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 */
}
```

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`.

Finally, we may wish to specify interface-specific annotations that help interfaces present the information. Eventually, we'd expect this information to end up in a separate file.

Decorations are specified in the definition of an attribute, immediately after the attribute name. At the moment the only annotation we have is the 'identity' annotation, which specifies that we'd like to identify objects in the database using these attributes. Note that this does not imply any uniqueness constraints. For example, we might specify that objects of type `Person` can be identified by the components of their name. E.g.:

```
class Person {
  attributes
    firstName (ID) : STRING
    lastName (ID) : STRING
    dateOfBirth : [DATETIME]
}
```

2.3 Types and user-defined enumerations

Booster has four built-in primitive types: `STRING`, `INT`, `BOOLEAN`, and `DATETIME`. Section 10.1 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 conbination 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	`true`
False	`false`
Negation	`not`
Conjunction	`&`
Disjunction	`or`
Implication	`=>`
Composition	`;`

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 ouptut 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 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 10.2. 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`
Muliplication 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

Invariants can be implicit, such as the constraints upon type, multiplicity and symmetry, as defined above. Alternatively, they can be user-defined, in one of the three forms: *static*, *dynamic*, and *derived attributes*. The first of these is the most generic, allowing arbitrary constraints to be placed on inter-connected classes. Dynamic invariants are a special form of static invariants, whose structure allows the heuristics to enhance the behaviour of methods, rather than constraining their availability. The effect of these is discussed fully in Section 11. Finally, derived attributes are a particular class of dynamic invariant whose use is common enough to warrant a special syntax.

2.5.1 Static Constraints

An invariant provides a constraint upon the class in which it is defined. Such a constraint is defined in the `invariants` section of a class, and again, multiple sections

may be defined within a single class definition. Invariants are defined using the same constraint language used for methods, described above.

For example, we may wish to express a constraint that the radius of a circle is always non-negative. Such an invariant could be defined as follows:

```
class Circle {
  attributes
    radius : INT
  invariants
    radius >= 0
}
```

Note that within an invariants clause, the context of any path is always `this`, and there is an implicit quantification which states that this constraint holds for all objects `this` of the class.

Multiple invariants may be defined using conjunction, but for convenience, the Booster language allows a list of invariants which are implicitly conjoined. For example, consider the further-constrained definition:

```
class Circle {
  attributes
    radius : INT
  invariants
    radius >= 0
    radius < 100
}
```

Which describes a class of `Circle` whose objects must have an attribute whose radius is both non-negative and less than `100`.

2.5.2 Dynamic Invariants

TODO!

- Dynamic invariants may refer to post-state attributes.
- There's an implication, where the antecedent gives a condition, and the consequent specifies a further constraint which applies. Typically, this consequent is intended as a postcondition which can be translated into an extra guarded command.

There's an example where if we put something into a set `S`, then we must take it out of `T`:

```
class A
  dynamic invariants
    S' \/ T' /= {} => T' = T' \ S'
```

Another example might be something inspired by aspect-oriented programming. Changing the 'last updated' attribute is fairly easy:

```
class A {
  attributes
    x : INT
    y : INT
    lastUpdated : DATETIME
  dynamic invariants
    x /= x' or y /= y' => lastUpdated' = CurrentDateTime
}
```

2.5.3 Derived attributes

TODO!

A derived attribute is a special form of dynamic invariant. When an attribute's value may always be calculated from the value of other attributes, we could write this as a dynamic invariant with a `true` antecedent. For example, to assert that the value of the attribute `noOfChildren` is always equal to the size of the set of children denoted by the attribute `children`, we could state a dynamic invariant:

```
invariants
  ...
  true => noOfChildren' = card(children')
```

In the case where the attribute `noOfChildren` is not updated in any other way, Booster allows the constraint to be declared as part of the attribute declaration. For example, the constraint above may be specified as:

