Formal assessment of hybrid functions

M. Fortes da Cruz[1], M. Kwiatkowski[2], S. Sharma[1]

1: Airbus Operations Ltd., New Tech Centre (09H—B1), Golf Course Lane, Bristol BS99 7AR, UK
2: School of Informatics, The University of Edinburgh, 10 Crichton Street, Edinburgh EH8 9AB, UK

Abstract:
It is commonly accepted in the academic community that if the use of formal methods were more widely spread and properly supported, then their use would provide substantial development time and cost benefits for the engineering of safety-critical systems.
The present work builds upon an increasing imperative to improve continuously the process, methods techniques and tools for analysing the functions/behaviours allocated to hybrid systems in the chosen avionics and mechatronics technologies.
Formalisation of requirements and formal expression of models remain the two major representational bottlenecks impeding the wider industrial usage.
Our investigations suggest that relatively minor improvements to the prevailing industrial practices can help to overcome these bottlenecks and to start benefitting from the power of formal methods.
In this paper, we present a novel method and workflow for formally analysing engineering specifications of hybrid systems that, semi-automatically, bring together requirement and domain engineering model. Both the method and workflow are validated with a case study of the Airbus A350 autobrake model.

Keywords:
Formal requirements, hybrid functions, model checking, ICP, SAT, LTL, CTL, DPLL, SMT.

1. Introduction and industrial motivation

Introduction:
For more than three decades, the aerospace industry has been using digital computers as an integral part of the civilian aircraft's flying safety-critical equipments. In turn, the proliferation of digital computers has created an increasing imperative to continuously improve the process, methods techniques and tools for analysing the functions allocated to systems in their chosen technologies.
The present work builds on this imperative by exploring ways and means for increasing the automation of formal verification. The proposed automation is through translation of requirements, and their respective design models, into formal logic transcriptions that can be merged and analysed from the same file.
The systems of interest are hybrid systems; these are multi-physics systems that exhibit continuous-time and discrete-time behaviours, with event-driven behaviours emanating from the embedded software components.
Naturally, hybrid functions are allocated to, and implemented by, hybrid systems. Further, the term hybrid function is used to describe those functions that feature both continuous and discontinuous functions from the multi-physics domain as well as the discrete-event functions from the embedded real-time control software applications.
Design is constructed through using models of both the physical elements (using CAD tools) and the avionics elements (using CASE tools). Conversely, the requirements have evolved from being disjoint sets of objective in structured documents, to integrated databases of indexed and linked textual objectives for validation and verification.
Applying formal methods to verification of embedded software for avionics requires constructing a coherent formal set of propositions that precisely captures the textual assertions and property objectives into formal logic propositions.
Besides, the engineers also need to *envision* consistently the behavioural logic of their models into a formalised set of declarations, relations, constraints and state transitions.
Before any meaningful formal assessments can be conducted, both sets of formal sentences—models constrained by requirements, and the attached objectives—must be correctly merged and prepared for analysis using a model checker.

Industrial motivation:

The Advisory Council for Aeronautics Research in Europe (ACARE) was set up in June 2001 to develop and maintain a Strategic Research Agenda (SRA) for aeronautics in Europe. This was referred to as the ACARE 2020 Vision. By 2004, ACARE identified five challenges and six high-level Target Concepts. These were gathered in two volumes [1].

This work aims to provide some contribution to the following three ACARE challenges:

a) The Challenge of Quality and Affordability:
   The Quality of the product is both its attributes and its properties; the attributes are subjectively assigned by the observer, whilst the properties are its intrinsic characteristics. We focus on the extent to which these intrinsic characteristics of the product fulfil its requirements.
   Affordability is also a subjective attribute; however, from the perspective of the manufacturers, it results from the ability to price the product correctly. Controlling the cost of Design, Development and Build allows the manufacturer to price the product for the market. Specifying the product concisely, so that the supply chain produces the required products, is a key contributor to affordability.
b) The Challenge of Safety
   Safety is the topmost driver for the aeronautics industry. The notions of Safety emerge from the behaviour of the system-elements and their interactions with their operations environment. In turn, the behaviour of the product emerges from the behaviour of its parts and their interactions. We focus on the automatic assessment of those safety properties that are allocated to the product. As one of the contributors to meeting the Challenge of Safety, ACARE identifies “Increase depth of product’s dependability assessments” during the Design, Operation and Maintenance phases. We focus on increasing the depth of the systems dependability during the design phase.
c) The Challenge of Air Transport System Efficiency
   Improvements in the efficiency of the ATS effects, and is affected by, the reliability of the components that constitute the system. An aircraft is a component of the Air Transport Systems (ATS). From an aircraft manufacturers’ perspective, the aircraft must be one of the most reliable components of the ATS. Further, the systems that constitute the aircraft must be reliable as they, in turn, contribute to the reliability of the aircraft.

