Improved Invariant Generation for Industrial Software Model Checking of Time Properties

Vassil Todorov, Safouan Taha, Frédéric Boulanger, Armando Hernandez

Improved invariant generation for industrial software model checking of time properties

Vassil Todorov
Quality and Engineering Department
LRI, Groupe PSA
Vélizy-Villacoublay, France

Safouan Taha
CS Department
LRI, CentraleSupélec
Gif-sur-Yvette, France

Frédéric Boulanger
CS Department
LRI, CentraleSupélec
Gif-sur-Yvette, France

Armando Hernandez
Quality and Engineering Department
Groupe PSA
Vélizy-Villacoublay, France

***Abstract*—Modern automotive embedded software is mostly designed using model-based design tools such as Simulink or SCADE, and source code is generated automatically from the models. Formal proof using symbolic model checking has been integrated in these tools and can provide a higher assurance by proving safety-critical properties. Our experience shows that proving properties involving time is rather challenging when they involve long durations and timers. These properties are generally not inductive and even advanced techniques such as PDR/IC3 are unable to handle them on production models in reasonable time.**

In this paper, we first present our industrial use case and comment on the results obtained with the existing model checkers. Then we present our invariant generator and methodology for selecting invariants according to physical dimensions. They enable the proof of properties with long-running timers. Finally, we discuss their implementation and benchmarks.

***Index Terms*—Software verification · Formal methods · Model checking · SMT solving · Invariant generation · Time properties**

I. Introduction and Motivation

Model-based design tools such as Simulink or SCADE are massively used in today's automotive embedded software design. These tools provide a higher level of abstraction compared to implementation code, and losen the coupling between system and software design. In the near future, with the advent of autonomous vehicles, car manufacturers could be asked to provide more guarantees and confidence to those systems by demonstrating the correctness of the software, at least for the safety-critical part. Actually, in industries such as *aviation* and *railway*, these guarantees are achieved by obtaining a certificate from a certification authority claiming that the software is safe (aviation) or that it is compliant with the safety standard of the domain (railway). For some use cases, it is possible to use formal methods for certification. For example, Airbus illustrates the use of Deductive Proof and Abstract Interpretation tools in [1]. Industrial experiments were done by the Ariane group in [2], by Dassault Aviation in [3], by Rockwell Collins in [4] and by other companies. Formal methods such as the B Method and Model Checking are used by some railway companies, and use cases are presented in [5] and [6].

In a previous paper [7], we shared our experience about applying tools that use formal methods (Abstract Interpretation, SMT-based Model Checking and Deductive Proof) on industrial software. As we are interested in automatic verification of safety properties, we will focus on SMT-based Model checking. Abstract Interpretation is well adapted for proving the absence of runtime errors, and Deductive Proof requires too much human intervention for our purpose.

We will use symbolic model checking to prove the correctness of properties about the deactivation of the cruise control of a car. These properties are safety related invariants about system requirements and are generally modeled as observers [8]. Observers are described in the same language as the model, and engineers can easily write them. When proved valid, they provide a higher assurance level of software correctness. Otherwise, the designer gets a valuable counterexample that helps him correct his design. In our case, we distinguish two kinds of safety properties, those that are stateless invariants and those that involve timers.

For our use case, we used SCADE with its integrated prover called Design Verifier (SCADE DV). We found that all our properties were proved immediately except those that were using long-running timers. For some models, it was impossible to prove them within 24 hours. The commercial tools that we used are black boxes and it is impossible to understand the reason of this proof failure by putting them into debug mode for example. As timers are something very common in industrial models, this motivated us to convert our SCADE model into Lustre [9], and then use some open source model checkers to understand why they were not proved, and to experiment new algorithms for proving this type of properties.

Our contribution, presented below, is about scaling the proof of time properties on production models. We propose a general improvement for invariant generators and a new methodology for obtaining better performance and scaling. As a proof of concept, we implemented it in the JKind model checker [10] and tested it on our representative production model, as well as on some production models from Rockwell Collins (now Collins Aerospace). The result is that properties previously very slow or impossible to prove are now proved within a few seconds and in a completely automatic manner.

II. Use case presentation

In this section, we first present the algorithms we used for proving properties then we present our model and the properties to be proved.

A. Preliminaries

1) Induction-based Model Checking: An inductive approach called k-Induction was proposed in [11]. It consists in proving that a property/specification is valid for all reachable states of a system i.e. it is *invariant*. It proceeds in two steps: firstly, it proves that the property is initially satisfied by the k first states, secondly it proves that if the property holds for k successive states it also holds for their successor. Common inductive-based model checkers iterate over k and send requests to SMT-solvers to check the validity of the k-Induction. At iteration k, each variable is duplicated k times, so the size of the SMT requests grows with the number of iterations and their solving time grows exponentially with k.

