An object-oriented formal specification of a configuration language for railway interlockings

Cydney Minkowitz  
cydney.minkowitz@transport.alstom.com

James Atkiss  
james.atkiss@transport.alstom.com

ALSTOM Signalling Ltd  
Post Office Box 146  
Trafford Park  
Manchester M60 1AX  
United Kingdom

1. Introduction

The Solid-state interlocking system (SSI) [1] is one of the most popular railway signalling systems in use today. It has been successfully installed in several hundred sites around the world, and has a proven safety track record. The concept of SSI was developed in the late 1970s by British Railways Research and Development Division, and the system was engineered by British Railways, GEC-General Signal Limited and Westinghouse Signals Limited in the early 1980s. Since then, the system has been enhanced by the three industrial partners.

Because the system was developed 15 years ago, much of the technology on which it is built is approaching obsolescence. A major project is underway in ALSTOM Signalling Ltd to re-engineer the SSI system using state of the art hardware and software safety techniques.

The re-engineering project is following an approved safety-related development process [2], which does not prescribe the use of particular software methods or notations, but rather encourages the use of a variety of methods that are appropriate to the task at hand. With this in mind, the requirements of the interlocking system have been re-specified using the Fusion method [3], which integrates the features of a number of object-oriented and formal modelling approaches.

The purpose of an interlocking system is to ensure the safe movement of trains [4]. It is a centralised control system that sends and receives information from trackside equipment, such as signals, points and track circuits. States of trackside equipment are stored in memory. The core of an interlocking system is divided into two distinct parts; configuration data for a particular geographic area and a generic interpreter of that data.

Configuration data is prepared specially for each signalling scheme. The data contains both geographic information and the application of signalling principles to that information. The source data is compiled into executable data which is interpreted at run-time. The source data is written using a special-purpose programming language which was designed by signalling engineers for signalling engineers. The language is constructed from expressions and statements. Primitive constructs of the language consist of tests and commands which are used to read and write to the interlocking memory. There is an established base of signalling engineers fluent in the language, so there is no cause to alter the language.
A semantic specification of the language was never written during the original SSI development project. The use of the language is described in a data preparation manual [5]. Although the language description is well written, it is open to misinterpretation and incomplete in parts. A full understanding of the language relies on knowledge gained by experienced signalling engineers. Intricacies of the language are not apparent to novices, especially to the software engineers whose job it is to develop a new compiler and interpreter for the language. A syntax specification of the language is included in the manual, by way of a BNF definition. However, the specification acts more as a description of the original language compiler rather than a description of the language's form.

As part of the re-engineering project, we proposed to provide a precise definition of the language in order to clarify its semantics. Although we were not at liberty to simplify the language's syntax, we intended to simplify its description by specifying it in a way that reflects the language's use.

A number of research projects have applied model-based formal specification techniques to specify the semantics of the interlocking configuration language [6], [7]. We wanted to use a similar technique, but one which would integrate easily with the object-oriented techniques we were using to specify the interlocking system requirements as a whole. The approach we adopted in the end was to use the Fusion modelling notation extended with a small set of mathematical notation comparable to that used in formal specification languages. In that way, the language specification would form a compatible addition to the Fusion model of the system requirements.

The language specification that we produced is composed of a semantics specification and syntax specification. The syntax specification consists of two parts: a concrete syntax specification and an abstract syntax specification.

The concrete specification defines the format of the text used to express the configuration data in source code. The concrete syntax is specified in the conventional Backus-Naur Form.

The abstract syntax specification describes the structure of the configuration data. It also defines language constraints that cannot be expressed easily using BNF notation. The abstract syntax is specified in terms of the elements that make up the data. It is used as a model on which to base the semantics specification. The model is graphically portrayed using the Fusion object model notation. A formal definition of the model is specified in a data dictionary using the mathematical notation extensions.

The semantics of the language are defined in terms of operations that process the elements of the abstract syntax. The operations define how the elements examine or affect the interlocking memory. The semantics are specified formally using the Fusion schema notation. The mathematical notation is used to specify operation pre- and post-conditions.

The rest of the paper is structured as follows. Section 2 presents an overview of the configuration language specification. For explanatory purposes, only a small, simplified extract of the specification is described in this paper. The Fusion object modelling notation defined in [3] is used to describe the essential concepts behind the specification. Section 3 describes the notation used to specify the configuration language. The specification extract is presented in Section 4. Section 5 concludes the paper with a discussion of our experience.
2. Specification Overview

The interlocking configuration data is interpreted with respect to the current states of trackside equipment stored in areas of memory. Each area is allocated to an id which uniquely identifies a unit of trackside equipment. Certain information is stored in the memory areas associated with each kind of trackside equipment. For example, the memory areas allocated to signals store information about the aspects that they currently display, and the memory areas allocated to sets of points store information about their current positions. Thus, for each type of trackside equipment, there is a distinct class of memory associated with it. Three of the memory subclasses are shown in the object model in Figure 2-1.

![Figure 2-1 Memories](image)

Figure 2-2 shows the state information stored for a track circuit, represented as attributes of track circuit memory objects. The first two boolean-valued attributes indicate whether the track circuit is clear or occupied. If a track circuit memory's clear and occupied attributes are simultaneously true and false, respectively, then the corresponding track circuit is considered to be clear. Conversely, a combination of clear and occupied values of false and true indicates that the track circuit is occupied. If each of the attributes is false, then the state of the track circuit is said to be undefined. The boolean-valued attribute available is used to authenticate the track circuit's state. The value of this attribute can be set to false as a result of a maintenance function, indicating that the track circuit is barred from use. Finally, the timer attribute stores the amount of time that the track circuit has been in its current state.

