set classes instead of frozen sets if a set becomes very large or infinite. This is not fully implemented at the current development level. Also pyB uses a external constraint solver to constrain the set of possible solutions. This usage of constraint solving will be extended in the future too.

3.6 Linking with ProB

Verification of States Below we use an example of a complex B-Machine (cruise control model) with a simple state. Figure 5 shows ProB in its animation-mode for this example. At some point the user can save the B-State to a file (Figure 6). Eventually pyB reads this file and outputs if a safety property was violated in this state. Currently this process is automated via a Python script but will be fully included into the official ProB release in the future.

Verification of State-changes Another application of pyB as second toolchain is not the verification of states, but the execution of operations (i.e the application of substitutions). In this approach, it is not of interest if a calculated state satisfies a safety condition, but rather which operations are enabled at a specific state and if the 'execution' of an operation leads to the same state calculated by other tools like ProB˙

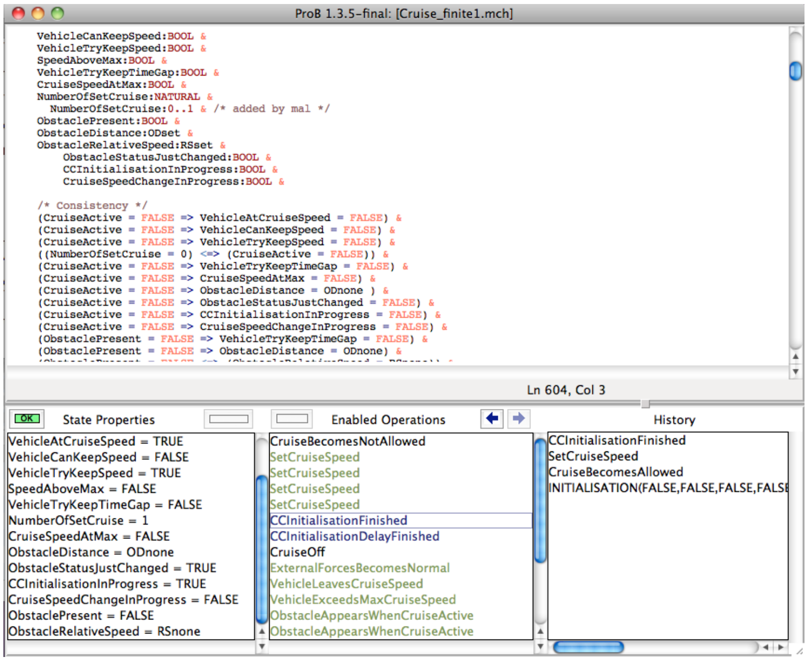
In this case, the second toolchain is not used to check if a safe or faulty state is really safe or faulty, but if the states (that violate the safety properties of a model) are really reachable.

Figure 7 shows the console output of an interactive animation of a simple scheduler B model.

4 Development Experience of PyB

4.1 Timeline and Effort

The developmet of pyB started late 2011 and continued until today. At approximately 10 hours work per week on average, the current tool is a result of



```
    VehicleCanKeepSpeed:BOOL &
    VehicleTryKeepSpeed:BOOL &
    SpeedAboveMax:BOOL &
    VehicleTryKeepTimeGap:BOOL &
    CruiseSpeedAtMax:BOOL &
    NumberOfSetCruise:NATURAL &
      NumberOfSetCruise:0..1 & /* added by mal */
    ObstaclePresent:BOOL &
    ObstacleDistance:ODset &
    ObstacleRelativeSpeed:RSset &
        ObstacleStatusJustChanged:BOOL &
        CCInitialisationInProgress:BOOL &
        CruiseSpeedChangeInProgress:BOOL &

    /* Consistency */
    (CruiseActive = FALSE => VehicleAtCruiseSpeed = FALSE) &
    (CruiseActive = FALSE => VehicleCanKeepSpeed = FALSE) &
    (CruiseActive = FALSE => VehicleTryKeepSpeed = FALSE) &
    ((NumberOfSetCruise = 0) <=> (CruiseActive = FALSE)) &
    (CruiseActive = FALSE => VehicleTryKeepTimeGap = FALSE) &
    (CruiseActive = FALSE => CruiseSpeedAtMax = FALSE) &
    (CruiseActive = FALSE => ObstacleDistance = ODnone ) &
    (CruiseActive = FALSE => ObstacleStatusJustChanged = FALSE) &
    (CruiseActive = FALSE => CCInitialisationInProgress = FALSE) &
    (CruiseActive = FALSE => CruiseSpeedChangeInProgress = FALSE) &
    (ObstaclePresent = FALSE => VehicleTryKeepTimeGap = FALSE) &
    (ObstaclePresent = FALSE => ObstacleDistance = ODnone) &
```