2) Invariant generation: An invariant generation procedure produces lemmas to allow the proof of properties that are not k-inductive. An invariant can be totally inductive or k-inductive for some $k \geq 2$.

3) Property Directed Reachability: Property Directed Reachability (PDR), which is also known as IC3[1], was developed initially for purely propositional systems and hardware verification [12], [13]. Some generalization for SMT was introduced in [14] and [15] for software verification. When a property is not inductive, PDR uses the counterexample provided by the SMT-solver to generate and/or strengthen the invariants. A sequence of frames blocking the dangerous states is constructed incrementally, mixing backward analysis of the proof obligation and forward propagation taking into account the initial states. Despite interval generation policies applied to the SMT counterexamples, PDR only generates invariants of simple forms that may not be strong enough to prove the property.

In practice, inductive model checkers often use a combination of the above techniques.

B. Model and Environment

We illustrate the use of symbolic model checking to prove the correctness of safety properties on a representative production model of a cruise control function. This function manages the speed of the car, switches to the right operating mode, manages the user interface, detects faults and decides whether the function should be turned on or off. It uses only linear arithmetic over integers.

We used ANSYS SCADE Suite (Safety Critical Application Development Environment) to design the model from low-level software textual requirements. The properties to be checked were also modeled in SCADE from high-level system safety requirements. The proof was done with SCADE DV [16], which is a symbolic model checker integrated in SCADE. We chose SCADE for our experiments because it has formal foundations [17] compared to Simulink, which is more simulation oriented and without a single formal background [18]. Actually, SCADE has a formal language based on Lustre, thus we could compare its internal model checker with other open source model checkers for Lustre.

[1]Incremental Construction of Inductive Clauses for Indubitable Correctness

At Groupe PSA, the embedded software is developed according to a standard V-Model methodology. High-level system requirements (HLR) are allocated to an Electronic Control Unit (ECU). Then they are decomposed into low-level software requirements (LLR) used to develop the code (handwritten or model-based).

We present our use case environment on Fig. 1. It is composed of a SCADE model, properties, and assumptions when needed. We used multiple model checkers to compare their performances (GATeL [19], SCADE DV [16], JKind [10], Kind2 [20]), and addressed multiple SMT solvers in the back-end (CVC4 [21], MathSAT [22], SMTInterpol [23], Yices2 [24], Z3 [25]). GATeL has its own SMT called Colibri developed at CEA. SCADE DV has its own SMT provided by Prover Technologies. JKind can use CVC4, MathSAT, SMTInterpol, Yices2 and Z3 via SMT-LIB [26]. Kind2, the successor of PKind and Kind, can use CVC4, Yices2 and Z3 also via SMT-LIB. At the moment of our experiments, Kind2 was unable to use IC3 with Yices2.

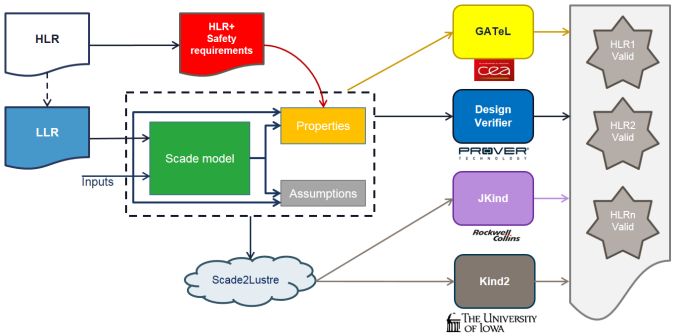

Fig. 1. Verification using multiple model checkers and multiple SMT solvers

Fig. 2 shows the principal blocks of our SCADE model. It contains an automaton for managing modes, a function for enabling/disabling the cruise control and a function for managing transitions. These components communicate with each other. We want to prove the correctness of the model by writing safety-properties as observers over the SCADE model.

C. Writing formal properties

As reported in [4], writing good formal properties shares many similarities with writing good requirements and is as much art as science. This report mentions that properties that cut across an entire system often find the most errors and that the best sources of formal properties are found in the safety-related requirements for the system.

We therefore formalized properties from the safety-related requirements, and extended them to all high-level requirements (HLR) concerning the deactivation of the function. We have safety-related requirements separated from the HLRs because they are written by a safety engineer and the HLRs are written by the function designer. We want to prove the validity of all these properties i.e. no matter what happens, the cruise controller will deactivate upon the specified conditions. Some of these requirements are listed in Table I.

