RULE-DIRECTED SAFETY VALIDATION OF SSI-BASED INTERLOCKING APPLICATION DATA MODELS

CYDNEY MINKOWITZ
Alstom Ferroviaria S.p.A., Italy

ABSTRACT
The Smartlock 400 (SML400) SSI-based interlocking product is one of a family of Alstom's railway interlocking products which was developed as a replacement for the Solid State Interlocking (SSI) product. A software tool has been introduced in the SML400 application engineering process to validate the application data against safety conditions, or rather to prove that the application data does not violate specified constraint violations. The aims when designing the tool were to: develop customized software based on a model of the application data generated by existing tools (rather than use a generic theorem prover, to avoid having to translate the data into another notation); use a dynamic technique similar to symbolic execution (as the nature of the data renders it difficult to use static model checking techniques); and employ application specific rules to make the technique manageable (i.e. to reduce the search space of proofs). The tool has demonstrated good performance on average sized and large interlocking applications. By customer request, it has been used principally to validate points free-to-move (PFM) conditions; it has found known data errors caused by points being commanded without having been tested free to move, imprecise definitions of PFM conditions and incomplete PFM tests across interlocking boundaries. The paper begins with the motivation behind the tool's introduction. It describes the context of the tool, including the characteristics of the application data, the way in which constraint violations are expressed and the operations performed by the tool. It contains descriptions of sample rules used by the tool to optimise the proofs. It compares the tool with other tools that have been used to verify safety properties of SSI-based data. The paper ends by proposing further work for the enhancement and use of the tool.
Keywords: railway interlocking systems, SSI, safety-critical systems, model-based reasoning, formal methods, VDM++.

1 INTRODUCTION

Solid State Interlocking (SSI) is the first generation processor-based interlocking developed in the 1980s by the research division of the UK's railway authority (British Rail at the time), GEC-General Signal and Westinghouse Signals Ltd. There is a large base of SSI installations in the UK. SSI has also been installed in Belgium, France, Australia, South Africa, Hong Kong and Indonesia. In the last 20 years, Alstom (which took over GEC-General Signal) and WRSL (formally Westinghouse Signals Ltd) have released next generation products (respectively Smartlock and Westlock) based on SSI – in particular, both products have adopted the language used to prepare the data for SSI applications.

Conventionally the process for the verification and validation of SSI-based applications involves:

- the independent manual checking of the application data;
- the automatic verification of the compiled data using diversity;
- the automatic set-to-work and independent principle testing of the data by simulation on the interlocking and/or office PCs with software that simulates the interlocking.

Weaknesses with this process are:

- manual checking is prone to human error;

WIT Transactions on The Built Environment, Vol 181, © 2019 WIT Press
www.witpress.com, ISSN 1743-3509 (on-line)
doi:10.2495/CR180161

- the verification of the compiled application data only performs checks to ensure that application data meets run-time operational constraints of the interlocking – it does not ensure that certain logical conditions, and in particular safety conditions, are satisfied;
- the setting up and running of tests by simulation are time consuming, and it is near on impossible to create test scenarios that would take into account every possible exception circumstance that could lead to a violation of a safety condition.

These weaknesses have led to situations where SSI-based interlocking failures have occurred on site resulting in unsafe states of the railway. These incidents have caused such concern that railway authorities were beginning to turn away from SSI-based products. In the UK, this resulted in Network Rail demanding more stringent safety process requirements on the development of new SSI-based interlocking applications, obliging its data suppliers to employ automatic tools to check for errors, specifically those related to the incorrect specification of points free-to-move (PFM) conditions which were the root cause of all incidents reported, and to actively engage both the railway industry and academia in the search for a solution to check the safety of commissioned and installed SSI-based interlockings. Most of the proposed solutions failed to convince Network Rail – some because they concentrated on automatic testing methods that did not guarantee the expected level of data coverage; others because they relied on the employment of general purpose theorem proving tools that require the translation of the SSI data into another notation suitable for static model checking and specialist skills to run them and which were judged to be of limited practical use for large scale interlocking applications. The solution presented by Alstom was the use of a bespoke software tool that proves the absence of violations of safety conditions, expressed simply in an SSI-like notation, using a dynamic proof technique and application specific heuristics on a software model already generated automatically from SSI data by pre-existing SML400 tools. A demonstration of this tool (integrated in the SML400 SSI-based application engineering process and toolset), showing its capability of finding known PFM related data errors (e.g. points being commanded without having been tested free to move, imprecise definitions of PFM conditions and incomplete PFM tests across interlocking boundaries), as well as its ability to check for the absence of PFM related errors in complex data, not only provided Network Rail with the confidence to continue considering Alstom as an approved supplier for future projects using the SML400 product, but it raised sufficient interest that Network Rail has since contracted Alstom to use the tool on the data of existing SML400 and SSI interlocking installations.

