Implementation Report

This document describes the implementation-details of a B-method tool written in python by John Witulski. The tool is a simple checker of solutions generated by the proB tool and will also be used as a tool for animating b-machines e.g to perform model-based testing of adhocracy or even direct execution of b code as part of the adhocracy project. In this text, the python B-tool will be called 'pyB'.

TODO: general informations about pyB and the b method, data-verification/validation

0. Index:

1. Tool SetUp and Configuration
2. Parsing of B files and future implementation
3. Processing of definitions
4. TypeChecking
5. Statespace and state-management, internal data representation
6. Enumeration and B-machine animation
7. UI, repl. and pretty printer
8. Interpretation and quick evaluation
9. Reasons for failure
10. Case Study: Alstom
11. Related work
12. Future work

1. Tool SetUp and Configuration

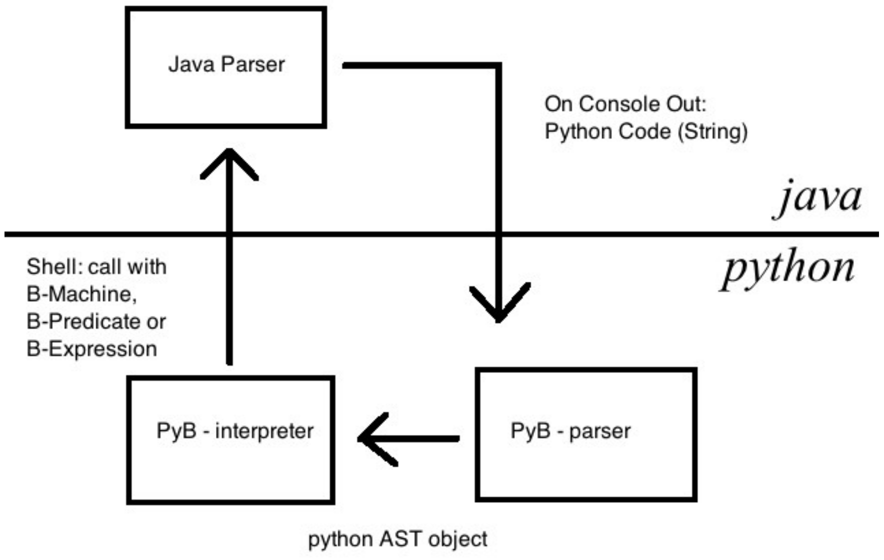
The Tool needs at least python 2.7 (json python module) and a java vm 5. To build the jars yourself you also need gradlescript (*./gradlew -jar*), check out the parser, replace the java files, move the visitors to bparser/src/main/java/de/be4/classicalb/core/parser/analyse/python (or json).
If desired, the tool can be configured via config.py.

2. Parsing of B files and future implementation

To avoid the new implementation of a bparser, pyB uses (a fork of) the same parser like proB. This parser is written in java (sablecc and some classes) and his jar files can be generated by a gradle buildscript. So an installed jvm is a requirement for parsing (in pyB).

After pyB calls the jvm/jars, a special python-visitor (ASTPython.java) prints the AST as python objects to the console. The dynamic exec python command generates the objects (on the python level) from the string for further processing.

To enable a translation of pyB via pypy it is necessary to eliminate all dynamic constructs at runtime. To make this possible a second json visitor has already been implemented. The parsing of a json-AST instead of a python-AST can be implemented without the dynamic exec. The python side of the json-parser is not fully done, while the visitor implementation on the java level has been finished.



3. *Processing of definitions*

The B definitions can be compared to C-makros. Before the typechecking and interpretation is done, pyB runs some recursive definition replacement functions over the parsed AST.