VehicleAtCruiseSpeed = TRUE
VehicleCanKeepSpeed = FALSE
VehicleTryKeepSpeed = TRUE
SpeedAboveMax = FALSE
VehicleTryKeepTimeGap = FALSE
NumberOfSetCruise = 1
CruiseSpeedAtMax = FALSE
ObstacleDistance = ODnone
ObstacleStatusJustChanged = TRUE
CCInitialisationInProgress = TRUE
CruiseSpeedChangeInProgress = FALSE
ObstaclePresent = FALSE
ObstacleRelativeSpeed = RSnone

CruiseBecomesNotAllowed
SetCruiseSpeed
SetCruiseSpeed
SetCruiseSpeed
SetCruiseSpeed
CCInitialisationFinished
CCInitialisationDelayFinished
CruiseOff
ExternalForcesBecomesNormal
VehicleLeavesCruiseSpeed
VehicleExceedsMaxCruiseSpeed
ObstacleAppearsWhenCruiseActive
ObstacleAppearsWhenCruiseActive

CCInitialisationFinished
SetCruiseSpeed
CruiseBecomesAllowed
INITIALISATION(FALSE,FALSE,FALSE,FALSE

Fig. 5. ProB animating a B-Machine: The B-Machine code, the B-State (values and constants), enabled operations, history

approximately 1,000 hours of work. It consists of over 7400 lines of code and 12000 lines of test-code.

Difficult parts of the implementation where type checking and the execution of nondeterministic substitutions. Especially the type checking implementation via unification consumed some time to assure compatibility with ProB. As usual small bugs caused by missing specification details inside the implementations consumed a lot of time.

Development was done using a version-control system. The progress of the project can be tracked via the commit-messages[2] of the git repository. Figure 8 shows the project time line.

4.2 Testing and Validation

pyB has been developed using test-driven development (TDD). The process of TDD is as follows:

1. Test-code is written for an unimplemented new feature. This code is called test-case.

[2] https://github.com/hhu-stups/pyB/commits/

```
/* 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
```

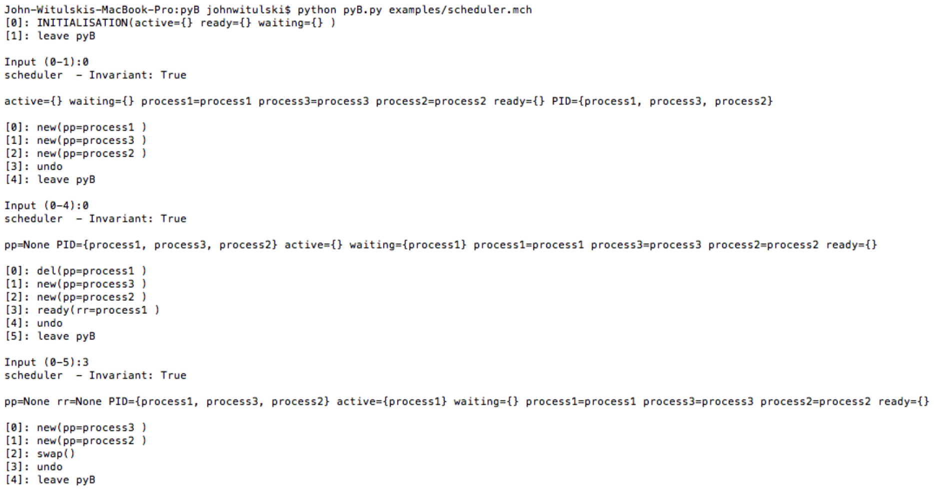
Fig. 6. A simple input example of **PyB**: This file contains all values and constants of a B-State, The first line #PREDICATE was added for parsing reasons

2. The first run (execution) of the test-case fails. I.e all assertions inside this test-case are false. This prevents the programmer from implementing an already implemented feature.
3. The feature is implemented and the test passes: All assertions of the test-case are true.
4. The code is refactored.
5. All previous written test-cases also pass. This ensures that the new implementation has not destroyed any previous functionality.