The remaining two challenges; the Challenge of the Environment and the Challenge of Security are outside the scope of this work.

In contribution to this industrial context, we address:

- The impediments that occur between the textual requirements and their manually interpreted formal expressions. The effort of interpretation is time and resource consuming, which defeats the object right from the start. One way to clear this bottleneck is to structure the engineering domain text and automate the formal translation.
- The need to assess both the continuous and discrete state space, and path of computations. Aircraft systems, such as the braking system, implement an aggregated set of hybrid functions in their chosen technologies of hardware, software and physical components, which are sensed, controlled and commanded. The Domain and Safety & Reliability Engineers interpret the normal and abnormal behaviours that lead to different envisioning of the models. A common and purpose-specific set of envisioning rules is necessary to assess the different perspectives of a system, for example, addressing a common V&V and safety purpose.

In order to address these areas this paper is organised into three parts. The first part, presented in Section 2, describes the formalisation of the constraints and properties of the models. The second part, presented in Section 3, describes the case study and a validation of the results. The final part, presented in Section 4, concludes with a discussion of our findings and suggests further work.

2. Formalising models and their properties

In order to analyse avionics models against design requirements automatically, it is necessary to remove all ambiguity from both areas. Furthermore, the resulting formal versions of models and requirements have to be compatible. This presents the non-trivial task of choosing the formal platform on which to conduct our analysis. Because the functions we aim to analyse are hybrid and non-linear, the formal framework has to be able to faithfully capture these two concepts.

2.1 Formal frameworks

A theory, a notation, a language and a platform support each of the formal frameworks below. The maturity of the first two frameworks led to industrial

applications in the area of hardware verification (e.g. Collins proprietary AAMP5 Microprocessor) [9].

Design Verifier (SLDV, DV plugin) [4]:

SLDV is an option for the MathWorks MATLAB/Simulink© and the SCADE© tool set. It performs an exhaustive formal analysis of dataflow and controlflow models, restricted to the interface boundaries and to the scope of the relevant selected set of sub-systems or functional operators. The model coverage assessment is deduced from the analysis, which follows from that scope assumption.
Dataflow refers to the Simulink and SCADE approach to modelling functions.
Controlflow refers to Stateflow/SSM approach to modelling behaviours and mode logic with transitions.
SLDV/DV can confirm or refute the consistency and correctness of Simulink/Stateflow or SCADE/SSM models with respect to the given properties. Domain engineers specify these properties directly in a Companion Observer Block Diagram, as claims to be guaranteed from assumptions about the interface. The constraints are specified in the models using assertion blocks.

NuSMV (LTL/CTL) [3]:

NuSMV is a symbolic model checker that originated from the re-design, re-implementation and extension of CMU SMV; a BDD-based model checker originally developed at CMU [McMillan 1993]. The input language of NuSMV is designed to take a description of Finite State Machines (FSM).
The types of FSM can be completely synchronous to completely a-synchronous, and can feature full details up to more abstraction. NuSMV specifications are expressed in either of two different temporal logics: Computation Tree Logic CTL, and Linear Temporal Logic LTL extended with Past Operators.
NuSMV evaluates CTL and LTL specifications in order to determine their truth or falsity in the FSM. If a specification does not to hold, then NuSMV constructs and presents a counterexample, i.e. a trace of the FSM that refutes the property.

