HAL open science

Industrial Grade Model Checking Use Cases, Constraints, Tools and Applications

Mathieu Clabaut, Ning Ge, Nicolas Breton, Eric Jenn, Rémi Delmas, Yoann Fonteneau

▶ To cite this version:

Mathieu Clabaut, Ning Ge, Nicolas Breton, Eric Jenn, Rémi Delmas, et al.. Industrial Grade Model Checking Use Cases, Constraints, Tools and Applications. 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), Jan 2016, TOULOUSE, France. hal-01291365v1

HAL Id: hal-01291365
https://hal.science/hal-01291365v1

Submitted on 21 Mar 2016 (v1), last revised 31 Mar 2016 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Industrial Grade Model Checking

Use Cases, Constraints, Tools and Applications

Mathieu Clabaut[1], Ning Ge[2,⋆], Nicolas Breton[1], Eric Jenn[2,⋆⋆], Rémi Delmas[3], and Yoann Fonteneau[1]

[1] Systerel, Aix-en-Provence — Toulouse, France `firstname.name@systerel.fr`
[2] IRT Saint-Exupéry, Toulouse, France `firstname.name@irt-saintexupery.com`
[3] ONERA, Toulouse, France `remi.delmas@onera.fr`

Abstract. *Model checking* has made a lot of progress since its infancy. For a long time, industrial applications were still limited to some very specific domains out of which the technique bumps into the state explosion wall. Nowadays things evolve and some tools are able to tackle real world use cases outside of the known domains.
We give here the feedback collected when using *model checking* on several industrial strength use cases and give indication on how we take into account the specific domain constraints.

1 Model Checking for Industrial Problems

Model checking refers to the problem of exhaustively and automatically checking whether a given model of a system meets a given specification.

Model Checking is now an old technique which takes its ground in the mid 1970s as a response to concurrent problem analysis. It was until recently essentially confined to some specific areas, such as hardware analysis or protocol verification. Extension to other domains such as software verification has always been difficult due to the combinatorial explosion problem (the size of the space state grows exponentially with the size of the problem to analyze).

However, recent developments in a variety of fields, ranging from symbolic *model checking* to SAT solver engines and including *model checker* parallelization lead to a broader range of application in industry including software analysis.

⋆ Seconded from Systerel, Toulouse, France
⋆⋆ Seconded from Thales Avionics, Toulouse, France

1.1 Industrial Use Cases

Model checking may be used for the following use cases, depending on the tool abilities:

Safety Proof consists in verifying some properties on a system model. The model may be designed by hand or automatically derived from existing artefacts, such as `Ada`, `C`, `Simulink` or `Scade` code…
If one of the properties is not being verified, the tool provides a counterexample explaining why the property does not hold.

System Debugging. One of the great features of *model checking* is its ability to provide the user with counterexamples. Such a feature may be used in different ways, but is generally of a good help for debugging a system.
For example, when designing a complex system which should ensure a safety property (e.g. a non collision property for an automatic train system), it is very useful to debug the root concepts which should ensure the safety upon a system model on which you can analyse counterexamples.

Equivalence Checking. The ability to prove some properties upon an existing model

may be used as a way to prove the equivalence of two models.

One application of *equivalence checking* is to verify that a software design in `C` satisfies its specification, by proving the equivalence between a model of the design and a manually written model of the specification.

Another application is to verify the correctness of a tool transformation — for example a translation from one formalism to another — by proving the equivalence between the before and after transformation artefacts.

Constraint Solving. A somewhat more contrived use of *model checking* is the use of its ability to provide counterexamples as a way to solve a constraint problem. The way to do it is to ask the tool to verify that the problem has no solution. If one exists, the tool will return a counterexample which is one solution to the problem.

Test Case Generation. Another use case, derived from the preceding one, is to use a *test objective* as a constraint. The provided counterexamples then give inputs and associated oracles which fulfill these test objectives and may be used as test scenarios.