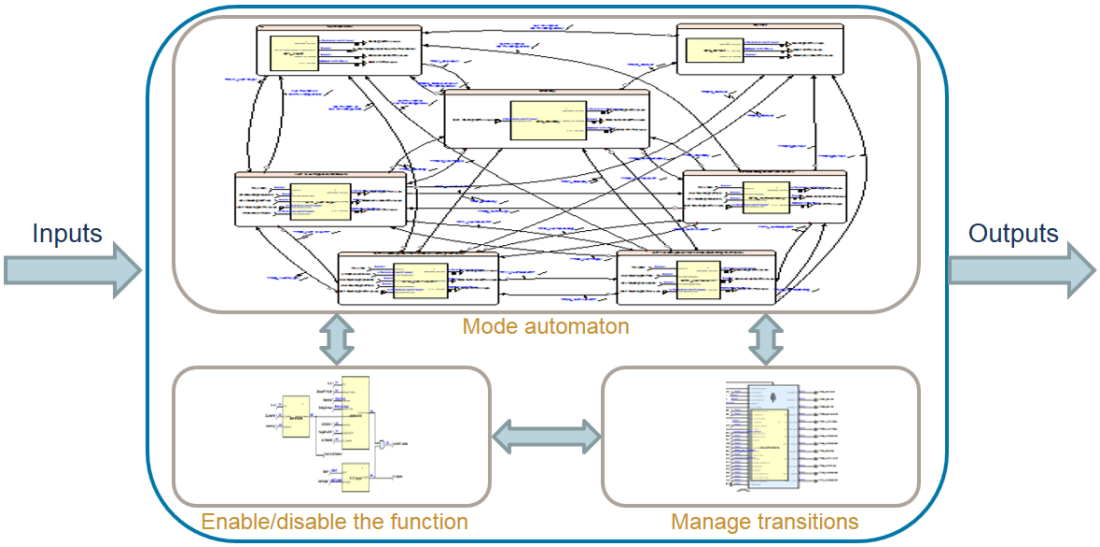

Fig. 2. Cruise controller SCADE model's principal blocks

TABLE I
SYSTEM REQUIREMENTS USED FOR MODEL CHECKING

REQ-01	A simple press on the *Cancel* button shall disable the cruise controller.
REQ-02	Switching off the ignition shall disable the cruise controller.
REQ-03	In order to respect the safety objectives when the brake pedal sensor is not working: a deceleration (*Decel*) under a defined threshold value (*T2*) and the brake pedal not seen pressed during 2 seconds shall turn off the function.

REQ-01 and REQ-02 are typical stateless invariants that are well handled and easy to prove with the actual model checkers.

REQ-03 uses time. When time is increased in the property and in the model, this makes the proof difficult because the number of states explodes. For our experiments, we modeled this property at three different levels:

- We name *PG* the *global property* that is checked at the bounds of the whole system (1300 lines of Lustre code and 78 nodes), see Fig. 3.
- Then we keep the model of the whole system but we rewrite *PG* into *PL*, which is the same property *expressed locally* on the bounds of the node that implements the authorization function, see Fig. 4.
- Finally, we *isolate* the node that implements the authorization function (320 lines of Lustre code and 3 nodes) to reduce the state-space, and call *PI* the property to be checked, which is the same as *PL*, only the context is different, see Fig. 5.

The inductive model checkers that we considered implement slicing algorithms such as the cone of influence (COI). Roughly, the cone of influence of a property is the structural part of the design on which the property depends. Before starting an analysis, the COI is computed in order to remove the parts of the design that have no influence on the property under analysis. We decided to check the *PL* property in order to see the efficiency of the slicing algorithms, and also to use it for compositional analysis, see section II-D. It is equivalent to *PI* but has some sort of environment that can give preconditions and reduce its state space.

In the case of *PI* (Fig.5), we noticed that two things made the proof difficult. Firstly, trying to prove a property over a *long period of time*, such as 2 minutes instead of 2 seconds, takes too much memory or time for some model checkers. We call this property *PI-X* where *X* is the number of 50 ms *time steps* (for example, 2 seconds represent 40 steps). Secondly, we want to check the difficulties that a model checker would have when checking a valid property that does not match exactly what the code does. We consider two variants of *PI-X* depending on the deceleration threshold:

- *PI-X-T2*: $(Decel < T2 \wedge X) \Rightarrow PI$ is the original property;
- *PI-X-T1*: $(Decel < T1 \wedge X) \Rightarrow PI$ has a stronger precondition because $T1 < T2$, thus it is a weaker property, which is valid when *PI-X-T2* is valid.