![Figure 2-2 Track circuit memory](image)

The interlocking configuration data contains a number of memory tests and commands that access the memories. Each test and command refers to the identifier of the memory that it accesses. Again, there is a distinct set of forms used to express the tests and commands for each type of memory, as shown by the class hierarchies in Figure 2-3.

![Figure 2-3 Memory tests and commands](image)
A track circuit memory test can assess whether a track circuit memory represents a clear, occupied or undefined state using a state qualifier of 'c', 'o', or 'x'. For example, the following syntax is used to test if the track circuit memory with identifier 'TXY' is clear.

\[ \text{TXY c} \]

In addition, a track circuit memory test can assess whether a track circuit memory has been in a certain state for an amount of time. To do this, it uses a comparator-value qualifier of '> t' or '< t' to check that the state has persisted for greater than or equal to 't' seconds or less than 't' seconds. For example, the following syntax is used to test if the track circuit memory with identifier 'TXY' has been occupied for 2 minutes or more.

\[ \text{TXY o> 120} \]

A track circuit memory command is used to set the state of a track circuit memory to one of clear, occupied or undefined. It also uses a state qualifier of 'c', 'o', or 'x' to achieve this. The following syntax, for example, is used to set the state of the track circuit memory with identifier 'TXY' to undefined.

\[ \text{TXY x} \]

The abstract syntax of the configuration language is defined in terms of the form that the configuration data takes. Figure 2-4 depicts the abstract syntax for track memory tests and commands. A track circuit memory test is cast as one of two forms: track circuit state test, which has a state attribute corresponding to a state qualifier of 'c', 'o' or 'x'; or track circuit timer test, which contains a track circuit state test and has comparator and value attributes corresponding to a comparator-value qualifier of '> t' or '< t'. There is only one form of track circuit memory command, which also has a state attribute corresponding to a state qualifier.

![Figure 2-4 Track circuit memory tests and commands](image)

The interlocking configuration data is interpreted in terms of how the memory tests and commands access the memories. When a memory test is processed, it reads a memory. The same memory may be read by many memory tests. When a memory command is processed, it changes an initial memory into a final memory. By implication, a memory can be changed by at most one memory command. The relationships in Figure 2-5 illustrate the two types of memory access. They are shown as partial relationships, because they do not come into force until the memory tests and commands are processed.
When each memory test and command is processed, an operation is performed to examine or affect a memory. Each memory test that is processed yields a value of true or false. Thus, the meaning of a memory test can be defined in terms of the operation that evaluates it. Similarly, each memory command that is processed performs some changes to the memory. Thus, the meaning of a memory command can be defined in terms of the operation that executes its changes. The definitions of these operations, and the operations that process the other types of configuration data not described here, constitute the semantic specification of the configuration language.

3. Specification Notation

Fusion provides a solid framework for formal specification. The Fusion object modelling notation has a well-defined semantics, where classes are interpreted as sets of objects and relationships as sets of tuples. As a consequence, it is easy to extend the notation in a formal way. Fusion suggests the use of a data dictionary to define information and constraints that are out of place on an object model, and a schema notation for defining operations on objects. It encourages the specification of type invariants and operation pre- and post-conditions, but is not prescriptive about the format used to express these. Thus, it is possible to customise the specification notation to suit the target readership.

The following is a description of the notation we adopted for our specification. Sections 3.1 and 3.2 describe the tabular formats used to define the types and operations of the specification, respectively. Section 3.3 describes the mathematical notation that we improvised in order to specify the conditions on the types and operations. This last subsection is presented in the same introductory manner used in the original text to fully define the notation and to explain formal specification concepts to uninitiated readers.

3.1 Data Dictionary Notation

The data dictionary contains complete definitions for each class and relationship in the object model, including type definitions of attributes. It also defines invariants that are additional to those expressed using the Fusion object modelling notation. Data dictionary entries for classes and relationships are presented as tables with the following headers and contents.
Class: The name of the class.

Description: A natural language description of the objects in the class and a description of any invariant on the objects in the class.

Superclasses: The names of the superclasses of the class.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The names of the attributes of an object in the class.
A natural language description of each attribute.
The definition of the type of value that each attribute may have expressed as either: **Type**, set of **Type**, or sequence of **Type**, where **Type** is the name of a primitive type or class.

Parts:

<table>
<thead>
<tr>
<th>Class</th>
<th>Cardinality</th>
<th>Ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The names of the classes of objects that are part of an object in the class.
For each class of part, the number of parts expressed as a Fusion cardinality symbol.
For each class of part, Yes if the parts are ordered, No otherwise.

Invariant: A condition which the objects in the class must satisfy. The condition is expressed using the mathematical notation described in Section 3.3.

Relationship:

<table>
<thead>
<tr>
<th>Description</th>
<th>The name of the relation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>A natural language description of the relation modelled by the relationship.</td>
</tr>
<tr>
<td>Participants</td>
<td>Role</td>
</tr>
<tr>
<td>--------------</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The names used to distinguish between participants in a relation of the same class.
The names of the classes of the objects that participate in a relation.
For each participating role, the number of participants expressed as a Fusion cardinality symbol.
For each participating role, Yes if each object in the class participates in a relation and No otherwise.

