Integrated Formal Verification of Safety-Critical Software

Ning Ge · Eric Jenn · Nicolas Breton · Yoann Fonteneau

Abstract This work presents a formal verification process based on the Systerel Smart Solver (S3) toolset for the development of safety-critical embedded software. In order to guarantee the correctness of the implementation of a set of textual requirements, the process integrates different verification techniques (inductive proof, bounded model checking, test case generation and equivalence proof) to handle different types of properties at their best capacities. It is aimed at the verification of properties at system, design, and code levels. To handle the floating-point arithmetic (FPA) in both the design and the code, an FPA library is designed and implemented in S3. This work is illustrated on an Automatic Rover Protection (ARP) system implemented on-board a robot. Focus is placed on the verification of safety and functional properties and on the equivalence proof between the design model and the generated code.

Keywords Integration, Formal verification, Safety-critical embedded software, HLL, S3, SAT, Floating-point arithmetic

Ning Ge
IRT-Saint Exupéry, Toulouse, France
Systerel, Toulouse, France
E-mail: ning.ge@irt-saintexupery.com | ning.ge@systerel.fr

Eric Jenn
IRT-Saint Exupéry, Toulouse, France
Thales Avionics, Toulouse, France
E-mail: eric.jenn@irt-saintexupery.com

Nicolas Breton
Systerel, Toulouse, France
E-mail: nicolas.breton@systerel.com

Yoann Fonteneau
Systerel, Toulouse, France
E-mail: yoann.fonteneau@systerel.com

1 Introduction

Even though significant progress has been made towards the integration of formal methods in the industry of safety critical systems, their usability is still impaired by their cost. It makes sense to formally verify the safety-critical parts of a system by combining different verification techniques at their best capacities. Moreover, the hope is that once the initial integration is done, subsequent verifications can be achieved at significantly lower costs. This work investigates how this could be achieved using a formal verification toolset, Systerel Smart Solver (S3)[1], and draw some lessons from our experience.

S3 [14] is built around a synchronous language and a model checker (S3-core) based on SAT [4] techniques. As the proof engine, S3-core relies on Bounded Model Checking (BMC) [3] and K-induction [38,6] techniques. S3 supports different activities of a software development process: property proof, equivalence proof, automatic test case generation, simulation, debug, and provides necessary elements to comply with the software certification processes. S3 is usually applied in the development process relying on SCADE [11]/Lustre [27] design models and the implementations in C and Ada. It has been exploited as industrial solutions to formally verify the railway signalling systems for years by various industrial companies in this field.

Critical applications used to rely on fixed-point arithmetic that requires less memory and less processor time than 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

[1] S3 is maintained, developed and distributed by Systerel (http://www.systerel.fr/).

between range and precision thanks to its formulaic representation which approximates a real number, and its standardization based on solid mathematical grounds [1]. Nowadays, FPA is more and more used in the space, aeronautics and automotive industries, as required by the increasing complexity of the computations and because FPUs are becoming standard for most processors. However, a common problem for the safety-critical software is the erroneousness due to the rounding and exceptions in floating-point computations [33]. It is necessary to provide verification means to guarantee the correctness of critical software with floating-point arithmetic.

This article is aimed at implementing a formal verification process, and providing approaches & tools to guarantee the correctness of the safety-critical embedded software. Our main contributions are twofold. On the one hand, we have designed and implemented a library of floating-point arithmetic (FPA-Lib) in S3 that is compliant with the IEEE Standard for FPA [1]. To evaluate the performance of this implementation, we show the experimental results on a triplex sensor voter using our FPA-Lib and other SAT/SMT solvers. On the other hand, we presents the integrated verification process using S3 on an Automatic Rover Protection (ARP) system that is deployed on a three-wheeled robot. Focus of the latter is placed on three main activities: (1) formally specify the textual requirements of the embedded software, (2) ensure correct design of the textual requirements by proving properties using appropriate formal techniques, and (3) guarantee the compliance of the generated code with respect to the design model by proving the equivalence between the design model and the generated code. An additional purpose of this work is to make the ARP use case publicly available to the research community.

This article is organized as follows: Sect. 2 presents the S3 toolset; Sect. 3 depicts the design and implementation of floating-point arithmetic in S3, and shows experimental results on the triplex sensor voter; Sect. 4 describes the ARP use case; Sect. 5 exposes the verification of safety and functional properties using inductive proof, BMC, and test cases generation techniques to guarantee the correctness of the design model; Sect. 6 illustrates the process of equivalence proof to guarantee the correctness of the generated C code; Sect. 7 discusses the experience derived from the verification activities, and Sect. 8 gives some concluding remarks and discusses perspectives.

2 The HLL Modeling Language and S3 Toolset

This work relies on the S3 verification toolset and its modelling language, called HLL (High Level Language). The architecture of S3 is depicted in Figure 1. S3 facilitates the construction of formal verification solutions compliant with certification standards, e.g. DO178C [22]. Towards this goal, S3 is organized in a set of small, independent components, from which the most critical ones - an equivalence model constructor, and a tool to verify the validity of the proof - are developed according to the highest integrity levels. We briefly introduce these components in S3 in the following part:

- A synchronous declarative language similar to the Lustre language [27], HLL, that is used to model the system, its environment constraints as well as expected properties on the system. To give an overview of the language constructs, Figure 2 shows the HLL model of a saturated counter and its property on the range of output value (respectively in the namespaces `Counter` and `Counter_Verif`). The counter reacts to the input command (modelled as an HLL enumeration): incrementation (`INC`), decrementation (`DEC`) or reset (`RESET`). The saturation range is defined by HLL constants. The behaviour of the counter is initialized by the value zero and then periodically updated. The effect of `INC` and `RESET` are directly defined in the schedule, while the effect of `DEC` is defined as a function contract by using HLL constraints and an intermediate variable `dec_input` of the HLL block `Fun_dec()`, without final implementation.
- Several translators to convert models (SCADE/Lustre) and code (C and Ada) to HLL.
- Two expanders to translate the HLL models into a bit level language, called LLL (Low Level Language) that only preserves Boolean streams and is restricted to three bitwise operators: negation, implication and equivalence.
- A SAT-based proof engine, named S3-core, to check LLL models. The performance of this proof engine allows users to manage the proof of industrial size problems: the size of those models routinely attains ten millions variables and several hundred millions clauses.
- Tools to build equivalence proof between models, or between models and code.
- Tools to animate and debug models (cex-simulator, why) that provide counterexamples. The counterexamples allow the users to developer simulators at user level.

S3 supports the following activities of a typical formal development process:

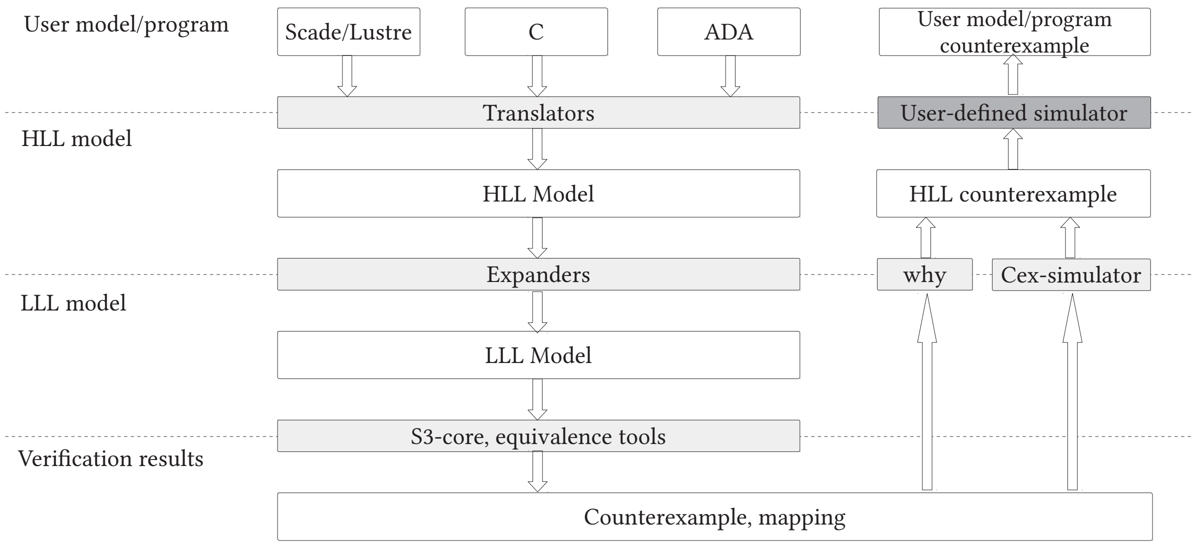

Fig. 1: The S3 Toolset

```
Namespaces:
Counter
{
Types:              enum {INC, DEC, RESET} Command;
Inputs:             Command in;
Constants:          int C_MIN := 0;
                    int C_MAX := 100;
Declarations:       int unsigned 32 cnt;
Outputs:            cnt;
Declarations:       int unsigned 32 dec_input;
Blocks:             Fun_dec(int pre_v) -> (int v)
                    {
                          v:= dec_input;
                    }
Constraints:
  ALL f: Fun_dec (f.pre_v = C_MIN -> f.v = C_MIN);
  ALL f: Fun_dec (f.pre_v > C_MIN -> f.v = f.pre_v1 – 1);
Definitions:
  cnt := 0,   if in==RESET  then C_MIN
            elif in==DEC    then Fun_dec(cnt)
            elif in==INC    then
              if cnt==C_MAX then C_MAX else cnt + 1
            else cnt;
}

Counter_Verif
{
Proof Obligations:
  Counter::cnt >= Counter::C_MIN;
  Counter::cnt <= Counter::C_MAX;
}
```

Fig. 2: An Example HLL Model

– **Static detection of runtime errors and standard conformance check**, including array bounds check, range check, division by zero check, over and underflow check, output and constraint initialization check, etc. Proof obligations are also generated to ensure that the generated HLL models show no undefined behavior with respect to the semantics of the source language.
– **Property proof**: Figure 3 presents the workflow of property proof. The design model, e.g. Lustre, is translated into an HLL model. Combined with properties expressed in HLL as well, it is then expanded to an LLL model that is fed to the S3-core. If a property is falsified, a generated counterexample can be simulated at the HLL level. This activity will be detailed in Sect. 5.

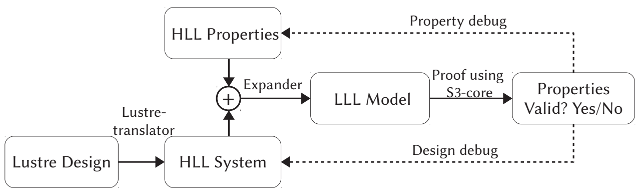

Fig. 3: Process of Property Proof

– **Equivalence proof**: Figure 4 presents the process of proving the equivalence between the design, e.g. the Lustre model, and the generated/implemented code, e.g. the C code. Models and code are translated into HLL models, which are then expanded to LLL models using diversified expanders[2]. Equivalence models are respectively constructed at the

[2] The diversified expanders are designed and implemented by different teams using different programming languages.

HLL level and the LLL level. Equivalence proof is performed on one of the equivalence models or both. This activity will be detailed in Sect. 6.

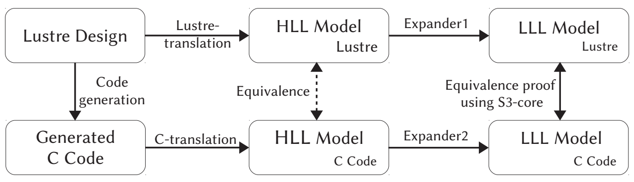

Fig. 4: Process of Equivalence Proof

– **Test case generation**: Test scenarios are generated from properties expressed as test goals using BMC. This activity will be detailed in Sect. 5.3.

3 Floating-Point Arithmetic Library in S3

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 [33]. 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 behaviour of floating-point programs, formal verification of floating-point programs is mandatory.

This part presents a new component in S3, the FPA library (FPA-Lib). This library addresses the verification of embedded software with FPA by means of bit-blasting (also called bit-flattening), which is a classic method that translates bit-vector formulas into propositional logic expressions.

3.1 Implementation of Floating-Point Arithmetic

The basic approaches to address formal verification of floating-point programs include abstract interpretation & static analysis, formal proof and bit-blasting. Abstract interpretation partially executes a program on an abstract domain. This approach performs well in static program analysis with floating-point variables to ensure that the critical software never executes an instruction with "undefined" or "fatal error" behavior, such as out-of-bounds accesses to arrays, overflows or division by zero [7]. Formal proof supported by proof assistants is a very powerful approach, but requires the guide from 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 [28]. 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 also implemented in the SMT solvers such as Z3 [19], MathSAT 5 [13], SONOLAR [29], CBMC [10], etc. Bit-blasting is the elementary but the most significant part of other floating-point verification strategies using SAT solvers as the backend. Since the publication of SMT-LIB theory of binary FPA [35], solvers are starting to support it using some advanced QF_FP solving strategies, such as mixed abstraction in CBMC [10], non-conservation approximations in Z3 [23], abstraction into interval arithmetic in MathSAT [8,9], translation into non-linear reals in Realizer [30], etc.

We have implemented an FPA standalone library to enable the floating-point verification by means of bit-blasting. This optimized FPA library establishes 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]. A single(double) precision floating-point number is expressed as a binary of 32(64) bits that contains a sign of 1-bit, a biased exponent of 8(11)-bit, and a mantissa of 23(52)-bit. The HLL modelling of the numbers is shown as follows.

```
Struct {                  | Struct {
s: bool;                  | s: bool;
e: int unsigned 8;        | e: int unsigned 11;
m: int unsigned 23        | m: int unsigned 52
} float;                  | } double;
```

We briefly present the implementation of addition/-subtraction to illustrate the principles of encoding. As shown in Figure 5, this implementation follows three steps: (1) align, to shift mantissa and render exponents equal; (2) addition/subtraction, to added/subtracted resulting mantissas; and (3) round, to shorten mantissa and obtain a number in $\mathbb{F}$.

The FPA library in S3 includes the constructs in Table 1. The square root and trigonometric operations are implemented using both the interpolation table and the function proposed by Cody & Waite [15].

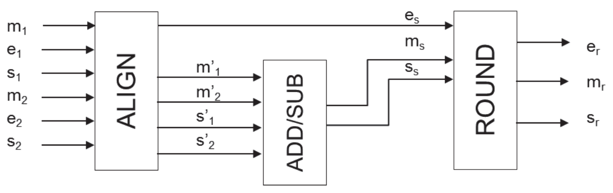

Fig. 5: Implementation of Addition/Subtraction in FPA-Lib

Table 1: Inclusions of FPA-Lib

Inclusions	Implementations
Binary interchange format	single/double precisions, user defined range
Numbers	Normal, subnormal, infinity, NaN
Rounding directions	roundTiesToEven, roundTiesToOdd, roundTiesToAway, roundTowardPositive, roundTowardNegative, roundTowardZero
Comparison operations	Equal, NotEqual, Greater, GreaterEqual, Less, LessEqual
Arithmetic operations	Addition, Subtraction, Multiplication, Division, SquareRoot
Conversion operations	convertIntToFloat, convertFloatToInt
Trigonometric operations	sin, cos, tan, ctan, etc.
Exception handling	invalid operations, division by zero, overflow and underflow

3.2 Evaluation of FPA-Lib on Triplex Sensor Voter

The triplex sensor voter[3] 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 [18], where real values were abstracted by integer values and integrators were not used. The work [20] analyzed the Simulink model with real numbers by both simulation and formal verification, and then estimated the impact of rounding errors caused by the floating-point implementation using SMT solvers and abstract interpretation. The work [12] strengthened the stability property by generating lemmas using a property-directed approach.

