Integrating Model Checking in an Industrial Verification Process: a Structuring Approach

Pierre-Alain Bourdil, Silvano Dal Zilio, Eric Jenn

Integrating Model Checking in an Industrial Verification Process: a Structuring Approach

Pierre-Alain Bourdil[1], Silvano Dal Zilio[2], and Eric Jenn*[1]

[1]IRT Saint-Exupéry, Toulouse, France
[2]LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France

An obstacle to the adoption of model-checking in large projects is a lack of guidelines on how to integrate formal methods with existing system engineering practices. In this context, a methodology should give answers to several questions: How to manage the models and abstractions used to verify a claim? How do we gain confidence on the soundness of these models? How can we build a structured argument from the verification results? In this paper, we describe a structured approach for managing verification arguments an apply it to check a critical function of an autonomous rover.

1 Introduction

Formal methods, such as model checking, are inexorably percolating in industry, especially in domains strongly constrained by safety and regulatory constraints. However, an obstacle to the adoption of model-checking is a lack of guidelines on how to integrate it with existing system engineering practices. In a "traditional" development process, checking that a system meets a given requirement is achieved using a combination of inspection, analysis and tests at each phase of the design process. The confidence in the verification process is basically expressed as a question of coverage: the process is deemed sufficient if coverage is sufficient. There are many reasons why this problem is more complex when dealing with formal verification techniques.

First, formal methods rely on models that have to preserve the properties to be verified. So we need to ensure that these models are faithful to the intent of the system designers. Next, no unique technique or model cover all aspects and objectives of the verification

*Seconded from Thales Avionics, Toulouse, France process. Verification can only be achieved by an appropriate combination of models (using different viewpoints, abstraction levels, etc.) and it is sometimes necessary to apply several transformation steps in order to obtain tractable verification problems. This means that we need to work with a collection of models that are logically connected to each other. It also means that it is necessary to justify that each abstraction is sound with respect to the current verification objective. Finally, formal techniques and tools are only applicable with specific restrictions: on the set of properties that can be checked; on the computational and communication models that can be used; ... It is therefore necessary to track whether these constraints are met and whether they are consistent with the hypotheses made about the system, its application and its environment.

In this paper, we describe a methodology for providing a convincing argument that a system design actually complies with a set of expected properties; what we call a *verification argument*. In practice, our methodology provides answers to several questions: How to manage the models and abstractions used to verify a claim? How to gain confidence on the soundness of these models? How can we build a structured argument from the verification results?

The question of providing valid verification arguments is not new. This is, for instance, the objective of *assurance cases* [10, 16]. Nonetheless, we focus here on a narrower problem and target only the proof of "low level" claims (technical properties) that rely on model-checking. In some sense, our approach can be seen as a pattern for integrating model-checking in an assurance case and is therefore complementary to it. Our methodology is based on the definition of formal claims and relies on the use of inference rules to combine them in a disciplined way. We also place a particularly strong emphasis on the transition from the informal to the formal world. This allows us to apply a small set of well-defined strategies (compositional reasoning, abstraction, etc.)

Contributions. We describe our methodology in Sect. 3 and provide several examples of "rule schemas" that can be applied to build a valid verification argument. We apply our approach on a non-trivial use case: the design and implementation of an autonomous rover developed at the IRT Saint-Exupéry. Actual documents related to the verification argument, as well as the source for all the formal models, have been made freely available online at `http://www.laas.fr/fiacre/examples/twirtee.html`.

Our test-bench, the *Three-Wheeled Integrated Rover Testbench for Equipment Engineering* (TwIRTee), is described in Sect. 2. It is used as an integrated demonstration platform for the various engineering activities carried out in the Ingequip project [4]: hardware/software co-design; design space exploration; modeling and formal verification; ... It is aimed at being representative of the architecture of actual space, aircraft or automotive systems designed by the project's industrial partner and to experiment with new design choices. In Sect. 4 we illustrate our methodology on the verification of a critical function of the rover, namely a distributed, fault-tolerant clock synchronization protocol over CAN [15].