Invariant: A condition which the relation must satisfy. The condition is expressed using the mathematical notation described in Section 3.3.

Any cell in a table without relevant information contains the following.

```
3.2 Schema Notation

The semantics of interlocking data is defined using the Fusion schema notation, extended with conventional mathematical notation described in Section 3.3. Schemas are used to define operations that evaluate or execute data with respect to a memory. A schema is presented as a table comprised of seven rows with the following headers and contents.

| Operation | The name of the operation. N.B. the same name can be used for more than one operation, as long as they are distinguishable by the types of the values that they read and change. |
| Reads | A list of declarations of object values that the operation reads. One of the following forms is used to declare a value that an operation reads. 
value\_i : Type\_i 
value\_i : set of Type\_i 
value\_i : sequence of Type\_i |
| Changes | A list of declarations of object values that the operation reads and writes. Each value is specified with a type as follows. 
value\_j : Type\_j 
If the operation does not change any values, then the following is entered. 
-- 
N.B. the above is shorthand for declaring that the operation reads: initial value\_j : Type\_j 
and writes: final value\_j : Type\_j, where initial value\_j and final value\_j denote an object’s values before and after the operation. |
| Delivers | The type of value that the operation delivers. 
If the operation does not deliver a value, then the following is entered. 
-- |
| Assumesh| A list of conditions which the values that the operation reads and changes must satisfy before the operation. These include conditions on how the values are related. The conditions are expressed using the mathematical notation described in Section 3.3. |
| Result | A list of conditions involving the values that the operation reads and changes. 
If the operation delivers a Boolean value, the result is interpreted as follows. 
*If each of the conditions listed evaluates to true, the operation delivers true.* 
*Otherwise the operation delivers false.* 
If the operation does not deliver a value, the result is interpreted as follows. 
*Each of the conditions listed must evaluate to true after the operation.* 
The conditions are expressed using the mathematical notation described in Section 3.3. |
| Comment | A natural language description of the assumptions and results of the operation. The description is written in terms of how the operation examines and/or affects the values that it reads and changes. |

3.3 Mathematical notation

The conditions defined in the data dictionary entries and schemas are expressed using a notation based on conventional predicate logic and set theory. A description of a portion of the notation used is described below. A complete list of the forms of expressions used is given in Table 3-1, together with their precedences and a brief description of their meanings.
3.3.1 Objects, relations and tuples

The operations in the schemas are defined in terms of values that represent objects of the various types in the object model. An object's value records the current values of the object's attributes and parts. Note that an object's value also records the object's unique identity, which never changes but is used to distinguish between objects whose values are otherwise equal. In the definition of an operation that affects an object named object, the notation:

initial object

denotes the object's value before the operation, i.e. the value that the operation reads. The notation:

final object

denotes the object's value after the operation, i.e. the value that the operation writes. If an operation does not affect an object named object, its value is denoted simply as follows.

object

Objects may be involved in relationships such as the reads and changes relationships. A relation represents a set of groupings of values of objects in a relationship. A binary relation between two object values is expressed using the following notation, where relationship is the relationship name, Class\textsubscript{1} is the class of the first object and Class\textsubscript{2} is the class of the second object.

relationship :: : Class\textsubscript{1} : Class\textsubscript{2}

A ternary relation between three object values is expressed using the following notation where relationship is the relationship name, Class\textsubscript{1} is the class of the first object, Class\textsubscript{2} is the class of the second object and Class\textsubscript{3} is the class of the third object.

relationship :: : Class\textsubscript{1} : Class\textsubscript{2} : Class\textsubscript{3}

For the purposes of the specification, relationships are considered to be uni-directional and therefore the ordering of the classes in a relation is significant. For example, the relation:

reads :: : Memory_test : Memory

is not the same as the relation:

reads :: : Memory : Memory_test.

A relationship involving objects of the same class, such as a changes relationship, has an implied ordering. For example, the relation:

changes :: : Memory_command : Memory : Memory

denotes a relation involving the value of a memory command, and the initial and final values of a memory, respectively.

A tuple represents an ordered grouping of values of objects in a relation, such as a pair of values of a memory test and a memory in a reads relation, or a triple of a memory command value and the initial and final values of a memory in a changes relation. Tuples are expressed as lists of object values between open and closed round parentheses and separated by commas. For example, the tuple:

(memory_test, memory)

represents a pair of object values in the relation: reads :: : Memory_test : Memory; and the tuple:

(memory_command, initial memory, final memory)

represents a triple of object values in the relation: changes :: : Memory_command : Memory : Memory.

The following operations on objects are used within the operation definitions in this document.

object.attribute_name

denotes the value of an object's attribute named attribute_name, e.g. track_circuit_memory.clear is the value of a track_circuit_memory's clear attribute.

object.Class

denotes the set of values that represent the parts of type Class in an object, e.g. track_circuit_timer_test.Track_circuit_state_test is the singleton set containing the value of the track circuit state test in a track_circuit_timer_test.
3.3.2 Sequences

Although not used in the specification extract presented in Section 4, a number of classes in the object model contain parts representing ordered collections of objects. The conventional sequence operators listed in Table 3-1 are used in the specification to express operations on these ordered sets.

3.3.3 Conditions

A condition represents a boolean expression that evaluates to true or false. Conditions are used in data dictionary entries to express invariants on classes and relations, and in schemas to express the assumptions and results of operations. In the data dictionary, a condition acts as a universal rule about the objects in a class or the tuples in a relation. In a schema, the conditions are expressed in terms of the values of the objects, or parts of the objects, that the operation reads and/or writes. The conditions used to express an operation's assumptions act as pre-conditions on these values that must be satisfied before the operation can proceed. The conditions used to express an operation's results act as post-conditions on the values that must be satisfied once the operation is complete. The conditions used to express the results of an operation that examines, rather than affects any, object values, act as tests on those values. The conditions used to express the results of an operation that affects object values act as assertions stating the effects or non-effects that the operation has on those values.

3.3.3.1 Simple conditions

The simplest type of conditions used in the operation definitions are operations on objects that deliver boolean values. For example, the condition:

\[ \text{track\_circuit\_memory\_test}.\text{clear} \]

acts as a test on the boolean valued clear attribute of a track\_circuit\_memory, i.e. the condition evaluates to true if the value of a track\_circuit\_memory's clear attribute is true, and to false otherwise. Similarly, the condition:

\[ \text{final \ track\_circuit\_memory}.\text{clear} \]

acts as an assertion that assigns true to the boolean valued clear attribute of a track\_circuit\_memory.

Simple conditions include inequality operations, which operate on integer values, such as the condition:

\[ \text{track\_circuit\_memory}.\text{timer} < \text{track\_circuit\_memory\_test}.\text{value} \]

which evaluates to true if the value of a track\_circuit\_memory's timer attribute is less than the value of a track\_circuit\_memory\_test's value attribute, and to false otherwise; and equality operations, which each operate on values of the same type, such as the condition:

\[ \text{final \ track\_circuit\_memory}.\text{timer} = 0 \]

which evaluates to true if the final value of a track\_circuit\_memory's timer attribute is equal to 0, and to false otherwise.

The other type of simple condition used in the operation definitions comprises set membership operations, such as the condition:

\[ \text{track\_circuit\_state\_test} \in \text{track\_circuit\_timer\_test}.\text{Track\_circuit\_state\_test} \]

which evaluates to true if the value of track\_circuit\_state\_test is in the set of values of track\_circuit\_timer\_test's track\_circuit\_state\_tests, and to false otherwise.
3.3.3.2 Post-conditions

Some of the operation definitions make use of the results of other operations defined in the specification. For example, the definition of an operation that evaluates a track circuit timer test makes use of the result of an operation that evaluates a track circuit state test. Two forms of notation are used to refer to operation results. The first form is:

```
result of operation reads value_1, ..., value_m
```

which represents the result of the operation named `operation` that reads the values in the list: `value_1`, ..., `value_m`. An example use of this form of notation is the expression:

```
result of evaluate reads track_circuit_state_test, memory
```

in the definition of the operation that evaluates a track circuit timer test, which represents the result of the operation that evaluates a `track_circuit_state_test` with respect to a `memory`. The second form of notation is:

```
result of operation reads rd_value_1, ..., rd_value_m writes wr_value_1, ..., wr_value_n
```

which represents the result of the operation named `operation` that reads the values in the list: `rd_value_1`, ..., `rd_value_m` and writes the values in the list: `wr_value_1`, ..., `wr_value_n`.

An expression of either of the above forms evaluates to a boolean value.

3.3.3.3 Compound conditions

A compound condition is a condition formed from other conditions. Compound conditions include the conventional logical operations of negation, conjunction, disjunction and implication.

Implications are used in operation post-conditions to perform case analyses. In this respect, an implication has the meaning:

```
if condition_1 then condition_2.
```

If the implication is acting as a test, it evaluates to `false` if its right hand condition evaluates to `false` whenever its left hand condition evaluates to `true`. Otherwise, it evaluates to `true`. If the implication is acting as an assertion, it states that the right hand condition must evaluate to `true` whenever the left hand condition evaluates to `true`. For example, the following list of implications performs a case analysis on the value of a `track_circuit_memory_test`'s state attribute in order to test what the values of a `track_circuit_memory`'s attributes should be.

```
track_circuit_memory_test.state = c ⇒
track_circuit_memory.clear and track_circuit_memory.available

track_circuit_memory_test.state = o ⇒
track_circuit_memory.occupied or not track_circuit_memory.available

track_circuit_memory_test.state = x ⇒
not track_circuit_memory.clear and not track_circuit_memory.occupied and
track_circuit_memory.available
```

Similarly, the following list of implications performs a case analysis on the value of a `track_circuit_memory_command`'s state attribute in order to assert what the final values of a `track_circuit_memory`'s attributes should be.

```
track_circuit_memory_command.state = c ⇒
final track_circuit_memory.clear and not final track_circuit_memory.occupied

track_circuit_memory_command.state = o ⇒
not final track_circuit_memory.clear and final track_circuit_memory.occupied