In our work, we start from a Lustre model of the voter and translate it to HLL using the SCADE(Lustre)-translator in S3. The FPA library is then applied to the verification of the stability properties by S3. Relying on this use case, we also evaluate the SMT solvers Z3 v4.4, MathSAT 5 and SONOLAR. Experiments are carried out using floating-point numbers of simple & double precisions, normal & subnormal numbers, and different rounding modes.

The experimental results in Table 2 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 inductive(also called step) instance in the K-induction proof, 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 inductive 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.

Note that this evaluation on the voter case does not lead to any conclusion about the performance of the concerning solvers, as the performance of SAT/SMT solvers depends on the target problem.

4 Specification and Design of the Automatic Rover Protection System

4.1 The Context of Use Case

The verification approach presented in this article has been applied on the Automatic Rover Protection (ARP) System embedded in TwIRTee, a small three-wheeled robot (or "rover") as the demonstrator of the the INGEQUIP project[4]. It is used to experiment and evaluate

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

[4] The INGEQUIP project is conducted at the IRT Saint-Exupéry.

Table 2: Experimental Results of Triplex Sensor Voter

Solver	**Strategy**	**Simple Precision**	**Double Precision**
S3	Bit-blasting	5m	(time out)
S3 + Glucose 4.0	Bit-blasting	6m	9h32m
Z3 v4.4	Bit-blasting + floating point	(time out)	(time out)
MathSAT5	Bit-blasting + abstract CDCL	(time out)	(time out)
SONOLAR	bit-blasting	(time out)	(time out)
SONOLAR+Glucose 4.0	Bit-blasting + aggressive restart + satellite preprocessing	10m	4h15m

various methods and tools in the domain of hardware/-software co-design, virtual integration of equipments [16], and formal verification [14,26,25,21,24]. TwIRTee's architecture and its software and hardware components are representative of typical aeronautical, spatial and automotive systems [17]. The overall system is composed of a unique stationary supervision station and a set of TwIRTee rovers moving in a controlled environment (Figure 6). The architecture of rover is composed of a mission and a power control subsystems. The power control subsystem is in charge of power supply, motor control and sensor acquisition. The mission subsystem is composed of a pair of redundant channels A and B. Each channel contains a monitoring unit (MON) in charge of monitoring the data and a command unit (COM) in charge of calculating commands for the rovers. The mission and power control systems communicate via CAN bus.

In the nominal case, each rover moves autonomously on a set of predefined tracks so as to perform its missions, i.e., moving from a start waypoint to a target waypoint under speed and positioning constraints. In this system, the ARP function is aimed at preventing collisions between the rovers. It generates the maximal accelerations and minimal decelerations that are taken into account by the rover trajectory management function. The communication between rovers are carried out via WIFI.

Here, we introduce several terms used in the paper. A **mission** is defined by a list of **waypoints** to be "passed-by" by the rover. A **segment**, defined by a couple of waypoints on the track, corresponds to a straight path. Segments only intersect at waypoints. The set of all waypoints and segments constitutes a **map**. Dedicated monitoring mechanisms ensure that if the rover gets out of the track, it is placed in a stopped safe mode and the supervisor is alerted. Accordingly, we consider that all displacements of rovers comply with the map. In the use case, we consider 3 rovers moving on a map of 45 segments and 150 waypoints. A mission contains at most 20 waypoints.

4.2 System-Level Safety and Functional Requirements

The requirements of ARP use case comes from the industrial partners of the INGEQUIP project. The safety requirements are defined in Table 3 and the functional & performance requirements are defined in Table 4.

The ARP is expected to ensure system-level safety requirement (REQ-SAF-1) in Table 3. This requirement states that at any time, the minimal distance between the centers of two rovers shall be greater or equal to 0.4m. It is split in two subsets of requirements: one is about the exclusive access to segments (REQ-SAF-1-1) and the other is about the design of a map (IR-F1, IR-F2 and IR-F3). The compliance with the requirements of map data is under the map supplier's responsibility.

The system-level functional requirements in Table 4 (REQ-F1 and REQ-F2) are mainly about excluding trivial implementations that would prevent collisions by, e.g., freezing all rovers. In the same manner, REQ-QoS-1 is introduced to guarantee the performance of the design, and to prevent trivial solutions of anti-collision, e.g., by performing missions sequentially. Note that the ARP is not to schedule the movement of the rovers but to ensure safety. Accordingly, if missions are schedulable, they shall remain schedulable with the ARP.

4.3 System Design Choice

Missions are elaborated off-line and transmitted via the supervision station. They are considered to be validated on-board (according to the REQ-F1 in Table 4). To ensure the main safety requirements, separation of rovers is implemented as follows: a rover may only enter a segment if it has been granted exclusive access

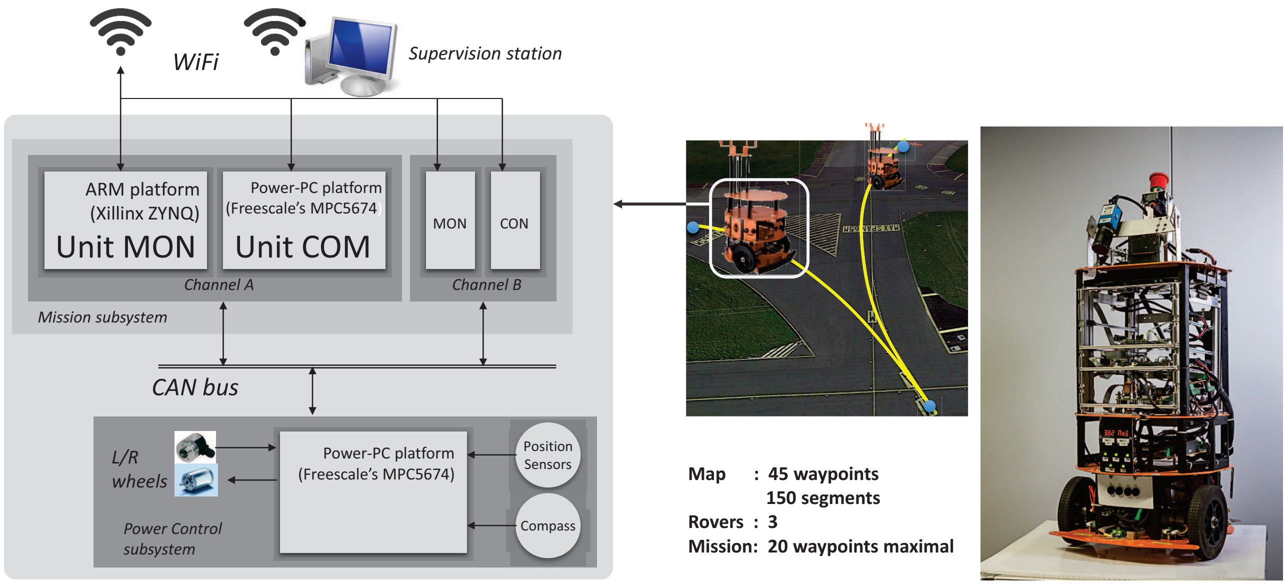

Fig. 6: Overview of the TwIRTee System

Table 3: System-Level Safety Requirements

REQ-ID	REQ
REQ-SAF-1	**Minimal separation**: At any time, the minimal distance between the centers of two rovers shall be greater or equal to 0.4m.
REQ-SAF-1-1	**Exclusive access to segment**: The ARP shall consider that it has been granted exclusive access to a segment.
IR-F1	**Length of segment**: The length of a segment shall be greater or equal to 0.4m.
IR-F2	**Distance between waypoint and segment**: The distance between a segment and a waypoint on a non-continuous segment shall be greater or equal to 0.4m.
IR-F3	**No intersection**: There shall not be any intersection between two segments.