HySAT [5, 6]:

In essence, satisfiability (SAT) is the problem of determining if the variables of a given Boolean formula can be assigned in a way that the formula evaluates to TRUE. Martin Davis and Hilary Putman (DP) devised the first algorithm in 1960.
In 1962, George Logemann and Donald W. Loveland proposed a refined DP as DPLL-algorithm.
HySAT is a SAT solver developed at the University of Oldenburg. It checks for satisfiability of arithmetical formulae that might involve non-linear and transcendental functions. HySAT processes elements of discrete and continuous domains (e.g. integer and real) as intervals on the real line and thus the solver is particularly suitable for analysis of hybrid functions.
HySAT integrates the DPLL-based SAT proof search algorithm with the interval-based arithmetic constraint propagation. This allows HySAT to take advantage of handling conventional model checking whilst handling similarly large Boolean combinations of non-linear arithmetic constraints involving transcendental functions.
Because the functions under analysis are hybrid and non-linear, the formal framework must faithfully capture these two concepts. Theory of hybrid satisfiability, and in particular the SMT solver extension in HySAT, appears suitable to solving this class of problems.

HySAT accepts input files consisting of four sections. The first three define the dynamic model that is the subject of analysis by listing all the state variables and the relationships between them (called *constraints*). The last section contains the *target formulae* that encode questions about the present and future behaviour of the model.
Design requirements are represented as target formulae and models as sets of constraints. This achieves the integration of the two domains; described in more detail in the next sections.

The choice of HySAT was based upon the following three reasons. Firstly, it handles discrete and continuous variables in a uniform fashion, namely as intervals on the real line; therefore, it provides an elegant and powerful (through the use of interval arithmetic) framework for hybrid analysis. Secondly, it supports the use of non-linear as well as transcendental operators in constraints and targets, and hence copes with the inherently non-linear characteristics of physical laws. Finally, it supports relatively free mixing of Boolean and arithmetical constructs, which are useful when analysing signal-processing diagrams. For a detailed comparison of HySAT with other satisfiability software, see [2].

Certain analysis and verification tasks can be performed directly by Design Verifier (DV) [4]. Whilst a useful tool, DV is not able to handle the complex non-linearities inherent in the majority of the requirements that industrial engineers deal with; further Simulink and Stateflow are not based upon a synchronous language.

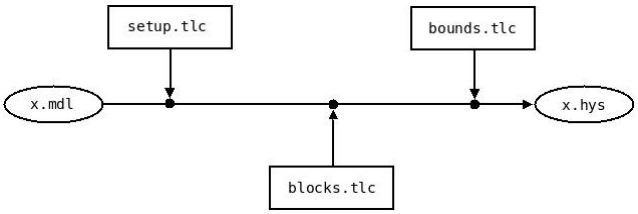
2.2 Formalising a Dataflow block diagram model

Simulink: The starting points of the model formalisation tasks are Simulink (*.m files) design specifications (*.mdl files). Here, the point of interest

is on signal-processing diagrams. Such diagrams are translated to HySAT model specifications. More precisely, a state variable is assigned at first to every line in the model. Then, one or more constraints are produced for every block, hence linking variables corresponding to its input and output signals in a way that reflects the block's intended behaviour. Finally, a limited type and range inference for the output variables is achieved. A by-product of this translation is an association table of signals and variables, which proved very useful for debugging.

As an example, consider a summation block with two input signals and one output signal. In the first step, variables for the signals are created; say *'x'* and '*y'* for the inputs and '*z'* for the output. The second step yields the invariant constraint *'x+y=z',* which obviously reflects the normal behaviour of the summation block. Finally, assuming that input signals (*x*, *y*) are floats with ranges ([0,100], [-50,50]), the type of the output signal, *z*, is inferred also to be a float, with the range ([-50, 150]). An output might be an input to another block; such information can also be used to infer type and range of the output of that block.

Implementation: The procedure described above was implemented using the Target Language Compiler (TLC), a code generation language for Simulink, allowing annotation of the model files and generation of a coded format file in HySAT format.



The Simulink to HySAT translation consists of three stages using TLC:

(a) Pre-processing of the Simulink model (*setup.tlc)*
(b) Creating appropriate constraints (*blocks.tlc)*
(c) Inferring the types and ranges of variables (*bounds.tlc*).