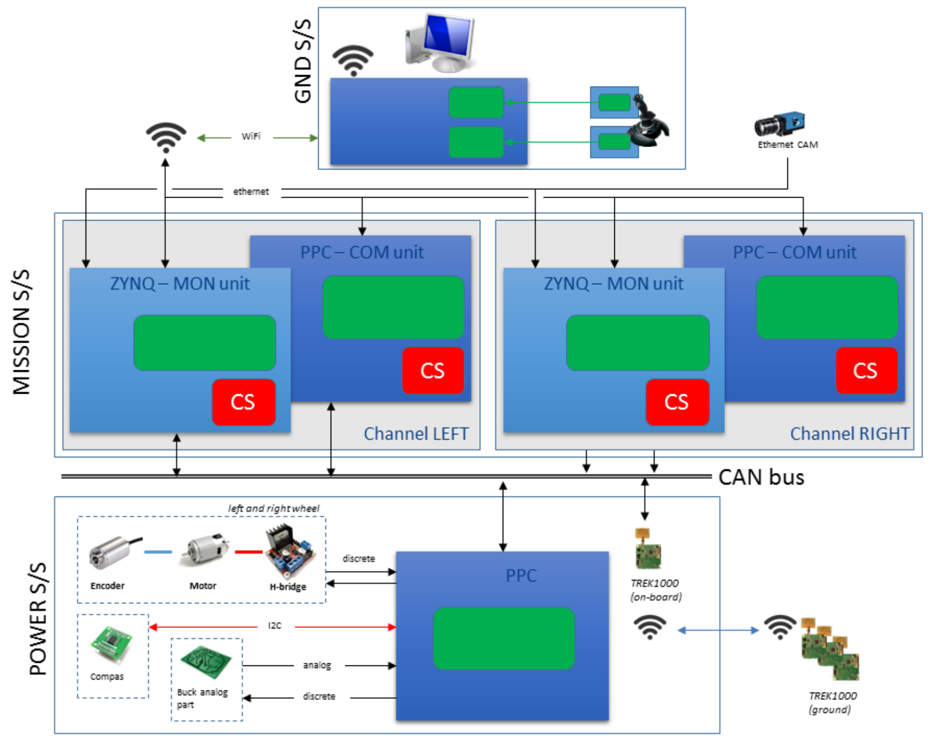

Figure 1: Architecture of the TwIRTee

2 Use Case: Synchronization Function for the TwIRTee

The TwIRTee rover is made of two subsystems, see Fig. 1; the Mission Subsystem (MSS) and the Power Subsystem (PSS). The complete functional chain involves a Localization Function (LF), a Mission Planning Function (MPF), a Trajectory Tracking function (TTF), and a low level Motor Control Function (MCF).

Functions LF, MPF and TTF are located in the MSS while function MCF is in the PSS. All computing systems are completely asynchronous; the different functions are allocated on multiple FPGA and connected through a CAN bus.

To protect the equipments, and the people working around TwIRTee, we define a “safety zone” around the rover large enough to allow emergency stop. The radius of the safety zone is computed from the maximal mission speed. An erroneous speed computation may lead the rover to overpass its authorized speed. In this case, a collision is possible; an event deemed as catastrophic. To mitigate this risk, we introduce a fail-safe Command/Monitoring pattern (COM/MON) to enforce safety [19]. In practice, all functions in the MSS are duplicated. For instance, both MSS/COM and MSS/MON send the speed to the PSS. If the difference of outputs received from COM and MON is above a given threshold, the COM is considered faulty. (Redundancy and fault recovery mechanisms are outside the scope of this paper.)

This design choice may have an adverse impact on the availability of the system. Indeed, due to different clocks rates, the clocks at various sites may diverge. Hence COM and MON may signal reaching a given speed setpoint at very different (local) time. In this case, PSS will almost systematically reject COM speed, causing inadvertent emergency stop of the rover.

