Formal Specification for Design Diversity: Two Case Histories, One Approach

Cydney Minkowitz

ALSTOM FERROVIARIA S.p.A., Information Solutions
Via Corticella 75, Bologna, 40128 Italy
cydney.minkowitz@transport.alstom.com

Abstract. Diverse programming is a recommended approach in the preparation of logic used to drive railway control systems, whereby different representations and processes are used to configure and validate the logic. This paper describes how two formal specifications have been used for the construction of a precise model of the logic, alternative to those represented using the user and machine notations, and for the construction of software tools to process the logic, following a rigorous refinement approach. The first specification was used to develop a redundant tool to check the results of a logic generator. The second specification was used to verify a logic compiler, both as an abstract representation, to compare against the compiled code, and as the design of a diverse code checker.

Keywords: railway interlocking systems, safety-related software, diversity, model-based formal methods, VDM++

1 Introduction

The Alstom Transport group supplies a family of interlocking systems, known as SMARTLOCK. Each interlocking system is a computer based railway signalling system, whose purpose is to ensure the safe movements of trains. Different hardware and software solutions have been used for the systems, but each is characterized as having a central subsystem containing generic software that processes code describing the interlocking logic for a specific signalling area. The logic for each system is configured using software tools, also developed by Alstom.

The interlocking logic is represented differently for each system. One system, referred to hereafter as System A, processes logic expressed as Boolean equations, which are generated using a rule-based tool, and the logic for another system, referred to hereafter as System B, takes the form of compiled procedural code.

Safety must be assured on all parts of an interlocking system, including the interlocking logic. The safety assurance techniques recommended for the fixed parts of the system are well documented, and the generic software of Systems A and B has been constructed and safety approved on the basis of such recommendations (e.g. the System A software has a primary-checker architecture, where a checker module checks the output of the primary software, and System B uses two-out-of three diverse-modular redundancy, where three software modules check each others' outputs). With regards to the safety assurance of the configuration data, the past publications (see [1], for instance) have concentrated on the promotion of appropriate data representations and configuration management procedures, both of which have also been employed for SMARTLOCK.

The best practice for the construction of railway control and protection software is documented in the CENELEC EN 50128 standard [2], which provides guidelines on the techniques to apply for systems with safety integrity levels (SILs) from 0 to 4, of which interlocking systems are rated as SIL4 (i.e. the maximum). Amongst others, the standard highly recommends two techniques for the configurable parts of the SIL4 systems - one is the use of diverse programming, whereby protection against random and systematic faults is achieved through diverse representations and processes leading to the same results, and the other is the use of formal methods.

The taxonomy of diversity techniques in [3] includes the following list of techniques applicable for *design diversity*, the term used for redundant software programs that compute the same outputs from the same inputs.

1. Separate (independent) developments,
2. Diverse development team,
3. Diversity in description/programming languages and notations,
4. Diverse development platforms and tools,
5. Diverse development methods,
6. Different expressions of substantially identical requirements,
7. Diverse requirements and specifications,
8. Different required properties implying the same behaviour,
9. Requiring different behaviours from the diverse versions.

In line with these recommendations, a software development approach using the above techniques has been applied for SMARTLOCK - first to develop a diverse logic generator for System A applications, and second to verify the logic compiled for System B applications, both as an abstract representation, to compare against the compiled code, and as the design of a diverse code checker. In particular, the use of a formal language as a specification notation, and as the basis of a development method, has led to alternative models of the interlocking logic and alternative means of implementing the tools.

Sections 2 and 3 give a description of the interlocking logic used for System A and System B applications and relate the background that motivated the development of the two aforementioned configuration tools and the use of the formal specifications.

Following a formal development approach, the specifications were refined into designs to be implemented directly in code. VDM++ was used as the specification language, and Common Lisp was used as the programming language. Section 4 discusses the rationale for this choice, with example excerpts of the VDM++ specifications of the two configuration tools, and with an explanation of how the approach was specifically applied to each tool.

The conclusions in Section 5 present the results accomplished by applying the approach to the two tools, both in its use as a technique for design diversity and as an effective software development method in general.

2 Diverse Logic Generator

The interlocking logic for a System A application is expressed as a sequence of Boolean equations. Each equation specifies a relation between Boolean-valued input variables and output variables. An input variable represents an indication received by the interlocking from some signalling component, i.e. some mechanism, such as a relay, used to control or monitor some physical device, such as a signal, set of points or track circuit. An output variable represents a command sent from the interlocking to some signalling component. The values of the variables represent the on/off states of the components. The expression defining the relationship between the inputs and outputs is written in terms of the logical 'not', 'and' and 'or' operators (expressed using the symbols ".N.", "*" and "+", respectively). Operating in split second cycles, the System A application continually receives the inputs, processes the Boolean equations, and transmits the outputs.

For example, Extract 1 contains Boolean equations that define the logic for the simple interlocking scheme shown in Figure 1, which contains two routes, RR1_4 from location 1 to location 4 in the right direction, and RL4_1 from location 4 to location 1 in the left direction, where there is one equation for each route.

