

Booster 2

Manual and Reference Guide



James Welch et al.
Department of Computer Science
University of Oxford

Contents

I	Introduction	1
1	Introduction	2
2	The Booster Language	3
2.1	System, comments	3
2.2	Classes and attributes	4
2.3	Types and user-defined enumerations	6
2.4	Methods	7
2.4.1	Constraints	7
2.4.2	Expressions	8
2.4.3	Method References	9
2.5	Invariants: Static and Dynamic	10
2.6	Inheritance	10
2.7	Workflows	10
3	A language of guarded commands	11
4	SQL and other implementation languages	12
II	Transformations and Heuristics	13
5	Transformations	14
5.1	Introduction	14
5.2	Structure	14
5.2.1	Parse	15
5.2.2	Elaborate	15
5.2.3	Compile	15
5.2.4	Simplify	15
5.2.5	Translate	15
5.3	Implementing in Spoofox	15
5.3.1	Explaining the Booster transformations	15
5.4	A Lookup Table	16

6	Type Deduction	17
III	Using The Booster Language	18
7	Default System	19
IV	The Spoofax Implementation	20
8	Implementation	21
8.1	Introduction	21
8.2	Compilation	21
8.3	Testing	22
9	How-Tos	23
9.1	Adding a new primitive type	23
9.2	Adding a new expression operator	23
9.3	Creating a new platform-specific output	23
V	Appendices	24
A	The Booster Syntax	25
B	Transformation Reference Guide	26
	References	27

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-sensitivity 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 */  
}
```

For ease of definition, multiple sections of each type may be defined. Each section is optional, although an empty class may trigger a warning in the editor. This subsection focuses on the contents of an `attributes` section; later subsections deal with methods and attributes.

An attribute definition consists in a name, and information about type, multiplicity and symmetry constraints. In addition, there may be additional hints for an interface about how attributes are used.

A simple attribute is defined by giving its name, and its type, separated by a colon (:). Here is an example of a simple attribute:

```
firstName : STRING
```

This defines an attribute whose name is `firstName` and whose type is the basic type `STRING`. In Booster it is conventional to start attribute names in lower case.

There is no separator between attribute definitions, e.g.

```
firstName : STRING  
lastName : STRING
```

Multiplicity information can also be defined. The examples above show simple, mandatory attributes. An attribute may be marked as *optional* by including the type in square brackets (`[]`). An optional attribute may take a value included in the type, or may be given the value `null`, a special value whose presence indicate that the value of the attribute is not defined.

For example, we may define an optional middle name of type `STRING`:

```
middleName : [STRING]
```

We may also allow an attribute to take a number of values. Initially, we simply allow an *unordered set* of values, but in future versions we intend to allow ordered *sets*, *bags*, and *sequences* of values.

To define a set-valued attribute, we use the keyword `SET`, round brackets `()`, and a multiplicity constraint. This defines the maximum and minimum number of values that an attribute may take. For example, to declare that an attribute has between zero and five values, we might define as follows:

```
middleNames : SET( STRING ) [0..5]
```

The syntax for multiplicity constraints is taken from UML. The table below shows the different forms of syntax, and the interpretations:

Description	Syntax Example	Min. Multiplicity	Max. Multiplicity
Min. and Max.	[3 .. 5]	3	5
Max. only	[.. 5]	0	5
Min. only	[3 ..]	3	No Maximum
Min. only	[3 .. *]	3	No Maximum
Fixed	[3]	3	3
Any	[*]	0	No Maximum

Attribute types are discussed in more detail in the next subsection. One attribute type of particular interest is a reference-valued attribute, which is a key concept in object-oriented programming. An attribute may hold values of references to other objects in the system. For example, the attribute `supervisors` of class `Student` may refer to one or more objects of type `Staff`. In `Booster`, this would be defined as follows:

```
class Student
  attributes
    ...
    supervisors : SET(Staff) [1 .. *]
    ...
```

The final constraint we might wish to add to an attribute is that of *symmetry*. This is the property that defines two attributes as *opposites*, we declare this pair of attributes as representing a *bi-directional association*. To illustrate, we assume the attribute `supervisors` defined in class `Student` as above; we define the opposite attribute `supervisees` in class `Staff`. The constraint we wish to capture is that given a `Student` object `s1`, for every `Staff` object referenced in its attribute `supervisors`, `s1` is contained within its `supervisees`.

To define such a bi-directional association, we simply add the opposite attribute name as part of the type definition. For example, the following code fragment implements the paired association described above:

```
class Student {
  attributes
    ...
    supervisors : SET(Staff . supervisees) [1 .. *]
```

```

    ...
}

class Staff {
  attributes
    ...
    supervisees : SET(Student . supervisors) [0 .. *]
    ...
}

```

This ensures the symmetry property between the attributes `supervisors` and `supervisees`.

2.3 Types and user-defined enumerations

Booster has four built-in primitive types: `STRING`, `INT`, `BOOLEAN`, and `DATETIME`. Section 9.1 describes how to add additional primitive types to the language. Class references may also be used as types—as illustrated in the symmetry description above.

It is often the case that enumerations are required to capture values of a particular attribute. In Booster, these enumerations can be specified using the set notation. A set may be defined in a system for use anywhere within it, and is defined outside of any class or workflow definitions.

The following example show the definition and use of an enumerated value:

```

system Calendar

set Weekday { Monday, Tuesday, Wednesday, Thursday, Friday}

class Appointment {
  attributes
    ...
    dayOfWeek : WeekDay
    ...
}

```

The elements of the set may be used as values in the definition of methods or invariants—see the next section for details. It is customary to use upper-case camel case for set names and values.

In the future, it is intended that sets be more flexible: that they may be extended at runtime. Values defined in the model may be used in constraints within the model, but additional values may be added to the running system.

2.4 Methods

All modifications to the data in a Booster-generated system must be performed through method calls: where it can be guaranteed that all business rules and invariants are maintained. Methods are defined in the context of classes, and consist of a name and a constraint upon values in the before- and after-states of the method call. Methods should be contained within a `methods` section, inside the class definition.

The simplest method is the one which is always available, and always succeeds, defined here within a class `C`, and given the name `m1`:

```
class C {  
    ...  
    methods  
        m1 { true }  
}
```

2.4.1 Constraints

Within the curly brackets is a simple logical constraint. This may be the basic constraints `true` and `false` (although typically these do not appear in real-life systems), or may be the combination of other constraints using the familiar conjunction (`&`), disjunction(`or`), implication (`=>`), or negation (`not`). The relational composition operator `;` is also available for more complex method specifications. The table below shows the complete set of logical operators:

Description	Booster Syntax
True	<code>true</code>
False	<code>false</code>
Negation	<code>not</code>
Conjunction	<code>&</code>
Disjunction	<code>or</code>
Implication	<code>=></code>
Composition	<code>;</code>

The simplest form of constraint is a comparison between two expressions. There are a number of comparison operators: equality (`=`), less than (`<`) and greater than (`>`) being the most frequently used. A complete list of comparison operators is given in the table below:

Description	Booster Syntax
Equal	=
Not equal	/=
Set membership	:
Set non-membership	/:
Less than	<
Less than or equal	<=
Greater than	>
Greater than or equal	>=
Strict subset	<:
Subset	<<:
Strict superset	:>
Superset	:>>

2.4.2 Expressions

Expressions make up the two sides of any comparison. The simplest form of expressions are defined as *value expressions*, corresponding to those expressions whose value may be simply evaluated. For example, the integer 1 or the string "Hello, World" are both examples of value expressions.

Expressions may also refer to the values of attributes in the model: for example we may refer to `o.a` to refer to the values stored in the attribute `a` for the object `o`. Path expressions may be created using the dot (`.`) notation: if `o.a` is also an object reference, we might refer to `o.a.b`. The familiar keyword `this` is used to refer to the current object.

Methods may also make use of input values (typically identifying parameters to a method), and output values (typically specifying objects created during the execution of a method). Inputs and outputs are denoted by the syntax `?` and `!` respectively. For example, `a?` represents an input named `a`. Input and output parameters do not need to be declared; instead the compiler finds all inputs and outputs for a given method and deduces their types from their usage. This is explained in more detail in Chapter 6.

Methods may constrain values in both the before- and after-states. For example, a method `m1` may constrain the attribute `a` to be less than 7 before the method is called and to be incremented by 1 after the method is called. In Booster, we denote values in the post-state with the decoration `'`, which will be familiar to users of the Z notation. The constraint described above might be written as:

```
m1 { a < 7 & a' = a + 1 }
```

This states that the value of `a` before the method is called is less than 7, and the value of `a` after the method is called is equal of the value of `a` before the method is called, incremented by one.

There is no need to order predicates within a method constraint, but it is conventional to place those applying just to the precondition first, for ease of reading.

The complete set of possible value expressions are given in the table below:

Description	Booster Syntax Example
Primitive value, such as a string or integer	13, "Hello World!"
Value taken from a defined set	Tuesday
Null value: an undefined optional attribute	null
Type name: the set of all instances of that type	INT, Student, Weekday
Method input or output	a?, b!
Path expression	this . a . b, a? . c
Set of expressions	{ 1 , 3 , 5 , 7 }, {}

Expressions may be combined through the use of operators. The list of operators is constantly changing, and should be adapted to fit the collection of primitive types in use. This is explained in more detail in Section 9.2. A default set of common operators are shown in the table below:

Description	Booster Syntax Example
Head element of a sequence, or a string	head(e)
Last element of a sequence, or a string	tail(e)
Size of a set, sequence, or string	card(e)
Negation of a numeric value	-e
Addition of two values	e1 + e2
Difference between two values	e1 - e2
Multiplication of two numeric values	e1 * e2
Division of two numeric values	e1 / e2
Maximum of two values	e1 max e2
Minimum of two values	e1 min e2
Intersection of two set values	e1 /\ e2
Union of two set values	e1 \/ e2
Concatenation of two sequence or string values	e1 ++ e2

The full syntax of expressions can be found in the appendices.

2.4.3 Method References

In order to simplify and rationalize the definition of method constraints, Booster allows the re-use of one method constraint inside the definition of another method. In this simplest case, this just requires using the method name in place of a predicate. For example, we might define the method `increment` as follows:

```
increment { a' = a + 1 }
```

and then re-use it in the subsequent method `maybeIncrement`:

```
maybeIncrement { a < 5 => increment() }
```

which performs the increment method if the value of attribute `a` has a value less than five before the method is called.

The rounded brackets indicate that we may also pass parameters to this method definition: essentially determining the values of any inputs to the method being referred to. For example, given the method specifications:

```
increment { a' = a + i? }  
  
maybeIncrement { a < 5 => increment(i? = 5) }
```

the method `maybeIncrement` would be equivalent to the following definition:

```
maybeIncrement { a < 5 => a' = a + 5 }
```

The parameter list is a comma-separated list of input names, and the expressions that are to be substituted, using the `=` sign to assert their equivalence.

We may also refer to a method of another object, by substituting a value for `this`. By convention, the notation is slightly different here - we place a path expression in front of the method name. For example, we might define:

```
maybeIncrementOther { a < 5 => obj? . increment(i? = 5) }
```

which includes the specification of `increment` from the class of which `obj?` is a member. Note that this is equivalent to the more verbose:

```
maybeIncrementOther { a < 5 => increment(this = obj?, i? = 5) }
```

which may prove confusing where a method named `increment` may be defined on multiple classes.

2.5 Invariants: Static and Dynamic

2.6 Inheritance

2.7 Workflows

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 re-cursed, 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.

Chapter 6

Type Deduction

Part III

Using The Booster Language

Chapter 7

Default System

Part IV

The Spoofax Implementation

Chapter 8

Implementation

8.1 Introduction

In this section we discuss the specifics of the Spoofax implementation

8.2 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/strategox.jar  
run trans.java-elaborate-booster -i test.boo2
```

where `run` is the java class that gets executed (defined in `strategox.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 applied must be appropriately wrapped in a 'main' function as described in <http://strategox.org/Spoofax/CommandLine> which allows input to be taken from the command line, and calls the top-level parser (to generate the initial ATree) manually.

8.3 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 9

How-Tos

9.1 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.

9.2 Adding a new expression operator

9.3 Creating a new platform-specific output

Part V

Appendices

Appendix A

The Booster Syntax

Appendix B

Transformation Reference Guide

References

- [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.