Table 4: System-Level Functional and Performance Requirements

REQ-ID	REQ
REQ-F1	Missions shall be structurally deadlock free.
REQ-F1-1	The initial waypoints of missions shall not be the same.
REQ-F1-2	The end waypoints of missions shall not be the same.
REQ-F1-3	The end waypoint of a rover's mission shall not exist in the missions of other rovers.
REQ-F2	**No deadlock**: At any time, if the definitions of scheduled missions are free of deadlocks, a deadlock shall not occur due to the ARP.
REQ-QoS-1	**Fairness**: At any time, any rover shall be given the opportunity to move.

to both the beginning and the end waypoints of the segment. As waypoints are global resources shared by all rovers, their reservation is ensured at system-level. Our system is designed as globally asynchronous and locally synchronous. Usually the synchronous programming schema used in synchronous languages, such as Lustre and HLL, supposes that time is defined as a sequence of instants. To preserve determinism, these languages use the concept of instantaneous broadcast [2] when several processes in parallel communicate, which means that message reception is synchronous (or simultaneous) with their emission.

To support the synchrony hypothesis, we rely on two components:

– A distributed clock synchronization protocol to guarantee a common clock for all units. This protocol has been modelled and verified using model checking relying on TINA[5].
– An implementation of the Physically Asynchronous Logically Synchronous (PALS) protocol [37]. The PALS protocol is aimed at providing logical synchrony on a physically asynchronous system. From a programming perspective, under the PALS protocol, a system composed of n units behaves as if all units were all ideally synchronized. The correctness of PALS protocol for distributed real-time systems has been proved by the work [32].

4.4 High-Level Software Requirements and Software Design

During the software design process, the system-level requirements are refined into a set of High-Level software Requirements (HLRs), given in Table 5.

The HLRs represent "what" to be implemented, while the Low-Level Requirements (LLRs) represent "how" to implement it. In this work, the LLRs are expressed by a Lustre model[6]. Some figures about the size of the design are provided. For an ARP system containing 3 rovers and missions of at most 20 waypoints performed on a map of 45 waypoints and 150 segments, there are about 50 variables and 1700 lines of code in the Lustre model. For space reasons, the Lustre model is not presented in the paper[7].We briefly describe some of its key points.

In order to validate the design by simulation, the C code generated from the Lustre design has been embedded in a simulation model developed in Scicos language[8]. Figure 7 shows the Scicos model. Here we use this simulation model to explain the architecture of our design. The ARP is split in two parts: the decision model that manages segments reservation and the behavior model that calculates the speed and position of the rover with respect to the reservation decision. As mentioned in Sect. 4.3, the problem of reserving a track segment can be reduced to the problem of managing access to critical sections in a distributed system. In our design, this problem is solved by decomposing time into "time-slots" and allocating a dedicated reservation slot to each rover: so that only one rover at a time can perform a reservation. Each time-slot is split in four sub-slots respectively for request, reply, reservation and empty tasks. For example, if there are two rovers (R_1 and R_2) in the system, the first time-slot (sub-slots t0 - t3) is assigned to R_1, while the second time-slot (sub-slots t4 - t7) is assigned to R_2.

5 Property Verification against Lustre Design

We have specified the system and produced a candidate validated Lustre design model in Sect. 4. Before generating C code from the Lustre model using lus2c generator, it is required to guarantee that this design actually complies with the set of requirements. S3 allows the users to formally express these requirements, and verify them against the design model. The verification process combines inductive proof, BMC, test case generation and equivalence proof techniques. The first three techniques are used to verify properties of the design model. The equivalence proof technique is applied to prove the correctness of the generated code againts the set of requirements by checking the equivalence between the design model and the generated code. We illustrate the property verification in this section and present the equivalence proof in Sect. 6.

5.1 The Workflow of Property Verification

Figure 8 depicts the workflow of property verification using S3. The Lustre model is translated into an HLL model, to which properties and environment constraints expressed in HLL are concatenated[9]. The HLL model is then expanded to the LLL model used as the input of the S3-core. This verification workflow can be split in two phases: first, the properties are checked for a certain time length n. If no property is violated, n is increased until either a counterexample (cex) is found, or some pre-known upper bound of n is reached. In case a safety property[10] fails, a cex in the form of a sequence of states is generated, where the last state

[5] http://projects.laas.fr/tina/home.php
[6] With respect to the DO178, the Lustre model is considered to express LLRs, since the source code is directly generated from the model with no other interpretation/refinement.
[7] Contact the authors for the specification document, design model and formal properties.
[8] http://www.scicos.org/

[9] It's the verifier's duty to translate the natural language requirements to the HLL properties.
[10] Usually, the safety referred by requirements means the system is safe, while the safety referred by properties is related to the deterministic process. Here is the latter case.

Table 5: High-Level Software Requirements

REQ-ID	REQ
HLR-01	**Mission validation**: The ARP shall validate the missions to be executed. A mission is an ordered set of waypoint indexes. (HLR-01-1) The mission shall have a starting waypoint. (HLR-01-2) The mission shall refer to existing waypoints in the map. (HLR-01-3) The mission shall not successively refer to the same waypoint. (HLR-01-4) Each waypoint in a mission shall have unique precedent waypoint except the starting waypoing (referred to as continuity in this document).
HLR-02	**Motor Request**: The ARP shall control the motor using one command out of emergency braking, acceleration, and deceleration.
HLR-03	**Emergency braking**: The ARP function shall send a non-null emergency brake request to the motor control if the distance to the end of the reserved area is less than or equal to [D_BRK] and the reserved end is not the mission end, or if the rover is at the reserved end.
HLR-04	**Deceleration**: The ARP function shall send a non-null deceleration request to the motor control subsystem if the distance to the end of the reserved area is less than or equal to [D_DEC] and greater than [D_BRK], and the reserved end is not the mission end.
HLR-05	**Acceleration**: The ARP function shall send a non-null acceleration request to the motor control if the distance to the end of the reserved area is greater than [D_DEC], or if the distance to the end of the reserved area is less than or equal to [D_DEC] and the reserved end is the mission end.
HLR-06	**Inside Reserved Area**: The ARP shall only allow a rover to enter an area that has been previously reserved. (HLR-06-1) The rover position shall be in front of or at the initial position of the reserved area. (HRL-06-2) The rover position shall be behind or at the final position of the reserved area. (HLR-06-3) The initial waypoint of the reserved area shall be reserved. (HLR-06-4) The final waypoint of the reserved area shall be reserved.
HLR-07	**Desired Reservation Zone**: At any time, the ARP shall require segments that enclose a zone of length [D_RSV] in front of the rover.
HLR-08	**Request of waypoints**: The ARP shall send reservation request for all the waypoints in the desired reservation area.
HLR-09	**Reply to requests**: The ARP shall reply to reservation requests sent by other rovers. It shall acknowledge positively (accept) a reservation for a waypoint if and only if the waypoint it not currently reserved by the local rover. The acknowledgement shall contain a continuous set of waypoints.
HLR-10	**Reservation of waypoints**: The ARP shall reserve a waypoint once it has received positive reservation acknowledgement from all other rovers.
HLR-11	**Release of waypoints**: The ARP shall release the waypoints of a segment as soon as the segment is on longer in its reserved area.
HLR-12	**Fairness of reservation**: The ARP shall send waypoint reservation requests when its reservation slot is activated. The APR shall have the possibility to perform a reservation if the required waypoint is not reserved by other rovers.
HLR-13	**End of mission**: The ARP shall ensure that the scheduled mission is completed within worst case mission time (WCMT).

contradicts the property. The cex trace is then directly exploited to debug the property, the design model, or the environment constraints.