```
class Parent {
  attributes
    children : SET(Child)[*]
    noOfChildren = card(children)
}
```

Note that for these derived attributes, the type of the attribute is deduced automtically.

2.6 Extension

Extension in Booster allows the modeller to capture the 'is a' relationship between classes. Put simply, if a class `B` extends a class `A`, then any attributes, methods and constraints that describe `A` also describe `B`. Class `B` may also have additional attributes, methods or constraints that describe the differences between it and class `A`.

For example, consider the following intuitive example:

```
class Rectangle {
  attributes
    width : INT
    height : INT
}

class Square extends Rectangle {
  invariants
    width = height
}
```

In practical terms, this would be equivalent to the following definition for `Square`:

```
class Square {
  attributes
    width : INT
    height : INT
  invariants
    width = height
}
```

This notion of extension is a good way of structuring models to maximise re-use. However, it's also beneficial because any method which was applicable to the class `Rectangle` is equally applicable to the type `Square`, for example in the following method:

```
addShapeToCollection { s? : myShapes' }
```

the input parameter `s?` may be an object of either type `Rectangle`, or type `Square`.

Methods in a sub-class may further constrain the method defined in a super-class. For example:

```
class A {
  ...
  methods
    m1 { x' = 5 }
}
class B extends A {

  methods
    m1 { x < 5 }
}
```

In this example, the method `m1` in class `B` receives the following constraint:

```
    m1 { x < 5 & x' = 5 }
```

Note that methods and invariants cannot be weakened in a sub-class: this enforces the 'is-a' relationship and enables a compositional approach to factorizing code.

In Booster it makes sense to allow multiple inheritance. In this case, attributes, methods and invariants from all super-classes are propagated to the sub-class. For example, consider the following model:

```
class Staff {
  attributes
    supervisees : SET(Student . supervisor)[*]
}

class Student {
  attributes
    supervisor : Staff . supervisees

}

class TA extends Student, Staff {
  invariants
    supervisor /= this
}
```

Where an attribute is inherited multiple times (imagine classes `Staff` and `Student` both extend a common class `Person`), then each attribute appears only once.

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]. A series of top-level transformations are defined, that

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 Main

5.2.2 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.

At this point, it is important that the entire domain of the lookup table is initialised with a value. This ensures that any time a part of the lookup table is read - for example in a pretty-printing of the table for debugging purposes - the lookup is guaranteed to succeed and handling the case of an unsuccessful lookup is not necessary.

The top-level parse transformation is show below. The transformation matches the entirety of a system definition, and returns it un-modified.

```
parse:
  System(name, components) ->
    System(name, components)
  with
    // retrieve the system information, component by component
    sets := <map(organise-set)> <getSets> components;
    classes := <getClasses> components;
    classes' := <map(organise-class)> classes;
    attributes := <map(organise-attribute)>
                    <concat><map(arrangeClassAttributes)> classes;
    methods := <map(organise-method)>
                    <concat><map(arrangeClassMethods)> classes;

    // insert all the retrieved values into the lookup table
    rules(
      LookupTable :+ "Name" ->  name
      LookupTable :+ "SetDef" -> sets
      LookupTable :+ "Class" ->  classes'
      LookupTable :+ "Attribute" ->  attributes
      LookupTable :+ "Method" ->  methods
    )
```

5.2.3 Elaborate

5.2.4 Compile

5.2.5 Simplify

5.2.6 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 consistently updating across the entire tree whenever a new piece of information is discovered. Without the knowledge of where you are in the tree, you cannot know how to update the rest of it.

This results in a style of transformations which is typically driven by *side-effects*. A typical top-level transformation may match the definition of a system as a whole, and will return it unmodified. However, upon matching a system definition, information will be taken from the current state of the lookup table, and the lookup table will

be updated with any new information. In such transformations, the value returned is irrelevant, since it will be typically unused by any subsequent transformations.

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, done,
                                        inputs, outputs))
```

Where `exts` and `done` in `method` are 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

Chapter 8

Good Modelling

8.1 Bi-directional associations

Mandatory-to-mandatory associations are difficult to work with, and often imply there is something wrong with the model! Object can't be created, except in pairs; they can't be deleted, except in pairs, and unless some careful swapping is arranged, links are difficult to update. Usually mandatory-to-mandatory associations can be refactored into one conjoined class, or may be better expressed as an optional-mandatory relationship.

