Booster 2
Manual and Reference Guide

James Welch et al.

Department of Computer Science

University of Oxford

Contents

List of Figures

Part I

Introduction

Chapter 1

Introduction

Chapter 2

The Booster Language

This section is intended to document the Booster syntax. As a reference, the complete abstract syntax is placed in an appendix.

2.1 System, comments

A booster system is described by a single text file, with the system name at the top. The system name is used as a namespace for any generated artefacts: databases, services etc. and hence should be both identifying and unique.

For example:

```
system ComputingLaboratory

// ... The system definition goes here
```

There are no restrictions on the type-senstivity of names in Booster, but convention is to use camel-case, with the initial letter capitalised for the names of systems, classes and sets, while methods and attributes use lower-case letters.

Comments can be (and should be) put into the code in the normal way: using the familiar notation of `//` for a single-line comment, and `/* ... */` for multi-line comments.

```
// This is a single-line comment

/* This is a multi-line
comment */
```

Comments may be placed anywhere inside the text file, and it is recommended that descriptive comments appear before the code that is being described.

2.2 Classes and attributes

Booster may be described as an object-based language: data, functionality and constraints are organised in structures known as classes. A class may be defined within the context of a system, and is given a name, to represent the real-world objects that it will be representing. Within a class, attributes, methods and invariants can be defined within separate sections using the keywords as shown in the example below:

```
class C {
  attributes
    /* attributes for the class C are defined here */
  methods
    /* methods for the class C are defined here */
  invariants
    /* invariants for the class C are defined here */
}
```

2.3 Types and user-defined enumerations

2.4 Methods

2.5 Invariants: Static and Dynamic

2.6 Inheritance

Chapter 3

A language of guarded commands

Chapter 4

SQL and other implementation languages

Part II

Transformations and Heuristics

Chapter 5

Transformations

5.1 Introduction

A 'staged compilation process', similar to that described on Page 8 of [1].

5.2 Structure

- Parse
  - Initialize lookup table
  - Populate lookup table
- Elaborate
  - Insert 'this' (requires that every term is annotated with the class it is contained in, and requires a function to lookup method and attribute names)
  - Generate inputs and outputs
    - Infer Types
    - Deduce Types
  - Populate lookup table
  - Inputs and outputs
  - Qualified invariants
  - Class-based invariants
  - Expanded workflows
  - (Expand Method References)
  - (Expand inheritance)
- Compile
  - 'Program'

    - Calculate postcondition
    - 'WP'

- Simplify
- Translate

5.2.1 Parse

In this initial stage of the process, the tree of abstract terms which comprises the abstract syntax is inserted into a lookup table, for later reference. At this point, only simple inferences are made; any complicated inferences are left until the 'elaborate' part of the process.

Before the lookup table can be created, it must first be initialised, which is useful in later steps: when the lookup-table is pretty-printed after a stage, perhaps for debugging purposes, there is no danger of an uninitialised part of the table.

5.2.2 Elaborate

5.2.3 Compile

5.2.4 Simplify

5.2.5 Translate

5.3 Implementing in Spoofax

5.3.1 Explaining the Booster transformations

Spoofax strategies typically take a 'graph re-writing' pattern: the tree is iteratively recursed, and when a term matching a particular pattern is found, an action is performed - typically a re-written version of the term is replaced in the tree.

However, this approach is not ideal for Booster transformations. Perhaps fundamentally, this is because the model is a graph, not really a tree. When iterating the tree, it is important to have a great-deal of contextual information present - for example, the list of attributes and their types for each class. This contextual information cannot easily be passed as a parameter during the tree exploration: it *could* be placed as annotations at suitable nodes in the tree. The main problem is that often the model may need updating somewhere other than the node currently visited.

These concerns may be illustrated by considering the function which deals with the expansion of the inheritance hierarchy. When a node is found which satisfies the property that

The node is an attribute 'a' in a class 'c1' The class 'c2' is a sub-class of the class 'c1' The class 'c2' does not currently contain the attribute 'a'

In this case it is necessary to have the relevant contextual information available during the examination of a node (which attributes are in which classes (other than the current), and which classes are sub-classes of others. Furthermore, it is necessary to be able to update this information in such a way that it can be used for the remainder of the tree-traversal, for otherwise further applications of this rule may be unapplicable, or, worse, repeated for the same node under the same pattern matching.

Under these constraints, two implementation choices make themselves clear. The first is that a mutable 'lookup-table' is required: a function which may be updated to reflect the latest understanding and is always available within any context.

The second is that the use of annotations in transformations such as this is not immediately useful. Annotations will need re-using and

5.4 A Lookup Table

There's a blurry line between a system and a booster specification. Since there is only one System in scope at any one time, then the lookup table acts as a lookup function for 'System'.