The BMC represents a partial decision procedure for a model checking problem, which is not complete. The completeness of a safety property can be achieved with k-inductive proof based on strengthening inductive invariants (also referred to as lemmas hereafter) if needed[11]. The k-induction relies on an iterative process to search for lemmas by analyzing the repeatedly produced step counterexamples, until the proof is com-

[11] Lemma searching is not a must. It is possible that a property is k-inductive.

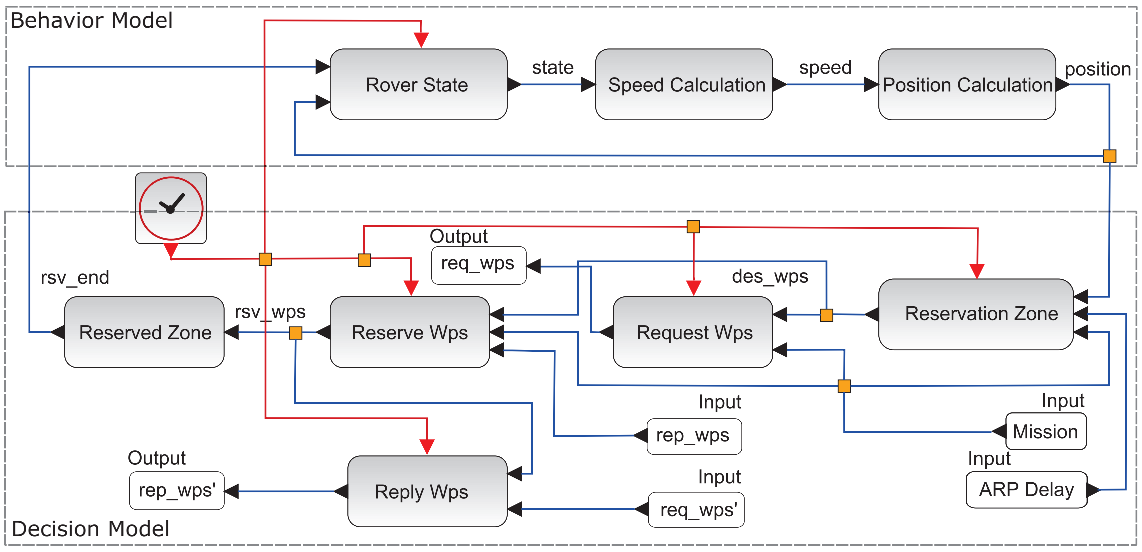

Fig. 7: Scicos Simulation Model of ARP

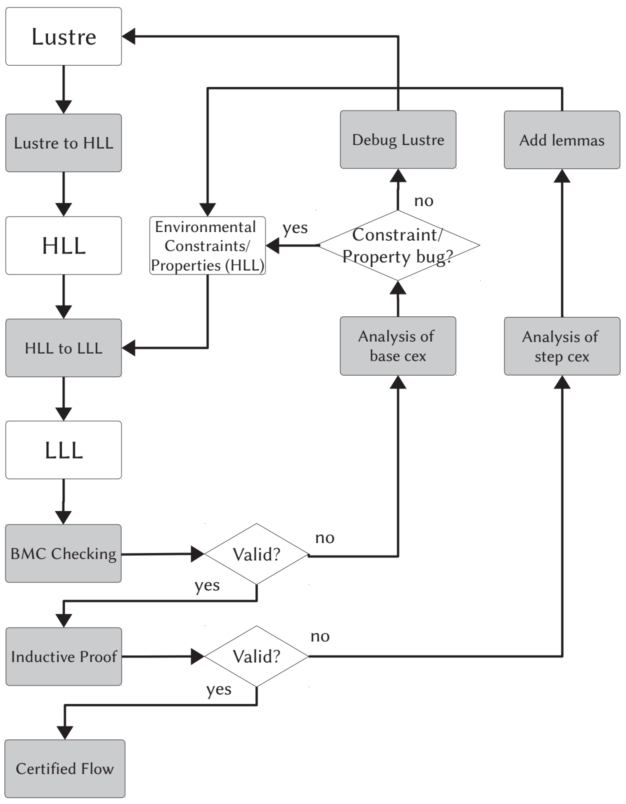

Fig. 8: The Workflow of Property Verification

plete. Examples of k-induction proofs and BMC verification are given in Sect. 5.2 and 5.3 respectively.

5.2 K-inductive Proof of Safety Property

Recent works have shown that k-induction often gives good results in practice when implemented by SAT or SMT based model checking [38,6]. Mathematical induction is the classical proof technique that consists of proving a base case (Eq. 1) and an inductive step case (Eq. 2). Let a transition system $\mathcal{S}$ be specified by an initial state condition $I(x)$ and a transition relation $T(x, x')$ where x, x' are vectors of state variables. A state property $P(x)$ is invariant for $\mathcal{S}$, i.e., satisfied by every reachable state of $\mathcal{S}$, if the entailments in Eq. 1 and Eq. 2 hold for some $k \geq 0$.

$$\begin{aligned} \mathtt{I}(\mathtt{x}_0) \ \wedge \ \mathtt{T}(\mathtt{x}_0, \mathtt{x}_1) \ \wedge \ \cdots \ \wedge \ \mathtt{T}(\mathtt{x}_{k-1}, \mathtt{x}_k) \\ \models \mathtt{P}(\mathtt{x}_0) \ \wedge \ \cdots \ \wedge \ \mathtt{P}(\mathtt{x}_k) \end{aligned} \tag{1}$$

$$\begin{aligned} \mathtt{T}(\mathtt{x}_0, \mathtt{x}_1) \ \wedge \ \cdots \ \wedge \ \mathtt{T}(\mathtt{x}_k, \mathtt{x}_{k+1}) \ \wedge \\ \mathtt{P}(\mathtt{x}_0) \ \wedge \ \cdots \ \wedge \ \mathtt{P}(\mathtt{x}_k) \models \mathtt{P}(\mathtt{x}_{k+1}) \end{aligned} \tag{2}$$

A counterexample trace for the base entailment indicates that the property P is falsifiable in a reachable state of $\mathcal{S}$. This is similar to the counterexamples produced by BMC, but a counterexample trace for the induction step entailment may start from an unreachable state or an over-approximated reachable state of $\mathcal{S}$. In Figure 9, we distinguish the reachable part of the state space and the over-approximated reachable state space. The transition $T(x_n, x_{n+1})$ starts from an over-approximated reachable state in step n, and ends in a unreachable state in step $n+1$. One way to rule out such step counterexamples is to increase the depth k of the induction. However, some invariant properties are not inductive for any k. So, instead of increasing k, the method to enhance k-induction of a property is to strengthen the induction hypothesis using new lemmas to reduce the over-approximation of the reachable state space.

Many recent efforts are dedicated to the automatic generation of invariants (used as lemmas in this work): automatic invariant checking based on BDDs [36]; unbounded model checking using interpolation [31]; property-directed reachability (PDR) [5]; quadratic invariant generation using templates based on abstract interpretation [34]. S3 provides a lemma generation tool based on a speculation strategy that searches for equivalent variables at bit-level. According to our experience, it is still very difficult for those tools to generate all necessary lemmas for an arbitrary system, and manual elaboration of lemmas to complete the proof remains important. So, to keep the approach as generic as possible, we do not apply invariant generation methods. Instead, we show how lemmas can be found "manually" on the basis of the step-counterexamples. We pick the property HLR-06-1 in Table 5 as an example to illustrate the process of inductive proof.