A solution will be to have a synchronous architecture, based on the use of a time-triggered protocol and a system wide global network time. Nonetheless, this may go against some non-technical constraints imposed on the system. Another solution, that we want to experiment in TwIRTee, is to synchronize the COM and MON by adding a distributed, fault-tolerant clock synchronization function to the MSS. Based on this design choice, an analysis of the possible failure modes of the system functions and their criticality will show that Clock Synchronization (CS) is critical. Moreover, because the CS function is distributed and asynchronous, the same analysis may quantify the likelihood to detect a design error by testing as medium. This warrant the use of more exhaustive verification methods, such as formal verification.

The Clock Synchronization Protocol. A clock synchronization algorithm provides a global time base in a distributed architecture where each process has its own local clock. Its purpose is to ensure two main properties [18]: *precision*, meaning that the difference between the clock values of any two process is bounded; and *accuracy*, meaning that the value of the clock is "not too far" from the actual (real) time. In this section we briefly present the clock synchronization algorithm we chose for TwIRTee, called *a-posteriori agreement* [14].

The algorithm is tailored for CAN and includes fault management mechanisms. We assume a fault model with omission faults for processes (a process does not send a message it should), transmission faults of the CAN (some messages needs to be retransmitted), arbitrary clock faults (e.g. drift too fast) and fail-silent processes. Fault occurrences are bounded by a constant f. The a-posteriori agreement algorithm is based on periodic resynchronization [18]. Processes initiate a new round of the algorithm at each period by broadcasting a *start* message over the CAN. Non-faulty processes receive the message. After $f + 1$ start messages, we know for certain that the round is started. This initiates a *synchronization round*. The remaining steps, which are also based on the exchange of typed messages, are used to solve two successive consensus problems. First a *vote phase*, to agree on a leader process whose clock is used as the reference. Next an *adjust phase*, to agree on an adjustment (a time delay) that should be applied on the local clock of every process to improve accuracy. The round terminates at the end of the adjust phase.

The algorithm is fully distributed, which means that every process has only a local view of the round. Since local clocks can be slightly out of sync, processes do not start their round at the same time; at the beginning of a round $n+1$, some processes still have their local round at n. Moreover, new processes may join the synchronization algorithm at any moment.

The a-posteriori agreement algorithm is an example of a complex, realtime, distributed protocol. The complexity arises from the large number of possible transitions; the presence of faults; the use of multiple timing constraints at different scale. For instance, the algorithm uses timeouts to detect faulty processes but also to minimize the number of exchanged messages. This is the reason why we chose to verify this algorithm formally using a realtime model-checker. Timing constraints are also introduced by the CAN. In Sect. 4, we sketch the argument used to assert the precision property on the algorithm. This property relies on some hypotheses of the CAN, like time bounds on message propagation and delivery. During a broadcast, each non-faulty process receive the same message with a time delay bounded by Γ_{tight}. This delay is due to physical time propagation and the processing time of local controllers. The maximal delay between the emission of a message at a correct process and its reception at any non-faulty process is bounded by Γ_{max}. The value of Γ_{max} take into account the worst case network load, maximum numbers of message retransmissions, … Precision of the clocks depends on Γ_{tight}, while timeouts are based on Γ_{max}.

3 A Methodology for Building Verification Arguments

We consider a top-down system development process, starting from a set of high level functional requirements and ending with a product (that is both the software and hardware parts actually implementing the system). In practice, a system design generally reuse existing components. When we decide to apply formal verification, the inputs are obtained from the system design (including reused components) which we assume described in a set of informal documents.

It is generally not possible to formally verify a design completely. There can be problems with cost, with scalability of the methods, but also with some theoretical limitations of the tools and models that we wish to use (for instance a lack of expressivity of the models or some undecidability results). Therefore, it is important to specify the *scope* of a model, that is which elements of the design are relevant to the given verification objective. The scope also includes constraints on the environment of the system. Our methodology associates explicitly with each model: (1) a set of constraints defining its scope; and (2) the properties that we want to verify.