1.2 Industrial Constraints

Putting aside the many use cases listed before, industry faces many constraints which may have prevented the use of model checking until now:

Regulatory Constraints Many companies must obtain approval from a suitable authority that the system they develop is acceptably safe to operate with regards to the applicable assurance standards (CENELEC EN-50128, DO-178C,…).

As such, it must be shown that the tools who have contributed to the system or to its verification have been qualified with respect to their usage and to their contribution to the global safety. Such a qualification bears a lot of constraints upon the tools and their development process. Few of the known *model checkers* are designed with such compliance in mind.

Cost Reduction The use of formal methods and *model checking* is of interest for industry only if they lead to cost reduction or to standard compliance. In a context where standard compliance is already achieved, the only motivation left for applying formal methods is to gain significantly over costs. Such an objective may be reached either by using *model checking* in order to automate some testing steps or in a less measurable way, by rising the overall quality beyond what is required by the regulatory constraints. Proving a property is indeed really an improvement over testing it, even in the frame of standard compliance, and can lead to finding bugs that would be otherwise discovered much later and would cost a lot more to be corrected.

2 The S3 Toolbox

2.1 S3

Systerel Smart Solver[4] (S3) is Systerel's response to the aforementioned use cases and industrial constraints. S3 is constructed around:

- a high level synchronous modelling language,
- several frontends for `C`, `Ada`, `Scade`, …
- a translation tool chain from the high level language towards a bit level language,
- a proof engine working over the bit level language,
- a proof verification engine,
- several tools for animation and debugging.

The performance of the proof engine allows us and our clients to manage the proof of industrial size problems some of them we will mention in the next section. The size of those models routinely topped ten millions variables and several hundred millions of clauses.

The qualification of S3 is made possible by the use of several diversified tool-chain with some small key tools built with respect to the higher integrity constraints that may be required — namely, an equivalence model constructor, and a tool to verify the validity of the proof (see fig. 1 on the following page).

[4] S3 is maintained, developed and distributed by Systerel.

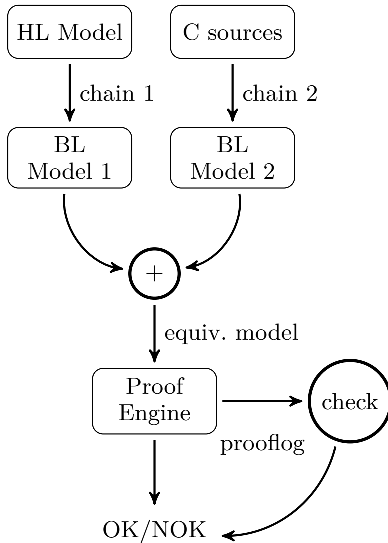

Fig. 1. Example of application for proving equivalence between C sources and a HL model specification. The two tool-chains are diversified and the software in the circles are developed with the higher level of integrity (HL = High Level, BL = Bit Level).

2.2 Floating-Point Arithmetic Library in S3

Critical applications used to use fixed-point arithmetic that requires less memory and less processor time then floating-point arithmetic (FPA) to perform non-integer computations on executing processors with no Floating-Point Unit (FPU), while leading to a limited-precision. Floating-point numbers support a trade-off between range and precision due to its formulaic representation which approximates a real number. Another advantage stands in the existence of a standard [1] based on solid mathematical grounds. Nowadays, FPA is more and more used in the space, aeronautics and automotive industries, due to the increasing complexity of the computations and because FPUs are becoming standard for most processors. Floating-point numbers are not real numbers. Floating-point operations behave in quite different way from the real counterparts, due, for instance, to rounding and cancellations [17]. Consequently, a software implementation of some mathematical expression usually provides results that are not strictly, mathematically, exact. As it is often difficult to foresee the behavior of floating-point programs, formal verification of floating-point programs is highly desired in the industry.

