Improving Predictability, Efficiency and Trust of Model-Based Proof Activity

Improving Predictability, Efficiency and Trust of Model-Based Proof Activity

Jean-Frédéric Etienne*, Manuel Maarek*†, Florent Anseaume*, and Véronique Delebarre*
*SafeRiver, 9 bis rue Delerue 92120 Montrouge, France
Email: see http://www.safe-river.com/contact
†Heriot-Watt University, EH14 4AS Edinburgh, UK

***Abstract*—We report on our industrial experience in using formal methods for the analysis of safety-critical systems developed in a model-based design framework. We first highlight the formal proof workflow devised for the verification and validation of embedded systems developed in Matlab/Simulink. In particular, we show that there is a need to: determine the compatibility of the model to be analysed with the proof engine; establish whether the model facilitates proof convergence or when optimisation is required; and avoid over-specification when specifying the hypotheses constraining the inputs of the model during analysis. We also stress on the importance of having a certain harness over the proof activity and present a set of tools we developed to achieve this purpose. Finally, we give a list of best practices, methods and any necessary tools aiming at guaranteeing the validity of the verification results obtained.**

***Index Terms*—Verification and Validation, Model-Based Design, Functional Hazard Analysis (FHA), Model Checking, Matlab/Simulink.**

I. Introduction

In this paper, we report on our industrial experience in using formal methods for the analysis of safety-critical systems developed in a model-based design framework. The choice of using a model-based design environment, as opposed to a purely formal method framework, mainly resides in the fact that models are more appealing to engineers and that they can serve different purposes. In particular, models can be used: to fine-tune requirements specification; to assess the feasibility of a system via simulation; for code generation purposes; for the verification and validation of safety requirements; for generating test cases to be executed on the target system; and as oracles to validate the outcome of each test run on the target system. Moreover, modelling suites such as SCADE [1] or MATLAB/SIMULINK [2], also integrate efficient model checkers that offer the possibility to prove significant and low-level implementation details, which normally requires cumbersome effort in proof assistant systems such as Coq [3], Isabelle/HOL [4] and B-Method [5]. Any counterexample generated from a failed proof attempt can also be simulated on the model for debugging purposes.

The use of formal methods in industrial projects generally requires formal proof analysis to be predictable and effective for very large and complex systems. These criteria are even more essential in a context whereby the safety-critical system under analysis is being developed in an incremental manner and that formal proof is used as a means to ensure non-regression. In particular, there is a need to be able to determine the mean cost and time necessary to prove the validity of a given safety requirement. This exercise mainly relies on the capacity in assessing the complexity of the models to be analysed. For instance, it can be decided that formal proof analysis may not be the appropriate validation strategy for a model containing complex mathematical computations, that approximation functions may be used as replacement for some complex mathematical operators, or even that optimisations may be required to ensure proof convergence within the budgeted cost and time. Moreover, the effectiveness of the proof activity in detecting any non-conformity or safety flaw may be undermined if no proper methodology is being put in place to validate the outcomes of the verification process. In regard to normative standards, such a methodology also has to provide the necessary evidence guaranteeing the absence of any false-positive result that can potentially hide a dangerous scenario leading to an undesired event.

In this paper, we present how we address the issues of the predictability, efficiency and trust of industrial model-based proof activity for safety-critical systems. Our contribution is threefold. We first present a formal proof workflow devised for the analysis of safety-critical systems developed in MATLAB/SIMULINK (see Section II). This workflow provides a systematic approach to quickly identify and address the critical issues that can impede the overall verification process when an automated proving technique (i.e., model checking) is being used as decision procedure. A practical method to avoid over-specification when modelling the environmental constraints is also considered. The workflow also illustrates how formal proof analysis can be used to validate the outcomes of the Functional Hazard Analysis (FHA) from which the safety requirements are derived.

Secondly, we developed a set of tools to achieve a certain harness over the formal proof activity performed with SIMULINK DESIGN VERIFIER (SLDV). These tools, namely Modeling Rules Verifier and Optimizer (see Section III), have for purpose to: facilitate the identification of incompatible constructs and any complex expressions that can influence proof convergence; verify a set of modelling rules ensuring the quality of the safety-critical system being developed; compute useful metrics that can be used to assess the complexity of models; perform some optimisations aiming at increasing

Preprint ICSE Software Engineering In Practice (SEIP) 2015. The definitive version was published by IEEE in the Proceedings of 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), DOI: 10.1109/ICSE.2015.142

the scalability of models being analyzed; and automatically generate auxiliary lemmas to help the proof engine find a solution more quickly.

Lastly, we present a list of best practices and methods that have to be considered throughout the verification and validation process to guarantee the validity of the results obtained with formal proof analysis (see Section IV). Their objective is to ensure that: the environmental constraints are not too restrictive; the safety properties are properly formalized (e.g., not a tautology); and the safety properties are sufficient enough to cover all the potential behaviours of the models being analysed.

II. Model-Based Formal Proof Workflow

As shown in Figure 1, a black box approach is generally adopted when attempting to prove a given property on a model in the MATLAB/SIMULINK environment. In particular, the properties to be proved are solely expressed in terms of the inputs and outputs of the model. An environment context may also be required for the properties to be satisfied by the model. This environment may be composed of some hypotheses specifying the constraints imposed on the inputs of the model and a data set characterising the constants being manipulated (e.g., topology or locations of fixed objects for an Automated Train Protection (ATP) system).