This process is automated to some degree. Because the tool was implemented by only one programmer, the distributed aspects and advantages of TDD are omitted in this overview. In early development-stages ASTs where constructed explicit by creating tree-nodes as input to the interpreter. Examples of assertions where simple arithmetic or boolean properties. When the development of the tool proceeds, the test-case become more complex.

After the successful usage of the Java parser, ASTs were created automatically by the parsing-module. Inputs also become more complex. Starting with easy inputs like ‘x=y+42‘, the tests quickly also includes sets, functions and relations. All this easy tests are still present and passed by pyB

At the current development new test-cases are full B-machines. Assertions are not longer made just on single predicates, but made up of whole B-machine states. Examples of assertions are true or false properties/ assertions/ invariants after the B-machine initialization, enabled or disabled operations inside a specific state or after some animation steps and of course the test if pyB gets the same result as ProB.



```
John-Witulskis-MacBook-Pro:pyB johnwitulski$ python pyB.py examples/scheduler.mch
[0]: INITIALISATION(active={} ready={} waiting={} )
[1]: leave pyB

Input (0-1):0
scheduler  - Invariant: True

active={} waiting={} process1=process1 process3=process3 process2=process2 ready={} PID={process1, process3, process2}

[0]: new(pp=process1 )
[1]: new(pp=process3 )
[2]: new(pp=process2 )
[3]: undo
[4]: leave pyB

Input (0-4):0
scheduler  - Invariant: True

pp=None PID={process1, process3, process2} active={} waiting={process1} process1=process1 process3=process3 process2=process2 ready={}

[0]: del(pp=process1 )
[1]: new(pp=process3 )
[2]: new(pp=process2 )
[3]: ready(rr=process1 )
[4]: undo
[5]: leave pyB

Input (0-5):3
scheduler  - Invariant: True

pp=None rr=None PID={process1, process3, process2} active={process1} waiting={} process1=process1 process3=process3 process2=process2 ready={}

[0]: new(pp=process3 )
[1]: new(pp=process2 )
[2]: swap()
[3]: undo
[4]: leave pyB
```

Fig. 7. Console output of PyB while animation a B-model. PyB prints the current state, the status of the invariant and the enabeld operations

Of course TDD can not guarantee the same reliability than a formal proof of PYB. But it still makes PYB more reliable and it is a much better approach than testing the tool afterwards: TDD reveals errors inside an implementation very soon.

5 Case Studies and Empirical Evaluation

5.1 Alstom Case Study

The case study consists of 6 industrial B-machines provided by Alstom. Every machine was model-checked with ProB. Two machines where faulty. They defined a partial surjection to an infinite set and initialized it with the empty set. After the correction to a partial function the machine still contains a deadlocked state.

The procedure of the double-checking was performed as follows:
Every machine was animated n-times with ProB. After every animation-step the state of the machine (only constants and variables) where written to a file. The data was loaded by PYB. PYB evaluated the properties and invariant of the machine and returned the result.

After the configuration of some tool properties (maximum size of integer-sets), PYB successfully checked 3 of 6 B-machines by double checking of 32 to 42 states of the machines in 5 minutes per machine. One B machine doesn't work with PYB because of the missing support of external functions like append (on B strings). The remaining machines fail at the same point as ProB (described above). The animation with PYB alone (without ProB's information) of all machines fails.

Fig. 8. Project time line

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
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
November 2013	added symbolic representation for large and infinite sets
December 2013	some systerel (industrial B-machines) successfully checked with pyB/ProB

5.2 Performance Evaluation

pyB is still in an early development phase. It may be possible to speed up checking by replacing data transfer by a socket communication (about 40% of the runtime) with ProB or refactoring the tool using object dispatching instead of the expensive Python “isinstance” built-in. Also using the pypy just-in-time compiler technology on pyB seams promising

Also the generation of large sets induces a serious performance issue. The evaluation of a cartesian product of two sets or the calculation of the power set of a set of 19 elements needs more than 7 seconds. The calculation of the power set of 22 elements already needs more than 440 seconds. Some performance issues can only be solved by better constraint solving.