The basic approaches to address formal verification of floating-point programs include abstract interpretation, formal proof and bit-blasting (also called bit-flattening). Abstract interpretation partially executes program on an abstract domain. This approach performs well in program analysis with floating-point variables [2]. Formal proof supported by proof assistants is a very powerful approach that requires to be guided by highly skilled expert to direct the reasoning towards target properties. Interactive theorem provers such as ACL2, Coq, HOL Light and PVS have been applied to floating-point verification [14]. Both of abstract interpretation and formal proof approaches lack ability to generate counterexamples when the property does not hold. Bit-blasting represents floating-point operations as circuits, which are then transformed to Boolean formula with bit-wise operators to be solved by SAT solvers. Bit-blasting relying on SAT solvers is a fully automatic reasoning benefiting from counterexamples for floating-point programs. It is implemented in the Satisfiability Modulo Theories (SMT) solvers such as Z3 [10], MathSAT 5 [7], SONOLAR [15] and CBMC [5]. It is also the elementary but the most significant part of other floating-point verification strategies using SAT solvers as the back-end. Since the publication of SMT-LIB theory of binary FPA [18], solvers are starting to support it using some advanced QF_FP solving strategies, such as mixed abstraction in CBMC [5], non-conservation approximations in Z3 [13], abstraction into interval arithmetic in MathSAT [3,4], translation into non-linear reals in Realizer [16], etc.

S3 will be used to verify part of the floating-point software embedded in a rover platform, TwIRTee (a three-wheeled autonomous rover) used as the demonstrator for the INGEQUIP project. The INGEQUIP research project at the *Institut de Recherche Technologique (IRT) Saint-Exupéry* in Toulouse addresses the following equipment engineering topics: architecture modelling and evaluation, early V&V using vir-

tual platforms, and formal verification. For this purpose, we implemented an `FPA` standalone library to enable the floating-point verification by means of bit-blasting. This optimized `FPA` library will establish a solid foundation and basic strategy for our future investigation on advanced `FPA` strategies in `S3`.

The implementation of `FPA` library in `S3` is based on the `IEEE FPA` standard 754-2008 [1]. The `FPA` library in `S3` includes:

- Binary interchange format encoding that allows user-defined ranges of exponent and mantissa, including single and double precisions
- Normal, subnormal, infinity, NaN numbers
- 5 rounding directions: `roundTiesToEven`, `roundTiesToAway`, `roundTowardPositive`, `roundTowardNegative`, `roundTowardZero`
- Comparison operations: `Equal`, `NotEqual`, `Greater`, `GreaterEqual`, `Less` and `LessEqual`
- Arithmetic operations: `Addition`, `Subtraction`, `Multiplication`, `Division` and `SquareRoot`
- Conversion operations: `convertIntToFloat` and `convertFloatToInt`
- Trigonometric operations: `Sin`, `Cos`, and `Tan`
- Default exception handling: invalid operations, division by zero, overflow and underflow

The *SquareRoot* and trigonometric operations are implemented by means of both interpolation table and the function proposed by Cody and Waite [8].

3 Industrial Applications

3.1 Railway Use Case — Interlocking Safety Proof

An interlocking is an arrangement of signal apparatuses that prevents conflicting movements through an arrangement of tracks such as junctions or crossings. An interlocking is designed so that it is impossible to display a signal to proceed unless the route to be used is proven safe.

The main challenge is to ensure that whatever scenario will happen, the designed interlocking will stay safe. And albeit its apparent simplicity one quickly understands the real underlying complexity — whatever rule you may imagine, it looks like one can always find a scenario breaking it (train can go backwards, can sometime appear to *fly* due to concurrency artefacts, …)

Instanciation Design Process In a recent work, we have considered a Computer Based Interlocking (CBI) designed through an instantiation process. In such a process, the *signaling principles* are captured by the engineers to produce the Generic Design in some suitable language. This design contains a set of generic descriptions of code parts that an interlocking system shall execute for some specific object of the system such as signals, switches, routes,… Each of these generic descriptions is given in a parametrized form which allows for the specificities of the object to which it applies. For example, the generic design for a route will probably be parametrized with the list of switches contained in the route. The generic design is thus specific to a set of signaling principles, but independent of a particular track layout.