Our approach starts with a very detailed initial formal model; a model with as few abstractions and hypotheses as possible with respect to the design. The goal is to limit, as much as possible, the work needed to justify the trustworthiness of the initial formal model with respect to the informal design model. Indeed, this activity is usually obtained by a review from domain experts and by simulation and is inherently error-prone. Except for the simplest cases, the detailed formal model is not suitable for (automatic) formal verification. For instance, with model-checking, it will generally gives an infinite-state system or be subject to the state space explosion problem. Therefore abstractions are mandatories. Each abstraction should be captured in a new formal model. This means that we need to build a (structured) collection of formal models in order to prove the main verification objectives. We also need to structure these models in order to build a *verification argument*. We propose a rule-based framework to: manage the resulting collection of models; to justify the soundness of each abstraction step; and to collect the verification artifacts: source code, verification results, counter-examples, etc.

Our methodology favors having multiple models with incremental abstractions rather than fewer models encompassing lots of abstractions. Indeed it is often easier to define (and justify) an abstraction once the system is well understood [5]. In the rest of this section, we give a high-level description of the methodology that we put in place. We try to give some examples based on the use case described in Sect. 2.

3.1 Models, Properties and Claims

Models. A model is a simplification of reality that gives a complete description a system from a particular perspective. It is the description of a mental representation of a system, existing or not, using a syntax. This syntax can be textual or graphical, formal or not. This very general definition allows us to define a homogeneous framework to deal with heterogeneous semantics. A model corresponds to a certain level of details (or abstractions). We recognize the design as the most detailed model of the system, in the sense that it describes some aspects of the system that are not relevant for verification. In our approach, we raise the level of abstraction from the design model until model-checking can be applied. In our use case, for instance, the system design is given as an algorithm in pseudo-code and a set of informal requirements; the simulation code (kindly provided by the authors of [15]) is a set of C-language source files (the simulation code is semi-formal and its semantics depends on the simulation framework used); and the formal models are based on the formal specification language Fiacre. Regarding the abstractions used, we abstract the behavior of the CAN network using timed FIFO; we over-approximate the effect of clock skews; we made some data abstractions [20]; etc.

Property. A property is a statement on a design intent [9]. A property can be used as a constraint on the environment, to ensure that we cover some specific parts of the system, or as a design specification. For the same reasons as previously, we choose a very general definition in order to cope with different types of syntax and semantics for properties. Properties can be expressed formally, using a logic, or informally, using natural language (in which case we should make provisions for possible imprecisions).

We take inspiration from the Property Specification Language (PSL) [1] and associate a *verification directive* to every property. These directives attach a role to the properties which is useful for integrating the verification process in the system design process. For the sake of simplicity, we consider only a subset of PSL verification directive, namely: assert, assume, restrict, cover. An *assert* directive is always associated with a design specification property and defines the verification objective. *Assume* and *restrict* properties constrain the design. The difference is their role in the verification process. An assumption must be validated by subject matter expert while a restriction can be introduced to simplify the verification problem. A restriction should be temporary, like a scaffold, otherwise it becomes an assumption. For the rest of this paper we will use the directives names as properties types.

Claims. Models and properties are two sides of the same coin: a property is always expressed with respect to a model and a model cannot exists without some assumptions. This strong relationship is captured as a triplet $\langle H; M; P\rangle$ which we call a *claim*, inspired by [16]. The claim $\langle H; M; P\rangle$ must be read as: "the model M satisfies P under hypothesis H".

Definition 1 (Claim)**.** *A* claim *is a triplet $\langle H; M; P\rangle$, where: H is a set of* assume *or* restrict *properties; M is a model; and P is a set of* assert *or* cover *properties.*

A claim makes explicit the relation between a model, its assumption and the verification goal. In the simplest cases, we can check the validity of a claim $\langle H; M; P\rangle$ by using a verification tool, such as a model-checker. In this case, the property P is expressed as a temporal formula that needs to be checked on the composition of the assumption with the model; we say that $\langle H; M; P\rangle$ is valid if and only if[1] $H \otimes M \vdash P$. For instance, if we can give a semantics for properties and models using "sets of execution traces"—that is associate a set of traces $[\![M]\!]$ to any model M and similarly for H and P—then validity is set inclusion and composition is set intersection. In this case we say that the claim $\langle H; M; P\rangle$ is valid if and only if $[\![H]\!] \cap [\![M]\!] \subseteq [\![P]\!]$.