In the MATLAB/SIMULINK environment, the satisfiability of a given property is determined using SIMULINK DESIGN VERIFIER (SLDV) [6], which is a plug-in to the SAT-based bounded model checker Prover (see [7], [8]) and whereby the K-induction principle [9] is implicitly used for invariance satisfaction. The formal proof workflow we devised for the analysis of safety-critical systems with SLDV is mainly organised in four steps (see Figure 2). Each of these steps is presented in more details in the subsequent subsections.

A. Step 1: Model Compatibility

For a model to be analysable by SLDV, there is a need to determine whether it satisfies the *compatibility* rules imposed by the proof engine. In particular, a compatibility check tool is provided in the MATLAB environment which attempts to determine whether each construct used in the model can be translated into the proof engine's internal formalism. When the compatibility check fails there is a need to identify and eliminate any source of incompatibility (see Example 3). This process is repeated until all the compatibility issues have been resolved.

In general, nondeterministic models or models that do not support code generation cannot be analysed with SLDV. Moreover, with regards to normative standards, models used for the developement of safety-critical systems shall be in conformity with a set of guidelines prohibiting modelling practices that could lead to nondeterminism (e.g., recursive function, unbounded loops, etc).

A model is therefore declared as not compatible whenever it contains: continuous constructs (e.g., differential equations, integral, etc); unbounded loop constructs; bounded loops for which the number of iterations cannot be determined statically at compilation time; recursive function calls; variable-size data type constructs; and complex mathematical operators such as square root, exponential, logarithm, trigonometric functions and power with non-integer exponents. As such, a nondeterministic model can be made compatible by ensuring that: each loop size is known statically at compilation time; no recursive function call is performed; and only fixed-size data type constructs are used. For models containing continuous constructs, these can be transformed into discrete ones via approximations. Similarly, complex mathematical operators can be replaced by appropriate approximation functions for analysis purposes. Nonetheless, the validity of the proof results can be established only when it is shown that the precision loss incurred due to the use of approximations is acceptable (see Section IV-C). Note that transforming a nondeterministic model into a deterministic one does not necessarily reduce the complexity of the model.

It should also be noted that, when a model is declared as incompatible, the MATLAB compatibility tool does not explicitly identify where the unsupported constructs are located. In order to ease the Model Compatibility step, we therefore developed the Modeling Rules Verifier tool whose main purpose is to identify the common sources of incompatibility (see Section III-A).

B. Step 2: Proof Convergence

Once a model is declared as compatible with the proof engine, the next step is to determine whether it facilitates proof convergence. For this purpose, a simple property is specified together with a minimal environment necessary for the property to be satisfied by the model. If SLDV is able to satisfy (or even falsify) this simple property (e.g., domain properties of the form $x \geq 10 \wedge x \leq 30$) within some reasonable analysis time (and without an excessive memory usage) then, the model may be considered as facilitating *proof convergence*. If the analysis ends with an *undecided* status or fails to complete due to excessive time and memory consumption then, there is a need to identify the constructs (or parts of the model) impeding proof convergence and to subsequently perform any appropriate optimisation. It should be noted that at this stage the analysis is carried out only on simple properties and that the model may no more converge for complex/higher-level ones. For instance, the *Cone of Influence* [10] of the property may not cover all the possible execution paths of the model (see Example 1).

Example 1 (Insufficient Simple Property)**.** *Figure 3 illustrates a situation whereby the property chosen to determine proof convergence is not significant enough to cover all the possible execution paths in the model. Indeed, since variable* `delay` *is only updated at lines 7 and 10, the* `else` *branch starting at line 11 will be discarded during analysis.*

```
1
2 function [delay, ...] = model(...)
3  %#codegen
4   persistent delay;
```

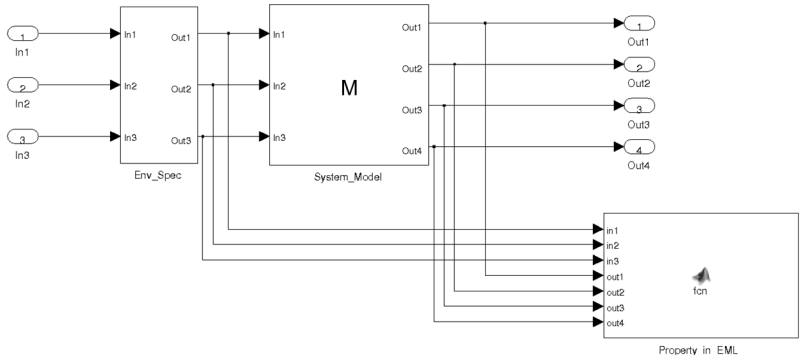

Fig. 1. General Proof Outline in MATLAB/SIMULINK

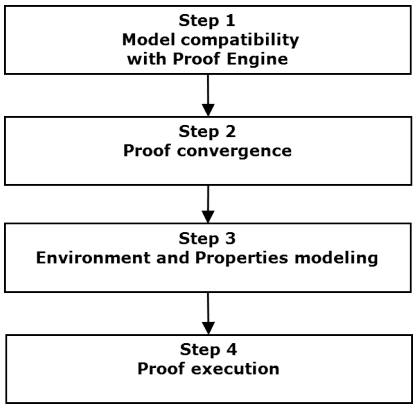

Fig. 2. Formal Proof Workflow with SLDV

```
5
6   if isempty(delay)
7     delay = int32(0);
8   else
9    if delay < MaxDelay
10     delay = delay + int32(0);
11   else
12     % complex computation performed
13   end
14
15  end
16
17 function simpleProperty(delay)
18  %codegen
19  sldv.prove (delay >= int32(0) & delay <= MaxDelay);
20
21 end
```