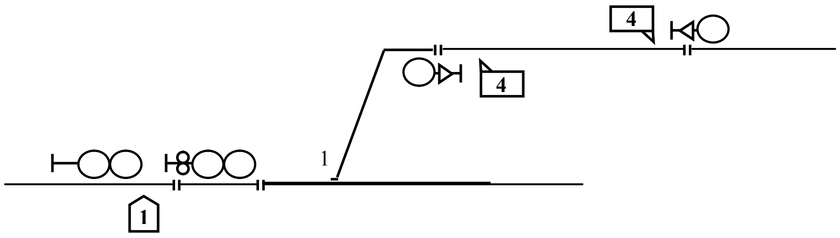

Fig. 1. Example interlocking scheme

```
BOOL    RR1_4_CR      =
                    ( RR1_4_LORCV * P1_4_DI * .N.P1_4_PULL_LORCV *
                      .N.RL4_1_NXC
                    + B1_4_PUSH_LORCV * P1_4_DI * .N.B1_4_PULL_LORCV *
                      .N.RL4_1_NXC )
BOOL    RL4_1_NXC      =
                    ( RL4_1_LORCV * P4_1_DI * .N.B4_1_PULL_LORCV *
                      .N.RR1_4_CR
                    + B4_1_PUSH_LORCV * P4_1_DI * .N.B4_1_PULL_LORCV *
                      .N.RR1_4_CR )
```

Extract 1. Logic expressed as Boolean equations

An alternative way of representing the interlocking logic is to specify the relations between the inputs and outputs using circuit diagrams. A circuit diagram consists of one or more terminal components, representing the outputs, and a network of non-terminal components, representing the inputs, linked together using 'inverse', 'serial' and 'parallel' connections. Because interlocking technology has its roots in relay technology, signalling engineers traditionally use circuit diagrams to design the interlocking logic. Being equivalent representations, it is possible to convert the circuit diagrams to their Boolean equation counterparts. For example, the two circuit diagrams in Figure 2, containing terminal components for setting the routes, and containing non-terminal components to detect whether the routes are requested (via the energised contacts "RR1_4" and "RL4_1"), the points on the routes are controlled in the correct position (via the energised contacts "P1_4" and "P4_1"), the devices used to request the routes are in the correct states (via the buttons "B1_4" and "B4_1") and the opposing routes are not set (via the de-energised contacts "RL4_1" and "RR1_4"), are an equivalent description of the Boolean equations in Extract 1.

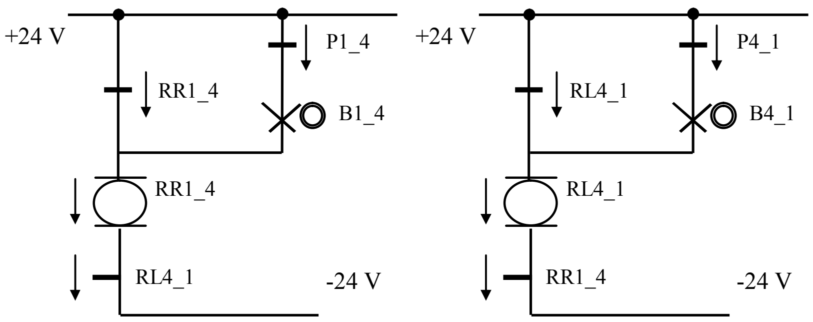

Fig. 2. Logic described as circuit diagram

The circuit diagrams are designed according to signalling principles, which incorporate safety and logistical constraints that govern how the interlocking must work for different railway operating companies. Analysis of the signalling principles has led to the identification of common network compositions, which may be reused for different interlocking applications, and the construction of schemas (generic templates and rules) for the design of the interlocking logic. This, in turn, gave rise to the requirement for a software tool that would generate automatically the interlocking logic from the schemas.

The design of the logic generator tool was commissioned nearly twenty years ago to the University of Bologna by the SASIB Railway Group in Bologna (which later became Alstom Ferroviaria S.p.A.), as part of a research project to investigate the use of rule-based techniques for interlocking data configuration. An early prototype of the tool was produced in the late 1980s using Quintus Prolog. The Prolog program was developed further within the Alstom group in the early 1990s, and has evolved over time to its current state. The program has been used since on numerous interlocking applications for different operating companies.

The program inputs files containing a given interlocking logic schema and files containing Prolog facts denoting the properties and layout of a given signalling area. The schema is expressed in an Alstom propriety meta-language, based on the Prolog syntax, which has features resembling a higher-order predicate calculus. As an example, Extract 2 contains a logic design rule for creating networks used for route setting, expressed in a simplified variant of the meta-language notation.

The program compiles a knowledge base from the schema and facts, constructs the logic and outputs the Boolean equations derived by applying the schema to the facts.

```
network_set_route(Dir, StartLoc, EndLoc) <--
  exists(route(StartLoc, EndLoc, Dir)) and (
    component(relay, route_set(Dir, StartLoc, EndLoc)) or
    component(button, route_set(StartLoc, EndLoc)) ) and
    network(points_controlled(StartLoc, EndLoc)) and
    not component(button, route_cancelled(StartLoc, EndLoc)) and
    if(
      exists(route(OppStartLoc, OppEndLoc, OppDir)),
      opposing_route(StartLoc, EndLoc, OppStartLoc, OppEndLoc) and
      not component(relay, set_route(OppDir, OppStartLoc, OppEndLoc)) )
).
```

Extract 2. Logic design rule in Prolog syntax

Having realized the requirement for the automatic generation of the interlocking logic, there remained the problem of validating the logic. In the early years, the logic was validated by manual inspection of the Boolean equations output. This practice, being time consuming and error prone, could not be sustained in the long term, especially with the increasing size and complexity of the new interlocking applications to come, and so new validation approaches were evaluated, and, in 2001, a decision was made to develop a diverse logic generator. The two tools could then be executed independently on the same input files, generating two output files which could be compared automatically by commercial off-the-shelf file comparison tools (see Figure 3).