Using bi-directional associations is usually the right thing to do. It's very rare that you want to know what's in a set, but you don't want to know what set a thing is in. Navigation on a website is so much better if it works both ways. There are some extreme examples though - consider the model:

```
class Country {
  attributes
    name : STRING
}

class Person {
  attributes
    bornIn : Country
}
```

Here, it may be appropriate to ignore the back-link, since it's unlikely to ever require the set of all people who were born in a country. This illustrates cases where the use of a class is really as an extensible enumeration, perhaps with extra attributes.

If backward-navigability is not required in the interface, it's more appropriate to disable this at the interface level, rather than constraining the model.

8.2 Inheritance and extension

Designed to capture the 'is-a' relationship, not a generic mechanism for extension. For example, a square is a rectangle, and so the extension is appropriate:

```
class Rectangle {
  attributes
    width : INT
    height : INT
}

class Square extends Rectangle {
  invariants
    width = height
}
```

Compare this with the following extension of Rectangle, which doesn't conform to the 'is-a' relationship:

```
class Cuboid extends Rectangle {
  attributes
    depth : INT
}
```

Whilst this makes sense in its current state, imagine a later revision to include the derived attribute `area` in class `Rectangle`:

```
class Rectangle {
  attributes
    ...
    area = width * height
}
```

This makes sense for the class `Square` but doesn't make sense for the class `Cuboid`.

Chapter 9

Implementation

9.1 Introduction

In this section we discuss the specifics of the Spoofax implementation

9.2 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 applied 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.

9.3 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 10

How-Tos

10.1 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.

10.2 Adding a new expression operator

10.3 Creating a new platform-specific output

Part IV

Theory

Chapter 11

Maintaining Invariants

Part V

Appendices

Appendix A

The Booster Syntax

The concrete syntax is presented here for reference. The comments here are included in the source files. The syntax is presented in a depth-first fashion for ease of reading.

A.1 Booster.sdf

We import some definitions from the Abstract Booster Model (which includes syntax for guarded commands), and from the relational model, which includes an SQL syntax.

```
imports Common AbstractBoosterModel Relational
```

The start token for any instance is the `system` token.

```
context-free start-symbols
  System
```

A system consists in the keyword `System`, along with an identifier, and a set of system components.

```
'system' ID ( SystemComponent )* -> System {cons("System")}

System -> BoosterTerm
```

Note that every construct in the language is a 'Booster term', which is a hook to allow some concrete syntax transformations.

System components are either set definitions, or class definitions.

```
SetDef -> SystemComponent
Class  -> SystemComponent

SystemComponent -> BoosterTerm
```

A set definition begins with the keyword `set`, and an identifier. A non-empty list of enumerated values is given, within curly brackets (`{}`).

```
'set' ID '{' {ID ','}+ '}' -> SetDef {cons("SetDef")}
```

```
SetDef -> BoosterTerm
```

A class is defined with the keyword `class` and an identifier. There is an optional extension clause, and then a list of class components, surrounded in curly brackets ({}).

```
'class' ID Extends? '{' ClassComponents * '}'
                                  -> Class  {cons("Class")}

Class -> BoosterTerm
```

An extends clause is the keyword `extends`, and a list of class names, which must be non-empty.

```
'extends' {Extension ','}+ -> Extends {cons("Extend")}

Extends -> BoosterTerm

ID -> Extension {cons("Extension")}
```

A class component is a section containing attributes, methods, invariants or workflows. Each section is prefixed by the appropriate keyword.

```
'attributes' Attribute*        -> ClassComponents {cons("Attributes")}
'methods' Method*              -> ClassComponents {cons("Methods")}
'invariants' Constraint*       -> ClassComponents {cons("Invariant")}
'workflows' WorkflowComponent* -> ClassComponents {cons("Workflows")}

ClassComponents -> BoosterTerm
```