With sufficient justification from a system expert, it is also possible to admit a claim as a *fact*. In all the other cases, when it is not possible to prove a "top-level claim", we propose to apply deduction rules in order to decompose the problem into sub-claims. This approach to structure an argument using an (inference) tree or a set of deduction rules is quite common, see e.g. [3, 6, 16].

3.2 Rules

We define a *rule* as a schema linking a claim (the conclusion), with a set of premises and a condition of application. Premises and conclusion are claims. The condition of application is a list of conditions required to apply the rule. In our methodology, we also attach a *justification* (also called an evidence is other settings, like [16]) to each application of a rule to show that the use of the rule is sound. This justification acts as a documentation when we generate the final verification argument.

Definition 2 (Rule schema)**.** *A* rule *is a set of premises claims, $C_1, \ldots, C_n$, together with a conclusion, C, and (optionally) a boolean condition E.*

$$\textit{(rule.id)}\;\frac{E \quad C_1 \quad \ldots \quad C_n}{C}$$

The abstract definition of a rule does not assume any semantics on the claims: the definition respects *semantic neutrality* [3]. Note that, even though our notation is inspired by logical inference rules, the soundness of a rule is not necessarily backed by any notion of a "proof calculus", like for instance natural deduction. This is the reason for adding an extra justification to each application of a rule. We say that the application of a rule, a *rule instance*, is sound when its conclusion is valid whenever all the claims in its premises are.

[1]The precise definition of the composition operator, $\otimes$, and of validity, $\vdash$, depends on the choice of a semantics for the models.

$$\text{(split)}\frac{(H \subseteq H_1 \cup H_2) \quad \langle H_1; M; P\rangle \quad \langle H_2; M; P\rangle}{\langle H; M; P\rangle} \qquad \text{(comp)}\frac{\langle H \wedge P_1; M; P_2\rangle \quad \langle H; M; P_1\rangle}{\langle H; M; P_1 \wedge P_2\rangle}$$

$$\text{(trace)}\frac{(\llbracket M_c\rrbracket \subseteq \llbracket M_a\rrbracket) \quad \langle H; M_a; P\rangle}{\langle H; M_c; P\rangle} \quad \text{(scope)}\frac{\langle H_s; M_s; P\rangle}{\langle H; M; P\rangle} \quad \text{(formalize)}\frac{\langle H; M; P\rangle}{\langle \mathit{Hyp}; \mathit{Sys}; \mathit{Prop}\rangle}$$

Figure 2: Example of rules used to build verification arguments

We propose some predefined rules to support this process, some of which are given in Fig. 2. Our first example, rule (split), can be used to decompose a proof goal, $\langle H; M; P\rangle$, into two sub-claims with stronger hypotheses, H_1 and H_2. This rule has a condition of application, which states that the constraints in H are covered by either H_1 or H_2 (the exact meaning of the condition $H \subseteq H_1 \cup H_2$ depends on the choice of semantics for the hypothesis). This rule is useful when H_1 and H_2 partition H. For example to decompose a set of initial conditions (defined in H) in two smaller subsets. Typically, a justification for rule (split) should explain how to interpret the assume and restrict properties in H as sets.

Our second example, rule (comp), is a simple example of applying compositional reasoning. We can interpret this "decomposition rule" as follows. If we know that M satisfies P_1 (with assumption H) then, to prove that M satisfies $P_1 \wedge P_2$, it is enough to prove that M satisfies P_2 with the stronger hypothesis $H \wedge P_1$. Therefore rule (comp) is useful to decompose a verification goal into simpler sub-goals. We could define more complex rules, where different components of M depends on each other, in the style of assume-guarantee reasoning [11,12]. In this case, a justification should be added to show how to break any possible circular reasoning and therefore ensure the soundness of the rule.