2.3 Formalising constraints from requirements

Formalisation starts with a design requirement given as an English sentence. The ideal route to take would be to automatically parse, recognize and translate such sentences; however, current state of natural language processing makes this an impractical—if not infeasible—option. As the first step, therefore, requirements are re-written into an equivalent, human-readable form; however, a form that is sufficiently constrained and parsimonious to be machine-readable. In a second step, a procedure was devised to automatically translate such semi-formal sentences to Linear Temporal Logic (LTL) expressions. Finally, those sentences were automatically translated into the HySAT target formulae.

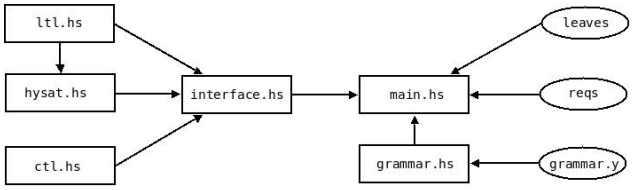
Step 1: A formal grammar is built to make requirements expressed in natural language to be machine-readable. A formal grammar is a set of rules (called *productions*) for generating text by successive rewriting a pre-defined string. Grammars generate sentences that resemble original natural language requirements. A computer can understand (parse) such sentences precisely because a grammar constrains their structure (syntax). To give an example, the original requirement « *The autobrake shall disarm after take-off* » becomes « *It shall* ***always*** *be* ***true*** *that the autobrake is* ***not armed after*** *the aircraft* ***takes off.*** » Here, this translation is done manually but can be enhanced by a user-interface to guide the requirements engineer (Appendix 1).

Step 2: Once the semi-formal requirement is parsed, it is possible to identify the atomic statements, i.e. those that are the most basic assertions about the system. In the above example, these are « *the autobrake is not armed* » and *« the aircraft takes off* ». We manually match them to appropriate conditions on state variables, such as *'AB_DISARMED = TRUE'*. Having done that, we use the internal structure of the semi-formal requirement—constructed using the grammar—to build an LTL expression encoding this requirement. This is further facilitated by the fact that most productions directly correspond to specific LTL constructs (Appendix 2).

Step 3: The translation of LTL expressions to HySAT target formulae is relatively straightforward and amounts to little more than syntactic manipulations. Here we have to bear in mind, however, that certain LTL expressions are too complex to have a HySAT representation. Moreover, it is possible that the requirement holds when the target formula is unsatisfiable, and dually, fails when the formula can be satisfied. These cases are properly reported during the automated translation (Appendix 4).

Implementation: The procedure detailed above was implemented as a standalone program using the functional programming language Haskell. The program requires three input files; one containing the semi-formal requirements, one matching the atomic sentences to simple conditions on state variables and one containing the HySAT model definition produced from the Simulink diagram. The program then outputs the complete HySAT input file, ready for analysis.

Architecture of the Requirement Analyser:



The main program file accepts two input files: i.e. requirements (*reqs*) and variable/condition association table (*leaves*), and communicates with the different target language specifications via a standardised interface (*interface.hs*)

3. Formal assessment of hybrid functions

3.1 Case study using HySAT

The Airbus A350 autobrake analysis model consists of 1288 Simulink blocks and a similar number of lines. A HySAT model consisting of 658 variables, 238 constants and 822 constraints were obtained after the translation. The type and range inference procedures were successful for 81% of signals; the remaining 19% were involved in feedbacks, where the naïve iterative method failed. Manually inserting proper types and bounds in the loops and re-running the algorithm refined this method with a final result (Appendix 3). Three requirements were positively verified on this model. Both the formalisation and verification tasks were completed within minutes on a standard desktop computer (Appendix 4).

3.2 Validation using Gryphon/NuSMV

The formalisation of Simulink models can be performed using more established software tools. The Reactis tool (by Reactive Systems) pre-processes the models to an intermediate format recognizable by the Gryphon tool (by Rockwell Collins) [7]. Gryphon produces a formal NuSMV specification and runs the NuSMV model checker. This NuSMV based workflow supports a wide range of Simulink blocks, but has one crucial shortcoming from our point of view—it does not yet fully support continuous variables, and hence hybrid systems.
In order to compare our approach with the Reactis, Gryphon and NuSMV workflow, we have isolated a fragment of the A350 braking and control system where all but one variable are discrete, and discretised the only continuous one. We then wrote several study requirements for this module and checked them using both workflows. The results were identical, which validated our approach and implementation.