2 SML400 APPLICATION DATA

A SML400 interlocking application contains one or more central interlockings (CIXLs). Each CIXL consists of hardware, software and communication links with other CIXLs and other systems, such as train control systems and systems connected to trackside equipment. SML400 application data is code that represents interlocking logic, i.e. an implementation of the signalling requirements that a CIXL must satisfy.

The interlocking logic is applied on zones of the signalling area that the CIXL controls. In other words, the application data is created from code related to one or more virtual interlockings (referred to as VIXLs). The logic for each VIXL of a CIXL is defined using a procedural, object-centred language, which has been designed to be backwards compatible with the language used for configuring SSI applications, where the data for one VIXL corresponds to the data for one SSI central interlocking.

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. There are different types of blocks of code, which are processed in different ways, as described in Table 1.

The application data is interpreted on the contents of reserved areas of memories used to record the states of the signalling functions that the CIXL controls. Iterating in cycles, the CIXL receives indications of the current states of the signalling functions, updates the memories accordingly as the logic demands, sends commands to control the signalling functions to their new states, and processes requests from signallers via train control systems (TCSs) or requests made within the application data blocks. Fig. 1 depicts the dynamic state of the CIXL on which the application data is processed. The CIXL state comprises data structures of different types. An *image* of the CIXL state represents the current values of the state variables.

The CIXL application data is prepared as source code. The source code syntax of the memory tests and commands uses mnemonics oriented to signalling engineers. The syntax also contains macros that encapsulate complex sequences of instructions. The source code is divided in the files listed in Table 2, where there is one set of files for each VIXL of the

Table 1: SML400 application data block types.

Block type	Description
SEQ	Contains code (in OPT, IPT and FOP blocks) executed in a sequential manner every processing cycle of the CIXL
PRR	Contains code performed on a request emanating from a train control system or from within the application data
PFM normal PFM reverse	Evaluated whenever other blocks need to test conditions for points being free to be moved normal or reverse
MAP	Evaluated whenever other blocks need to test certain route related conditions
Evaluation Execution	Evaluated/executed when called from other blocks (similar to subroutines)

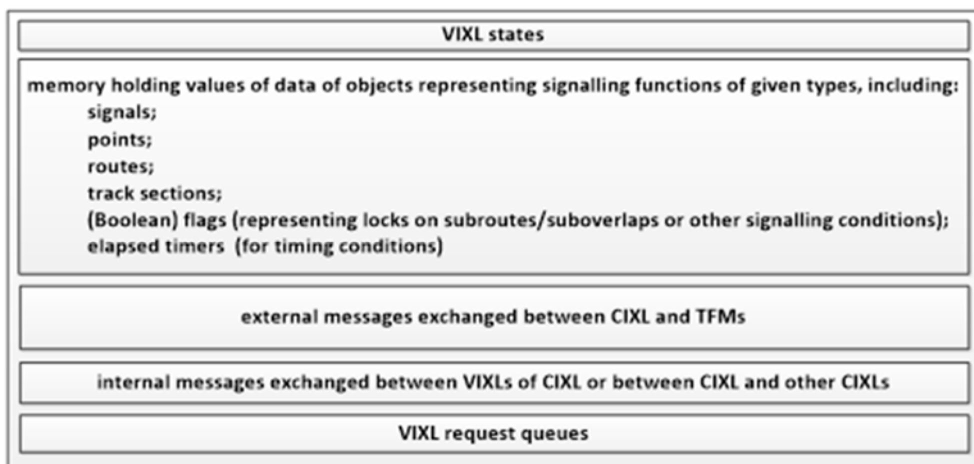

Figure 1: CIXL state.

Table 2: VIXL application data source files.