These are the fields that are originally stored in the lookup table.

```
"Name" -> name

"SetDef" -> (name, [elements])

"Class" -> (name, ([subclasses], [attributes], [methods],
                                        [constraints], [workflows]))

"Attribute" -> ((cname, aname), (decorations, type,
                                        (opposite), minmult, maxmult))

"Method" -> ((cname, mname), (constraint, guardedCommand, exts))
```

Where `exts` in `method` is used solely in the elaboration phase, to keep track of which constraints from subclasses have been conjoined into the method definition.

Part III

Using The Booster Language

Chapter 6

Default System

Part IV

The Spoofax Implementation

Chapter 7

Implementation

7.1 Introduction

In this section we discuss the specifics of the Spoofax implementation

7.2 Compilation

The spoofax transformations can be executed in one of two ways: either as a compiled, java program, or as an interpreted CTree. Brief details can be found here: `http://strategoxt.org/Spoofax/FAQ`. By default, new Spoofax projects use the CTree implementation; we prefer the java implementation as it is more portable: in particular, the test framework uses a compiled jar file.

To change between CTree and Java implementations, you need to change the project in two places: the Ant script that compiles the transformations, and the code which tells the editor how to apply transformations to a `.boo2` file.

The first change is in the file `build.main.xml`, and you need to change line 43, which determines the main targets for the ant build. To use a Java implementation, the line must be as follows:

```
<!-- Main target -->
<target name="all" depends="meta-syntax, spoofaximp.default.jar"/>
```

Whereas if you require a CTree interpreter, this line can be re-written:

```
<!-- Main target -->
<target name="all" depends="meta-syntax, spoofaximp.default.ctree"/>
```

Note that of course you could instruct the builder to do both, but this makes the build take twice as long, which is inconvenient when developing and testing the transformations.

The second step is to change which transformation engine is provided by the editor for files with an extension of `.boo2`. To do this, you need to edit the file `editor\Booster2-Builders.esv` and replace the line:

```
provider : include/booster2.ctree
```

with the lines:

```
provider : include/booster2-java.jar
provider : include/booster2.jar
```

Or vice-versa, if you're planning on using the CTree interpreter. Remember that the project will need re-building after these two changes have been made.

To execute the transformations on the command line, you will need the java implementation. Ensure that your files `include/booster2.jar` and `include/booster2-java.jar` are up to date. Then from the top-level directory, you can execute the command (all as one line):

```
java -cp
include/booster2.jar:include/booster2-java.jar:utils/strategoxt.jar
run trans.java-elaborate-booster -i test.boo2
```

where `run` is the java class that gets executed (defined in `strategoxt.jar`), `java-elaborate-booster` is the name of the transformation that you'd like to apply, and `test.boo2` is the input source file. The transformation that gets aplied must be appropriately wrapped in a 'main' function as described in `http://strategoxt.org/Spoofax/CommandLine` which allows input to taken from the command line, and calls the top-level parser (to generate the initial ATree) manually.

7.3 Testing

The tests are defined as *regression tests* - i.e. they are in place to ensure that after any changes to the transformations are made, the existing functionality continues to work as expected. Currently these tests take the form of a booster source file, and a set of expected results: additional source files which should match the expected result of any of the key transformation stages. For example, for a given input file `test.boo2`, there would be suitable result files `test.parsed.boo2`, `test.elaborated.boo2`, etc.

The ANT files which runs the tests can be found in `test/regression/runTests.xml`. This process first empties the results of any previous testing (which are kept so as to be available for inspection after testing), and then runs the test again. For each source file it finds in the `source` directory, it runs each transformation and moves the result into the output directory. It uses the host machine's `diff` function to compare the output with the expected result: using options to ignore white space and empty lines where applicable, to allow for small changes in the output format. Based on the result of this `diff` command, it moves the file into the appropriate `success` or `failure` directory for later inspection.

Finally, the process counts the number of files in the `success` and `failure` directories to provide a test report.

Chapter 8

How-Tos

8.1 Adding a new primitive type

To add a new type to Booster:

- Add it to the Booster2.sdf file, and add a corresponding 'Extent'
- Make sure the syntax is disallowed by the tokenizer (common.sdf)
- Add the rules for the type in library/basicTypes.str
- Ensure there is a corresponding type in the Relational.sdf syntax, and that the output methods are appropriate.

8.2 Creating a new platform-specific output

Part V

Appendices

Appendix A

The Booster Syntax

Appendix B

Transformation Reference Guide

Bibliography

[1] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, 2010.