An attribute declaration contains an identifier, an optional list of decorations, and a type declaration. The type is separated by a colon `:`.

```
ID Decoration* ':' TypeDecl -> Attribute {cons("Attribute")}

Attribute -> BoosterTerm
```

At the moment, the only decoration currently supported is that of identity, which is simply the keword `ID`, surrounded in brackets.

```
'(' 'ID' ')' -> Identity {cons("Identity")}

Identity -> Decoration
```

A type declaration uses the 'primitive' type declaration in one of three forms: undecorated, to mean mandatory; in square brackets (`[]`) to indicate optionality; with the keyword `set`, and round brackets, to indicate a set-valued attribute. In the latter case, a mutiplicity is also required in square brackets (`[]`).

```
PrimTypeDecl               -> TypeDecl
'[' PrimTypeDecl ']'       -> TypeDecl {cons("Optional")}
```

```
'set' '(' PrimTypeDecl ')' '[' Multiplicity ']'
                          ->  TypeDecl {cons("Set")}

TypeDecl -> BoosterTerm
```

A primitive type declaration is either one of the basic types; or it is the name of a class, or set definition; or a class identifier with an attribute name, signifying the opposite attribute.

```
BasicTypeDecl -> PrimTypeDecl {cons("BasicType")}

ID            -> PrimTypeDecl {cons("UniDirectional")}
ID '.' ID     -> PrimTypeDecl {cons("BiDirectional")}

PrimTypeDecl -> BoosterTerm
```

A basic type is either a string, an integer, a datetime, or a boolean value:

```
'STRING'   -> BasicTypeDecl {cons("String")}
'INT'      -> BasicTypeDecl {cons("Int")}
'DATETIME' -> BasicTypeDecl {cons("DateTime")}
'BOOLEAN'  -> BasicTypeDecl {cons("Boolean")}
```

Mupliticties can come in many forms. See section 2.2 for more details. Here the different formalisms are enumerated explicitly:

```
INT '..' INT -> Multiplicity {cons("MultMinAndMax")}
    '..' INT -> Multiplicity {cons("MultJustMax")}
INT '..'     -> Multiplicity {cons("MultJustMin")}
INT '..' '*' -> Multiplicity {cons("MultJustMin")}
    INT      -> Multiplicity {cons("MultSingle")}
    '*'      -> Multiplicity {cons("MultAny")}

Multiplicity -> BoosterTerm
```

A method is defined with an identifier, and a constraint within curly brackets ({}). For the purposes of auto-completion, it is useful to define allow an expression instead of a constraint: this is marked as deprecated to warn the user. For the purposes of debugging, a method body may also be defined as a guarded command.

```
ID '{' Constraint '}'     -> Method {cons("Method")}
ID '{' Expression '}'     -> Method {cons("Method"), deprecated}
ID '{' GuardedCommand '}' -> Method {cons("Method")}

Method -> BoosterTerm
```

In its simplest form, a constraint is either `true`, `false`, or a relation between expressions:

```
'true'                          -> Constraint {cons("True")}
'false'                         -> Constraint {cons("False")}
```

```
Relation                          -> Constraint
```

Alternatively, a constraint may be made up of smaller constraints, using the logical combinators *not*, *and*, *or*, *implies*, or *relational composition*. A constraint may also appear in brackets, to denote operator precedence.

```
'not' Constraint                  -> Constraint { cons("Not")}
Constraint '&' Constraint         -> Constraint { cons("And"), assoc}
Constraint 'or' Constraint        -> Constraint { cons("Or"), assoc}
Constraint '=>' Constraint        -> Constraint { cons("Implies"), assoc}
Constraint ';' Constraint         -> Constraint { cons("Then"), assoc}
"(" Constraint ")"                -> Constraint { bracket}
```