The generic design is then instantiated upon a particular track layout to produce the running software.

Verification Process We graft our verification process onto the instantiation scheme by designing a Generic Safety Specification made of:

- some high level proof obligations (3 to 4 properties about absence of derailments and collisions),
- some intermediate predicates modeling the domain (such as topological predicates which encode the track layout, integration predicates which associate input and output variables with their object instance, helper predicates conveying high level relations between objects such as reachability,…)
- an environment model upon which the proof obligations are expressed (trains behavior,…).

and then instantiate those properties and models with the specific track layout data. The overall uncertified process is given in figure 2.

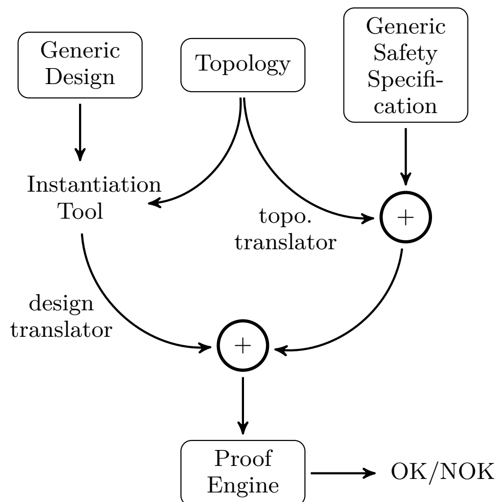

Fig. 2. Overview of an uncertified verification process for an *instantiated design*. The specific *topology translator* tool translates topological data in an High Level Language model. The specific *design translator* tool translates the instantiated design in a High Level Language model. The concatenation of all models is then feed to S3 proof engine. Manual lemmas may be added into the Generic Safety Specification to help the proof.

As an example of intermediate predicates used in the Generic Safety Specification, the fact that "every block belonging to a given route" is free can be expressed as:

```
ALL b:BLOCKS (contains(route,b) -> free(b))
```

where `ALL` is the universal quantifier, `b:BLOCKS` indicates that `b` shall range on every instances of the `BLOCKS`, and `->` is the implication operator.

Results The proof process was applied to a CBI of more than 200 routes and permit us to pinpoint 3 safety counterexamples which where then corrected whether in the design or in the topology data. The overall proof took less than 25 mn in the worst case, which would go as high as 1 hour for a certified process implying the use of a second tool chain, an equivalence verification and a prooflog check.

The verification was successful and proved to be usable enough to assist the design team in debugging the CBI. It also paved the way for a certified verification that would be used by the safety team.

In the competitive interlocking market, this would clearly give an edge to the CBI manufacturer when comparing to other solutions integrating or not formal verification.

3.2 Aerospace and Automotive Use Case

The floating-point verification by S3 has been applied to two case studies: the avionic triplex sensor voter and the automatic rover protection system. It is also used to automatically generate the test cases.

Triplex Sensor Voter Case Study The triplex sensor voter[5] is used in a common form of redundant aircraft system Triplex Modular Redundancy (`TMR`), which relies on three identical sensors to compute an output value from the three input values by the voter. It is implemented using linear arithmetic operations as well as conditional expressions (such as saturation). Its formal analysis covers functional and non-functional properties including stability, absence of runtime errors, and also to parameterize certain parts of the model to help the formal analysis. The formal analysis of triplex sensor voter was first studied by Dajani-Brown et al. in [9], where real values were abstracted by integer values and integrators were not used. In [11], Dierkes analyzed the `Simulink` model with real numbers by both simulation and formal verification, then estimated the impact of rounding errors caused by the floating-point implementation using `SMT` solvers and abstract interpretation. In [6], Champion et al. strengthened the stability property by generating lemmas using a property-directed approach.