Fig. 3. An Example of Simple Property in EML

Based on our experience, we identified several factors that can influence the complexity of a SIMULINK model. For instance, the use of excessive vector operators can lead to the generation of costly implicit loops during the analysis process, especially when dealing with arrays/matrices of significant size. It is therefore recommended to factorise implicit loops as far as possible by replacing vector operators with scalar ones. An example of such an optimisation is given in Section III-A. A proof analysis may also end with an *undecided* status in the presence of non-linear arithmetic or complex mathematical computations in the model. Non-linear arithmetic mainly arises when mixing integer and floating-point arithmetic or when certain specific type conversion operations are performed in the model. Another aspect that can influence proof convergence is the size of the state space necessary to resolve the problem at hand. An excessive use of *persistent* variables[1](or memory blocks) can drastically increase the size of the search space during analysis. These variables can be defined explicitly in the model or are inherent to built-in constructs such as stateflow, time delays and temporal operators. One way to reduce the size of the search space during analysis is to specify auxiliary lemmas characterising the inductive behaviour for each of the variables concerned.

C. Step 3: Environment and Property Modeling

Step 3 of the formal proof workflow consists in formalising the safety properties to be proved together with any necessary environmental constraints required for the inputs of the model under analysis. The hypotheses as well as the properties are usually expressed in EMBEDDED MATLAB (EML) for readability, traceability and maintainability purposes. Indeed, EML is a high-level programming language that eases specification without having to deal with the complexity inherent to graphical block constructs. In EML, assumptions may be specified using the `sldv.assume` directive, while the `sldv.prove` directive is used to designate the proof objective to be satisfied (see Example 2)

Referenced Data Set. In order to ease proof convergence, a data set instance is generally provided when attempting to prove a given safety property. In fact, the use of a data set instance generally prevents the modelling of costly engineering rules as assumptions on the inputs of the model and significantly reduces the size of the search space during analysis. However, the choice of a specific data set instance has to be performed in such a way as to guarantee that it allows to cover all the possible execution paths of the model. Computing the *proof converage* of all proven safety properties can be considered as one feasible solution to make sure that

[1]Variables retaining their values from one cycle to another

none is left uncovered (see Section IV-B).

Over-Specification Avoidance on Environmental Constraints. When specifying the environmental constraints, one must make sure that they are not too restrictive so as not to compromise the analysis results. In essence, a too restrictive assumption may hide a dangerous scenario leading to the occurrence of an undesired event. A systematic approach to avoid over-specification is to start with a property (or set of properties) requiring a minimal environment specification (e.g., only the necessary data set instances are provided). The environment context may afterwards be refined as and when required based on feedback from failed proof attempts. For instance, a new environmental constraint may only be added when the generated counterexample exhibits an unfeasible scenario.

Safety Property. When attempting to model a safety requirement there is a need to determine whether it can solely be expressed on observable variables and to clarify any imprecision or ambiguity due to the use of natural language. There is also a need to establish whether the safety requirement is cycle dependent or not so as to introduce any appropriate memorisation mechanism (e.g., *persistent* variables) during the modelling process.

Example 2 (Property and Hypothesis modelled in EML)**.** *Figure 4 illustrates how a safety property for the localisation function of an Automated Train Proctection system is modelled in* EML*.*

```
function safetyProperty(Segments, TrainChar, Positions)
 %#codegen

 extR.segment = Positions.externalRearSegment;
 extR.abscisse = Positions.externalRearAbscisse;
 extR.orientation = Positions.externalRearOrientation;

 intF.segment = Positions.internalFrontSegment;
 intF.abscisse = Positions.internalFrontAbscisse;
 intF.orientation = Positions.internalFrontOrientation;

 % Positions not set to their default value
 hyp = extR.segment > int32(0);

 params.intF_orient = intF.orientation;
 params.trainLength = TrainChar.trainLength;

 % Path p exists between externalRear and internalFront AND
 % Length of Path p = trainLength
 % Orientation of the internalFront = Orientation of Path p

 p = seg_path(extR, intF, Segments, params, @pathPredicate);
 goal = p.nbSeg > int32(0);
 sldv.prove(implies(hyp, goal));
end

function out = pathPredicate(path, params)

 out = path.pathLength == params.trainLength & ...
       params.intF_orient == path.pathOrientation;
end
```

Fig. 4. An Example of Safety Property in EML

This property simply states that if the computed positions for a given train are not set to their default value then, there should exist a path p between the external rear and internal front positions of the train s.t.: path p follows the direction specified by the orientation of the external rear position; the length of path p corresponds to the train's length; and the internal front position is oriented w.r.t. path p. Figure 5 shows how hypotheses can be specified in EML*.*

```
function env(TrainChar, ...)
 %#codegen

 hypNbUnit = TrainChar.nbUnit >= int32(1) & ...
             TrainChar.nbUnit <= NbMaxUnit;
 sldv.assume(hypNbUnit);

 computedLength = TrainChar.nbUnit * TrainChar.unitLength;
 hypLg = TrainChar.trainLength == computedLength;
 sldv.assume(hypLg);
 ...
end
```

Fig. 5. An Example of Hypothesis in EML

In the above example, `nbUnit` *and* `trainLength` *are free variables, while* `unitLength` *and* `NbMaxUnit` *are constant values.*

D. Step 4: Proof Execution - An Iterative Process

As illustrated in Figure 6, the proof workflow is an iterative process especially when the formal analysis does not end with a *valid* status. Each of the different non valid statuses is discussed in more details in the following subsections.