More complex combinators are provided in the quantifiers: existential and unviersal. These use the keywords `exists` and `forall` respectively, and declare a quantified variable, a set expression denoting the range, and a constraint:

```
'exists' ID ":" Expression "@" Constraint -> Constraint {cons("Exists")}
'forall' ID ":" Expression "@" Constraint -> Constraint {cons("Forall")}
```

The final type of constraint is that of a method reference, which is defined later.

```
MethodReference -> Constraint {cons("MethodRef")}

Constraint -> BoosterTerm
```

A relation, or comparison between two expressions, is simply the two expressions separated by an operator:

```
Expression BinRel Expression -> Relation {cons("BinRel")}

Relation -> BoosterTerm
```

The list of relational operators are as follows:

```
'='     -> BinRel {cons("Equal")}
'/='    -> BinRel {cons("NotEqual")}
':'     -> BinRel {cons("In")}
'/:'    -> BinRel {cons("NotIn")}
'<'     -> BinRel {cons("LessThan")}
'>'     -> BinRel {cons("GreaterThan")}
'<='    -> BinRel {cons("LessThanEquals")}
'>='    -> BinRel {cons("GreaterThanEquals")}
'<:'    -> BinRel {cons("Subset")}
'<<:'   -> BinRel {cons("SubsetEquals")}
':>'    -> BinRel {cons("Superset")}
':>>'   -> BinRel {cons("SupersetEquals")}
```

An expression is either a simple value expression, or is formed of sub expressions: either unary operators (*head*, *tail*, *card*, or *unary minus*. Or, expressions can be combined with a binary operator - the list of these is defined below. Brackets are also permitted to declare precedence.

```
ValueExpression         -> Expression
'head' Expression       -> Expression {cons("Head")}
'tail' Expression       -> Expression {cons("Tail")}
'card' Expression       -> Expression {cons("Cardinality")}
'-' Expression          -> Expression {cons("Negative")}

Expression BinOp Expression -> Expression {cons("BinOp"), left}

"(" Expression ")"      -> Expression {bracket}

Expression -> BoosterTerm
```

A value expression may be either a basic value, such as an integer or string; a type extent, such as the set of all integers; a path expression; the *null* value; or a (possibly empty) set of further expressions. In the final case, the expressions are presented in a comma-separated list, surrounded by curly brackets ({}):

```
BasicValue                -> ValueExpression {cons("BasicValue")}
TypeExtent                -> ValueExpression {cons("TypeExtent")}
Path                      -> ValueExpression {}
'null'                    -> ValueExpression {cons("Null"), prefer}
"{" {Expression "," }* "}" -> ValueExpression {cons("SetExtent")}

ValueExpression -> BoosterTerm
```

A basic value can either be a string, or an integer:

```
INT             -> BasicValue {cons("Integer")}
STRING          -> BasicValue {cons("String")}
```

A type extent is simply the name of one of the basic types:

```
'string'        -> TypeExtent {cons("String"), prefer}
'int'           -> TypeExtent {cons("Int"), prefer}
'datetime'      -> TypeExtent {cons("DateTime"), prefer}
'boolean'       -> TypeExtent {cons("Boolean"), prefer}
```

A path is either a *path start*, or it is a path followed by a path component, using the standard 'dot' (.) notation.

```
PathStart -> Path
Path '.' PathComponent -> Path {cons("Path"), prefer}

Path -> BoosterTerm
```

A *path start* can be an input, or an output, the keyword *this* in a primed or unprimed fashion, or an attribute identifier, which again, may be optionally primed.

```
Input         -> PathStart
Output        -> PathStart
```

```
This          -> PathStart
ThisPrimed    -> PathStart
ID Decorator? -> PathStart {cons("PathStart")}

PathStart     -> BoosterTerm
```

Inputs are simply an identifier with the `?` decoration; ouptuts are identifiers with the `!` decoration.

```
ID "?" -> Input {cons("Input")}
ID "!" -> Output {cons("Output")}
```

The keyword `this` is used to indicate the current object. It may appear in primed form.

```
'this' -> This {cons("This")}
'this' "'" -> ThisPrimed {cons("ThisPrimed")}
```