File	Description
OPT	Prepares operating information for transmission to TFMs and other CIXLs
IPT	Directs proving information to update CIXL memory
FOP	Specifies processing of flags, such as subroute and suboverlap releasing
PRR	Includes availability tests and commands to set and lock routes and move points
PFM	Defines for each set of points conditions for them to move normal and reverse
MAP	Used to search for trains in relation to given route setting and release conditions

CIXL. As indicated by their names, each file is used to define blocks of specific types. The source code makes reference to the addresses of telegrams, used for the exchange of external messages with trackside function modules (TFMs) and internal messages between VIXL or between the CIXL and other CIXLs (or other external systems), and to the ids of various signalling functions managed by the SML400T application. The source files are processed together with internal and external communications data files, which list the telegram addresses, and identity files, which list, for each VIXL of the CIXL, the ids of signalling functions by type.

Fig. 2 contains example source code (with comments) for a PRR block (in the PRR file) used to satisfy a request from a TCS to set a route. After undergoing a series of checks and translations (including the expansion of macros intro primitive data constructs), the source code is compiled to Motorola S3 object code, which is programmed on a memory device to be installed on the CIXL. Following the design of SSI, the object code instructions are derived directly from the source code syntax, as shown by the listing in Fig. 3.

Because Alstom has no access to definitive reference material on the SSI language, and because the SML400 application data language was to be extended to exploit features provided by the SML400 product, a decision was made to formally specify the new language, in order to understand its semantics, i.e. what requirements it must satisfy to conform to the on-line processing requirements of the CIXL (exported CIXL application data constraints) and how it is interpreted by the CIXL, and, as a consequence, clarify its syntax [1]. The formal specification, written using the VDM++ notation, 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 Fig. 4, which contains (commented) code equivalent to that expressed in Fig. 2 and Fig. 3.

The intermediate code serves as an API to construct objects representing the application data (in its primitive form after source code to source code translation). This object-oriented representation renders it easier to design tools to reason about, manipulate and process the CIXL application data. Fig. 5 illustrates how the intermediate code, and the object model created from it, is used to verify and validate CIXL application data..

The top of Fig. 5 shows the activities used as part of a diversified process to compile and verify the CIXL application data. The intermediate code is used in the process as a means of verifying that the application data object code is consistent with the source code from which

```
*VMANUP.QR5(M)
 if VMANUP.R5(M) a    / if route 5(M) available and
  VMANUP.P2 cnf          / points 2 controlled or free to move normal
 then VMANUP.R5(M) s   / then set route 5(M);
  VMANUP.P2 cn              / set points 2 controlled normal;
  VMANUP.S5 clear bpull     / clear signal 5 button pulled
 \
\
```

Figure 2: PRR block in source code syntax.

```
memory map:
VMANUP.S5                       5
VMANUP.P2                       2
VMANUP.R5(M)                    9
block map:
VMANUP.QR5(M)            14780
instructions:
[N°14780 : 0x0001 14787 14787 => if ]
[N°14781 : 0x0621 9  0    => VMANUP.R5(M) a ]
[N°14782 : 0x0541 2  1    => VMANUP.P2 cnf ]
[N°14783 : 0x0002 0  0    => then ]
[N°14784 : 0x0620 9  0    => VMANUP.R5(M) s ]
[N°14785 : 0x0541 2  0    => VMANUP.P2 cn ]
[N°14786 : 0x040A 5  0    => VMANUP.S5 clear bpull ]
[N°14787 : 0x0008 0  0    => \ ]
[N°14788 : 0x0008 0  0    => \ ]
```

Figure 3: PRR block in object code syntax.

```
data.definePRRBlock(
 "VMANUP.QR5(M)",   -- block label
 data.statementList([ -- block statements
  data.conditionalStatementList(
   data.conditionList([  -- conditions to evaluate
   data.routeAvailableTest("VMANUP.R5(M)"),
   data.pointsNormalStateTest(  -- Check,
    "VMANUP.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,
    "VMANUP.P2",                  -- in points memory,
    {                             -- all data bits, comprising
    mk_token(<controlN>)},        -- controlN data bit,
    1),                           -- to 1.
   data.signalStateCommand(       -- Assign,
    "VMANUP.S5",                  -- in signal memory,
    {                             -- all data bits, comprising
    mk_token(<bpull>)},           -- bpull data bit,
    0)]),                         -- to 0.
   nil)]));                       -- no 'else' statements to execute
```

Figure 4: PRR block in intermediate code syntax.