[5] Triplex sensor voter case study is provided by Rockwell Collins to make it publicly available to the research community.

In our work, we start from a SCADE model of the voter and translate it to HLL model using the SCADE-translator. Thanks to our FPA library, the HLL model is then verified using the S3 solver. In parallel, we use SMT-solvers Z3 v4.4, MathSAT 5 and SONOLAR to verify the SMT model. Experiments are carried out using both simple and double precision floating-point numbers with or without subnormal numbers and with different rounding modes.

The results show that neither Z3 v4.4 (bit-blasting strategy, floating-point strategy) nor MathSAT 5 (bit-blasting strategy, abstract CDCL algorithm) or SONOLAR are able to handle the step instance, be it in simple or double precision. We managed to prove the inductive instance using a combination of SONOLAR bit-blasting to a CNF and a pure SAT solver (glucose 4.0 multithread with 8 threads and aggressive restart strategies, satellite preprocessing) in 10min of computation for the simple precision instance, and 4h15min of computation for the double precision instance (wall clock time). S3 proved the step instance in 6min using glucose 4.0 and 5mins using S3's own solver for the simple precision instance, and in 9h32min using glucose 4.0 for the double precision instance.

Test Generation In order to conform to the domain standards, companies sometimes have to setup processes where a structural test coverage must be achieved by mean of manual testing activities. Those activities consist mainly in finding the inputs and oracles such that the structural coverage criterion is respected.

We successfully used *model checking* with S3 in order to setup an automatic process of finding the set of tests that will achieve the structural node coverage on a Scade model, where the criterion was that at most one entry of a given node should change at once. Automation of such a task leads to great *cost reductions.*

Formal Verification for ARP Case Study The Automatic Rover Protection (ARP) system, a simple collision avoidance function, is developed for the twIRTee platform in the INGEQUIP project. It performs a predefined trajectory (a "mission") on a predefined track and avoid collisions with other rovers. The set of tracks are statically defined and embedded in the rover. The ARP is based on the following principles:

1. Missions (trajectories) are precomputed using an adapted shortest path algorithm[12].
2. A rover shall only move on a reserved path if there is a free reserved space ahead.
3. A reserved path is a stack of reserved nodes.
4. A rover resends reservation request to get the exclusive access to the tracks located ahead of it, if there is not enough reserved space (D_REQ) ahead.
5. A rover stops if there is not enough reserved space (D_STOP) ahead.
6. A rover reserves enough space (D_RSV) ahead for each request.
7. In order to ensure a consistent management of node reservations, a distribution strategy using id-priority is implemented.

In this case, S3 is exploited to verify several properties applicable to all independent rovers, such as $\mathbf{P}_1$: *rovers are never granted simultaneous access to the same area* (safety), and $\mathbf{P}_2$: *all rovers eventually reach their destinations* (bounded liveness), etc.

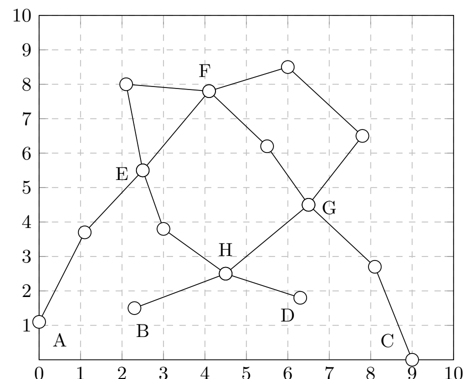

Fig. 3. Rover Trajectories for ARP

We present experimental results from a ARP case study with two rovers. The trajectories of rovers within an enclosed space of 10m×10m are predefined (see fig. 3). The rover R_1 proceeds with a speed $v_1 = 0.4$m/s from node A, through node E, F, G and H, then stops at node