track_circuit_memory_command.state = x ⇒
not final track_circuit_memory.clear and not final track_circuit_memory.occupied
```
3.3.3.4 Qualified conditions

Qualified conditions are used to state conditions involving a collection of object values. There are two types of qualified condition: universally qualified condition and existentially qualified condition.

A universally qualified condition states a condition that each value in a qualifying set must satisfy. The notation used for universally qualified conditions is:

\[
\text{for all } v_1, ..., v_n : T_1, ..., T_n \bullet \text{condition}
\]

where \( v_1, ..., v_n \) are variables from each of the types \( T_1, ..., T_n \). An example of the use of this notation is:

\[
\text{for all } \text{memory_test} : \text{Memory_test}, \text{memory} : \text{Memory} \bullet
\]
\[
(\text{memory_test}, \text{memory}) \in \text{reads} :: : \text{Memory_test} : \text{Memory} \Rightarrow \text{memory} \cdot \text{id} = \text{memory} \cdot \text{id}
\]

which is used to state a rule about \( \text{memory_test} \)s and the \( \text{memory} \)s that they read. That is, for each \( \text{memory_test} \) and \( \text{memory} \) pair involved in a \( \text{reads} \) relationship, the \( \text{id} \)s of the \( \text{memory_test} \) and \( \text{memory} \) are equal.

As the form:

\[
\text{for all } v_1, ..., v_n : T_1, ..., T_n \bullet \text{condition}
\]

is used frequently to express rules of this kind, the following abbreviation is used.

\[
\text{for all } (\text{memory_test}, \text{memory}) \in \text{reads} :: : \text{Memory_test} : \text{Memory} \bullet
\]
\[
\text{memory_test} \cdot \text{id} = \text{memory} \cdot \text{id}
\]

An existentially qualified condition states a condition that at least one value in a qualifying set must satisfy. The notation used for existentially qualified conditions is:

\[
\text{there exists } v_1, ..., v_n : T_1, ..., T_n \bullet \text{condition}
\]

where \( v_1, ..., v_n \) are variables from each of the types \( T_1, ..., T_n \). An example of the use of this notation is:

\[
\text{there exists } \text{track_circuit_state_test} : \text{Track_circuit_state_test} \bullet
\]
\[
\text{track_circuit_state_test} \in \text{track_circuit_timer_test} \cdot \text{Track_circuit_state_test} \text{and result of evaluate reads } \text{track_circuit_state_test}, \text{track_circuit_memory}
\]

which is used to state a property about \( \text{track_circuit_timer_tests} \), i.e. a \( \text{track_circuit_state_test} \) is contained in a \( \text{track_circuit_timer_test} \) and it evaluates to \( \text{true} \) w.r.t. a \( \text{track_circuit_memory} \).

Again, as the form:

\[
\text{there exists } v_1, ..., v_n : T_1, ..., T_n \bullet \text{condition}
\]

is used frequently to express properties of this kind, the following abbreviation is used.

\[
\text{there exists } (\text{track_circuit_state_test}) \in \text{track_circuit_timer_test} \cdot \text{Track_circuit_state_test} \text{and result of evaluate reads } \text{track_circuit_state_test}, \text{track_circuit_memory}
\]

Thus, the property above is expressed more succinctly as follows.

\[
\text{there exists } \text{track_circuit_state_test} \in \text{track_circuit_timer_test} \cdot \text{Track_circuit_state_test} \bullet
\]
\[
\text{result of evaluate reads } \text{track_circuit_state_test}, \text{track_circuit_memory}
\]
<table>
<thead>
<tr>
<th>Notation</th>
<th>Precedence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>1</td>
<td>the value of object</td>
</tr>
<tr>
<td>initial object</td>
<td>1</td>
<td>the value of object before the operation</td>
</tr>
<tr>
<td>final object</td>
<td>1</td>
<td>the value of object after the operation</td>
</tr>
<tr>
<td>relationship :: : Class₁ : Class₂</td>
<td>1</td>
<td>the binary relation named relationship of type Class₁ × Class₂</td>
</tr>
<tr>
<td>relationship :: : Class₁ : Class₂ : Class₃</td>
<td>1</td>
<td>the ternary relation named relationship of type Class₁ × Class₂ × Class₃</td>
</tr>
<tr>
<td>(object₁, object₂)</td>
<td>1</td>
<td>the tuple of object₁ and object₂</td>
</tr>
<tr>
<td>(object₁, object₂, object₃)</td>
<td>1</td>
<td>the tuple of object₁, object₂ and object₃</td>
</tr>
<tr>
<td>attribute names of object</td>
<td>2</td>
<td>the set of attribute names of object</td>
</tr>
<tr>
<td>object.attribute_name</td>
<td>2</td>
<td>the value of object’s attribute named attribute_name</td>
</tr>
<tr>
<td>object.Class</td>
<td>2</td>
<td>the set of values of the parts of type Class in object</td>
</tr>
<tr>
<td>closure of relation</td>
<td>2</td>
<td>the transitive closure of relation</td>
</tr>
<tr>
<td>[value₁,..., valueₙ]</td>
<td>1</td>
<td>the sequence containing value₁,..., valueₙ</td>
</tr>
<tr>
<td>[]</td>
<td>1</td>
<td>the empty sequence</td>
</tr>
<tr>
<td>values of sequence</td>
<td>2</td>
<td>the set of values of sequence</td>
</tr>
<tr>
<td>indices of sequence</td>
<td>2</td>
<td>the set of indices of sequence</td>
</tr>
<tr>
<td>length of sequence</td>
<td>2</td>
<td>the length of sequence</td>
</tr>
<tr>
<td>sequence(index)</td>
<td>2</td>
<td>the value of sequence at index</td>
</tr>
<tr>
<td>first of sequence</td>
<td>2</td>
<td>the first value of sequence</td>
</tr>
<tr>
<td>rest of sequence</td>
<td>2</td>
<td>the subsequence of sequence containing all but the first of the values of sequence in the order in which they appear in sequence</td>
</tr>
<tr>
<td>sequence₁ append sequence₂</td>
<td>3</td>
<td>the sequence containing the values of sequence₂ appended to the values of sequence₁</td>
</tr>
<tr>
<td>sequence₁ subseq sequence₂</td>
<td>3</td>
<td>sequence₁ is a subsequence of sequence₂</td>
</tr>
<tr>
<td>value₁ &lt; value₂</td>
<td>4</td>
<td>value₁ is less than value₂</td>
</tr>
<tr>
<td>value₁ &gt; value₂</td>
<td>4</td>
<td>value₁ is greater than value₂</td>
</tr>
<tr>
<td>value₁ = value₂</td>
<td>4</td>
<td>value₁ and value₂ are equal</td>
</tr>
<tr>
<td>value ∈ set</td>
<td>4</td>
<td>value is a member of set</td>