Contradictory Status. When the proof execution ends with a contradictory status, SLDV indicates that there is a contradiction in the model and that the analysis cannot proceed any further. This situation mainly arises when either there is at least two conflicting assumptions constraining the inputs of the model or when there are some assumptions in contradiction with the data set characterising the constants being manipulated by the model. Contradictory assumptions can only be identified through a trail and error approach as no feedback from the proof engine is provided for debugging purposes. For instance, each assumption can be removed one at a time until the contradictory situation is resolved.

Falsified Status. When the proof execution ends with a falsified status, a counterexample is generated for debugging purposes. This counterexample has to be analysed in order to determine whether it is due to: some missing hypotheses in the environment context; an erroneous data set instance; a wrongly formulated safety property; or a non-conformity situation. It is evident that any feedback from failed proof attempts also allows to strengthen the Functional Hazard Analysis (FHA) of the corresponding safety-critical system. Indeed, all the safety hypotheses necessary to prove the validity of a given safety requirement are identified explicitly. Similarly, any wrongly formulated safety property due to ambiguities,

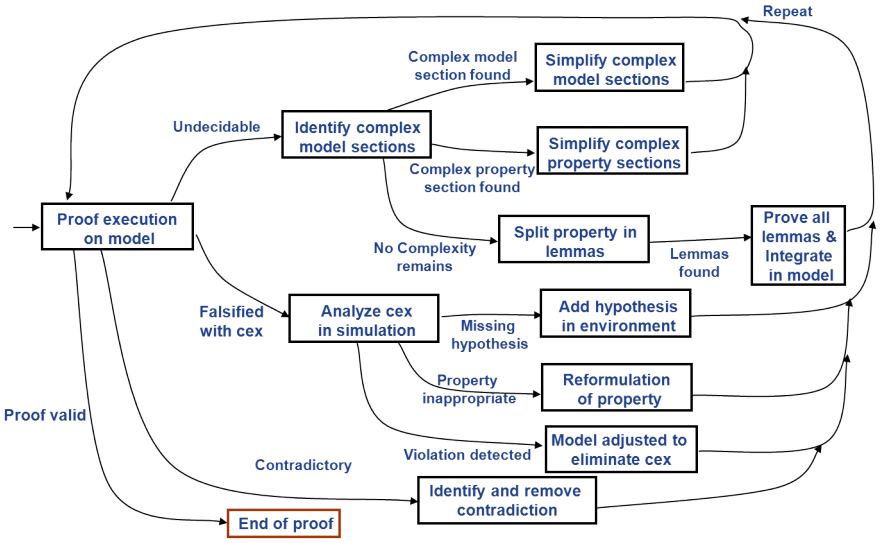

Fig. 6. Proof Execution Step

imprecisions or even conflicting requirements in the safety analysis may be detected. Any necessary reformulation may even lead to the introduction of complementary failure modes not covered by the FHA.

Undecided Status. When the proof execution ends with an undecided status, SLDV indicates that it was not able to satisfy or even falsify the given proof objective. This situation mainly arises when the model/safety property is too complex or when the size of the state space necessary to represent the problem at hand makes an exhaustive proof analysis intractable. As mentioned in Section II-B, one way to address this issue is to reduce the complexity of the model (or property) via appropriate optimisations. However, when the complexity of the model cannot be reduced any further, it might be helpful to decompose the safety property into intermediate lemmas. Each intermediate lemma may either correspond to a specific functional case or to a lower-level safety property to be satisfied by a sub-module. In the former case, the intermediate lemmas are used to provide the necessary reasoning steps influencing the capacity of the proof engine in finding a solution more quickly. In the latter case, the problem to be resolved is decomposed into manageable lower-level ones so as to reduce the complexity during analysis.

For instance, we may have a model $\mathcal{M}$ for which we would like to prove a given safety property P under the assumption of some environmental constraints Q. Using a Hoare-style notation, the proof outline is therefore specified as following:

$$\{Q\}\ \mathcal{M}\ \{P\}$$

Suppose that model $\mathcal{M}$ is decomposed into three *sequential* sub-modules M_1, M_2 and M_3. In order to reduce the complexity of the problem to be resolved, a compositional proof paradigm may be adopted whereby model $\mathcal{M}$ is replaced by the sequential composition $M_1; M_2; M_3$. The new proof outline obtained is as follows.