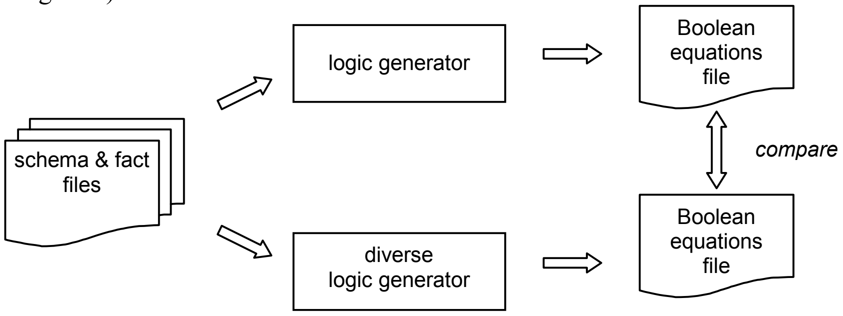

Fig. 3. Logic generation using diverse software

The second tool was to be developed independently from the first tool, using a different team and alternative techniques and tools. Given that the only documentation on the first tool, available at the time when the second tool was conceived, consisted of a user manual and the Prolog program itself, in order to understand the nature of the schema meta-language and how it was to be interpreted, it was considered necessary to begin the development with a precise specification written in a suitable notation, for which VDM++ was chosen.

The main purpose of the specification was to:

– construct a model of the circuit diagrams used to describe the interlocking logic;
– formalise the conversion rules used to generate Boolean equations from the logic;
– construct a model of the schema definitions embodying the templates and rules used to construct the interlocking logic;
– formalise the reasoning mechanisms used to apply the schema to the facts.

So as not to be influenced by the design of the first tool, the specification of the second tool was created from first principles (based on prior knowledge of formal logic and rule-based systems), using the first tool as a black box in order to analyse the required behaviour, by observing its results when applied to example scenarios.

Although the formal specification was used initially as an aid to understanding the requirements, in the end, it served as an alternative representation of the meta-language. For example, Extract 3 shows how the design rule in Extract 2 is expressed in VDM.

```
kb.defineDesignRule(
  "network_set_route",
  ["Dir","StartLoc","EndLoc"],
  kb.guardedDesignInstruction(
    kb.predicate(
     "route",
     ["StartLoc","EndLoc","Dir"]),
  kb.andConstruction(
    kb.orConstruction(
      kb.componentAssociation(
        <relay>,
        kb.componentFunction(
          "route_set",
          ["Dir","StartLoc","EndLoc"])),
      kb.componentAssociation(
        <button>,
        kb.componentFunction(
          "route_set",
          ["StartLoc","EndLoc"]))),
    kb.andConstruction(
      kb.designRuleApplication(
        "points_controlled",
        ["StartLoc","EndLoc"]),
      kb.andConstruction(
        kb.notConstruction(
          kb.componentAssociation(
            <button>,
            kb.componentFunction(
              "route_cancelled",
              ["StartLoc","EndLoc"]))),
        kb.impliedDesignInstruction(
          kb.predicate(
            "route",
            ["OppStartLoc","OppEndLoc","OppDir"]),
          kb.guardedDesignInstruction(
            kb.ruleApplication(
              "opposing_route",
              ["StartLoc","EndLoc","OppStartLoc","OppEndLoc"]),
            kb.notConstruction(
              kb.componentAssociation(
                <relay>,
                kb.componentFunction(
                  "set_route",
                  ["OppDir","OppStartLoc","OppEndLoc"]))))))))));
```

Extract 3. Logic design rule in VDM syntax

3 Diverse Code Checker

The System B interlocking system has been developed in recent years to replace the Solid-state interlocking system (SSI), which, after more than a quarter of century of wide-spread use and proven safety track record, has reached a state of obsolescence due to the out-dated software and hardware configuration technology that it uses.

Using the same design as SSI, the System B system interprets binary code representing the interlocking logic. The code is interpreted on the contents of reserved areas of memories used to record the states of the signalling functions (e.g. signals, points, track sections and routes) controlled by the interlocking. Iterating in cycles, the interlocking receives indications of the current states of the signalling functions, updates the memories accordingly as the logic demands, and sends commands to control the signalling functions to their new states.

The interlocking logic is defined using a procedural, object-centred language, which has been designed to be backwards compatible with the language used for configuring SSI applications. The logic is organized into blocks of code containing tests and commands that access the signalling object memories, which are combined together in conditions and statements, using typical imperative language constructs, to be evaluated and executed by the interlocking.

The logic is prepared as source code. The source code syntax of the memory tests and commands uses mnemonics oriented to signalling engineers. Extract 4 contains an example source code for an execution block used to set a route (with comments describing the code). The source code is compiled to Motorola S3 object code, which is programmed on a memory device to be installed on the interlocking system. Following the design of SSI, the object code instructions are derived directly from the source code syntax, as shown by the listing in Extract 5.

Because Alstom has no access to definitive reference material on the SSI language, and because the language was to be extended to exploit features provided by the new interlocking system, a decision was made to formally specify the new language, in order to understand its semantics, i.e. how it is interpreted by the interlocking, and, as a result, clarify its syntax. The formal specification was constructed in VDM++, based on a formal model created previously using the Fusion notation (see [4]). The formal specification defines, in an object-oriented manner, the essential entities, properties and relations of the code elements processed by the interlocking interpreter.

The formal specification uses an abstract syntax to describe the interlocking logic, in a language that is independent from both the source code and object code notations. An example of the syntax is contained in Extract 6, which contains (commented) code equivalent to that expressed in Extract 4 and Extract 5.

The source code notation includes macro-like constructs, called specials, which, unlike SSI, are not supported by the System B object code interpreter. Also, unlike SSI, the System B interpreter requires the object code blocks to be ordered differently from the source code blocks. As a consequence, before the source code is compiled to object code, a source-to-source translation must be performed, in order to expand the specials to equivalent code using more primitive language constructs, and to order the blocks appropriately. (The formal specification models the interpretable code only and organizes the code blocks in an alternative way using VDM maps – see Extract 10 in Section 4.2.)