Example 1 HLR-06-1 states that *the rover position shall be in front of or at the initial position of the reserved area.*

It is formally expressed in Eq. 3, where `pos_r(t)` is the position of the rover `r` at time `t`, and $\mathtt{pos_r(init_{rsv})}$ is the initial position of the reserved area of rover `r` at time `t`. This requirement is expressed in HLL using the FPA operators, given by Eq. 4, where `i` is the id of rover `r`, and the `FLT_ge()` is the floating point greater-or-equal operator. The notion of time cycle does not appear in Eq. 4, because time is implicit in HLL. To simplify the explanation, we suppose that the mission of each rover contains at most 5 waypoints.

$$\forall \mathtt{r} \in \mathtt{Rovers}, \mathtt{t} \in \mathtt{Time}\big(\mathtt{pos_r(t)} \geq \mathtt{pos_r(init_{rsv})}\big) \qquad (3)$$

$$\mathtt{FLT\_ge(pos_i, init\_rsv_i)} == \mathtt{true}; \qquad (4)$$

Following the workflow defined in Figure 8, BMC is executed first, and no counterexample is found within a time length of 20 cycles. Then k-induction is executed. With k = 1, a step counterexample is found in the next inductive depth (depth = 2), shown in Figure 10. The FPA-lib of S3 follows the IEEE 754 FPA standard, thus a variable of float type (here variables `pos1` and `init_rsv1`) is composed of a sign, an exponent, and a mantissa. To facilitate the explanation, the converted decimal values of floating numbers are given in Figure 10. The Boolean variable `rsv1[i]` represents the reservation status (by the local rover) of waypoint i of a rover's mission. Values of variables `pos1`, `init_rsv1` and `rsv1` are given for steps 0 - 3, where a step-counterexample is produced in step 2.

```
[depth 2] > pos1
$1: pos1 is 0.45 0.55 [0.75] 1.05

[depth 2] > init_rsv1
$2: init_rsv1 is 0 0 [1.20] 0.607

[depth 2] > rsv1
rsv1 is a composite
$28*: rsv1[0] is t t [f] f
$29:  rsv1[1] is f f [f] t
$30:  rsv1[2] is t t [t] t
$31:  rsv1[3] is f f [f] f
$32:  rsv1[4] is f f [f] f
```

Fig. 10: Step-counterexample of Property HLR-06-1

This step-counterexample contradicts the property HLR-06-1 because of `pos1 < init_rsv1` in step k=2. This means that the rover locates outside the reserved area. The reserved area is in fact a set of continuous[12] reserved waypoints of rover's mission, therefore the calculation of `init_rsv1` depends on the reservation status of the waypoints (variable `rsv1`). We notice that in step k=1, the waypoints `P0` and `P2` in the mission are reserved (i.e., `rsv1[0]=t` and `rsv1[2]=t`), but the waypoint `P1` is not (i.e., `rsv1[1]=f`), which means that ***the reserved area is not continuous***. This step-counterexample does not indicate a design error. Indeed, HLR-09 in Table 5 requires that any positive reply to a reservation request shall contain a set of continuous waypoints. Unfortunately, we cannot use it as lemmas of this property because its inductive proof also produces step-counterexamples and needs to be analyzed. We thus have two solutions: (1) express and prove a property about the continuity of the reserved area, if valid, use it as a lemma to prove HLR-06-01; (2) investigate the step-counterexamples of HLR-09 to make it proved, and then use HLR-09 as a new lemma to prove HLR-06-1.

For the first solution, the added lemma is expressed in HLL as Eq. 5, where N is the number of waypoints in a mission. Using this additional lemma, HLR-06-1 as well as all other indeterminate[13] properties are proved

[12] As explained by the REQ-01-4 in Table 5, we use *continuous (continuity)* hereafter for the fact that each waypoint has a unique precedent waypoint in a mission or in a reserved area, except that it is the initial one.

[13] Indeterminate means neither valid nor violated, or unknown.

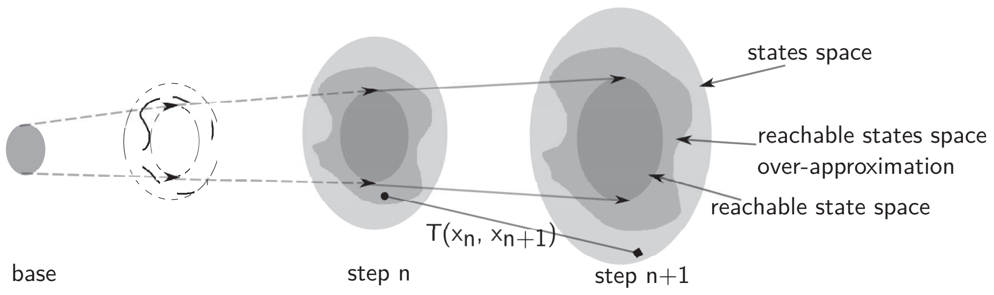

Fig. 9: Step Counterexample in Inductive Proof

by 1-induction. Although this step-counterexample is not due to any missing or wrong property in the specification, we still suggest to report it to the designer. Then s/he might decide to add the new lemma as a complementary requirement about the continuity of reserved areas in the specification. This may reduce the re-verification effort. In this case, as the designer thinks this implicit property is important, and he decides to add it in the specification as a derived requirement from the development process.

$$\texttt{ALL i} : [0, \texttt{N}-3],\ \texttt{j} : [2, \texttt{N}-1]\ \big(\texttt{rsv1[i]} \wedge \texttt{rsv1[i+1]} \wedge \texttt{i}+\texttt{j} \leq \texttt{N}-1 \rightarrow \texttt{rsv1[i+j]}\big); \tag{5}$$

For the second solution, we can first consider HLR-09 as an axiom. Inductive proof demonstrates that even if HLR-09 were proved, HLR-06-1 would remain indeterminate and a step-counterexample similar to the one in Figure 10 would be produced again. Following the same idea, we assume all indeterminate properties except HLR-06-1 are valid, all the step-counterexamples indicate that the step k+1 contains non-continuous reserved areas. This leads the verifier to add the same lemma as the one in the first solution.

5.3 BMC and Test Case Generation

The formal verification cannot completely replace the testing. Model-based test case generation has been developed as an important technique to perform software testing and system testing. Usually, we derive the abstract test suite from formal specification of requirements against the model of the system under test. Relying on the information for converting abstract test case to executable one, the executable test suite is derived.

In general, properties are classified as *safety* or *liveness* properties. The former declares what should not happen (or should always happen), while the latter declares what should eventually happen. The vast majority of properties in the ARP system are safety ones, except the system-level functional property REQ-F2 in Table 4 and the software-level functional property HLR-13 in Table 5.

Example 2 REQ-F2 states that *at any time, if the definition of schedulable missions are free of deadlock, a deadlock shall not occur due to the ARP.* HLR-13 states that *the ARP shall ensure that the schedulable mission is completed within worst case mission time (WCMT).*

HLR-13 is a bounded liveness property because an over-approximated WCMT can be used as the upper bound of checking. Hence it is a good candidate for BMC. If no counterexample[14] is found before the time bound, the property is valid. In the case of HLR-13, a counterexample is easily produced using BMC. A precondition of HLR-13 is REQ-F2, because a rover may not complete its mission when deadlocks occur. The validation of REQ-F2 requires that missions are schedulable, otherwise it is possible that deadlocks occur, and HLR-13 fails. As we cannot check these two properties considering the actual mission schedules, we use BMC to generate test case scenarios containing deadlocks due to unschedulable missions. These test cases can be used later to check the verification tool of mission schedules.