$$\frac{\begin{array}{l} Q \Longrightarrow \beta_1 \wedge \beta_2 \wedge \beta_3 \\ \{\beta_1\}\ M_1\ \{\alpha_1\} \\ \{\beta_2 \wedge \alpha_1\}\ M_2\ \{\alpha_2\} \\ \{\beta_2 \wedge \alpha_2\}\ M_3\ \{\alpha_3\} \\ \alpha_3 \Longrightarrow P \end{array}}{\{Q\}\ M_1; M_2; M_3\ \{P\}}$$

Moreover, it can be noticed that precondition Q can be split into sub-conditions (i.e., β_1, β_2 and β_3) to match the input flow decomposition from $\mathcal{M}$ to M_1, M_2 and M_3. Similarly, part or all of the postconditions for each specific sub-module may be required to establish safety property P (e.g., when M_1, M_2 and M_3 have outputs in common with $\mathcal{M}$). For the above decomposition, we assume that only M_3 produces the outputs for $\mathcal{M}$. It can be noted that such a decomposition can be performed as from the Functional Hazard Analysis phase to determine whether a system level mitigation has properly been propagated down to the component level.

III. Harness over Proof Activity

The use of formal methods in industrial projects requires formal proof analysis to be predictable and effective for very large and complex systems. These criteria are even more essential in a context whereby the safety-critical system under analysis is being developed in an incremental manner and that formal proof is used as a means to ensure non-regression. To this end, we developed a set of tools that allows to have a certain harness over the formal proof activity performed with SLDV. They are presented in the following subsections together with a brief account of the industrial project on which they have been applied successfully.

A. Modeling Rules Verifier

The Modeling Rules Verifier tool has for purpose to ease Step 1 and 2 of the formal proof workflow presented in

Section II. When a model is declared as *incompatible*, the MATLAB compatibility check tool does not provide any clue as to where the unsupported constructs are located. As such, the process of identifying the sources of incompatibility can be cumbersome, especially when the models concerned are quite complex. The Modeling Rules Verifier tool allows to explicitly identify all the sources of incompatibility and any complex expression that can influence proof convergence in a model. It is based on symbolic execution and performs type inference as well as constant propagation when analysing a given model w.r.t. the semantics of MATLAB/SIMULINK built-in constructs. Hence, in addition to the identification of recursive function calls and unsupported mathematical operators, it is able to determine: when a loop size cannot be known statically at compilation time; when a variable-size data type construct is being used; when a variable is being inferred with two or more types within the same function definition[2]; which expressions generate costly implicit loops or when loop factorisation is possible; when unnecessary array/matrix copying can be avoided; and where floating-point expressions are introduced due to untyped constant values[3].

Example 3 (Unknown Loop Bound at Compilation Time)**.** *The following error message is generated by Modeling Rules Verifier for the function definition given in Figure 7, provided that function* `find_idx` *is being called more than once with different values for parameter* `vsize`*.*

```
find_idx.m:5:In function find_idx: Compatibility : Unbounded
For loop - Index End value is not a constant
```

It indicates that the upper bound value of the 'for' loop introduced at line 5 cannot be determined.

```
1  function out = find_idx( value, vect, vsize )
2    %#codegen
3     out = int32(-1);
4     found = false;
5     for x = int32(1:vsize)
6      if ~found && vect(v) == value
7         out = x;
8         found = true;
9      end
10    end
11 end
```

Fig. 7. Unknown Loop Bound

The recommended coding pattern for function `find_idx` *is given in Figure 8, whereby directive* `numel` *is being used to determine the size of array vect at compilation time.*

```
1  function out = find_idx( value, vect)
2    %#codegen
3     out = int32(-1);
4     found = false;
5     for x = int32(1:numel(vect))
6      if ~found && vect(v) == value
7         out = x;
8         found = true;
9      end
10    end
11 end
```

Fig. 8. Bounded Loop Coding Pattern

Example 4 (Implicit Loop Detection)**.** *The following messages are produced by Modeling Rules Verifier for the function definition given in Figure 9, where each* `checkX` *function returns a boolean vector denoting whether each element of vector* `X` *satisfies a given condition.*

```
validate.m:3:In function validate: Implicit loop - in
expression check1(X) | check2(X)
validate.m:4:In function validate: Implicit loop - in
expression check3(X) | check4(X)
```

They indicate that implicit loops are generated due to the application of the OR logical operator at lines 3 and 4. In fact, the definition given in Figure 9 will produce six loops, i.e., one loop for each `checkX` *function and one loop for each OR logical operator. This can impede proof convergence especially when vector* `X` *has a significant size.*

```
1  function [Y, Z] = validate(X)
2    %#codegen
3    Y = check1(X) | check2(X);
4    Z = check3(X) | check4(X);
5  end
```

Fig. 9. Implicit Loops due to Vector Operators

The complexity of function `validate` *can be reduced by replacing each vector operator by a scalar one such that only a single loop is required. This is illustrated in Figure 10, where each* `checkX_sc` *function is a scalar version for each* `checkX` *function. The compiler directive* `coder.nullcopy` *is used only to declare vectors* `Y` *and* `Z` *with the proper size and type but without performing any initialisation.*

```
1  function [Y, Z] = validate(X)
2    %#codegen
3    Y = coder.nullcopy(X);
4    Z = coder.nullcopy(X);
5    for i= int32(1:numel(X))
6      Y(i) = check1_sc(X(i)) | check2_sc(X(i));
7      Z(i) = check3_sc(X(i)) | check4_sc(X(i));
8    end
9  end
```

Fig. 10. Loop Factorisation

Modeling Rules Verifier also offers the possibility to compute useful metrics that can be used to assess the complexity of each MATLAB Function Block [4] within a SIMULINK model. These metrics can provide clues as to where optimisations can be performed, whenever the proof analysis does not converge. Based on our experience, we also devised a set of modelling

[2]In EML, the type of a variable is being inferred through assignment statements. As such, it can be the case that a variable can change type at different points in a function definition.