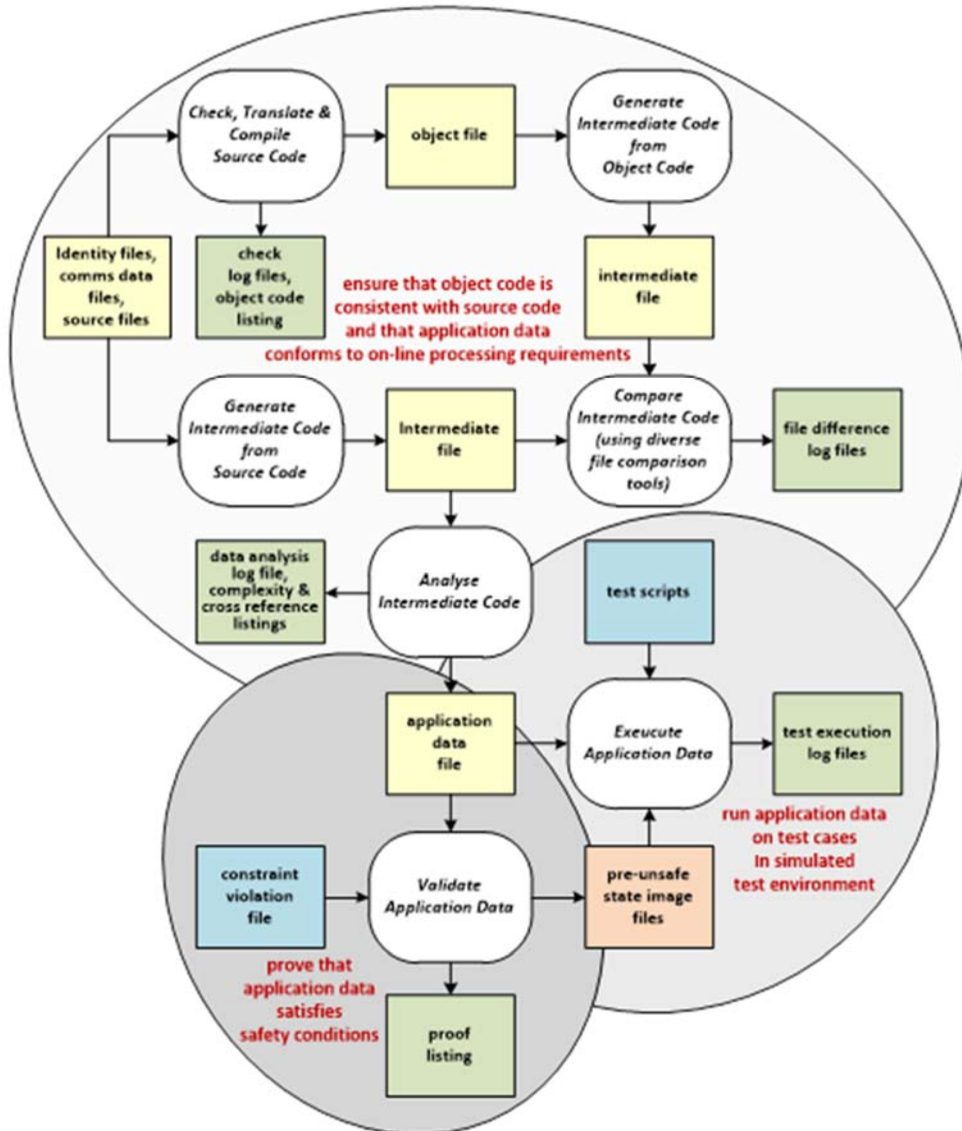

Figure 5: Application data verification and validation.

it was compiled, by comparing files containing intermediate code generated independently from the source code and the object code, and to perform a static analysis of the application data, as an independent verification to the checks performed on the source code, to verify that the application data meets the exported constraints. Diverse means were used to develop tools to automate the process as described in [1]. The VDM++ specification of the SML400 application data language was used as the software design of a tool that analyses the intermediate code (the design was translated in Common Lisp using VDM++ to Common Lisp implementation rules and delivered as an application using LispWorks). This tool uses