4. Conclusion

4.1 Discussion

Our work was exploratory in nature; we set out to gauge the potential of formal verification techniques in the context of industrial design of hybrid systems. This is why the integration of our framework with the existing industrial tools is not as complete as possible. The same is true of our specific analysis results, which lack in breadth. Nevertheless, we can report the following several conclusions with conviction.

Simulink diagrams are not formal enough. Simulink diagrams, like the one formalised during the course of this project, do not contain enough information on their own to make the formalisation process fully automatic. This is because Simulink was primarily used as a simulation engine, and actually not as a formal modelling language. Quite often, Simulink signals do not carry any information about their domain (range), units, and their loose type information cannot be reliable upon. Whilst this information can be partially recovered from supporting documents, such as Interface Control Documents (ICDs), this still requires human intervention. In summary, Simulink is a modelling tool used for simulations that can also serve as a general-purpose modelling and formal analysis environment if the diagrams carry more annotations on their lines of input and output ports as well as interface boundaries.

S-functions are difficult to verify. While native Simulink blocks have reasonably clear meaning (function) and can be readily represented in any formal setting, the semantics of S-functions is often unclear. The supporting documentation, while perhaps sufficient for everyday modelling, falls short of the rigour required for formal verification. As a result, the engineer performing formal verification of a diagram containing S-functions must access, or possess expert knowledge of this subject, in order to write tailored representations of their behaviour in the target formalism. Moreover, we have found that the use of S-functions promotes the lack of rigour described in the preceding paragraph.

Verification can be fully automated. The three instances where human intervention in the formal verification process was found to be necessary were:

(a) setting the proper types and ranges of variables,
(b) reformulating the textual requirements in the constrained language, and
(c) linking the basic propositions with model signals.

We believe that all three human interventions can be eliminated. Elimination of (a) can be achieved through more standardised and rigorous modelling practices; (b) can be eliminated by introducing a standard requirement language, where textual requirements can be parsed by a computer directly; and (c) can be performed during the modelling stage.

Medium-sized models can be verified efficiently. The autobrake model analysed here consisted of 1288 blocks. The translation to HySAT had 658 variables, 238 constants and 822 constraints. The requirements could be verified in a matter of seconds on a desktop computer. While these results are encouraging, some sensitivity to the number of variables and to their domains was found. Hence, we believe that models that are larger by an order of magnitude than the autobrake model—for example, the full A350 BSCS model—cannot be practically verified, unless strict typing and range discipline is followed (see the paragraph on Simulink) and/or advanced modular (*compositional*) verification techniques are developed.

Requirements exhibit different levels of ambiguity. Most of the autobrake requirements we analysed were ambiguous, in the sense that it was possible to assign more than one distinct formal representations to a single requirement. Some of them were simply too high-level to be translated to the language of arithmetical formulae; others had insufficient indication of their applicability domain; finally, some merely lacked proper tolerance levels. Elimination of these ambiguities, at least of the two latter kinds, would greatly increase the potential of automated requirement verification.

High-level programming languages should be used for verification tools. The Simulink to HySAT translator was implemented using TLC, an internal Simulink scripting language. Lack of advanced programming concepts in TLC makes it difficult to mirror the abstract mathematical constructs in the code, and consequently produces monolithic, inflexible software. In contrast, the use of Haskell for requirement analysis proved here to be the right choice because the result was a flexible and stable program. Of course, these remarks are not intended to cast doubt on the usefulness of TLC within its intended application areas: e.g. code generation (C, FORTRAN).

4.2 Future work

We have identified the following directions for further development:

Grammar inference It is possible—albeit to a limited degree—to determine a grammar, given a sufficient number of sample sentences it produces, by a process known as *grammar induction* or *grammar inference*. Given the abundance of example requirements, it is conceivable to use this theory to build a requirement grammar from these examples instead of just postulating one, as we did. The main advantage of this approach is that the resulting grammar would produce sentences identical to the requirements as they are currently used. The main downside is that not all productions of this automatically generated grammar would correspond to any discernible logical or temporal operators.