</tr>
<tr>
<td>result of operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reads value₁,..., valueₙ</td>
<td>4</td>
<td>the result of the operation named operation that reads the values in the list : value₁,..., valueₙ</td>
</tr>
<tr>
<td>result of operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reads rd_value₁,..., rd_valueₙ</td>
<td>4</td>
<td>the result of the operation named operation that reads the values in the list: rd_value₁,..., rd_valueₙ and writes the values in the list: wr_value₁,..., wr_valueₙ</td>
</tr>
<tr>
<td>not condition</td>
<td>5</td>
<td>not condition</td>
</tr>
<tr>
<td>condition₁ and condition₂</td>
<td>6</td>
<td>condition₁ and condition₂</td>
</tr>
<tr>
<td>condition₁ or condition₂</td>
<td>7</td>
<td>condition₁ or condition₂</td>
</tr>
<tr>
<td>condition₁ ⇒ condition₂</td>
<td>8</td>
<td>if condition₁ then condition₂</td>
</tr>
<tr>
<td>for all v₁s : T₁,..., vₙs : Tₙ •  condition</td>
<td>9</td>
<td>for all v₁s of type T₁,..., vₙs of type Tₙ, condition</td>
</tr>
<tr>
<td>for all v₁s ∈ T₁,..., vₙs ∈ Tₙ •  condition</td>
<td>9</td>
<td>for all v₁s in the set T₁,..., vₙs in the set Tₙ, condition</td>
</tr>
<tr>
<td>there exists v₁s : T₁,..., vₙs : Tₙ •  condition</td>
<td>10</td>
<td>there are v₁s of type T₁,..., vₙs of type Tₙ, condition</td>
</tr>
<tr>
<td>there exists v₁s ∈ T₁,..., vₙs ∈ Tₙ •  condition</td>
<td>10</td>
<td>there are v₁s in the set T₁,..., vₙs in the set Tₙ, condition</td>
</tr>
</tbody>
</table>

Table 3-1 Summary of mathematical notation used in the specification
4. Specification Extract

The data dictionary entries and schemas that define the classes, relationships and operations described in Section 2 are presented in this section. The definitions refer to the primitive types listed below.

- **Integer** the set of integer values
- **Boolean** the set of boolean values: \{true, false\}
- **Id** the set of identifiers of trackside equipment
- **Timer** the set of timer values, with an implicit ordering
- **Timer_comparator** the set of comparators of timer values: \{less_than, greater_than_or_equal_to\}
- **Track_circuit_state** the set of track circuit state values: \{clear, occupied, undefined\}

The following are the data dictionary entries for the Memory and Track_circuit_memory classes shown in Figures 2-1 and 2-2.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Superclasses</th>
<th>Attributes</th>
<th>Parts</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Values in memory of the states of trackside equipment. No two distinct memories have the same id value.</td>
<td>--</td>
<td>id: A handle that uniquely identifies a unit of trackside equipment.</td>
<td>--</td>
<td>for all memory, other_memory: Memory • not memory = other_memory ⇒ not memory.id = other_memory.id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Superclasses</th>
<th>Attributes</th>
<th>Parts</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track_circuit_memory</td>
<td>Values in memory of track circuit states.</td>
<td>Memory</td>
<td>available: true iff the track circuit is not blocked.</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>clear: true iff the track circuit is clear.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>occupied: true iff the track circuit is occupied.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>timer: The amount of time (measured in seconds) for which a track circuit has been in its current state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>--</td>
<td>Ordered:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The data dictionary entries defining the abstract syntax of the memory test and command classes described in Figures 2-3 and 2-4 are as follows.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Superclasses</th>
<th>Attributes</th>
<th>Parts</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory_test</td>
<td>Tests that examine the states of trackside equipment.</td>
<td></td>
<td>Name</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>id</td>
<td>A handle that uniquely identifies a unit of trackside equipment.</td>
<td>Id</td>
</tr>
<tr>
<td>Memory_command</td>
<td>Commands that affect the states of trackside equipment.</td>
<td></td>
<td>Name</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>id</td>
<td>A handle that uniquely identifies a unit of trackside equipment.</td>
<td>Id</td>
</tr>
<tr>
<td>Track_circuit_memory_test</td>
<td>The type of track circuit memory tests. Example text corresponding to objects in the class are the following.</td>
<td>Memory_test</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• TXY o</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• TXY c</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• TXY x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• TXY o&gt; 60</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• TXY o&lt; 60</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Track_circuit_state_test</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Description | The type of track circuit memory tests that examine the boolean valued attributes of track circuit memories. Example text corresponding to objects in the class are the following.  
- TXY 0  
- TXY c  
- TXY x |
<p>| Superclasses | Track_circuit_memory_test |
| Attributes | Name | Description | Type |
| state | The state corresponding to a track circuit memory test's qualifier. If the qualifier is o, state is o. If the qualifier is c, state is c. If the qualifier is x, state is x. | Track_circuit_state |</p>
<table>
<thead>
<tr>
<th>Parts</th>
<th>Class</th>
<th>Cardinality</th>
<th>Ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invariant</td>
<td>--</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Track_circuit_timer_test</th>
</tr>
</thead>
</table>
| Description | The type of track circuit memory tests that examine the boolean valued and timer attributes of track circuit memories. Example text corresponding to objects in the class are the following.  