the instructions in the intermediate code to build the application data objects, ensuring that the objects satisfy the invariants defined in the specification, checks that the objects satisfy other conditions defined in the specification, and saves the objects to a file read by other tools that support the validation activities at the bottom of Fig. 5 (also implemented with LispWorks using the same specification to design approach). One of these tools is used to validate the application data of one or more VIXLs/CIXLs against logical requirements, including safety constraints, by executing the application data models according to the sections of the VDM++ specification that define how the CIXL interprets the SML400 language, in a test environment which simulates the interfaces between the CIXL and other subsystems of the interlocking. As stated in the introduction, because test scenarios cannot cover every possible circumstance that could lead to an unsafe state, another tool is used to validate the application data by proving that there is no execution path in the object model that would result in a CIXL state that constitutes a violation of a safety constraint. Starting from an initial state, this latter tool performs an exhaustive search on parts of the object model that could lead to an unsafe state, by executing those parts on that state, and all new safe states that result from the execution, until an unsafe state is reached or all states have been searched.

3 CONSTRAINT VIOLATIONS

Constraint violations are defined in text files using SML400-like syntax. Like the CIXL application data, constraint violations are contained in blocks. Each constraint violation file contains one or more violation blocks. An example of a violation block is contained in Fig. 6.

As seen in the figure, each violation block has a block label and contains one or more constraint violations. The block label of each violation block in the constraint violation file must be unique. Each constraint violation contains a list of conditions that define an unsafe CIXL state (i.e. a CIXL state resulting from some execution path of the CIXL application

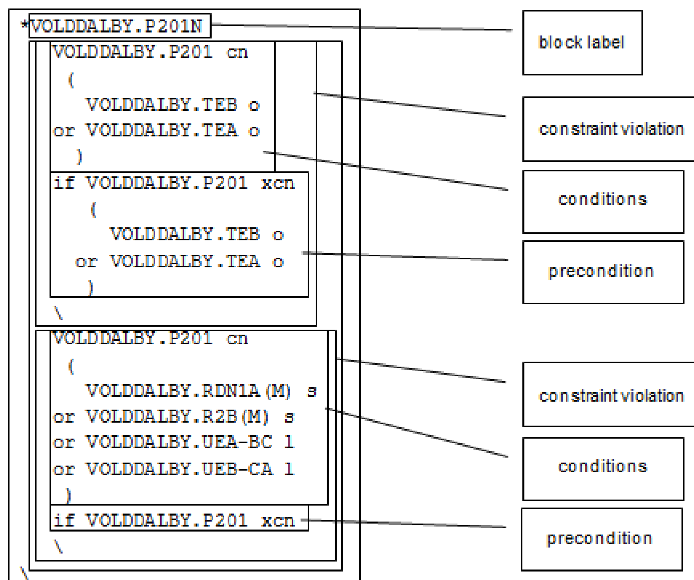

Figure 6: Violation block.

data which is deemed unsafe) and an optional list of conditions that define a precondition on the CIXL state prior to the unsafe state (i.e. the conditions that the CIXL state must satisfy in order to run an execution path of the CIXL application data). The conditions and optional precondition of a constraint violation are expressed in terms of tests on the contents of the CIXL memories associated with signalling functions declared in the identity files from which the CIXL application is created (recall Fig. 1). The memory tests access data of the memories of various types that represent different attributes of the signalling function objects.

To explain better, the following two clauses, expressed in natural language, paraphrase the two constraint violations defined in Fig. 6.

The state of the CIXL is unsafe whenever the points "201" of VIXL "OLDDALBY" are positioned normal and at least one of the track sections "EB" and "EA" of the VIXL becomes occupied, in the case when the points are not lying normal and/or at least one of the track sections is already occupied.

The state of the CIXL is unsafe whenever the points "201" of VIXL "OLDDALBY" are positioned normal and at least one of the subroutes "EA-BC" and "EB-CA" of the VIXL is locked, in the case when the points are not lying normal.

In the second constraint violation, the expressions "`VOLDDALBY.UEA-BC  l`" and "`VOLDDALBY.UEB-CA  l`" denote that two flags are set false, where each flag represents the state of a subroute (a part of route over a given track section in a certain direction – if the subroute is locked, the corresponding flag is false; if the subroute is free, the corresponding flag is true). In the SML400 application data syntax, each object id is prefixed with a VIXL name tag, which is composed of the upper case letter "V", the name of the VIXL that contains it and the full stop character ".", and an upper case letter that signifies the type of object.