To explain the generation of deadlock scenarios, we consider a system with two rovers. REQ-F2 is satisfied if the property expressed in Eq.6 is false, where rovers r_i and r_j are stopped, r_i (r_j) requests waypoint p_j (p_i), but p_i (p_j) is reserved by r_i (r_j). Both rovers wait for a locked resource.

$$\forall \texttt{p}_\texttt{i} \in \texttt{Mis}_\texttt{i}, \texttt{p}_\texttt{j} \in \texttt{Mis}_\texttt{j}, \texttt{r}_\texttt{i}, \texttt{r}_\texttt{j} \in \texttt{Rovers}, \texttt{t} \in \texttt{Time}\ \big(\texttt{i} \neq \texttt{j} \wedge \texttt{state}(\texttt{r}_\texttt{i},\texttt{t}) = \texttt{STOP} \wedge \texttt{state}(\texttt{r}_\texttt{j},\texttt{t}) = \texttt{STOP} \\ \wedge \texttt{req}(\texttt{r}_\texttt{i},\texttt{p}_\texttt{j},\texttt{t}) \wedge \texttt{req}(\texttt{r}_\texttt{j},\texttt{p}_\texttt{i},\texttt{t}) \wedge \texttt{rsvd}(\texttt{r}_\texttt{i},\texttt{p}_\texttt{i},\texttt{t}) \wedge \texttt{rsvd}(\texttt{r}_\texttt{j},\texttt{p}_\texttt{j},\texttt{t})\big) \tag{6}$$

[14] The counterexample of liveness property is a path to a loop that does not contain the desired state. This implies that with an infinite loop path, the system never reaches the specified state.

We launch BMC for this property for some time length, and test case scenarios are extracted from the generated counterexamples.

5.4 Safety Property and Map Data Validation

Once the design is delivered to the verifier, it is the verifier's duty to express and verify the properties. S/He might have several ways to express one property. Some safety properties can hardly be verified by induction or BMC due to, for example, the complexity of calculation. In that case, we may take benefit of divide and conquer strategy by decomposing the property into a set of simpler ones, even static ones. Take the REQ-SAF-1 in Table 3 as an example.

Example 3 REQ-SAF-1 states that *at any time, the minimal distance between the centers of two rovers shall be greater or equal to 0.4m.*

This property can be verified by calculating the distance between two rovers at any time and then checking its value, unfortunately this solution is expensive due to the nonlinear floating-point arithmetic. To alleviate this problem, REQ-SAF-1 is split in another safety property about the reservation of waypoints (REQ-SAF-1-1) and a set of properties about the map data (IR-F1, IR-F2 and IR-F3) in Table 3. REQ-SAF-1-1 is proved by k-induction using similar process as described in the Sect. 5.2. IR-F1, IR-F2 and IR-F3 are requirements about the length of segment, the distance between a waypoint and a segment, and the absence of intersection between segments. In this work, the map data are modeled in Lustre, as same as the software. In fact, these static requirements on the map data could be easily checked using a dedicated verification program. However, when these map properties are used as sub-properties of the safety property REQ-SAF-1, they need in any case to be re-verified in the Lustre model. Our work uses a unique tool chain for the data validation. This approach allows the users to reuse the properties expressed on the map data for the verification of software.

5.5 Property Verification Results

The safety, functional and performance properties of ARP are formally expressed. As shown in Table 6, some safety properties can be directly proved by 0-induction or 1-induction, while some others need additional lemmas. REQ-QoS-1 is a system-level performance property. It is difficult to verify it at system-level without having software design, it is thus expressed as HLR-12 and verified at software-level by inductive proof.

6 Equivalence Proof between Design and Generated Code

The property verification activities depicted in Sect. 5 demonstrate that the design model complies with the set of requirements. However, there is still a gap between the design model and the code embedded in the system. The code can be either implemented by the developer or be generated automatically from the Lustre model. In our case, we use the lus2c translator[15] to generate the C code from the Lustre model. However, as this translator is not qualified[16], it is still unknown whether this C code satisfies the requirements.

To prove the correctness of the generated C code, two approaches are applicable. The first one follows the strategies presented in Sect. 5. We first translate the C code into the HLL model using the C-translator in S3, and verify that this HLL model satisfies all requirements defined in Sect. 4. The second approach demonstrates that the code is equivalent to the design model, i.e., the same inputs generate the same outputs. This guarantees that the properties (related to inputs and outputs) satisfied by the design model will be satisfied by the code.

Figure 11 presents several verification activities (A) in the process of equivalence proof: A1 generates C code from Lustre model with a qualified translator; A2 translates Lustre models into HLL models, where properties are combined and verified; A3 translates C code into HLL models, where properties are combined and verified; A4 proves that the HLL models generated from the Lustre model and the C code are equivalent; A5 proves that the LLL models generated from the Lustre model and the C code (through the HLL model) are equivalent.

Based on different development contexts and the activities in Figure 11, we summarize a set of strategies (S) for the verification of the C code, as follows:

- S1: The code generator is qualified as a development tool at the same level as the application. Properties are verified on the Lustre model (A2). Thanks to the qualified translation (A1), properties are preserved in the generated C code.
- S2: The code generator is not qualified at the same level as the application.
  - S2a: Properties are directly verified on the C code (A3).
  - S2b: Properties are verified on the Lustre model (A2). The C code is proved to be equivalent to the Lustre model (A4 or A5). Thanks to the

[15] The translator lus2c is provided by Lustre v4 toolset.
[16] Qualification is a requirement in getting a system certified.

Table 6: Property Verification Results of the ARP Design

Verification Techniques	**REQ-ID**	**Verification Results**
Inductive Proof	IR-F1, IR-F2, IR-F3, REQ-F1, HLR-01, HLR-03, HLR-04, HLR-05, HLR-07, HLR-08, HLR-11	Valid by 0-induction
	REQ-SAF-1-1, HLR-10	Valid by 1-induction
	HLR-02, HLR-06-1, HLR-06-2, HLR-06-3, HLR-06-4, HLR-09, HLR-12 (REQ-QoS-1)	Valid by 1-induction using additional lemmas
Data Validation	REQ-SAF-1 (IR-F1, IR-F2, IR-F3)	Valid
BMC and Test Case Generation	HLR-13, REQ-F2	Test cases generated

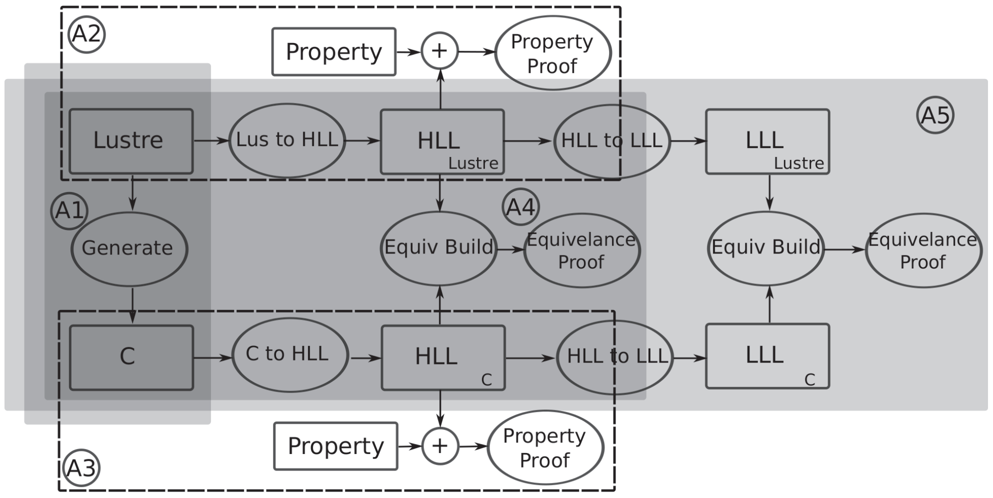

Fig. 11: Activities in the Process of Equivalence Proof

equivalence proof, properties are preserved by the C code.
  - S2b1: The equivalence is proved at HLL level (A4).
  - S2b2: The equivalence is proved at LLL level (A5).
- S2c: Properties are verified at both Lustre and C code level (A2 and A3).