- TXY 0> t  
- TXY c< t  
- TXY x< t |
| Superclasses | Track_circuit_memory_test |
| Attributes | Name | Description | Type |
| comparator | The comparator corresponding to a track circuit memory test's first qualifier. If the first qualifier includes <, comparator is less_than. If the first qualifier includes >, comparator is greater_than_or_equal_to. | Timer_comparator |
| value | The integer value corresponding to a track circuit memory test's second qualifier, e.g. if the second qualifier is t, value is t. | Timer |
| Parts | Class | Cardinality | Ordered |
| Track_circuit_state_test | 1 | No |
| Invariant | for all track_circuit_timer_test : Track_circuit_timer_test •  
there exists track_circuit_state_test ∈  
track_circuit_timer_test.Track_circuit_state_test •  
track_circuit_state_test.id = track_circuit_timer_test.id |
Class : Track_circuit_memory_command

Description : The type of track circuit memory commands. Example text corresponding to objects in the class are the following.

- TXY o
- TXY c
- TXY x

Superclasses : Memory_command

Attributes : | Name | Description | Type |
---|---|---|---|
state | The state corresponding to a track circuit memory command's qualifier. If the qualifier is o, state is o. If the qualifier is c, state is c. If the qualifier is x, state is x. | Track_circuit_state |

Parts :

Class : Track_circuit_state

Cardinality : Ordered :

Invariant :

The data dictionary entries for the memory access relationships shown in Figure 2-5 are as follows.

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Description</th>
<th>Participants</th>
<th>Cardinality</th>
<th>Total</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>reads</td>
<td>A binary relation of type Memory_test × Memory. The ids of a memory_test and the memory that it reads are equal.</td>
<td>Memory_test</td>
<td>*</td>
<td>No</td>
<td>for all ((\text{memory_test}, \text{memory}) \in \text{reads} ) :: (\text{memory_test.id} = \text{memory.id})</td>
</tr>
<tr>
<td>changes</td>
<td>A ternary relation of type Memory_command × Memory × Memory. The ids of a memory_command and the memory that it changes are equal.</td>
<td>Memory_command</td>
<td>1</td>
<td>No</td>
<td>for all ((\text{memory_command}, \text{initial_memory}, \text{final_memory}) \in \text{changes} ) :: (\text{memory_command.id} = \text{initial_memory.id}) and (\text{memory_command.id} = \text{final_memory.id})</td>
</tr>
</tbody>
</table>

Finally, the following schemas define the semantics of the track circuit memory tests and command. The first schema specifies an operation that evaluates a track circuit state test with respect to a track circuit memory, the second schema specifies an operation that evaluates a track circuit timer test with respect to a track circuit memory, and the third schema specifies an operation that executes a track circuit memory command with respect to a track circuit memory.
### Operation: evaluate

**Reads:**
- `memory_test : Track_circuit_state_test`
- `memory : Track_circuit_memory`

**Changes:** --

**Delivers:** Boolean

**Assumes:**
- `(memory_test, memory) ∈ reads :: Memory_test : Memory`

**Result:**
- `memory_test.state = c ⇒ memory.clear and memory.available`
- `memory_test.state = o ⇒ memory.occupied or not memory.available`
- `memory_test.state = x ⇒ not memory.clear and not memory.occupied and memory.available`

**Comment:**
A `track_circuit_state_test` reads a `track_circuit_memory`. The operation that evaluates a `track_circuit_test` performs a case analysis on the value of the `memory_test`'s `state` attribute. If the value is `c`, it delivers `true` if each of the values of the `memory`'s `clear` and `available` attributes is `true`. If the value is `o`, it delivers `true` if the value of the `memory`'s `occupied` is `true` or the value of the `memory`'s `available` attributes is `false`. If the value is `x`, it delivers `true` if each of the values of the `memory`'s `clear` and `occupied` attributes is `false` and the value of the `memory`'s `available` attribute is `true`. Otherwise it delivers `false`.

### Operation: evaluate

**Reads:**
- `memory_test : Track_circuit_timer_test`
- `memory : Track_circuit_memory`

**Changes:** --

**Delivers:** Boolean

**Assumes:**
- `(memory_test, memory) ∈ reads :: Memory_test : Memory`

**Result:**
- `there exists track_circuit_state_test ∈ memory_test.Track_circuit_state_test • result of evaluate reads track_circuit_state_test, memory`
  - `memory_test.comparator = less_than ⇒ not memory.timer = stopped and memory.timer < memory_test.value`
  - `memory_test.comparator = greater_than_or_equal_to ⇒ not memory.timer = stopped and ( memory.timer > memory_test.value or memory.timer = memory_test.value )`

**Comment:**
A `track_circuit_timer_test` reads a `track_circuit_memory`. The operation that evaluates a `track_circuit_timer_test` performs a case analysis on the value of the `memory_test`'s `comparator` attribute. If the value is `less_than`, it delivers `true` if the value of the `memory`'s `timer` attribute is not `stopped` and is less than the value of the `memory_test`'s `value` attribute. If the value is `greater_than_or_equal_to`, it delivers `true` if the value of the `memory`'s `timer` attribute is not `stopped` and is greater than or equal to the value of the `memory_test`'s `value` attribute. Otherwise it delivers `false`. 
<table>
<thead>
<tr>
<th>Operation</th>
<th>execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>memory_command : Track_circuit_memory_command</td>
</tr>
<tr>
<td>Changes</td>
<td>memory : Track_circuit_memory</td>
</tr>
<tr>
<td>Delivers</td>
<td>--</td>
</tr>
<tr>
<td>Assumes</td>
<td>(memory_command, initial memory, final memory) ∈ changes :: : Memory_command : Memory : Memory</td>
</tr>
</tbody>
</table>
| Result            | memory_command.state = c ⇒
|                   | final memory.clear and not final memory.occupied |
|                   | memory_command.state = o ⇒