Our last example, rule (trace), illustrates the use of abstractions during the verification process. In this case, we assume that we have two different models of the system—an abstract (M_a) and a concrete (M_c) one—both equipped with a trace semantics. Rule (trace) simply states that we can check properties on the abstract model if any execution trace of M_c (the set $\llbracket M_a\rrbracket$) is also a trace of M_a. For instance, in our use case, we use rule (trace) to check a specific property on a timed system (considering the set of traces with all timing information erased) by proving it on an equivalent system with more relaxed timing constraints. In this particular case, we need to add a justification that properties in P depends only on the order of the events and not their date, so that timing information can be safely omitted.

We can add rules for more abstractions. Some abstractions are purely automatic, like: predicate abstraction; Counter-Exemple Guided Abstraction Refinement; symmetry; . . . while other are hand-crafted, such as cut-point, counter-abstraction, or data independence.

3.3 Verification Argument and Evidence-Based Rules

Rules can be applied successively until we reach an *axiom*, that is a rule with no premises. In our context, an axiom is either a claim proved using a model-checker or it is a fact. This gives a derivation tree where the root is the main verification objective. We call this tree a *verification argument.* Hence we build the argument bottom-up, starting from the verification objective and ending with axioms.

The main goal of our work is to prove the verification objective. By construction, the verification objective (the root claim) of an argument is valid when all the rules instances in the argument are sound. We can back the soundness of the three rules defined in the previous section by reasoning on the semantics of the models. This is not always the case and this is one of the motivation for adding a justification to every rule instance. Next, we give two examples of "evidence-based rules", (scope) and (formalize), which are rules whose soundness is supported only by subject matter expertise. Both rules are necessary to build an argument in all but the simplest of cases. Also, while these rules share a common, very basic schema, they are quite different in nature.

Scoping rule: in general, design documents describe more elements than necessary for the verification objectives. Therefore we often need to simplify the models and the set of requirements in order to focus on some elements in the design. This is achieved by rule (scope), see Fig. reffig:infer, where M_s, H_s are obtained by simplifying M, H with respect to the property P. The soundness of this rule relies only on its justification. For instance to explain why some parts of the design documents have no influence on the proof of the verification objective P. Means for the justification are those accepted in the context of aeronautical certification [8]: analysis (including proof), tests and peer-review. For instance, in our use case, we use scoping to select the assumption made on: the clock characteristics; the CAN; the processes implementing the clock synchronization algorithm (scheduling, H/W initialization, . . .). Other elements in the design are abstracted by the designer as irrelevant for our verification objective, for instance some hypothesis on the environment (maximal velocity of the robot) or the functional specification of the LF, MPF, TTF and MCF functions.

Formalization rule: another instance where formal reasoning is not enough to simplify a claim is related to the steps where we move from an informal to a formal model. Rule (formalize) states that H, M, P are formal interpretation of the corresponding models; $\mathit{Hyp}, \mathit{Sys}, \mathit{Prop}$. The construction of $\langle H; M; P\rangle$ starts with the definition of *modeling requirements.* Concerning the assumptions made on the environment, *Hyp*, the modeling requirements should set the (range of) parameters pertaining to: the model of computation (message passing, shared variables, etc.); the temporal constraints (network delay, processing time, etc.); the bounds on the interactions (size of buffers, FIFO, etc.); etc. This activity is valuable for the designer as it explicits important characteristic of the environment and helps detect imprecision on the design, such as undefined FIFO size. Requirements for properties in *Prop* states how assert properties have to be checked. For instance, a formal property can be expressed as a temporal logic formula or checked using an observer. Finally, modeling requirement for the design model *Sys* should be as close as possible to those expressed on *Hyp*, so that we start with a formal model that relies