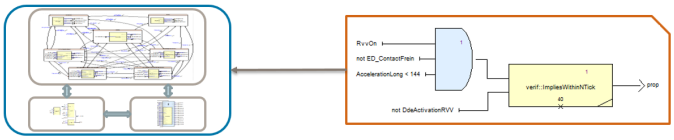

Fig. 3. Property *PG-40* expressed on the bounds of the model

Fig. 4. Property *PL-40* expressed on a sub-node

Fig. 5. Property *PI-40* expressed on an isolated sub-node

Our final goal was to prove the global property *PG* (involving the entire model) directly, but we noticed that for long-running time properties it was impossible to scale. We decomposed it in two smaller properties and used a compositional approach to prove the property on the entire model. Property *PL* (expressed locally on the node implementing it) was used for compositional reasoning as discussed in the next section.

D. Compositional approach

A compositional approach reduces the complexity of the verification of a big model by dividing it into 2 or more components. We divided our model into two components as shown on Fig.6:

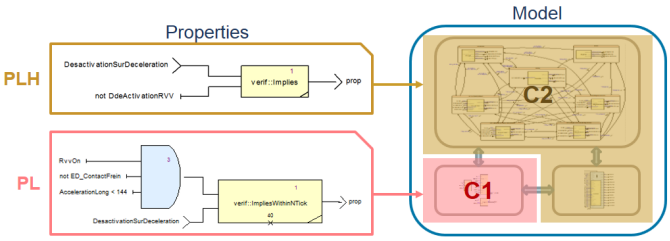

Fig. 6. Compositional approach for properties PL and PLH

- *C1* (pink): The authorization function takes into account the inputs of the model and produces an intermediate result.
- *C2* (brown): The rest of the model that produces outputs which uses the intermediate result.

Then we used two properties applied on C1 and C2, that put together, are equivalent to the global property *PG*:

- *PL* (pink): Property *PL* is expressed locally on the node implementing the authorization function. It takes into account the inputs of the whole model and the intermediate output.
- *PLH* (brown): The local output of the authorization function is used to prove the global model output.

E. Results analysis

In this section, we comment on the results obtained for *PG*, *PL* and *PI* with a small number of time steps (40, which is equivalent to 2 seconds) and with a large number of time steps (2400, equivalent to 2 minutes). We also used two different values for the properties deceleration threshold: *T1* and *T2*, where $T1 < T2$. Experiments were run on an Intel® Xeon® CPU E5-2609 v2 @ 2.50GHz and 64 GiB of memory with SCADE DV, Kind2 1.1.0 and JKind 4.0.1. We used all available SMT-solvers with Kind2 and JKind but found that Kind2 generally works best with Z3 (4.7.1) and JKind with Yices2 (2.5.2). Our results listed below are obtained with these solvers. Kind2 does not support PDR/IC3 with Yices2 and thus cannot be compared to JKind with Yices2, which supports it. The timeout option of the model checkers was set to 2 hours (wall-clock time). The results obtained by GATeL were unsound and we do not comment on this model checker in the rest of the paper.

1) Invariant generation is mandatory: Our first experiment was to disable PDR and invariant generation processes and we found that this type of time properties were not *k*-inductive even for 2-step models. We needed additional invariants to strengthen the property.

2) PDR/IC3 only for small timers and models: Our results show that PDR is a good strengthening algorithm only for small numbers of time steps and small models. With the time span of properties and the size of the model, there is a combinatorial explosion.

3) Threshold impact: The deceleration threshold *T1* affected essentially SCADE DV. We noticed that Kind2 and JKind had no problem with it, even if it slowed down the proof. The SMT solvers behind them have stronger theories on integers. The threshold *T2* affected essentially Kind2, as long-running timers were impossible to prove with it.

4) Subnode property PI: Increasing the time span from 40 to 2400 steps for the small model with property *PI* took more time with Kind2, but resulted in a timeout for JKind. JKind and Kind2 use a different implementation of the template-based invariant generation techniques described in [27]. Even when the PDR process produces the proof, it sometimes uses invariants provided by the invariant generator, we noted it *PDR+Invgen* in the results below.

5) Compositional approach with property PL: We used property *PL* combined with *PLH* to decompose the complex property *PG* into two simpler problems. Proving *PL* is almost equivalent to *PI* when using slicing because it eliminates the code that is not concerned by the property. JKind was unable to prove the long time *PL* property, and Kind2 showed that for the *T1* threshold it was possible to prove it using its invariant generator, but not for *T2*.

6) Global property PG: Our final goal was to prove the global property *PG*, taking into account the entire model with a long-running timer (2400 steps). We encountered some difficulties with SCADE to prove it when using the *T1* threshold, and Kind2 was unable to prove it with the *T2* threshold. JKind was unable to prove long-running timers at all. This motivated us to try to understand these difficulties.

Table II and Table III present the results obtained with different number of time steps and thresholds. The measured time is in seconds of wall-clock time. As Kind2 and JKind run multiple engines such as PDR, *k*-Induction and invariant generation in parallel, we put the engine that provided the first result. The second one helped the first with a useful invariant.