Compositional verification Our estimates of the computational load of formal testing of requirements suggest that large avionics/mechatronics models cannot be cost-effectively tested yet with existing methods. Developing modular (also called *compositional*) verification techniques, where a requirement is verified for submodels and the verification results are then combined to yield an assessment of the whole model, would make it possible to also verify large models. While this is an extremely difficult and ambitious challenge in general, it should nevertheless be at least undertaken in a limited industrial scope.

Context-sensitive grammars In this work, we used a *context-free* grammar, mainly because of their conceptual and theoretical simplicity. However, it may be the case that a more complex *context-sensitive* grammar is a better fit for an industrial-quality requirements language. The first step towards settling of this question should be a detailed analysis of the fine structure of actual requirements.

5. Acknowledgements

This work was partially funded by the EPSRC through PhD Internship Programme of the UK Knowledge Transfer Network for Industrial Mathematics (KTN-Mathematics) 2009 and funded and supported by the Airbus Systems Methods and Tools department within an ongoing internal project in the Filton (Bristol, UK) group.

We thank all the reviewers of previous draft of this paper for their supportive and guiding comment.

6. References

[1] Advisory Council for Aeronautics Research in Europe, *"Strategic Research Agenda Volume 1", "Strategic Research Agenda Volume 2"* Oct 2004 access from http://www.acare4europe.org/html/documentation.asp

[2] Clarke, Grumberg, and Peled: "*Model Checking"*, MIT Press, 2000.

[3] A. Cimatti, E. M. Clarke et al.: "*NuSMV 2: An OpenSource Tool for Symbolic Model Checking"* In Proceedings of International Conference on Computer-Aided Verification (CAV 2002). Copenhagen, Denmark, July 27-31, (2002).

[4] M.Au. Fortes Da Cruz: "*Two-ways traceability between Requirements Based Engineering and Model Based Engineering using DOORS and SCADE tools",* SCADE USERS Conference, Toulouse (France, 2006).

[5] M. Fränzle: *Verification of Hybrid Systems*, CAV 2007: **38** (2007).

[6] M.Fränzle et. al.: "*Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure*", JSAT **1**(3—4), 2007

[7] S. P. Miller: "*Formal Methods for Critical Systems"*. FMICS 2008: 1, (2008).

[8] S. P. Miller: "*Bridging the Gap Between Model-Based Development and Model Checking"*. TACAS 2009: 443-453, (2009).

[9] S.P. Miller et al: "*Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods*". *WIFT '95:* IEEECS, Boca Raton, FL. April, 1995, Pages 2–16.

7. Appendices

Appendix 1: Grammar

The definition of the formal grammar underlying the requirements analysis is in the format of Happy (`http://haskell.org/happy`), a parser generator for Haskell. The first main part, beginning with the `%token` directive, contains the syntactic atoms processed by the grammar. The second, captioned "production rules", effectively defines the form of every correct sentence produced by the grammar. This file can be fully automatically compiled to a parser compatible with Haskell.

```
%name parser Req
%tokentype { Token }
%error { parseError }

-- Lexical tokens
%token assert           { TokenAssert $$ }
       not              { TokenNot $$ }
       always           { TokenAlways $$ }
       never            { TokenNever $$ }
       possible         { TokenPossible $$ }
       impossible       { TokenImpossible $$ }
       alwayspossible   { TokenAlwaysPossible $$ }
       neverpossible    { TokenNeverPossible $$ }
       if               { TokenIf $$ }
       then             { TokenThen $$ }
       else             { TokenElse $$ }
       unless           { TokenUnless $$ }
       until            { TokenUntil $$ }
       after            { TokenAfter $$ }
       while            { TokenWhile $$ }
       and              { TokenAnd $$ }
       or               { TokenOr $$ }
       sentence         { TokenSentence $$ }

-- Precedence and associativity of tokens.
   Tokens mentioned early bind weakly.
%right assert not
%right always never
%right possible impossible
%right alwayspossible neverpossible
%left if then else
%left unless until after while
%left or and
```

```
%%

-- Production rules
Req  | assert Req                { Assert $1 $2 }
     | not Req                   { Not $1 $2 }
     | always Req                { Always $1 $2 }
     | never Req                 { Never $1 $2 }
     | possible Req              { Possible $1 $2 }
     | impossible Req            { Impossible $1 $2 }
     | alwayspossible Req        { AlwaysPossible $1 $2 }
     | neverpossible Req         { NeverPossible $1 $2 }
     | if Req then Req else Req  { IfThenElse ($1, $3, $5) $2 $4 $6 }
     | Req unless Req            { Unless $2 $1 $3 }
     | Req until Req             { Until $2 $1 $3 }
     | Req after Req             { After $2 $1 $3 }
     | Req while Req             { While $2 $1 $3 }
     | Req and Req               { And $2 $1 $3 }
     | Req or Req                { Or $2 $1 $3 }
     | sentence                  { Sentence $1 }
     | error                     { Error "???" }  -- (*)
```