As with SSI, it is assumed that the System B object code is generated using a compilation system approved at a SIL4 safety integrity level. SSI ensures this level by using compiler and decompiler tools. For System B, this proved inadequate, owing to the difficulty of decompiling back to the pre-translated source code. Instead, it was considered necessary to compile and decompile to an intermediate representation of the logic, for which the abstract syntax of the formal specification proved ideal.

```
*R5(M)
   if R5(M) a           / if route R5(M) available and
      P2 crf            /    points P2 controlled or free to move normal
   then R5(M) s         / then set route R5(M),
        P2 cr           /      set points P2 controlled normal and
        S5 clear bpull  /      clear signal S5 button pulled
   \
\
```

Extract 4. Execution block in source code syntax

```
memory map:

S5                                 5
P2                                 2
R5(M)                              9

block map:

R5(M)                              14780

instructions:

[N°14780  : 0x0001 14787  14787   => if ]
[N°14781  : 0x0621 9      0       => R5(M) a ]
[N°14782  : 0x0541 2      1       => P2 cnf ]
[N°14783  : 0x0002 0      0       => then ]
[N°14784  : 0x0620 9      0       => R5(M) s ]
[N°14785  : 0x0541 2      0       => P2 cn ]
[N°14786  : 0x040A 5      0       => S5 clear bpull ]
[N°14787  : 0x0008 0      0       => \ ]
[N°14788  : 0x0008 0      0       => \ ]
```

Extract 5. Execution block in object code syntax

```
data.defineExecutionBlock(
   "R5(M)",                  -- block label
   data.statementList([      -- block statements
      data.conditionalStatementList(
         data.conditionList([ -- conditions to evaluate
            data.routeAvailableTest("R5(M)"),
            data.pointsNormalStateTest(    -- Check,
               "P2",                       -- in points memory,
               mk_(                        -- if either
                  {                        -- all data bits, comprising
                   mk_token(<controlN>)},  -- controlN data bit,
                  1),                      -- are set to 1, or
               true)]),                    -- points free to move.
         data.statementList([ -- 'then' statements to execute
            data.setRouteCommand("R5(M)"), -- Set route memory data bit.
            data.pointsNormalStateCommand( -- Assign,
               "P2",                       -- in points memory,
               {                           -- all data bits, comprising
                mk_token(<controlN>)},     -- controlN data bit,
               1),                         -- to 1.
            data.signalStateCommand(       -- Assign,
               "S5",                       -- in signal memory,
               {                           -- all data bits, comprising
                mk_token(<bpull>)},        -- bpull data bit,
               0)]),                       -- to 0.
         nil)]));            -- no 'else' statements to execute
```

Extract 6. Execution block in intermediate code syntax

Figure 4 illustrates the architecture of the compilation system eventually designed to compile and verify System B interlocking logic. The system contains three subsystems - a compiler to compile the logic in object code format (constructed using conventional means), and a redundant compiler and decompiler to verify the logic in intermediate code format, each developed using diverse teams, techniques and tools.

In order to ensure that the interlocking interpreter processes the logic correctly, before converting the source code to object code, the compiler checks the logic to ensure that it is syntactically correct, self-consistent and complete. Whereas, with SSI, similar checks are also performed on the object code by the interpreter itself, with System B, redundant checks are performed on both the source and intermediate code by two separate program components of the redundant compiler. The first program checks that the source code is syntactically correct and compliant to the invariants defined on the intermediate code (to ensure that it can be transformed correctly). The second program checks that the intermediate code is self-consistent and complete.

Design diversity was achieved by separate, independent, developments. The three subsystems were implemented using different programming languages. The compiler was implemented in C, the redundant compiler was implemented using Bison/Flex and C++ (for the source to intermediate code generator) and Common Lisp (for the intermediate code checker), and the decompiler was implemented in Prolog.

Furthermore, different specification documents were produced for the development teams of the three subsystems to follow, written using different notations. The first two documents, used by the compiler and redundant compiler development teams, define the source code syntax (in BNF notation) and the requirements of the compilation system, by specifying informally the consistency, completeness and capacity constraints of the interlocking logic (expressed in natural language on the source code syntax terms) and the source code to source code translation rules and source code to object code conversion rules (expressed in a semi formal notation). The third document, used by the redundant compiler and decompiler development teams, defines the structure of the intermediate code (in the UML notation) and the source code to intermediate code and object code to intermediate code conversion rules (expressed in a semi formal notation). The fourth document, used by the redundant compiler development team, specifies the semantics of the interlocking logic (i.e. the VDM++ specification), which, in order to implement the intermediate code checker program, was extended to define, on the intermediate code, the same constraints that were expressed informally on the source code. In this way, the redundant compiler not only performs a redundant check of the interlocking logic, but it performs the checking diversely on a different representation of the logic.