TABLE II
RESULTS USING DECELERATION THRESHOLD *T1*

	SCADE DV	Kind2 / Z3	JKind / Yices2
PI-40	2972	141.7 | PDR+Invgen	10.7 | Invgen
PI-2400	Timeout	139.6 | PDR+Invgen	Timeout
PL-40	Timeout	156.8 | Invgen	12.6 | PDR
PL-2400	Timeout	1353 | PDR+Invgen	Timeout
PG-40	Timeout	373.7 | Invgen	7064 | PDR
PG-2400	Timeout	155.2 | Invgen	Timeout

TABLE III
RESULTS USING DECELERATION THRESHOLD *T2*

	SCADE DV	Kind2 / Z3	JKind / Yices2
PI-40	3	3.8 | Invgen	0.8 | Invgen
PI-2400	2	8.1 | Invgen	Timeout
PL-40	7	23.9 | Invgen	12.5 | PDR
PL-2400	11	Timeout	Timeout
PG-40	9	1370 | Invgen	51.2 | PDR
PG-2400	11	Timeout	Timeout

III. APPROACH AND CONTRIBUTION

Although our model used linear arithmetic over integers, we noticed that increasing the time span of our global property based on REQ-03, e.g. from 2 seconds to 2 minutes, made the proof with SCADE DV fail in a reasonable time (24 hours). To understand the problem, we translated the SCADE model with its properties into the Lustre language, and used open source SMT-based model checkers, putting them into debug mode to analyze the situation.

A. SCADE to Lustre transformation

As SCADE has a textual language inherited from Lustre, we developed a tool based on an XSLT transformation called *SCADE2Lustre*. We used SCADE to convert our model into the SCADE textual language and then we transformed this textual representation of our model into Lustre code using our *SCADE2Lustre* tool. As JKind does not support SCADE automata, we rewrote our automaton in Lustre and checked using JKind and Kind2 that we had the same proof results as those obtained by SCADE DV.

B. Understanding the problem

We analyzed our model with different numbers of time steps, different algorithms such as *k*-Induction, PDR/IC3 and invariant generation, at different levels of abstraction (properties *PG*, *PL*, *PI*). We also used different model checkers and different SMT solvers as back-ends. We found use cases with long-running timers in production models that all available model checkers were unable to prove. In order to understand the problem we decided to use and modify JKind for its particular implementation of Inductive Validity Cores (IVC) [28]. We used IVC to get the invariants that had enabled the proof. It was useful for understanding what candidates we needed to generate for the proof.

1) k-Induction: The basic idea behind *k*-Induction is to make use of invariants that are not 1-inductive. With the increase of *k*, there is a combinatorial explosion, so it can run for a very long time. It was the case for our property involving time because it was not *k*-inductive for a small *k*. This is why we needed a smarter invariant generator to help strengthen the property before *k* goes too high.

2) PDR/IC3: PDR can strengthen the property, but the number of invariants it constructs from the property explodes when the time span of the property and the size of the model increase. Because of the interval generation, most invariants are useless for our proof and just slow down the proof process. Furthermore, these invariants appear to find relations only between variables and constants but not between multiple variables. For long-running time properties on production models, PDR suffered from the same combinatorial explosion problem as *k*-Induction.

3) The JKind invariant generator: JKind uses a template-based lemma generation, as described in [27], for its invariant generation procedure. In order to obtain invariants to strengthen the proof, JKind creates a list of candidates representing literals. Four different types of candidates are generated automatically:

- **Boolean candidates**: all boolean system variables and their negations e.g. *a* and *not a* where *a* is a boolean.
- **Init candidates**: integer variables are compared with $\geq$ and $\leq$ operators to their initial values e.g. $(i \geq 0), (i \leq 0)$ where 0 is the initial value of *i*.
- **Subrange candidates**: variables of an integer subrange type are compared for equality to all the values in the subrange, e.g. $(s = 0), (s = 1), (s = 2)$ for $s \in [0..2]$.
- **Enum candidates**: variables of an enum type are compared for equality to all the values of the enum.

The invariant generator checks all propositional formulae (with boolean operators) involving these literals, whether they are invariants or not. The number of formulae grows exponentially with the number of literals. To avoid this combinatorial explosion, JKind reduces its candidates to those listed above and no relational candidates (explained in subsection III-C) are considered.

All these candidates were not strong enough for proving our long-running timer properties.