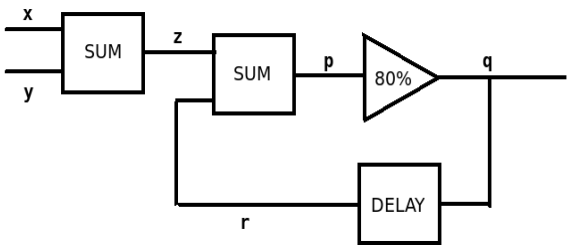
Appendix 2: Model & Formal requirements

A snippet of the HySAT input file encoding the A350 autobrake model together with a requirement. The file has four sections: the first (DECL) defines the model variables. Observe that some of them are actually constants, thereby reducing the state space; variables are annotated with the name of the Simulink block to whose output they correspond; the variable marked with the asterisk (*) is involved in a feedback loop, and thus its type and range are set to the most conservative values. The second section (INIT) defines the constraints linking the variables in the initial state of the model, while the third (TRANS) defines the pre- and post-conditions to be met during the evolution of the model. Observe how the initial state conditions are duplicated in order to ensure model consistency after a transition takes place. Finally, the last section (TARGET) contains the target formula produced by the Requirement Analyser. As indicated by the appropriate annotation, the requirement in question is valid iff the target formula is *unsatisfiable*.

```
DECL
      define x0 = 3.5;                  -- AP2633_IN_LBP4
      define x1 = 3.0;                  -- AP2633_IN_LBP2
      define x2 = 2.5;                  -- AP2633_IN_LBP1
      define x3 = 2.0;                  -- AP2633_IN_LBP3
      int [0, 1] y27x9;                 -- MRTRIG1
      ...
      int [0, 1] y27x136;               -- OU1
      define y27x137 = 0;               -- Ground3
      int [0, 1] y27x138;               -- AB COMMAND
      float [-9999, 9999] y27x139;      -- AB COMMAND (*)
      ...
      int [0,1] prev_y27x9;
      ...
INIT
      ...
      (y27x136 >= 1) <-> (1 - y27x34 >= 1);
      ...
TRANS
      ...
      (y27x136' >= 1) <-> (1 - y27x34' >= 1);
      prev_y27x9' = y27x9;
      ...
TARGET
      (!(y27x136 <= 0)) and (((1 - prev_y27x9) * y27x9) >= 1); -- Refute
```

Appendix 3: Bounds

Propagation of types and ranges of variables through an example signal processing diagram involving feedback. The method requires all the information about the inputs to infer type and range of the output and thus it fails for all the variables involved in the feedback loop (Table 1). The solution is to manually specify the type and range of one of those variables and run the algorithm again (Table 2).



Round 0	Round 1	Round 2	Round 3,4,...
x: float in [0,5] y: float in [-5,0] z: ? p: ? q: ? r: ?	x: float in [0,5] y: float in [-5,0] z: float in [-5,5] p: ? q: ? r: ?	x: float in [0,5] y: float in [-5,0] z: float in [-5,5] p: ? q: ? r: ?	x: float in [0,5] y: float in [-5,0] z: float in [-5,5] p: ? q: ? r: ?