|                   | not final memory.clear and final memory.occupied |
|                   | memory_command.state = x ⇒
|                   | not final memory.clear and not final memory.occupied |
|                   | final memory.available = initial memory.available |
|                   | memory.available and
|                   | ( not final memory.clear = initial memory.clear or
|                   | not final memory.occupied = initial memory.occupied ) ⇒
|                   | final memory.timer = 0 |
|                   | not memory.available or
|                   | ( final memory.clear = initial memory.clear and
|                   | final memory.occupied = initial memory.occupied ) ⇒
|                   | final memory.timer = initial memory.timer |
| Comment           | A track_circuit_memory_command changes a track_circuit_memory. The operation that executes a track_circuit_memory_command performs a case analysis on the value of the memory_command's state attribute. If the value is c, it writes true to the memory's clear attribute and false to the memory's occupied attribute. If the value is o, it writes false to the memory's clear attribute and true to the memory's occupied attribute. If the value is x, it writes false to each of the memory's clear and occupied attributes. It makes no difference to the memory's available attribute. If the value of the memory's available attribute is true and the operation makes a difference to either of the memory's clear and occupied attributes, it writes a zero value to the memory's timer attribute. If the value of the memory's available attribute is false or the operation makes no difference to either of the memory's clear and occupied attributes, it makes no difference to the memory's timer attribute. |
5. Conclusions

The extract presented in Section 4 is a simplified subset of the language specification used to illustrate the nature of the language and specification style. The actual specification is considerably more complicated in parts and is over 150 pages in its entirety. The language specification is also an addition to the overall requirements specification of the interlocking system, which contains over 50 pages of Fusion models. A comparatively small amount of time (approximately four months) was invested to produce and check the language specification. The return on our investment was a sound basis on which to continue software development. In that respect, the specification work has served two purposes:

- to clarify the syntax and semantics of the language.
- to provide a software specification of the language compiler and interpreter.

The first benefit to be gained from the work was a simplification of the concrete syntax specification, which was revised after the forms of the language constructs were revealed in the abstract syntax model. As a result, there is a direct correspondence between the concrete syntax and abstract syntax. As the abstract syntax also serves as a representation of the executable configuration data that the interlocking system interprets, this will make it easier to design the translation process that the compiler will perform.

The specification was validated by one of the original designers of the language, who desk-checked it against the descriptions in the language user guide. Despite having had little personal experience in the use of formal, or object-oriented, specification techniques (although he had previously instigated the use of formalism for this application [6]), he was able to identify a number of logical errors in the specification. More interestingly, he was able to confirm the specification of a number of details which were described ambiguously, or not addressed, in the user guide. He was aided in his work by the natural language commentary that annotated the specification. With this intention in mind, an attempt was made to paraphrase the specification in a way that closely reflected the mathematics. In some instances, the comments became obtuse due to the complexity of some of the language constructs. In these cases, unless there was recourse to simplify the configuration language, a clear interpretation could only be gleaned from the mathematics.

The formal specification cannot replace the language user guide as a comprehensible description; the latter document is still seen as essential reading for a comprehensive understanding of the use of the language. However the ease in which the formal specification can be communicated to readers untrained in the relevant specification techniques says much about the style of the specification. In particular, the graphical notation used in the object models helps readers to visualise its structure. The mathematical notation was intentionally restricted to one page's worth of familiar and phonetic forms in order to level the learning curve. On the negative side, the use of a non-standard notation meant that we could not take advantage of the tool support provided with many of the propriety formal specification languages.

Finally, we recognise the benefit gained from viewing the language specification as a software specification. In particular, the structure of the specification suggests a natural implementation model for the interpreter. We intend to refine the specification into an implementation for the interpreter using a rigorous rather than formal approach. The target implementation language is SPARK Ada [8]. The classes of the abstract syntax model will be represented by appropriate data types, and the operations will be implemented as procedures that operate on those types. The implementation will be annotated with pre- and post-conditions refined from those in the specification. The static analysis facilities of SPARK Ada will help us to validate the implementation against the specification.
References


Acknowledgements

We would like to thank David Stratton, Rosamund Rawlings, Alan Cribbens, Brian Lewis and Bob Barnard for their advice and comments on the specification.