C. Contribution

JKind uses multiple cooperative engines in parallel, including *k*-Induction, PDR and template-based invariant generation. We worked on the improvement of the invariant generation. We noticed that our property used a constant value for the number of time steps and the code also used a constant for it. The same was true for other clauses in the property. We needed invariants that could provide information about the relation (essentially a comparison) between constants and variables of the property and constants and variables of the model. In order to find relations between the property and the model, we propose two new additional categories of relational candidates (atoms):

- **INT** $\times$ **INT**: for all integer variables in the model and the property, add a comparison relation with the $\geq$ operator, e.g. $Variable1 \geq Variable2$

- **INT** $\times$ **CONST**: for all integer variables and constants in the model and the property, add comparison relations with the $\geq$ and $\leq$ operators, e.g. $Variable1 \geq Constant1$; $Variable1 \leq Constant1$

We implemented this new invariant generation algorithm in JKind and applied it to a sub-node of our model with a large number of time steps (property PI-2400). We were able to prove the property within a few seconds although it was impossible to prove before. Once we could prove the property, it was possible to use IVC to find the invariant that had enabled the proof. We used it to understand what were the most useful candidates we needed to generate for the proof.

Next, we wanted to prove the entire model with a long time property (property PG-2400). With our new invariant generator, we had the needed candidates but for the entire model their number was too big. We noticed also that numerous candidates did not make sense, e.g. when comparing a variable about speed to a constant about deceleration, or comparing counters with non counter elements. To get interesting invariants, we propose to use the physical type (speed, deceleration, counter, etc.) of the variables and the constants, and to keep only candidates that compare elements of the same physical type. We explain this in details in the next section.

1) Physical types methodology: A physical quantity is a physical property that can be quantified by measurement. A physical quantity can be expressed as the combination of a number and a unit. For example, in the physical world, we measure the quantity of speed using the unit $\mathrm{m\,s^{-1}}$ and its derivations. The same is true for other physical quantities. In the automotive and other industries, most of the external interfaces of a function represent a physical quantity (speed, deceleration, battery voltage, etc.) and has a physical unit.

At the code level, information about units is lost and only numbers are present. Fortunately, at the software architecture design level, the physical units are present. As most of the software is designed using model-based design tools, this information can be used for model verification. We propose to use this semantic information to have a deeper understanding of the variables and to generate less invariant candidates while increasing their usefulness. As a methodology, we propose the introduction of physical types at the model level for tools such as Simulink or SCADE. Instead of using a base representation types such as *int*, we declare a type for each physical quantity, e.g. *tSpeed*, *tDeceleration*, *tVoltage*, *tCounter* etc. Then all the variables and constants are typed according to their appropriate physical type. Actually, these new types are just aliases of the base representation types, but they carry more semantics for our algorithm. Thus we can recognize data of the same physical type and reason on them using the appropriate relations, see Fig. 7.

For our use case, we defined physical types in SCADE during the design phase. Then, we used them for all the constants and variables of the model. This takes little when done during the design stage. As types are immediately evaluated and shown on the SCADE model, it also gave a better readability during the review of the model. Once the model was validated, we converted it into Lustre code using our SCADE2Lustre converter, which preserves types.

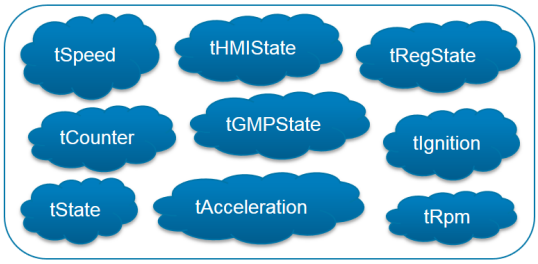

Fig. 7. Constants and variables partitioned by their physical types

2) Timers patterns: We wanted to optimize further our algorithm, and to push only the most relevant candidates. By analyzing the useful candidates from the minimal invariants used for the proof (see IVC above), we noticed that all the variables that were useful were assigned a previous value (they correspond to state variables). We propose to eliminate variables that are not assigned a previous value, which correspond to combinatorial variables for which SMT solvers are very efficient, so invariant generation is not necessary. An example of state and combinatorial variables is shown on Fig. 8

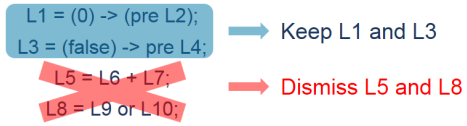

Fig. 8. The interesting variables for proving properties about time are those that encode a state. The others are dismissed.

3) Implementation in JKind: We introduced our improvement on a GitHub branch of JKind[2] called "invgen-timers". We modified JKind to be able to preserve the original Lustre types because they were lost after inlining. We introduced an option "-inv_gen_level" proposing more and more candidates when the level increases:

- Level 0: Default JKind level before our improvements
- Level 1: Use the physical types methodology with INT $\times$ INT and INT $\times$ CONST relational candidates, restricted to state variables (variables with an assignment of a value from a previous state). This level performs best if our physical types methodology is applied.
- Level 2: Uses INT $\times$ INT and INT $\times$ CONST relational candidates no matter their type, restricted to state-variables. This level works for models that do not use physical semantic types.
- Level 3: Uses INT $\times$ INT and INT $\times$ CONST relational candidates including state-variables and combinatorial variables. This level can be used if the other levels do not provide the necessary invariants.