$$
\dfrac{\cdots \qquad \dfrac{\dfrac{\dfrac{}{\langle H; M; G_1^0\rangle}\,(mc.6) \qquad \dfrac{([\![M_c]\!] \subseteq [\![M_a]\!]) \qquad \dfrac{(G_1^i \subseteq G^i_{1,1} \cup \cdots \cup G^i_{1,k}) \qquad \bigwedge_{j\in 1..k} \overline{\langle H_{abs} \wedge G_{1,j}; M_{abs}; G_1^{i+1}\rangle}}{\langle H_{abs} \wedge G_1^i; M_{abs}; G_1^{i+1}\rangle}\,(.8)}{\langle H \wedge G_1^i; M; G_1^{i+1}\rangle}\,(trace.7)}{\langle H; M; G_1^n\rangle}\,(induction.5)}{\langle Hyp; Sys; P_1\rangle}\,(formalize.4)}{\cdots}\,(comp.2)
$$

Figure 3: Verification argument for the clock precision property

on as few abstractions as possible. Once again, this activity may eliminate imprecision in the design model.

Once the model requirements are defined, we need to define one or more *model element* for every model requirement. Model elements are relative to the formal language used to build the claim. They give a detailed description of the formal model and capture modeling choices that "implement" the modeling requirements. For each model element a rationale is given that shows how the modeling choice meets its requirement. Model requirements can be seen as High Level Requirements (HLR) while model elements describes Low Level Requirements (LLR). Rationale helps to show that LLR meet their HLR. This decomposition supports traceability from requirements to formal model code. This process ease peer-review and raise the overall confidence on the formal modeling process. This is, for instance, analogous to the way assurance process reduce the risk a test oracle does not detect an incorrect system [16].

4 Illustration of our Methodology on the Rover Use Case

Figure 3 gives an overview of the structure of a *verification argument* for the clock synchronization algorithm. We focus on one initial claim related to the precision of the clocks. The derivation tree does not display the justification for each rule application. In practice, each rule application is associated to a report. As an example, the report for rule (formalize.4), as well as all the source for the formal models, are available online at `http://www.laas.fr/fiacre/examples/twirtee.html`.

We give some insights on the soundness of each rule application below. As a first step, rule (comp.2), we decompose the precision property as the conjunction of two simpler properties: (1) that each round of the algorithm terminates and reaches a consensus on an adjustment (property P_1); and (2) that, assuming P_1, we can bound the clock difference by a constant. The last claim can be shown by analysis, so we focus our discussion on claim $\langle Hyp; Sys; P_1\rangle$. Next, we provide formal models for the system design and the hypothesis using rule (formalize.4). In the same step, property P_1 is interpreted as a LTL liveness property, G_1^n, where n is the index of the current round. Since the number of rounds is unbounded, it is not possible to check the property using an enumerative model-checker and more abstractions are needed.

The following step is to use an induction on the number of rounds, rule (induction.5). At this level, even though the state space of the models is finite, it is too big for model-checking (with more than 10^8 states). We apply rule (trace.7) to consider a more abstract model where propagation and hardware processing time (Γ_{tight}) is zero. Informally, this is justified by the fact that there are more “execution traces” in the abstract model (we check condition $\llbracket M_c \rrbracket \subseteq \llbracket M_a \rrbracket$) and by the fact that property G_1^n is untimed. Finally, we apply rule (split.8) to obtain more tractable model-checking problems by partitioning the set of initial conditions. For instance, if we consider a configuration of the system with 5 nodes and 2 faults, we end up with about thirty sub-claims to solve, each with a manageable state space size (a few millions states). In this case we can conclude the argument by providing model-checking results as evidence.

5 Conclusion and Related Work

We propose a methodology for structuring the formal models—and the reasoning—necessary to prove a claim on a system design. Our approach relies on the construction of a structured verification argument in which claims are linked with formal models and with a rationale showing how the model elements meet their requirements. We have followed this methodology to prove properties on a critical function of an autonomous rover, TwIRTee, that has been used as a test-bench for various engineering activities, see e.g. [2, 4]. While we focus on realtime model-checking in our illustration (Sect. 4), other formal methods have been used on the design of TwIRTee. For instance, SAT-based model-checking and the event-B method have both been used to check its collision avoidance function. We plan to apply our methodology to this use-case and to integrate all these results in a complete assurance case.