Each constraint violation has a type, derived from the label of the violation block in which it is contained, which is taken into account by the rules used by the validation tool. At the time of writing, the tool has only been used to search for PFM related violations, such as the ones defined in Fig. 6, and related subroute/suboverlap release and route release violations, which define unsafe states related to the releasing of locked subroutes/suboverlaps once trains pass over them and the cancelling of routes, because past experience has shown that these types of violation are most likely to occur, however the tool has been designed in a general way to run proofs on other types of violations. Different types of violations are expressed in standard ways following generic or customer specific guidelines.

The validation tool translates constraint violation files into an intermediate code format, similar to that used for the CIXL application data. The intermediate code format comprises an API for creating objects representing the violation blocks and the constraint violations contained in them. The tool loads the intermediate code, creates the objects, ensuring that they comply to specified invariants (e.g. the violation block labels are distinct, and the constraint violations refer to signalling functions declared for the CIXL application data) and reasons about them as it performs the proofs.

4 APPLICATION DATA VALIDATION

Fig. 7 shows the main operations that the validation tool performs to run a proof.

The purpose of the first operation is to reduce the search space by masking out data that is not relevant to the violation under proof. During this operation the tool performs a kind of two phase "look-back" through the application data, starting with primary commands in the data that could lead directly to the violation. The tool applies generic rules about logic, rules specific to signalling and SML400 application data and rules related to the violation's type (examples of which are described in Section 4.1), to prevent unreachable states being

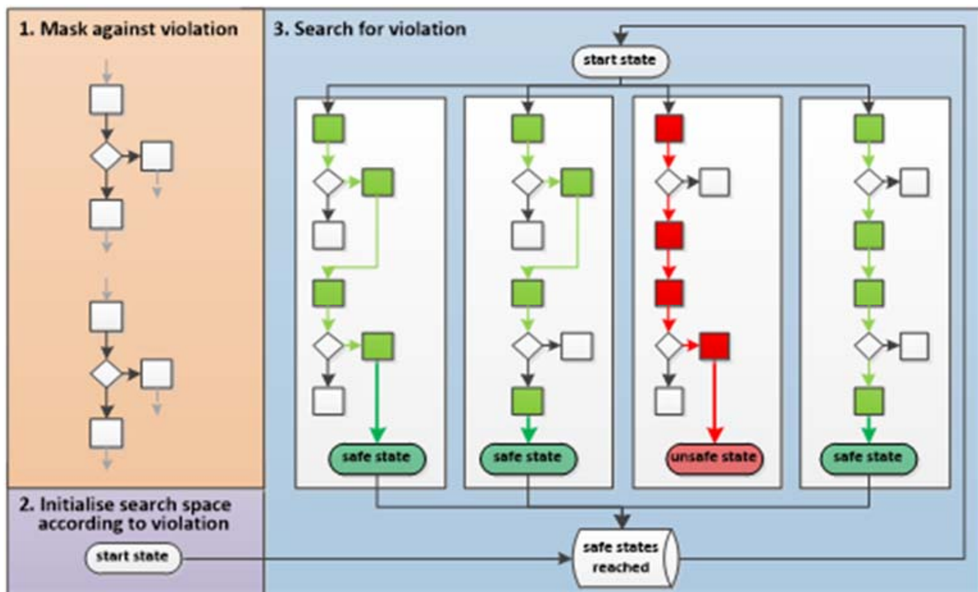

Figure 7: SML400 application data validation tool operations.

searched and false unsafe states being detected. The purpose of the second operation is to start the search on a CIXL state that is most likely to lead to an unsafe state as quickly as possible, by presetting the state's variables to appropriate values deduced by the masking operation. As an example, for PFM related violations, the tool presets flags representing subroutes/suboverlaps (i.e. sets them as free) and therefore, in the first instance, it does not look back on commands that set them. This is a valid start state since, in its start-up operations, the CIXL application data should wait a specified time, to ensure that all trains have halted before processing requests, during which time subroutes/suboverlaps will become free if tracks are unoccupied, and it is assumed that the start-up operations and route release conditions are validated by running the tool on related violations or by some other means. The tool does however mask in secondary commands that set preset flags if they are in the same statements as the primary commands in the case when they could override or enable the primary commands.