[3]By default, a constant value is of type 'double'.

[4]SIMULINK blocks containing functions written in EML.

guidelines which have for purpose to guarantee the compatibility of SIMULINK models with SLDV and to ensure the quality of the safety-critical systems being modelled. Modeling Rules Verifier implements most of the modelling rules specified in this set of guidelines.

B. Optimizer

The Optimizer tool mainly has for purpose to perform some optimisations in order to increase the scalability of models being analysed with SLDV while having a certain harness over time and memory consumption during proof execution. In particular, it first generates a new SIMULINK model whereby all library links are broken[5] and all model references are inlined. As such, the original models/library components are left unchanged during the optimisation phase. The tool afterwards type-checks the flatten model and uses symbolic execution to optimise the model by performing constant propagation, dead-code elimination and arithmetic simplifications (only for integer and fixed-point arithmetic). Once these optimisations are done, it automatically generates lemmas for some specific built-in constructs present in the model. The purpose of these lemmas is to influence the capability of the proof engine in finding a solution more quickly. For the time being, the optimisation and lemmas generation phases are only performed on MATLAB Function Blocks contained within a SIMULINK model. It is expected to extend this capability to StateFlows and SIMULINK blocks as well (see Section III-D).

Example 5 (Lemmas for Built-In Constructs)**.** *The following lemmas are generated whenever an array access is encountered in a* MATLAB *Function block.*

1. $\forall i, \exists j \in [1..size(A)], i = j \longrightarrow A[i] = A[j]$
2. $\forall i, \forall j \in [1..size(A)], i \neq j \longrightarrow A[i] = \bot$

The second lemma is introduced to check for out of bound array access. It can also be noticed that array indices start at 1 *in* MATLAB/SIMULINK*.*

It should be noted that the maturity of the SLDV proof engine is such that it can handle quite complex and heterogeneous models. However, for certain specific problems, the convergence of the proof analysis is not always guaranteed. The optimisations performed by the Optimizer tool allow to reduce the analysis computation time by a factor of 50 (or even more) and the memory consumption by a factor of 10. We believe that further optimisations can still be performed to increase the scalability of SIMULINK models subjected to formal proof analysis.

C. Success Story

The Modeling Rules Verifier and Optimizer tools have allowed to establish a precise estimation of the time and cost necessary to carry out the formal proof activities for a complete Automated Train Protection (ATP) system (i.e., both trackside and onboard) modelled in SIMULINK, with most of the important functionalities (e.g., train tracking, localisation, anti-collision, etc) specified in EML.

Thanks to Modeling Rules Verifier, an average of 8 man-hours was required to determine whether a newly developed functionality satisfied the compatibility rules imposed by the proof engine (or to identify any incompatible constructs), whether the modelling rules were well applied, or whether appropriate optimisations were required to reduce the complexity of the model. Thanks to Optimizer, an average of 20 man-hours was required to model and prove each of 528 high-level safety properties identified by the Functional Hazard Analysis. This ratio also comprises the time spent in analysing any counterexample generated, in identifying any necessary environmental constraint required, and in determining the correctness of the safety property being analysed (see Section IV-B).

The specification of at least 472 environmental constraints was also necessary for the high-level safety properties to be satisfied by the models under analysis. Note that the number of input data handled by the different functionalities of an ATP system are quite voluminous. For instance, some of the SIMULINK models analysed are defined with up to 5610 inputs encapsulated within Bus objects[6]. As such, some of the environmental constraints specified were even more complex than the models and the safety properties to be validated.

With the help of the Optimizer tool, the worst computation time observed when attempting to prove the satisfaction of a high-level safety property was approximately 10 hours. Nearly 1440 hours of computation time (i.e., 60 days) were required for a complete replay of all proofs, which results in a mean computation time of ≈ 3 hours per safety property. In practice, only 15 days were required as the proof executions were performed in parallel. It should also be noted that without the use of the Optimizer tool the analysis of some high-level safety properties would require weeks to complete.

D. Work in Progress

For the time being, the Modeling Rules Verifier and Optimizer tools only consider MATLAB Function Blocks embedded within a SIMULINK model. Any other block constructs (e.g., StateFlows or built-in SIMULINK blocks) are left unchanged during the optimisation and automatic lemma generation phases. In order to efficiently apply the verifications and optimisations described in the previous sections on those graphical constructs, we expect to first translate them into EML when parsing the SIMULINK model. The use of EML as pivot language is necessary as it allows to exhibit all the variables implicitly encapsulated within the graphical constructs (e.g., state variables, loop indices, temporal counters, etc). Indeed, access to these variables is essential for the automatic lemma generation phase.

[5]An explicit copy is made for each custom-made library block or for each EML file.

[6]In SIMULINK, a Bus object is analogous to a structure definition in C.