We can find some related works that include the use of formal methods within an assurance case. For instance, Denney et al. [7] use the results of a static analysis tool as part of an argument for an UAV. A similar work, with Event-B, is proposed in [13]. Likewise, Jee et al. [10] integrate real-time model-checking in their argument about the safety of a pacemaker. Nonetheless, they use a single model, as an evidence, but do not provide an incremental process to help derive this model from the specifications.

At the moment, our approach is not supported by any tooling. For future work, we envisage to provide end users with means to automate, or at least simplify, the management of claims. We would also like to help the user with selecting the appropriate verification tool. For instance, some of the rules given in Sect. 3.2 can be generalized into patterns that can be used to guide the generation of a valid argument. A possible solution for implementing our approach could be to integrate it with the Evidential Tool Bus of [17], that supports the integration of multiple verification tools in order to build assurance cases.

References

[1] Standard for property specification language (PSL). *IEC 62531:2012(E) (IEEE Std 1850-2010)*, 2012.

[2] Mathieu Clabaut, Ning Ge, Nicolas Breton, Eric Jenn, Remi Delmas, and Yoann Fonteneau. Industrial grade model checking - use cases, constraints, tools and applications. In *Int. Conf. on Embedded Real Time Software and Systems (ERTSS)*, 2016.

[3] Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. Tool integration with the evidential tool bus. In *Proc. of VMCAI*. Springer, 2013.

[4] P. Cuenot, E. Jenn, E. Faure, N. Broueilh, and E. Rouland. An experiment on exploiting virtual platforms for the development of embedded equipments. In *Int. Conf. on Embedded Real Time Software and Systems (ERTSS)*, 2016.

[5] Abhishek Datta and Vigyan Singhal. Formal verification of a public-domain DDR2 controller design. In *21st Int. Conference on VLSI Design*. IEEE, 2008.

[6] E. Denney and G. Pai. Evidence arguments for using formal methods in software certification. In *Software Reliability Engineering Workshops (ISSREW)*, 2013.

[7] Ewen Denney, Ganesh Pai, and Josef Pohl. Heterogeneous aviation safety cases: Integrating the formal and the non-formal. In *17th Int. Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2012.

[8] RTCA DO. 178c. *Software considerations in airborne systems and equipment certification*, 2011.

[9] Harry Foster. *Applied assertion-based verification: An industry perspective*. Now Publishers Inc, 2009.

[10] Eunkyoung Jee, Insup Lee, and Oleg Sokolsky. Assurance cases in model-driven development of the pacemaker software. In *Leveraging Applications of Formal Methods, Verification, and Validation*. Springer, 2010.

[11] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4), 1981.

[12] Amir Pnueli. *In transition from global to modular temporal reasoning about programs*. Springer, 1985.

[13] Yuliya Prokhorova and Elena Troubitsyna. Linking modelling in Event-B with safety cases. In *Software Engineering for Resilient Systems*. Springer, 2012.

[14] Luís Rodrigues, Mário Guimaraes, and José Rufino. Fault-tolerant clock synchronization in CAN. In *19th IEEE Symp. on Real-Time Systems*. IEEE, 1998.

[15] J. Rufino, P. Verissimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. In *28th Int. Symp on Fault-Tolerant Computing*, 1998.

[16] John Rushby. On the interpretation of assurance case arguments. In *2nd Int. Workshop on Argument for Agreement and Assurance (AAA)*, 2015.

[17] John M Rushby. An evidential tool bus. In *7th International Conference on Formal Engineering Methods (ICFEM)*, volume 3785 of *LNCS*. Springer, 2005.

[18] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM (JACM)*, 34(3), 1987.

[19] Pascal Traverse, Isabelle Lacaze, and Jean Souyris. Airbus fly-by-wire: A total approach to dependability. In *Building the Information Society*. Springer, 2004.

[20] Pierre Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic. In *13th Symp. on Principles of Programming Languages*. ACM, 1986.