6 Related and Future Work

6.1 Related Work

PredicateB and PredicateB++ are similar tools to pyB. In contrast to pyB they only evaluate predicates and need additional software like ovado[3] [LBL12] or the DVT tool to validate data like B-states. Predicate B (written in Java) and PredicateB++ (in C++) were created by the company ClearSy in 2005 and 2008. They have been successfully used by Brama [Ser07] inside the Rodin Platform [ABH06]. Another, recent tool is the JEB animator [YJS12] written in JavaScript. Systerel + Ovado [BDP12]

Outside of the B community, there is of course considerable work on proving tools correct. We just want to mention the grand challenge of the verifying compiler [Hoa03] and the work on the Compcert verified compiler (e.g., [TL08]).

6.2 Future Work

The most important issue that has to be solved, to guarantee the correct checking of solutions generated by other tools (like ProB), is the handling of large or infinite sets and the implementation of better constraint-solving. Even if all variables and constants of a state are known (calculated by an other tool), there may be (quantified) predicates for which checking could cause the generation of large sets.

At the current development state there are critical performance issues. Some of them will be solved by a more efficient implementation of the tool. Other problems are a result of the choice of Python as implementation language. These problem can be solved by refactoring the tool to R-Python, a statically typeable subset of Python which can be translated to c using the pypy technology. Performance can also be increased by the use of a better constraint-solver.

A completely different aspect of this research is to see how fast can this tool be by using the pypy just-in-time compiler technology. The tool is not far away from becoming a full model-checker when a constraint-solver is successfully integrated in this project. Also the tool is able to animate B-machines. This will be useful to check if the animation of ProB is correct, i.e if the same states are enabled and the same deadlocks are found.

6.3 Conclusion

pyB has been successfully used to validate data generated by ProB for many simple B machines and some more complex, industrial B machines. Once all performance issues are solved, we will have a reliable, independently developed, double chain for model checking and data validation for B models.

[3] More Informations about Ovado and PredicateB at http://www.data-validation.fr/

References

[ABH06] Jean-Raymond Abrial, Michael Butler, and Stefan Hallerstede. An open extensible tool environment for Event-B. In Zhiming Liu and Jifeng He, editors, *Proceedings ICFEM'06*, LNCS 4260, pages 588–605. Springer-Verlag, 2006.

[Abr96] Jean-Raymond Abrial. *The B-Book*. Cambridge University Press, 1996.

[BDP12] Frédéric Badeau and Marielle Doche-Petit. Formal data validation with event-b. *CoRR*, abs/1210.7039, 2012. Proceedings of DS-Event-B 2012, Kyoto.

[CEN11] CENELEC. Railway applications – communication, signalling and processing systems – software for railway control and protection systems. Technical Report EN50128, European Standard, 2011.

[Cle09] ClearSy. *Atelier B, User and Reference Manuals*. Aix-en-Provence, France, 2009. Available at `http://www.atelierb.eu/`.

[Hoa03] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J.ACM*, 50(1):63–69, January 2003.

[LB03] Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[LB08] Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[LBL12] Thierry Lecomte, Lilian Burdy, and Michael Leuschel. Formally checking large data sets in the railways. *CoRR*, abs/1210.6815, 2012. Proceedings of DS-Event-B 2012, Kyoto.

[LFFP09] Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. Automated property verification for large scale B models. In A. Cavalcanti and D. Dams, editors, *Proceedings FM 2009*, LNCS 5850, pages 708–723. Springer-Verlag, 2009.

[LFFP11] Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. Automated property verification for large scale b models with ProB. *Formal Asp. Comput.*, 23(6):683–709, 2011.

[PL10] Daniel Plagge and Michael Leuschel. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. *STTT*, 11:9–21, 2010.

[Ser07] Thierry Servat. Brama: A new graphic animation tool for B models. In Jacques Julliand and Olga Kouchnarenko, editors, *Proceedings B'2007*, LNCS 4355, pages 274–276. Springer-Verlag, 2007.

[TL08] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th symposium Principles of Programming Languages*, pages 17–27. ACM Press, January 2008.

[YJS12] Faqing Yang, Jean-Pierre Jacquot, and Jeanine Souquières. The case for using simulation to validate event-b specifications. In Karl R. P. H. Leung and Pornsiri Muenchaisri, editors, *APSEC*, pages 85–90. IEEE, 2012.