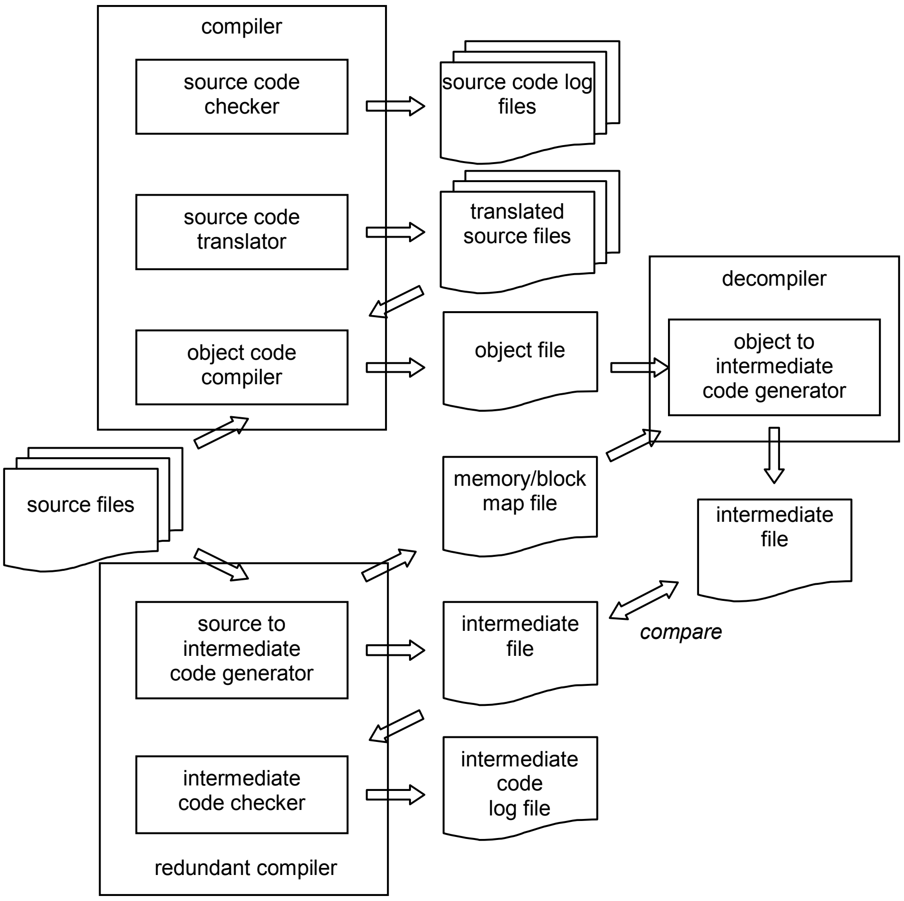

Fig. 4. Code compilation using diverse software

4 Formal Development Approach

The choice of VDM++ [5] as the formal specification language for the tools was based on two criteria: 1) the provision of a standard model-based notation appropriate to the application, which, being based on the ISO/VDM-SL standard, and having object-oriented extensions that were ideal for describing the elements of the interlocking logic and the rules used to generate and check the logic, was easily met by VDM++; and 2) the availability of tool support, which manifested itself in the form of the VDM++ Toolbox, offering syntax and type checking facilities for verifying the specification, an interpreter for validating the specification on test scenarios, and including the Rose-VDM++ Link add-in, rendering it easy to generate UML class diagrams (using the Rational Rose CASE tool) to accompany the formal specifications for documentation purposes.

For those not familiar with the VDM++ notation, Extract 7 and Extract 8 contain simplified extracts of the specification for the diverse logic generator, used to model the components of the interlocking logic and define the rules for constructing the Boolean variables in the equations generated from the logic, which may be understood from the comments (after the ‘--' sign) and the explanation that follows.

```
class Component
-- a Boolean variable is transcribed from the component’s name and
-- an extension string derived from the component’s extension type
   types

      public BooleanVariable ::
                name : seq1 of char
                extStr : seq1 of char
      inv boolVar ==
             boolVar.extStr in set rng exts union {"_CR", "_NXC"};
      public ExtensionType = <state> | <ilv> | <iln> | … ;

   values

      protected exts : map ExtensionType to seq1 of char =
                  { <ilv>|->"_DI", <iln>|->"_LORCV", … };

   instance variables

      protected name : seq1 of char;
      protected extType : ExtensionType;

end Component
```

Extract 7. Example VDM++ class (with abridged extension type and string definitions)

Every component has a name (see Figure 2 of Section 2) and an extension type associated with a descriptive string (for example, the extension types ‘ilv’ and ‘ilv’ are associated with the extension strings “_DI” and “_LORCV”, standing for direct input and locally received input, respectively). The extension type ‘state’, assigned to relays appearing in multiple circuits, is associated with two extension strings: "_CR" and "_NXC", which are used to distinguish whether the values of the corresponding Boolean variables are to be accessed in the current cycle or in the next cycle.

```
class TerminalComponent is subclass of Component
-- a terminal component is assigned to at most one circuit
   instance variables
      -- index of that circuit
      protected ctInd : [ nat1 ] := nil;

   operations
      -- output variable of equation generated from that circuit
      protected
      outputVariable : () ==> BooleanVariable
      outputVariable () == is subclass responsibility;

end TerminalComponent

class NonTerminalComponent is subclass of Component
-- a non-terminal component may be assigned to many circuits
   instance variables
      -- indices of those circuits
      protected ctIndSet : set of nat1 := {};

   operations
      -- input variable of equation generated from a given circuit
      protected
      inputVariable : nat1 ==> BooleanVariable
      inputVariable (ctInd1) == is subclass responsibility;

end NonTerminalComponent

class Relay is subclass of TerminalComponent, NonTerminalComponent
-- a relay may be used both as a terminal or non-terminal component
   operations

      public
      inputVariable : nat1 ==> BooleanVariable
      inputVariable (ctInd1) ==
         let extStr =
                if extType <> <state> then exts(extType)
                else
                   if ctInd = nil then "_CR"
                   else if ctInd < ctInd1 then "_CR" else "_NXC"
         in return mk_BooleanVariable(name, extStr);

      public
      outputVariable : () ==> BooleanVariable
      outputVariable () ==
         let extStr =
                if extType <> <state> then exts(extType)
                else
                   if ctIndSet = {} then "_CR"
                   else
                      if forall i in set ctIndSet & i <= ctInd
                      then "_NXC"
                      else "_CR"
         in return mk_BooleanVariable(name, extStr)
      pre ctInd <> nil;

end Relay
```

Extract 8. Example VDM++ operations

The terminal component of a circuit is transcribed as the output variable of the corresponding equation. Each non-terminal component of a circuit is transcribed as an input variable of the corresponding equation. Each circuit is identified by an index which determines the order in which its corresponding equation is listed.

If the extension type of a relay is not the state type, the extension string of the Boolean variable transcribed from the relay will be that associated with the extension type. Otherwise, the extension string will be determined from the index of the circuit containing the relay, the rule being that if the value of an output variable of one equation is set in the current cycle it may be referenced by an input variable in a later equation in the current cycle, otherwise it may be referenced by an input variable in an earlier equation of the next cycle. Likewise, the value of an output variable of an equation is set for the next cycle if all input variables that reference it are contained in prior equations, otherwise it set for the current cycle. (See Extract 1 of Section 2).