B, while the rover R_2 proceeds with a speed $v_2 = 0.3m/s$ from node C, through nodes G, F, E and H, then stops at node D. The length of each trajectory is about 20m. As the first step of our study, we assume that the space occupied by rovers are ignored, and that the clocks of rovers are synchronized. Consequently, rovers are considered to perform actions synchronously. We proved the property $\mathbf{P}_1$ in almost no time using induction and the property $\mathbf{P}_2$ in several seconds using bounded model checking in `S3`.

3.3 Other Use Case — Constraint Solving for Matrix Wiring

S3 was used as a constraint solver to automate finding of solution to a wiring problem over a 100×100 matrix with several layers of wires and several limitations over the deformations that wires may undergo.

Once the counterexample obtained, a small and easy tool allowed to translate it into a *netlist*.

4 Conclusion and Future Work

Our experience with *model checking* has shown that S3 is a great tool to manage different types of industrial size problems with respect to their specific regulatory constraints. However, the tool is still at pains for some peculiar problems and has some intrinsic limitations that prevent its application on a wider range of problems, one of it being its current inability to handle designs which make use of floating-point arithmetic.

In order to tackle the floating-point limitation, we have designed an `FPA` library and integrated it in S3. Our next goal is to investigate how well it works on a wider range of industrial designs from automotive, aeronautic, industry, energy… and to establish benchmark (or reuse existing `SMT` benchmarks) to evaluate the performance of floating-point verification by S3.

The proof engine at the core of S3 is also undergoing heavy work in order to improve its efficiency for big size models.

This significant improvement, associated to the heavy work ongoing on the proof engine at the core of S3, shows great promises for dealing with future industrial challenges.

References

1. IEEE Standards Association. Ieee standard for floating-point arithmetic. 2008.
2. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN Notices*, volume 38, pages 196–207. ACM, 2003.
3. Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Interpolation-based verification of floating-point programs with abstract cdcl. In *Static Analysis*, pages 412–432. Springer, 2013.
4. Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 45(2):213–245, 2014.
5. Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed abstractions for floating-point arithmetic. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 69–76. IEEE, 2009.
6. Adrien Champion, Rémi Delmas, and Michael Dierkes. Generating property-directed potential invariants by quantifier elimination in a k-induction-based framework. *Science of Computer Programming*, 103:71–87, 2015.
7. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.
8. W.J. Cody and W.M.C. Waite. *Software manual for the elementary functions*. Prentice-Hall series in computational mathematics. Prentice-Hall, 1980.
9. Samar Dajani-Brown, Darren Cofer, Gary Hartmann, and Steve Pratt. Formal modeling and analysis of an avionics triplex sensor voter. In *Model Checking Software*, pages 34–48. Springer, 2003.
10. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
11. Michael Dierkes. Formal analysis of a triplex sensor voter in an industrial context. In *Formal*

Methods for Industrial Critical Systems, pages 102–116. Springer, 2011.

12. Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
13. Andreas Fröhlich, Armin Biere, Christoph M Wintersteiger, and Youssef Hamadi. Stochastic local search for satisfiability modulo theories. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.
14. John Harrison. Floating-point verification using theorem proving. In *Formal Methods for Hardware Verification*, pages 211–242. Springer, 2006.
15. F Lapschies, J Peleska, E Gorbachuk, and T Mangels. sonolar smt-solver. *Satisfiability modulo theories competition; system description*, 2012.
16. Miriam Leeser, Sayan Mukherjee, Jaideep Ramachandran, and Thomas Wahl. Make it real: Effective floating-point reasoning via exact arithmetic. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–4. IEEE, 2014.
17. David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12, 2008.
18. Philipp Rümmer and Thomas Wahl. An smt-lib theory of binary floating-point arithmetic. In *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.