The tool performs the search in the third operation as described previously. For performance reasons, the tool partitions the masked application data, and corresponding parts of the state that the masked application data accesses, in *execution contexts* that it searches separately. Each execution context contains a PRR block that commands parts of the state that could lead to the constraint violation, or a SEQ block that directly commands data values referred to in the constraint violation, together with all mutually dependent SEQ blocks with statements that command parts of the state that the PRR/SEQ block tests. During the search operation the tool simulates the CIXL execution cycles by rerunning the contexts on each relevant safe state reached.

The tool executes the blocks in the contexts according to the VDM++ specification. Since the tool is meant to validate the application data that the CIXL interprets rather than the other functions of the CIXL, including the functions that interact with TCSs and the trackside communication systems, there are some deviations from the specification, for example in the

management of request queues, external messages and elapsed timers. The tool considers that all requests coming from a TCS are in the queue for the relevant VIXL (whereas it expects internal requests to be queued by commands in the masked application data) and, since it runs the execution contexts separately, it does not need to maintain an ordering of the requests in the queues. The tool does not look back on input message tests, which means that whenever they appear in conditions, it evaluates them as uncertain, since it has no way of determining the contents of the messages, which causes the tool to *swing* in both senses of the conditional statement lists in which the messages appear. To manage this uncertainty, the tool uses fuzzy evaluation and efficient branching algorithms to ensure that all paths in execution contexts are taken as and when needed, thereby forcing itself to search on all relevant images of the CIXL state. The tool operates on ranges of timer values converted from specific values in elapsed timer tests and commands.

The tool may run proofs on one or more violations. As it performs its operations, it prints information about each proof to a listing, which contains details of how it performs the mask operation, a description of the proof, including the constraint violation under proof and the execution contexts created for the proof. For each unsafe state reached from a given start state, it prints to the listing a description of the start state, the execution path that led to the unsafe state, and a description of the unsafe state. If no unsafe state is reached, or if the option is set to continue searching after an unsafe state is reached, on completion of the proof, it prints to the listing the results of the proof, including the number of safe states reached and searched in the execution contexts. The tool has an option to save an image of the state prior to the unsafe state which can be loaded by the tool that executes the application data (see Fig. 5) to examine the circumstances leading to it. The tool is capable of recovering a proof that has been interrupted for some reason. The tool enables users to overwrite the default masking rules, via a dialog that allows changes to the resulting selection of preset and redundant flags and PRR/SEQ blocks masked in and out of the data.

4.1 Masking rules

The tool uses generic rules about logic to simplify masked in lists of conditional statements, e.g. to eliminate redundant branches containing commands that would never be reached, and to deduce redundant flags, i.e. flags that are only tested in conditional statement lists with only one branch.

The states of track sections are normally set according to the contents of input messages, i.e. via commands in conditional statement lists that test input messages. Because the tool swings on these tests as indicated above, these commands serve to generate all permutations of track section states that the program needs to search on. To reduce the amount of swinging, the tool imitates a practice that was necessary in early SSI applications for performance reasons, which is to split the processing requirements for a request into separate PRR blocks that are processed in successive execution cycles, where the split PRR blocks are chained together via internal requests. The tool uses the same technique to split requests on artificial internal requests to divide the workload during the search operation. Assuming that PRR blocks test but not command track section state values, the tool moves statements of a PRR block that test distinct track section state values into virtual PRR blocks, replacing the statements with commands to set the artificial requests.

From the tool's perspective an elapsed timer can be in any state at the start of during any execution path that it takes, so one might think that the tool should evaluate timer tests as uncertain. However, to avoid swinging on these tests, the tool makes a judgement on timer values based on an analysis of their ranges, considering that each timer value may be set to

one of three state values: 0 (for started), 1 (for set) and 2 (for stopped). The tool presets the timer values to started or stopped depending on how the timers are used in the data (for example, it assumes that elapsed timers used for managing requests over cross boundaries are reset to stopped at the start of each execution cycle, otherwise an error would occur).

The tool applies masking rules that are specific to applications for Network Rail to simplify data related to track sections. One rule is applied to tests and commands that prevent the misdetection of track section states due to bobbing conditions. The other rule (applied only for PFM related violations) is for track sections that are duplicated across VIXL boundaries. Both rules rely on naming conventions used for the track section ids.