IV. Validity of Verification Results

This section gives a list of best practices, methods and any necessary tools aimed at guaranteeing the validity of the verification results obtained with formal proof analysis. These have for objective to enhance the quality of the safety-critical system produced. Moreover, it is important to note that the methods/techniques intending to increase the level of confidence in the formal analysis results can be applied regardless of the modelling environment and the formal methods tools used.

A. Environment Constraints

The methodology, described in Section II-C, provides a systematic approach to avoid over-specification when defining the hypotheses characterising the environmental constraints for the model under analysis. However, each and every hypothesis must undergo a formal specification review to guarantee that it has been well formalised and that it is covered by (or is less restrictive than): either a safety property satisfied by an interfacing system, a field equipment or some exploitation procedures; or a safety property satisfied by an upstream module whenever a compositional proof approach is adopted (i.e., no disparity with what has been proved and what is being used as assumption, see Section II-D); or an engineering rule guaranteed by the data preparation process.

In general, the data preparation process for safety-critical systems is also subjected to formal proof activites in order to guarantee that each specific instance is in conformity with the expected engineering rules. Nonetheless, there is still a need to determine whether the instances of data set used allow to cover all the possible execution paths of the model under analysis. Determining the *proof coverage* of all the proven safety properties can be used as a means to address this issue (see Section IV-B).

B. Proven Properties

A formal specification review must also be performed for each proven safety property to make sure that any notion/-concept described in the FHA is captured properly during the formalisation process. However, specification/code review is generally a tedious process and can be error-prone, especially when the safety property to be reviewed is complex. In order to increase the level of confidence in the safety properties declared as *valid*, it is proposed to: ensure that the negation of the premise (if any) for each safety property always generate a counterexample; perform fault injection in the model to see whether the expected errors are captured by the formal proof analysis. The first verification has for purpose to determine the reachability of the context in which the goal of the safety property has to be satisfied. The second verification is complementary to the first one as, in addition to the identification of too restrictive environments, it also allows to ensure that the safety property is well formalised and is not a tautology.

The pertinence of the safety properties in detecting all the possible bugs/safety flaws in the model under analysis can also be reinforced by computing the *proof coverage* for each proven safety property. Indeed, the computation of the proof coverage allows to:

1) explicitly determine whether some execution paths are completely left uncovered due to too restrictive environment constraints or to incomplete data set instances;
2) validate the outcomes of the safety analysis by determining whether the proven safety properties are sufficient enough to cover all the potential behaviours of the model. For instance, an execution path can be left uncovered due to either an improper formalisation or to a failure mode not detected by the safety analysis;
3) provide the necessary justifications with regards to the normative standards (e.g., EN 50158 [11], DO-178C [12]) for the use of formal methods as a complete replacement for unit and integration testing.

C. Approximation Functions

In general, the use of floating-point is not recommended for the development of safety critical systems. Nevertheless, approximation functions may still be required for complex mathematical operators when a fixed-point representation is used in the models to be analysed. When approximation functions are used, the validity of the proof results can be established only if it is shown that the precision loss incurred is acceptable for the application context at hand. This precision loss can be evaluated by performing an equivalence testing between the original function and the approximated one through the use of an automatic test generation tool such as STIMULUS [13]. For the equivalence testing to be effective, there is also a need to ensure that the test cases generated allow to cover all the intervals considered when defining the approximation function. In essence, there is a need to guide the random generation to ensure that each interval is covered at least once during simulation. In some simulation tools like STIMULUS, the end user has the possibility to influence the test case generation through the use of *weighted choice* operators when specifying the constraints characterising the test environment. Moreover, if the precision loss incurred is deemed acceptable, the approximation functions may also be used for code generation purposes. As such, there will be no disparity between the model subjected to formal proof analysis and the one used to generate the embedded code. This may also benefit the application's computation time as approximation functions are generally more efficient in terms of execution time and computational resources.

D. Verification Tools

1) Translators/Optimisers: The development of translators/optimisers used to encode the model under analysis in the formalism expected by the proof engine must be performed in conformity with the recommendations prescribed by the normative standards such as EN 50128 [11], DO-178C [12], and IEC 62279 [14]. To have a certain level of confidence in the analysis results obtained, there is a need to provide evidence that the transformations/optimisations performed meets the qualification level expected for tools

supporting the verification process (e.g., T2 level in EN 50128 standard and TQL4 level in DO-178C). For instance, a specification document describing the transformations applied must be produced together with any theoretical justification establishing the soundness of the process (e.g., semantics of the original model is preserved). Moreover, a safety analysis must be performed on the specification document to identify all the possible failure modes that may negatively impact the outcome of the verification result. Finally, a set of test cases allowing to cover all the failure modes identified by the safety analysis and the expected nominal behaviours of the tool must be provided.

2) Solvers: For the solvers used as decision procedures (industrial or open source), it is expected that they provide the same guarantees as required for translators/optimisers. Nevertheless, if the solvers used do not meet the qualification criteria described above, then one of the following strategies can be applied:

1) A 2oo3 voting system can be put in place whereby three solvers are invoked to determine the satisfiability of a given problem. If at least two out of three solvers produce an unsatisfiable (resp. satisfiable) result then the corresponding property is declared as valid (resp. falsified); This method allows to fulfill two purposes: firstly, it increases the chance of detecting a problem (if any); secondly, it reduces the investigation process if there isn't really a problem (e.g. one of the solvers generates an erroneous result).
2) If the solver used offers the possibility to produce a refutation proof when a problem is declared as unsatisfiable, the resolution tree can be cross-checked by a theorem prover to obtain a high degree of confidence in the result obtained [15].