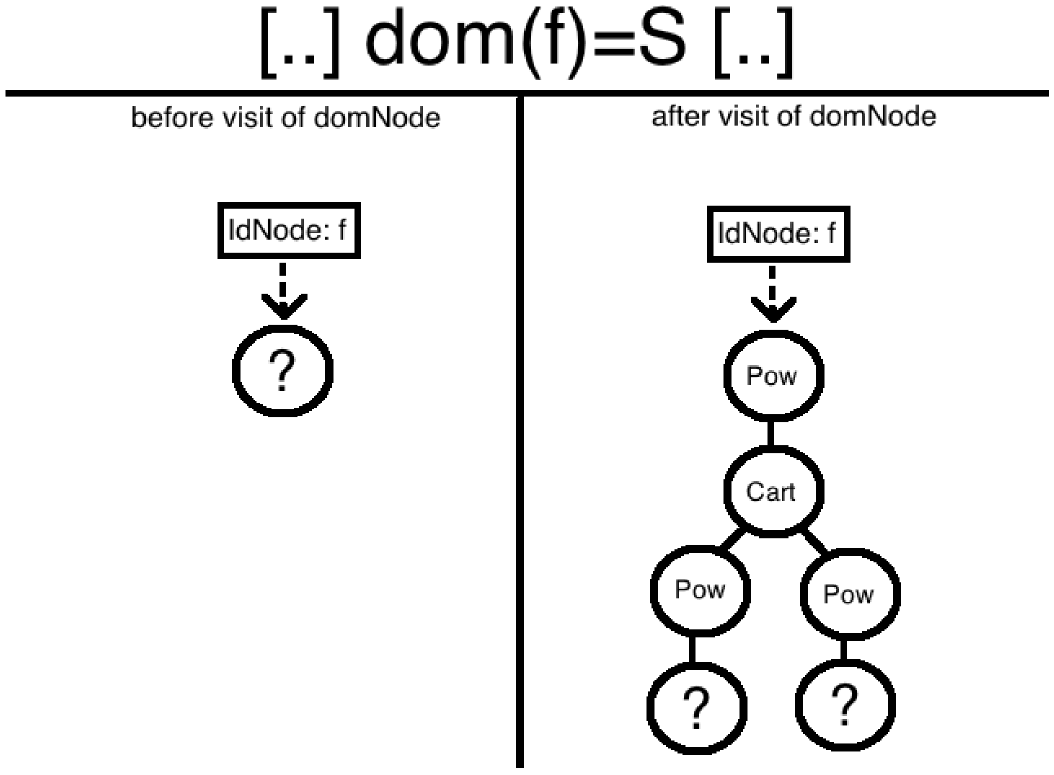
The definition processing consists of two phases. The first phase collects all expression-, predicate- and substitution-definitions defined in the DEFINITIONS section into a map. The second phase searches the remaining AST for matching definitions and replace the sub-ASTs with the saved definition-body

4. *TypeChecking*

It is necessary to evaluate quantified predicates, lambda expressions and set comprehensions to animate b-machines by pyB, but also to check proB solutions, because the prob-solution-files do not contain any informations about bound variable values inside such predicates. This is the reason why pyB needs typeinformations.

Like proB, pyB uses unification to gain typeinformations. The goal is the feature to type every predicate by pyB, that can be typed by proB i.e. the order of subpredicates inside a predicate doesn't effect the typing-success. Unlike prolog (proB) python (pyB) has no build in unification, so it is more complicated to write such a typechecker. Another problem are the possible identical names of identifiers in different scopes and the ambiguity of the subtraction and multiplication of integers or sets.

The typechecker walks the AST and collects informations about the nodes via unification. The typechecking method returns for every subtree (or leaf) a b-type object or an unknown-type object. Unification is implemented by a pointer of an unknown-type object to an concrete type object whenever possible. One important case is the visit of an identifier node: in this special case the idNode (**not** the id name) is mapped to a type object. This could be a unknown type object that points to a concrete or another unknown type object. After the processing of the hole predicate, every chain of unknown types objects must point at a concrete type or the typechecking fails. This final mapping of idNodes to b-types is called resolve, but it turned out that a single resolve-phase after the processing of a scope doesn't succeed in all cases. So every (unknown) type object is unified with an expected type corresponding to the visited node(see Figure 2), to gain information even before the resolve-phase. So the processing of an AST-node returns at least the minimum typeinformations known at this time.



In the resolve-phase every idNode is mapped to an basic (string, integer, boolean, set) type object or an composed (cart, power, struct) type object. IdNodes that point to the special unknown type objects (PowORIntegerType type or PowCartORIntegerType) must be resolved in the final resolve-phase or the typechecking fails.

Typechecking is performed before any interpretation for all b-machines and there seen, includes, used or extended b-machines.

5. Statespace and state-management, internal data representation

The pyB Tools was originally developed to evaluate single predicates. Later it processed expressions, substitutions and than entire b-machines. Today pyB has many modes of operation and what a state is and how it is represented differs from case to case.