The specifications, which, as stated in the previous sections, were initially constructed as an aid to the understanding of requirements, were refined to software design models, intended to be programmed directly to code. The data structures and algorithms were redesigned to be efficient, without compromising the clarity of the specifications. For example, data types employing suitable data management strategies were introduced to replace the use of predefined VDM data types, class methods were redefined to use state designators to update instance variables represented as sequences and maps in place of concatenation and override operators, and functions using tail recursion were redesigned using accumulator arguments.

Common Lisp was used as the programming language for the following reasons. With its rich dynamic data types and support for both functional and iterative programming styles, it is straightforward to translate specifications expressed in the VDM notation into code. Because functions can be created within code expressions and returned from, and passed to, other functions, it is easy to implement VDM higher-order predicate, set and sequence expressions (such as the ‘forall’ expression in Extract 8, which can be coded using the Common Lisp ‘every’ function). The class types of the specification are realized easily as data types of the Common Lisp Object System (CLOS), which is comparable to the VDM++ object system (for example, in its conception of polymorphism and inheritance). Like VDM++, the Common Lisp language has been made a standard, and good programming environments exist to support it. Allegro CL was selected as the environment, mainly because its incremental compiler enables it, on the one hand, to produce efficient code and, on the other hand, to evaluate the code interactively, in an analogous way to which the VDM++ Toolbox interpreter evaluates specifications.

In general, the major part of a VDM++ specification may be realized as simple applications of standard Common Lisp facilities. Where this is not possible, special purpose functions were provided in a Common Lisp package, named “basics”, which was used as a library by the two programs. In order to formalize the implementation process, a VDM++ to Common Lisp guide was written, which suggests how statements, expressions and data types specified using the VDM++ notation should be written using equivalent predefined Common Lisp facilities, or functions from the basics package, and prescribes conventions for structuring the program code and documenting its interfaces, so as to make it easy to trace back from the program to the specification. Where performance was critical, based on a sound understanding of the specification, a judgment was made to stray from the implementation guide and use more efficient Common Lisp facilities that have similar, but not equivalent, meanings to the recommend ones. For example, many functions in Common Lisp have "destructive" counterparts, which may cause side effects on their arguments, but can be used safely in certain circumstance. Optimizations were also made by careful consideration of the invariants defined on the data types used in the specification.

Rather than generating the program code manually, an evaluation was made of the use of the C++ code generator tool (included in the VDM++ Toolbox Professional version), which generates code built using the services of predefined C++ classes that realize the VDM++ constructs. However, it was decided that the use of the classes would result in code that was less efficient, and less traceable to the specification, than hand-crafted Common Lisp code, and that the small overhead of manual translation, using the implementation guide, and the slight risk of generating code that was non-conformant to the specification, were not outweighed by the potential advantages of a code generator that has no software safety integrity claim and produces less maintainable code.

4.1 Approach Applied to Diverse Logic Generator

Following the prescribed approach, an initial specification of the diverse logic generator described in Section 2 was constructed in VDM++ as a means of understanding the nature of the schema meta-language and how it is interpreted by the Prolog program. Using the VDM++ Toolbox interpreter, the specification was validated by testing it on the example scenarios.

The VDM++ specification was then refined into a design, using approved software design practices, with particular attention to the definition of data structure invariants and operation preconditions. All aspects of the software were specified formally, including user interaction, localization and error handling.

The software provides the following three main functions.

Load the knowledge base	reads the expressions in the input files and converts them into data structures conforming to the specified schema representation.
Generate the interlocking logic	constructs the circuit diagrams that describe the interlocking logic by applying the schema's rules to its facts.
Output the Boolean equations	converts the circuit diagram representation of the interlocking logic into Boolean equations to be output as the result of the application.

The functions are realized in the design by classes organized into self-contained packages, which represent the different layers of the software. Each package is contained in a separate specification document. The packages were assembled together as a project, in the VDM++ Toolbox project, along with predefined VDM++ libraries providing input and output operations and mathematical functions, so that the specification could be checked and interpreted as a whole.

The packages of the software design are summarized as follows.

Interface	provides user operations, including operations to load the knowledge base, construct the interlocking logic and output the Boolean equations.
Interlocking Logic	contains classes used to represent the interlocking logic as circuit diagrams, including classes defining the various types of components and network connections, and operations to construct the logic and format the logic as Boolean equations.
Knowledge Base API	comprises the interface used to define the schema templates and rules and to assert the facts.
Interlocking Logic Schema	contains classes used to represent the schema definitions, and operations to apply the schema to the facts.
Deductive Reasoning	provides the deduction mechanism used by the schema applications, by furnishing classes representing logical conditions and operations for resolving them.
Pattern Matching	provides a mechanism for unifying patterns in the schema definitions against asserted facts or deduced predicates.
Data Management	supplies classes used to store and retrieve data in the knowledge base in an efficient manner.
Support Operations	defines values, types and functions used by the other packages, such as string handling utilities and error codes.

The design choices for the deductive reasoning, pattern matching and data management packages were critical to the performance of the software, and the data structures and algorithms were optimized in these packages during the refinement process.

The knowledge base API package provides operations, corresponding to the commands and expressions of the Prolog based meta-language, to construct the knowledge base. In order to ensure the integrity of the knowledge base, the specification in the package defines the invariants on the schema definitions and the error messages that are output if the invariants are not satisfied by the operation calls

The calls to the operations of the knowledge base API package are made by the interface package as part of the user operation to load the knowledge base. The interface package contains a model of the user interaction of the software, using preconditions that act as guards on the user operations.