The idea behind this new option is to provide different amount of invariants so that the user can begin with the lowest level. If

[2]JKind on GitHub: https://github.com/agacek/jkind

the property cannot be proved with it, the next level could be used until the property is proved. Beginning with the highest level may degrade the performance for properties where a lower level would be sufficient.

IV. Results and Benchmarks

In this section, we examine the results obtained by our invariant generation algorithm and methodology using physical types compared to the results obtained with the official versions of JKind and Kind2. We also used JKind's and Kind's benchmarks to find use cases about timers and compare performances. Finally, we asked Rockwell Collins for use cases about timers and found that some properties on production models, which were not proved before with JKind, were now proved within a few seconds thanks to our improvements.

A. Our use cases

We summarize here the results obtained with our cruise controller model.

In tables IV and V we present the results in seconds obtained using our methodology (JKind new) based on physical types compared to the previous results (Kind2 and JKind official versions).

TABLE IV
Results using our new invgen and types for threshold *T1*

	SCADE DV	**Kind2**	**JKind (official)**	**JKind (new)**
PI-40	2972	141.7	10.7	0.2 | Invgen
PI-2400	Timeout	139.6	Timeout	0.2 | Invgen
PL-40	Timeout	156.8	12.6	4.7 | Invgen
PL-2400	Timeout	1353	Timeout	5.3 | Invgen
PG-40	Timeout	373.7	7064	4.3 | Invgen
PG-2400	Timeout	155.2	Timeout	4.8 | Invgen

TABLE V
Results using our new invgen and types for threshold *T2*

	SCADE DV	**Kind2**	**JKind (official)**	**JKind (new)**
PI-40	3	3.8	0.8	0.1 | Invgen
PI-2400	2	8.1	Timeout	0.1 | Invgen
PL-40	7	23.9	12.5	4.2 | Invgen
PL-2400	11	Timeout	Timeout	4.2 | Invgen
PG-40	9	1370	51.2	4.2 | Invgen
PG-2400	11	Timeout	Timeout	4.4 | Invgen

We notice that JKind new (Level 1) outperforms the official versions of the other model checkers for both deceleration threshold values $T1$ and $T2$.

B. JKind benchmark

JKind provides with its source files, 56 Lustre programs with properties to be proved. We used it to compare the performance of our different algorithms with the official one. We did not find long-running timers in this benchmark and the new levels of invariant generation we introduced in JKind did not bring better results. We suppose that this benchmark was tuned for the current JKind version, as there are no unsolvable problems in it (everything can be proved or invalidated).

C. Kind benchmark

We also used a suite of 1047 Lustre programs developed as a benchmark for Kind [29]. Most of them were very small and not containing timers. Their properties were proven in less than a second. However, we found some programs that were using timers. We present their results in Table VI using our implementation of the three different levels of invariant generation (L1, L2 and L3), compared to the JKind and Kind2 official versions. The timeout was set to 10 minutes, Z3 was used with Kind2 and Yices2 with JKind.

TABLE VI
Results using our new invgen on Kind benchmark

Program	Kind2	JKind	JKind-L1	JKind-L2	JKind-L3
P1	Timeout	13.2	4.3	9.2	6.6
P2	Timeout	9.9	7.9	10.3	4
P3	Timeout	13.2	9.4	7.6	13.6
P4	Timeout	6.5	8.7	10.1	3.8
P5	Timeout	9.1	58.7	6.9	5.6
P6	Timeout	Timeout	Timeout	1.7	1.3
P7	Timeout	Timeout	1.8	1.4	Timeout
P8	19.3	50.4	6.3	4.4	7.7
P9	0.4	1.3	1.3	0.2	0.2

The full names of these programs are:

- P1: DRAGON_11.lus
- P2: DRAGON_11_e1_2450.lus
- P3: DRAGON_11_e1_2450_e1_5887.lus
- P4: DRAGON_11_e1_2450_e2_1483.lus
- P5: DRAGON_11_e2_5396_e3_282.lus
- P6: durationThm_3_e3_442_e6_113.lus
- P7: durationThm_3_e7_334_e8_369.lus
- P8: microwave05.lus
- P9: twisted_counters.lus

This benchmark does not use physical types. All the variables and constants are of type integer, real or boolean. Our level 1 invariant generator is more suitable when physical types are used. However, we can see that levels 2 and 3 performed well on these programs containing counters.