5 COMPARISON WITH RELATED WORK

[2] and [3] present studies of the use of the general purpose NuSMV model checking software to verify SSI data, whereby the data is translated automatically as a model in the SMV format and the model is checked against specified safety properties. Only a subset of the data is included in the model – the rest of the data is not considered safety related or is abstracted in the model, and the model checking is done only on a single SSI application. [2] reports difficulties in optimizing the software to provide acceptable performance without the need to access its internal representations and concludes that a purpose-build SSI data checker, integrated with a parser of the SSI data, would be more efficient and usable. [3] reports that is possible to gain better performance via customization of the software's algorithms. Both [2] and [3] suggest that the approach would be improved if the underlying model representation was hidden and the safety properties were automatically generated from track layouts.

In contrast the SML400 application data validation tool was developed as custom-made software that operates on a model that resembles the target data that is interpreted by the CIXL. All of the target data are represented in the model, taking into account that the misuse of apparently non-safety related data could lead to an unsafe state. The tool addresses performance issues by splitting the application data into manageable partitions and using bespoke algorithms to search them efficiently, together with the use of domain-specific heuristics (knowledge about signalling applications in general and SSI-based applications in particular) that are derivable automatically from the types of constraint violations. These techniques serve to limit the search space for each proof, thus making it feasible to use on complex data.

To give an indication of the speed of the validation tool, on a microprocessor that runs at 2.67 GHz, it takes a few minutes to prove the absence of constraint violations for the movement, in both directions, of all sets of points in an interlocking area of medium complexity using violation blocks akin to those in Fig. 6. For a single set of points on a track section that allows multiple movements across interlocking boundaries, a proof takes several days (although the time can be reduced significantly by splitting the conditions involving the track sections and the subroutes into separate violation blocks). Even so, the performance is favorable when compared to the time it takes to run similar test scenarios.

The validation tool may be run on the application data of single VIXLs or on data related to multiple VIXLs of the same CIXL or different CIXLs, thus enabling safety requirements of data related to communicating interlockings to be checked. The validation can be performed in phases, by running separate proofs to validate safety requirements related to different signalling functions of different VIXLs, to enable the validation activity to be performed in parallel.

The key competence required to use the tool is a basic fluency in signalling to be able to deduce, from scheme plans, unsafe conditions and to formulate them as constraint violations

in the required syntax. Although it would be possible to generate the constraint violations automatically, it is intended that the safety properties of the CIXL application data be checked by an independent validator, i.e. the same person who is responsible for ensuring that the right properties are validated, and hence the user of the tool would need to control the contents of the violation blocks in any case. The validation tool is integrated in a user-friendly software system that automates the generation and flow of files between the tools used for the interlocking configuration activities, including those shown in Fig. 2, in a way that is transparent to the user, so limited informatics skills are required to run the tool.

6 FUTURE WORK

Until now, the tool has been applied only to UK SSI-based applications and, following the specific requirements from Network Rail, it has only been used to validate PFM related safety conditions. The tool has been tried on other kinds of safety conditions, but mainly as a means to test the genericity of its design, so the next challenge is to employ the tool on a wider set of example data and conditions, using appropriate masking rules oriented to the signalling principles of other countries where SSI-based interlockings are installed, building up a set of *templates* to express the constraint violations in standard ways as was done with the PFM related violations. Due to the urgency of delivering the tool for the UK market, the Network Rail specific masking rules have been hard-coded in the tool's software, however it would be straightforward to enable the tool to import rules that are specific to given signalling authorities from configuration files.

Given that the language used to prepare SML400 application data is an extension to the SSI language, the SML400 tools can be used on legacy SSI data without the need to rewrite it. Because the data language for Westlock is not strictly backward compatible with SSI, it would be interesting to see how much effort would be necessary to enable the SML400 tools to import Westlock data to allow it to be checked with the SML400 data validator tool.

ACKNOWLEDGEMENT

Special thanks go to Don Hayward of Alstom UK for help with the identification and formulation of the constraint violations and some application specific rules.

REFERENCES

[1] Minkowitz, C., Formal specification for design diversity: Two case histories, one approach. *ADBIS CEUR Workshop Proceedings*, **639**, pp. 41–60, 2010. CEUR-WS.org.
[2] Huber, M. & King, S., Towards an integrated model checker for railway signalling data. *Lecture Notes in Computer Science*, **2391**, Springer: Berlin, 2002.
[3] Busard, S., Cappart, Q., Limbrèe, C., Pecheur, C. & Schaus, P., Verification of railway interlocking systems. *4th International Workshop on Engineering Safety and Security Systems*, ESSS, pp. 19–31, 2015.