In every case pyB holds exactly one environment. This environment encapsulates the state. In case of a simple predicate or expression evaluation the state is just a map from names of identifiers to a value.

In case of the processing of a entire B-machine or even an animation, the state is something more complicated. The environment holds a single statespace object. At this moment it is just a stack of state objects to enable the undo function while animation. If pyB develops to a modelchecker (for some reason), this will be replaced with some other data-structure.

A state object represents the state of the machine at a distinct time. Every state is a stack of maps to enable scoping. The maps map from identifier-names to values. The reason for a push on the map is for example the interpretation of a operation-body or the evaluation of a quantified predicate.

The 'primitive B values' string, integer and bool are represented with the corresponding python types. Sets are represented via the frozenset datatype and not the set datattype to enable set of sets. Functions are represented as sets of python two-tuples. The elements of enumerated sets are represented by strings with the same name.

6. Enumeration and B-machine animation

To evaluate quantified predicated, lambda expressions, set comprehension or enable animation of b-machines it is necessary to generated some sets or find solutions for some predicates.

The interpreter of pyB is designed only to evaluate predicates, compute expressions and perform substitutions on the machine state. This is not enough to enable an efficient animation.

At the beginning pyB used a “brute force enumeration” to check e.g. quantified predicates by generating all values of the quantified variables and simple checking for a false or true predicates in the domain of the variable type and ignoring every constraint inside the predicate. It is easy to construct examples where this strategy leads to big performance problems.

Because of this problems pyB now uses an extern constraint solving module (integer finite domains) which can constrain the size of possible solutions of a predicate while using a pyB B-notation wrapper. This constraint solver uses a (potentially large) domain and a predicate to return some solution candidates. If the constraint solving fails, e.g because of missing implementations inside the wrapper, the bruteforce (generate all values) strategy is used as fallback. Of course every solution is doublechecked by the interpreter.

The animation uses top level precondition and select predicates of operations to check which operations are enabled.

7. UI, repl. and pretty printer

PyB was intended to be a tool just used for checking proB results. A tool like that doesn't need a userinterface.

Today pyB is much more. It is intended to expand its capabilities to animation of B-machines and later partial integration into adhocracy. For this reason there exists a minimalistic ui. The ui is just a console output. If there is a need for a graphical ui in the future it is intended to use an other gui like that of rodin. The first reason is reusing existing work done in the past and the most important second reason is the fact that it is not possible to translate a gui with pypy.

PyB has also a read-eval-print-loop. This mode reads the userinput from the console and give it as an predicate to the parser. If this fails it trys to give it as an expression. The generate AST is typechecked and interpreted. The result is printed to the console.

PyB uses helper methods to print the result of an expression (internally represented by python datatypes) in a b-notation. Never the less pyB has also a pretty printer to print ASTs to the console in b-notation. This is uses for error reporting, e.g. To print the false subpredicate of an invariant to the user.

8. Interpretation and quick evaluation

The pyB interpretation module is designed only as a simple checker. It will be some work to build a model checker out of this tool because of refactoring of the eval. of the non deterministic substations.

The interpreter uses some strategies to avoid the computation of large sets. On is the quick member check. If the left side of a memberOfNode is known, pyB doesn't generate the right side but only checks if the element can be generated by the expression on the left side.

Also the the interpreter trys to use constraint solving when ever possible to produce a better performance.

9. Reasons for failure

TODO:

10. Case Study: Alstom

The case study consists of 6 B-machines. 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 deadlock-state.

The procedure of the double-checking was as follows:
Every machine was animated n times with proB. After every animation 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 correct configuration of the MAX_INT and MIN_INT values 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 machine doesn't works with pyB because of the missing support of extern functions like append (Strings). The remaining machines fail at the same point like proB (described above). The animation with pyB (and without ProB) of all machines fails.

It may be possible to speed up this slow checking by replacing the file by a socket communication (~40% of the time) with proB or using the pypy technology on pyB

TODO: image

11. Related work

TODO: (Ovado, ProB,...)

12. Future work

1. *Use Rodin as userinterface*
2. *Integration with adhocracy to enable model based testing*
3. *Better constraint solving to enable successful animation of more B-machines*
4. *Eliminate all dynamic code to enable pypy translation to C*
5. *more...*

The tool can be downloaded at https://github.com/hhu-stups/pyB

18.04.2013 John Witulski hhu