3) Code Generators: Similar to translators and solvers, code generators must also be certified in order to guarantee the equivalence between the embedded code and the model subjected to formal proof analysis. In case of a non-certified code generator, one of the following validation strategies can still be applied:

1) The easiest solution is to perform an equivalence testing between the embedded code and the model. This entails two steps: firstly, there is a need to generate test cases necessary to obtain a 100% coverage on the source model; secondly, the generated test cases are executed on the embedded code with the source model being used as oracle to validate each test run. This method allows to reduce the risk of any impairment introduced by the code generator but does not guarantee the absence of errors.
2) Proving the equivalence between the embedded code and the model provides a means to exhaustively establish that the semantics of each construction is preserved during translation. For this matter, there is a need to translate both the model and the embedded code in a common formalism, to construct an equivalence relation, and to determine the satisfiability of the equivalence relation.

V. Future work

We are currently developing our own formal verification tool based on Bounded Model Checking (BMC), which uses the K-induction [9] principle for invariance satisfaction and several SAT/SMT solvers as decision procedures. Much of the effort is focused on the encoding phase to reduce as far as possible the number of variables/contraints to be handled at the SAT/SMT solver level. The results obtained so far are very promising. We also expect to extend our formal verification tool so that it can accept several modelling formalisms as inputs (via dedicated translators), thus providing a framework that would allow the integration and validation of models/components described in different formalisms. Finally, we would like to provide the possibility of computing the proof coverage for each property declared as valid. The computed coverage would provide meaningful information about the execution paths covered during analysis, about the branching conditions discarded by the premises (if any) of a given property, or about the execution paths falling outside the *Cone of Influence* [10] of a given property.

VI. Conclusion

The formal proof workflow, presented in this paper, has been applied successfully for the verification and validation of an entire Automated Train Protection (ATP) system developed in MATLAB/SIMULINK. We believe that it can easily be adapted for other model-based design environments modulo any tool support provided. The Modeling Rules Verifier and Optimizer tools, developed in support of SLDV, have also allowed to establish a precise estimation of the time and cost required to carry out the formal verification activities relative to this industrial project. Indeed, for any newly developed functionality, it took an average of 8 man-hours to determine whether it satisfied the compatibility rules imposed by the proof engine (or to identify any incompatible constructs), to ensure that the modelling rules were well applied, or to propose any appropriate optimisation to reduce the complexity of the model. Moreover, with a factor of more than 50 in computation time gained when applying the Optimizer tool, an average of 20 man-hours was required to formalise, prove and assess the correctness of one high-level safety property for a given functionality. Approximately 1440 hours of computation time (i.e., 60 days) were required for a complete replay of all proofs. In practice, only 15 days were required as the proof executions were performed in parallel.

We also believe that the Modeling Rules Verifier and Optimizer tools can be generalised for any modelling formalism if a well-defined intermediate language is provided to properly capture the semantics of each graphical construct. The need for such an intermediate language is necessary as it would ease the assessment of the computational complexity for a given model as well as the automatic generation of auxiliary lemmas for specific built-in constructs. It should also be noted

that, even if the preprocess performed by the Optimizer tool allows to increase the scalability of the SIMULINK models being analysed with SLDV, the optimisations done are still limited as they are applied at the EML level. More efficient optimisations, derived from the compiler domain, may be applied if the problem at hand is first transformed into a Static-Single Assignment (SSA) -like form extended with *latch* notations for state variables.

REFERENCES

[1] Esterel Technologies, "SCADE," http://www.esterel-technologies.com/products/scade-suite/.

[2] MathWorks, "SIMULINK," http://www.mathworks.com/products/simulink/.

[3] The Coq Development Team, Coq, *version 8.4*, INRIA, Feb. 2012, http://coq.inria.fr/.

[4] T. Nipkow, C. Lawrence, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, University of Cambridge/Technische Universität München/Université Paris-Sud, Dec. 2013, http://isabelle.in.tum.de/.

[5] J. R. Abrial, *The* B *Book, Assigning Programs to Meanings*. Cambridge (UK): Cambridge University Press, 1996, iSBN 0521496195.

[6] MathWorks, "SIMULINK DESIGN VERIFIER," http://www.mathworks.com/products/-sldesignverifier/.

[7] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," *Intl. Conf. on Formal Methods in Computer-Aided Design*, vol. 1954, 2000.

[8] Prover Technologies, http://www.prover.com.

[9] A. Biere, "Bounded model checking," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, pp. 457–481.

[10] A. Biere, E. Clarke, R. Raimi, and Y. Zhu, "Verifying safety properties of a PowerPC - microprocessor using symbolic model checking without bdds," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, vol. 1633.

[11] *EN 50128, Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems*, CENELEC, Jun. 2011.

[12] *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc, Dec. 2011.

[13] Argosim, "Stimulus," http://www.argosim.com.

[14] *IEC 62279, Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems*, International Electrotechnical Commission, Sep. 2002.

[15] T. Weber and H. Amjad, "Efficiently checking propositional refutations in HOL theorem provers," *Journal of Applied Logic*, vol. 7, no. 1, pp. 26–40, 2009.