In our case study, we have proved the equivalence between the Lustre model (including the map data) and the generated C code using the strategy S2b. The reasons for choosing this strategy are explained as follows:

1. The C code generator `lus2c` is non-qualified. (rule out S1)
2. It is reasonable to assume that only a subset of the requirements will be formally expressed and verified. One will probably use other more classical approaches, such as testing. The cost of test increases as the abstraction level decreases, thus test is less expensive at Lustre level than at C level. (rule out S2a and S2c)
3. Specific formal verification techniques can be applied on Lustre thanks to its abstract semantics, which is lost once the C code is generated. This implies that proving properties at Lustre level is simpler than at C level. (rule out S2a and S2c)
4. S2b supports two complementary approaches of equivalence proof S2b1 and S2b2. S2b1 allows debugging counterexamples at the HLL level, but might need additional lemmas for some cases. S2b2 automatically searches and adds necessary lemmas using speculation techniques, but counterexamples are still difficult to exploit. Usually S2b2 is performed first; if a property is falsifiable or indeterminate, the S2b1 is used to analyze the (step-)counterexample. (keep S2b)

7 Discussion

In this part, we provide more details and discuss some lessons learnt from the experiments.

7.1 Proof of Generated Code

The strategies of equivalence proof discussed in Sect. 6 have pros and cons. One can select appropriate strategies under the development contexts and the available resources.

- S1 requires a qualified code generator. This was not an option in our case, but this is the usual strategy in the domain of safety critical applications where the cost of a failure largely exceeds the cost of qualification. A qualified code generator saves a lot of effort, but is very expensive.
- S2a and S2c require to express and verify properties and lemmas at code level. As the code is less abstract and more complex than the Lustre model, property verification requires more effort.
- S2c seems redundant as property proofs are performed at both Lustre and C level. However, it might be useful to determine the origin of an error: a property satisfied in Lustre but falsifiable in C reveals probably an error during translation.
- S2b is "S1 without qualified generator". The equivalence proof between Lustre model and C code ensures that the generated C code implements exactly the properties expressed in Lustre. S2b does not need expensive qualified generator, but needs more effort to carry out equivalence proof. Each time the Lustre model is modified and the new code is generated, the equivalence needs to be re-proved.

7.2 Proof-Driven Design Guidance

The formal verification of a system could fail because of the complexity of the system, the lacking of complete requirements to support formal verification, etc. For instance, in Sect. 5.2, the HLR-06-1 is proved by k-induction after searching and adding a lemma. If we consider that the verifier has not a complete or deep knowledge about the design, s/he reports a scenario that contains the step-counterexample to the designer. If necessary, the designer may then decide to add a complementary requirement derived from this lemma in the initial specification, in order to reduce the cost of subsequent verification. The other way round, the verifier may ask the designer to state as many detailed requirements as possible about the system. These properties may be written as comment and/or assertions to be checked at runtime.

Sometimes, a lemma may not be provable from the initial hypotheses. This might be the case that some environment hypotheses have been considered as granted by the designer without ever being explicitly expressed. This case could be handled either by a modification of the requirements to make the hypothesis explicit or by a modification of the design to make it independent from these hypotheses.

8 Conclusion and Perspectives

This article shows how multiple formal verification techniques (inductive proof, BMC, test case generation, and equivalence proof) can be integrated to verify an actual system with an industrial grade toolset. Some significant activities of a typical verification process have been addressed, from the specification and design to the formal verification. Focus has been placed on the verification of safety and functional properties and on the equivalence proof between the design model and the generated code. We have drawn some lessons about the equivalence proof and the proof-driven design guidance from this experiment. This verification process is classic when the proof of property is based on SAT/SMT solvers. The main effort lies in searching for lemmas for the property proof using k-induction. This needs good understanding of the proof techniques. As our verification tool provides step-counterexamples feedback, the debug process can be seen as an engineer work. This case study is built on the Lustre modelling language and S3 toolset. Similar property proof process can be applied to other modelling languages and SAT/SMT tools.

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, etc, and to establish benchmark (or reuse existing SMT benchmarks) to evaluate the performance of floating-point verification by S3.

Acknowledgement

This work has been funded by the INGEQUIP project. The authors would like to thank the members in the project for their good cooperation. We are thankful to Rémi Delmas and Micheal Dierkes for sharing the experimental results on the triplex sensor voter use case.

References

1. IEEE Standards Association. Ieee standard for floating-point arithmetic. 2008.
2. Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

3. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.
4. Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.
5. Johannes Birgmeier, Aaron R Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *Computer Aided Verification*, pages 831–848. Springer, 2014.
6. Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer-Aided Design*, pages 409–426. Springer, 2000.
7. 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.
8. 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.
9. 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.
10. 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.
11. Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From simulink to scade/lustre to tta: a layered approach for distributed embedded applications. In *ACM Sigplan Notices*, volume 38, pages 153–162. ACM, 2003.
12. 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.
13. 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.
14. Mathieu Clabaut, Ning Ge, Nicolas Breton, Eric Jenn, Rémi Delmas, and Yoann Fonteneau. Industrial grade model checking - use cases, constraints, tools and applications. In *International Conference on Embedded Real Time Software and Systems*, 2016.
15. W.J. Cody and W.M.C. Waite. *Software manual for the elementary functions*. Prentice-Hall series in computational mathematics. Prentice-Hall, 1980.
16. Philippe Cuenot, Eric Jenn, Eric Faure, Nicolas Broueilh, and Emilie Rouland. An experiment on exploiting virtual platforms for the development of embedded equipments. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.
17. Philippe Cuenot, Eric Jenn, Eric Faure, Nicolas Broueilh, and Emilie Rouland. An experiment on exploiting virtual platforms for the development of embedded equipments. In *International Conference on Embedded Real Time Software and Systems*, 2016.
18. 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.
19. 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.
20. 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.
21. Arnaud Dieumegard, Ning Ge, and Eric Jenn. Event-b at work: some lessons learnt from an application to a robot anti-collision function. In *9th NASA Formal Methods Symposium (NFM 2017)*, 2017.
22. RTCA DO. 178c. *Software considerations in airborne systems and equipment certification*, 2011.
23. 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.
24. Ning Ge, Arnaud Dieumegard, Eric Jenn, Bruno d'Ausbourg, and Yamine Aït-Ameur. Formal development process of safety-critical embedded human machine interface systems. Technical report, IRT Saint-Exupéry, 2017.
25. Ning Ge, Arnaud Dieumegard, Eric Jenn, and Laurent Voisin. From event-b to verified c via hll. *arXiv preprint arXiv:1610.07410*, 2016.
26. Ning Ge, Eric Jenn, Nicolas Breton, and Yoann Fonteneau. Formal verification of a rover anti-collision system. In *International Workshop on Formal Methods for Industrial Critical Systems and Automated Verification of Critical Systems (FMICS-AVoCS 2016)*, volume 9933, pages 171–188, 2016.
27. Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
28. John Harrison. Floating-point verification using theorem proving. In *Formal Methods for Hardware Verification*, pages 211–242. Springer, 2006.
29. F Lapschies, J Peleska, E Gorbachuk, and T Mangels. sonolar smt-solver. *Satisfiability modulo theories competition; system description*, 2012.
30. 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.
31. Kenneth L McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification*, pages 1–13. Springer, 2003.
32. José Meseguer and Peter Csaba Ölveczky. Formalization and correctness of the pals architectural pattern for distributed real-time systems. In *International Conference on Formal Engineering Methods*, pages 303–320. Springer, 2010.
33. David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12, 2008.
34. Pierre Roux, Romain Jobredeaux, and Pierre-Loïc Garoche. Closed loop analysis of control command software. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 108–117. ACM, 2015.
35. Philipp Rümmer and Thomas Wahl. An smt-lib theory of binary floating-point arithmetic. In *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.
36. John Rushby. Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. In *Theoretical and Practical Aspects of SPIN Model Checking*, pages 1–11. Springer, 1999.
37. Lui Sha, Abdullah Al-Nayeem, Mu Sun, Jose Meseguer, and Peter C Olveczky. Pals: Physically asynchronous logically synchronous systems. 2009.
38. Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal methods in computer-aided design*, pages 127–144. Springer, 2000.