Table 1: Propagation of types and bounds through a signal processing diagram involving feedback.

Round 0	Round 1	Round 2	Round 3
x: float in [0,5] y: float in [-5,0] z: ? p: ? q: ? r: float in [-20,20]	x: float in [0,5] y: float in [-5,0] z: float in [-5,5] p: ? q: ? r: float in [-20,20]	x: float in [0,5] y: float in [-5,0] z: float in [-5,5] p: float in [-25,25] q: ? r: float in [-20,20]	x: float in [0,5] y: float in [-5,0] z: float in [-5,5] p: float in [-25,25] q: float in [-20,20] r: float in [-20,20]

Table 2: Propagation of types and bounds through the same diagram, this time with manual injection of type and bounds into the feedback loop.

Appendix 4: Session

A recording of a short command-line session with the requirement analyser. Seven commands are issued (bold lines numbered in square brackets). Commands [1] and [2] display two files containing the requirement to be verified; in the second case, the requirement has an empty “while” clause, which makes it nonsensical. The third command displays the basic property--variable condition association table. Command [4] invokes the analyser on the correct requirement, and the result is a logical breakdown of the sentence, together with the HySAT target formula. Commands [5] and [6] produce CTL and LTL target formulae, respectively. Finally, an attempt to analyse a grammatically incorrect requirement is made, and results in an appropriate error message [7].

```
[1] marek@belafonte:~$ cat req
it shall always be true that the autobrake is disarmed after the aircraft takes off
[2] marek@belafonte:~$ cat malformed_req
it shall always be true that the autobrake is disarmed after the aircraft takes off while
[3] marek@belafonte:~$ cat leaves
the autobrake is disarmed = y27x136 <= 0
the aircraft takes off = (1 - y27x9) * y27x9' >= 1
[4] marek@belafonte:~$ ./analyser.exe -R req -L leaves
This is the Requirement Analyser.
```

Results:

```
it shall always be true that the autobrake is disarmed after the aircraft takes off:

it shall always be true that
              the autobrake is disarmed
       after
              the aircraft takes off

Refute (BMC mode): (!(y27x136 <= 0)) and (((1 - prev_y27x9) * y27x9) >= 1);
================================================================================
[5] marek@belafonte:~$ ./analyser.exe -R req -L leaves -T ctl
This is the Requirement Analyser.
```

Results:

```
it shall always be true that the autobrake is disarmed after the aircraft takes off:

it shall always be true that
              the autobrake is disarmed
       after
              the aircraft takes off

AG (((1 - y27x9) * y27x9' >= 1) -> (AX ((1 - y27x9) * y27x9' >= 1)))
================================================================================
```

```
[6] marek@belafonte:~$ ./analyser.exe -R req -L leaves -T ltl
This is the Requirement Analyser.
```

Results:

```
it shall always be true that the autobrake is disarmed after the aircraft takes off:

it shall always be true that
             the autobrake is disarmed
       after
             the aircraft takes off

G (((1 - y27x9) * y27x9' >= 1) -> (X (y27x136 <= 0)))
================================================================================
[7] marek@belafonte:~$ ./analyser.exe -R malformed_req -L leaves
This is the Requirement Analyser.

================================================================================
it shall always be true that the autobrake is disarmed after the aircraft takes off while:

it shall always be true that
                    the autobrake is disarmed
             after
                    the aircraft takes off
       while
             ???

This requirement is structurally malformed.
================================================================================
```

8. Glossary

DPLL	Davis-Putnam-Logemann-Loveland backtracking algorithmic procedure
HySAT	An implementation of an SMT technique focusing on hybrid and non-linear aspects of the analysed problems.
ICD	Interface Control Document.
LTL	Linear Temporal Logic. A formal mathematical language for expressing properties of deterministic systems.
SAT	Satisfiability
SLDV	Simulink Design Verifier. An add-on to Simulink and Stateflow providing basic model verification capabilities.
SMT	Satisfiability Modulo Theories. A field of theoretical computer science concerned with algorithms for solving complex arithmetical decision problems using background knowledge of structures involved (e.g. associativity of addition).
SMV	Symbolic Model Verifier
TLC	Target Language Compiler. An internal Simulink code generation language.