The specification was transformed into a Common Lisp program, using the implementation rules and the basics library described previously. The program is organized in Common Lisp packages in line with the specification packages. Because the program uses the same input files as the Prolog program, an additional package was implemented to convert the schema definitions and facts in these file to an alternative representation, based on the VDM syntax used for the specification of the knowledge base API package (see Extract 3 of Section 2). The user interface of the program consists of a simple form, constructed using the Common Lisp graphics library, containing controls to invoke the user operations, which are enabled according to the preconditions defined in the specification of the interface package.

4.2 Approach Applied to Diverse Code Checker

The definitions of the various types of code objects in the formal specification of the interlocking logic for System B applications (such as those nominated in Extract 6 of Section 3) contain invariants that specify the constraints that the objects must satisfy. In other words, the objects may only be constructed if they comply with these constraints. The constraints are either expressed individually for each type, as type invariants, or as consistency constraints between different types. A code object may only be constructed if its properties satisfy its type invariants. Collections (or blocks) of code objects may only be constructed if the blocks satisfy the consistency constraints mutually imposed on them. Furthermore, the blocks may only be processed by the interlocking interpreter if the code satisfies certain completeness constraints. Extract 10 shows how the completeness and consistency constraints are specified on the interlocking logic as a whole, and Extract 9 shows how the consistency constraints are specified on a given type of code object.

```
class StatementList

   instance variables

      private stmnts : seq1 of Statement;

   operations

      public
      initialise : seq1 of Statement ==> StatementList
      initialise (stmnts_) == (stmnts := stmnts_; return self; );

      -- is consistent if all statements are consistent
      public
      consistent : InterlockingLogic ==> bool
      consistent (logic) ==
         return
            forall stmnt in set elems stmnts &
               stmnt.consistent(logic);

      -- executes each statement in turn, provided they are consistent
      public
      execute : InterlockingState * InterlockingLogic ==> ()
      execute (state, logic) ==
         for stmnt in stmnts do stmnt.execute(state, logic)
      pre consistent(logic);

end StatementList
```

Extract 9. Example VDM++ specification of code object type

```
class InterlockingLogic

   instance variables
      -- memory allocations by function type- map ids to memory indices
      private memoryInds :
                 inmap Memory`TypeName to
                         inmap Name`FunctionId to
                            InterlockingState`MemoryIndex := {|->};
      -- execution block declarations- map block labels to block indices
      private execBlockInds :
                 inmap Name`BlockLabel to BlockIndex := {|->};
      -- execution block definitions – map block indices to code blocks
      private execBlocks : inmap BlockIndex to ExecutionBlock := {|->};

   operations

      public -- all execution blocks called must be defined, etc.
      complete : () ==> bool
      complete () ==
         return
            dom execBlocks = rng execBlockInds and …
            ;

      public -- id is declared if allocated in memory map
      memoryAllocated : Memory`TypeName * Name`FunctionId ==> bool
      memoryAllocated (typeName, functionId) ==
         return functionId in set dom memoryInds(typeName);

      public -- if no error detected, adds execution block definition
      defineExecutionBlock : Name`BlockLabel * StatementList ==> ()
      defineExecutionBlock (blockLabel, statementList) == (
         if not executionBlockAllocated(blockLabel) -- declare block if
         then declareExecutionBlock(blockLabel);    -- no call made yet
         let blockInd = executionBlockIndex(blockLabel)
         in ( -- if statements inconsistent or recursive call to block
            if not statementList.consistent(self) or
               ( exists blockCall in set statementList.blockCalls() &
                    blockCall.targets(self, <execution>, blockInd) ) or
               executionBlockDefined(blockInd) -- or block redefinition
            then error; -- flag error
            execBlocks(blockInd) :=
               new ExecutionBlock().initialise(blockInd, statementList);
         );
      );

      public -- constructs set route command
      setRouteCommand : Name`RouteId ==> SetRouteCommand
      setRouteCommand (routeId) == (
         if not memoryAllocated(mk_token(<route>), routeId)
         then return undefined; -- exception if route id not declared
         return
            new RouteStateCommand().initialise(
                   memoryIndex(mk_token(<route>), routeId)
                );
      );

end InterlockingLogic
```

Extract 10. Excerpt of simplified VDM++ specification of interlocking logic

The source code to intermediate generator of the System B compilation system (see Figure 4 of Section 3) ensures that the type invariants of the intermediate code are respected. The intermediate code checker program checks the other constraints on the interlocking logic, which are defined in the formal specification.

The intermediate file input to the intermediate code checker takes the form of a sequence of operation calls used to define blocks of code (see again Extract 6 of Section 3), where the interface used to express the operation calls is defined by the signatures of the operations of the formal specification (e.g. see the signatures of the defineExecutionBlock and setRouteCommand operations in Extract 10). By reading and evaluating each operation call in the file in turn, the intermediate code checker program constructs a model of the interlocking logic, in accordance with the formal specification. For each block definition, the program constructs a block object, with the code objects that it contains, and associates the block with the interlocking logic. Once the model is fully constructed, the program checks that the interlocking logic satisfies the specified completeness constraints.

By definition, the model of the interlocking logic only permits code objects that have well formed and consistent properties to be assigned to and associated with the logic. This behaviour is imposed in the formal specification via the use of the undefined expression and error statement, where an undefined expression implies a failure to construct an object of the code, due to a non-conformity in the object's definition, and an error statement implies a failure to assign a property to the interlocking logic or a failure to associate an object with it, due to an inconsistency with other properties or associations of the code. The semantics of the undefined expression and error statement are such that, if the formal specification were to be executed directly, the execution would abort on the first occurrence of a non-conformity. Because the intermediate code checker program is required to report all existing non-conformities in the code, the implementation does not strictly apply the interpretations of the undefined expression and error statement. Instead, wherever in the specification there is an error statement in the definition of an operation intended to update the interlocking logic, the corresponding Common Lisp function flags an error and continues with its execution. Similarly, wherever in the specification there is an undefined expression in the definition of an operation intended to return a code element to a caller operation, the corresponding Common Lisp function flags an error and returns an "undefined" result, allowing the caller function to continue execution. In order to report all possible errors in compositions or collections of code objects, four functions named 'f-and', 'f-or', 'f-every' and 'f-some' have been provided in the basics library and used, where appropriate, in place of the Common-Lisp 'and' and 'or' macros and 'every' and 'some' functions normally used, which unlike their Common-Lisp counterparts, have been defined to evaluate all of their arguments.