The primed decoration is simply the character `'`.

```
"'" -> Decorator {cons("Primed")}

Decorator -> BoosterTerm
```

A path component is simply an attribute name, which may optionally be primed:

```
ID Decorator? -> PathComponent {cons("PathComponent")}
```

For auto-completion purposes, it is handy to understand the unfinished path: the empty path component.

```
  -> PathComponent {cons("PathComponent"), deprecated}

PathComponent -> BoosterTerm
```

The list of binary expression operators is given below:

```
"+"     -> BinOp {cons("Plus")}
"-"     -> BinOp {cons("Minus")}
"*"     -> BinOp {cons("Times")}
"/"     -> BinOp {cons("Divide")}
"max"   -> BinOp {cons("Maximum")}
"min"   -> BinOp {cons("Minimum")}
"/\\"   -> BinOp {cons("Intersection")}
"\\/"   -> BinOp {cons("Union")}
"++"    -> BinOp {cons("Concat")}
```

A method reference consists in a path, which denotes the name of the method, and the object on which the method is to be performed. A list of substitutions is provided in brackets: an input name and an expression it is to be replaced with, separated by an equals (=) sign.

```
Path '(' { ( ID '?' '=' Expression) "," }* ')'
                            -> MethodReference {cons("MethodReference")}

MethodReference -> BoosterTerm
```

A workflow component is either a sequential workflow, or a parallel workflow.

```
SeqWf -> WorkflowComponent
ParWf -> WorkflowComponent
```

Both sequential and parallel workflows are introduced with the keywords `seq` and `par` respectively. An identifier is provided, followed by a colon (`:`). Finally an expression of the appropriate type is given.

```
'seq' ID ':' SeqWfExpression -> SeqWf {cons("SeqWf")}
'par' ID ':' ParWfExpression -> ParWf {cons("ParWf")}
```

A sequential workflow expression is either the keyword `skip`, or the choice between two guarded actions, or a wait expression:

```
'Skip'                                    -> SeqWfExpression {cons("Skip")}

GuardedAction '->' SeqWfExpression
'[]'
GuardedAction '->' SeqWfExpression        -> SeqWfExpression{cons("Choice")}

'WAIT.' INT '.' INT '->' SeqWfExpression -> SeqWfExpression {cons("Wait")}
```

A sequential workflow may also be defined as a reference to another sequential workflow, a guarded sequential workflow, or may be bracketed to resolve ambiguities. These definitions are intended to be syntactic sugar, and should be simplified before any precondition calculation is attempted.

```
ID                                  -> SeqWfExpression {cons("
   WorkflowReference")}
GuardedAction '->' SeqWfExpression -> SeqWfExpression {cons("Prefix")}
"(" SeqWfExpression ")"             -> SeqWfExpression {bracket}
```

A guarded action is simply a guard expression, followed by a method reference:

```
Guard '&' MethodReference -> GuardedAction {cons("GA")}
```

And a guard is a constraint preceded by the keyword `Normal` or `Delayed`. A guard may also be bracketed to resolve ambiguities.

```
'Normal' Constraint  -> Guard {cons("Normal")}
'Delayed' Constraint -> Guard {cons("Delayed")}
"(" Guard ")"        -> Guard {bracket}
```

Parallel workflows may be a reference to another workflow, or the parallel combination of another workflow and a parallel workflow expression.

```
ID                        -> ParWfExpression {cons("Single")}
ID '|||' ParWfExpression -> ParWfExpression {cons("Multiple")}
```

Some priorities are set to help the parser disamiguate complex constraints:

```
context-free priorities
  'not' Constraint              -> Constraint
  > Constraint '&' Constraint   -> Constraint
  > Constraint 'or' Constraint  -> Constraint
  > Constraint '=>' Constraint  -> Constraint
  > Constraint ';' Constraint   -> Constraint
```

A.2 AbstractBoosterModel.sdf

A.3 Relational.sdf

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.