5 Conclusions

Returning to the list of techniques for design diversity in Section 1, it is clear that the diverse logic generator and diverse code checker were conceived and realised using the first four techniques, as they each are part of a set of redundant tools developed independently by separate teams using different programming languages and tools. It is also evident that the fifth technique was applied, given that the diverse logic generator was developed using a formal refinement approach, in contrast to the prototyping approach used for the Prolog program counterpart, and given that the specification used for the diverse code checker also constitutes its design, whereas the specifications used for the source code checker of the System B compiler were used for reference purposes only. The sixth technique was intentionally applied for the diverse code checker, in order to derive an abstract syntax of the interlocking data that is different from both the source code and object code syntax. Similarly, the formal specification of the diverse logic generator provided a different model of the interlocking data, in both its meta-language and circuit diagram forms, and also of the rules used to process them. Given that the diverse logic generator was designed with no working knowledge of the Prolog logic generator, apart from what was documented in its user manual, some assumed requirements of the diverse logic generator resulted in behaviour that was complementary (in a positive sense) to that of the Prolog program, and given that the diverse code checker and the diverse logic generator (which also performs checks on the data) defined the constraints to check in an alternative way to their counterparts, and generated different error messages accordingly, the seventh and eighth techniques were also, less apparently, applied.

The implementation of the diverse logic generator program was verified using the same tests that were performed on the VDM++ specification. After delivery, the program has been used, in collaboration with the Prolog program, to prepare the interlocking logic for a variety of System A applications. Only two program anomalies have been reported, one defect due to an incorrect transformation of an operation of the VDM++ specification used to output the Boolean equations to a file, and one capacity problem due to an inefficient algorithm used as part of the logic to Boolean equation conversion process. New versions of the program have been delivered since completion of the first version, as have new versions of the Prolog program, due to new requirements. For each new version, the formal specification was updated and validated, using the VDM++ Toolbox, and the Common Lisp program altered accordingly.

The intermediate code checker program has been verified, along with the other System B compilation system components, using an industry standard safety approval process, and is now being used to configure System B applications. Rigorous requirements were defined for the compilation system as a whole, including operational and non-functional requirements not covered by the functional requirements defined in the diverse specification documents that were used for the individual subsystem developments. Specially designed tests tracing back to all the requirements were performed as part of a formal V&V process that provided input to safety approval process. In particular, a single dataset containing non-conformities violating all constraints specified on the interlocking logic was used for tests on both the compiler and redundant compiler, to ensure that they yield similar results.

Apart from its use as a diverse representation, the formal specification of each program was essential for understanding its requirements, assisting, particularly, in the analysis of the software whose behaviour the program was meant to, diversely, reproduce. The specification also provides indispensable documentation for the program’s software design. The VDM++ to Common Lisp implementation guide served as a coding standard for the program's construction. With the aid of the guide, the implementation of each program took a, nearly insignificant, fraction of its overall development time, the bulk of which was spent in formalizing a software model that was correct and conformant to requirements.

In its analysis of the pros and cons of the design diversity techniques, [1] warns that the additional overhead of using diverse specifications may lead to an increase in system design effort. The previous argument would suggest, however, that an alternative specification may decrease overall development costs, especially if it is used as the basis of a formal refinement approach, such as the one described here.

The approach itself has proved to be an efficient software development in general, particularly recommended for applications of considerable complexity with unapparent requirements (such as the ones presented here, which are more complex than may appear from their simplified descriptions) and whose performance is not the most crucial concern (e.g. tools like the SMARTLOCK configuration tools, which are intended to be run in single-shot batch mode). The same approach has, in fact, been used subsequently to develop another System B configuration tool (a less complex SIL0 program, whose requirements were not obvious at the start of the development, and whose specification benefited from the use of the higher-order functional style support provided by VDM), which took only a month to design and program.

Of course, the approach does not exclude the use of other specification notations for defining diverse representations, providing that they are substantially different from the notations used for the alternative representations, and that they have enough expressiveness to define representations that lend themselves easily to code transformation, either manually or using a code generators.

In conclusion, it is hoped that this paper offers a contribution to the literature promoting best practices for the configuration of safety-related data, which, at the outset of the work reported here (as confirmed by [5] in its analysis of the subject), was notably lacking.

References

1. A. G. Faulkner, P. A. Bennett, R. H. Pierce, I. H. A. Johnston, N. Storey, The Safety Management of Data-Driven Safety-Related Systems, Proceedings of the 19th International Conference on Computer Safety, Reliability and Security, Springer-Verlag Lecture Notes in Computer Science, Vol. 1943, 2000
2. CENELEC EN 50128, Railway applications—Communications, signalling & processing systems—Software for railway control & protection systems, March 2001
3. L. Strigini, B. Littlewood, A discussion of practices for enhancing diversity in software designs, Centre for Software Reliability, Technical Report LS_DI_TR_04, 2000
4. C. Minkowitz, J. Atkiss, An object-oriented formal specification of a configuration language for railway interlockings, Proceedings of the Third Northern Formal Methods Workshop, September 1998
5. Fitzgerald, J. S., Larsen, P. G., Sahara, S. VDMTools: advances in support for formal modeling in VDM. CS-TR-1057, 2007