D. Rockwell Collins use cases

At Rockwell Collins, Lustre is used as an intermediate language to make formal proofs of high-level properties. Some models have properties with long-running timers. First, they provided us with a representative version of their production model with a property using 6000 time steps that was impossible to prove before. We proved it in a few seconds. Then we shared our new version of JKind with them so that they try it on their internal production models that they could not share with us. They told us that it proved in a few seconds properties that were not proved before.

V. Conclusions and Future Work

In this paper, we have proposed an algorithm that brings an improvement to invariant generation, enabling the automatic proof of properties involving long-running timers, which are present in most embedded software. This algorithm consists in

injecting new relational invariant candidates to enrich the invariant generation. However, if too few candidates do not allow concluding, too many candidates can slow down the proof and lead to timeouts. That is why we propose a new methodology using physical types (speed, deceleration, etc.), which restricts the number of candidates to only those that make sense (e.g. deceleration variables compared to deceleration constants or speed variables compared to other speed variables) and may therefore be useful for the proof. Our algorithm is applicable to all forms of inductive model checkers. We have implemented it as part of the open source model checker JKind. We have shown that it outperforms the official versions of JKind, Kind2 and SCADE Design Verifier on benchmarks and on several industrial use cases.

With this paper, we also want to show a way to improve the state of the art in formal methods. When using real production models we do not have access to advanced academic model checkers and solvers because industrial companies essentially use black box tools that cannot be put in a debug mode or modified. If the proof is not possible, it is very difficult to understand why. We used the Lustre language as an intermediate language between the black box tools and the open source model checkers. This allowed us to understand what was missing to automatically strengthen the proofs, and to implement our new algorithm in JKind as a proof of concept.

The presented method for generating invariants is working fine when the counter in the property and the counter in the model evolve at the same rate and sequentially. This is the most common case for industrial models. There can exist models that increment counters at different rates. For the moment, these models need additional invariants provided by the user as assertions.

Our invariant generation technique and methodology using types could also be applied to a more difficult problem: proving properties on nonlinear systems. Modern SMT solvers take into account some nonlinear theories, but it is time consuming to obtain models for complex nonlinear queries. We plan to investigate how the nonlinear barrier could be removed using invariant generation.

REFERENCES

[1] J. Souyris, V. Wiels, D. Delmas, and H. Delseny, "Formal Verification of Avionics Software Products," in *Proceedings of the 2Nd World Congress on Formal Methods*, ser. FM 09, Berlin, Heidelberg, 2009, pp. 532–546.

[2] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin, "Space Software Validation using Abstract Interpretation," in *Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA 2009)*, vol. SP-669. ESA, May 2009, pp. 1–7.

[3] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate, "Testing or Formal Verification: DO-178c Alternatives and Industrial Experience," *IEEE Soft.*, vol. 30, no. 3, 2013.

[4] D. Cofer and S. P. Miller, "Formal Methods Case Studies for DO-333," Tech. Rep., Jan. 2014.

[5] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier, "Météor: A Successful Application of B in a Large Project," in *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems Toulouse, France, September 20–24, 1999 Proceedings, Volume I*, J. M. Wing, J. Woodcock, and J. Davies, Eds., 1999, pp. 369–387.

[6] M. Petit-Doche, N. Breton, R. Courbis, Y. Fonteneau, and M. Güdemann, "Formal Verification of Industrial Critical Software," in *FMICS 2015, Oslo, Norway, June 22-23*, 2015, pp. 1–11.

[7] V. Todorov, F. Boulanger, and S. Taha, "Formal Verification of Automotive Embedded Software," in *Proceedings of the 6th Conference on Formal Methods in Software Engineering*, ser. FormaliSE '18. New York, NY, USA: ACM, 2018, pp. 84–87.

[8] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," in *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Jun. 1993.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A Declarative Language for Real-time Programming," in *POPL '87*. ACM, 1987, pp. 178–188.

[10] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani, "The JKind Model Checker," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer, 2018, pp. 20–27.

[11] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *FMCAD 2000*, pp. 127–144.

[12] A. R. Bradley and Z. Manna, "Property-directed incremental invariant generation," *Formal Aspects of Computing*, vol. 20, pp. 379–405, 2008.

[13] N. Een, A. Mishchenko, and R. Brayton, "Efficient Implementation of Property Directed Reachability," ser. FMCAD '11, Austin, 2011, pp. 125–134.

[14] A. Cimatti and A. Griggio, "Software Model Checking via IC3," in *CAV 2012, Berkeley, CA, USA*. Springer, 2012, pp. 277–293.

[15] K. Hoder and N. Bjørner, "Generalized Property Directed Reachability," in *Proceedings of SDAT'12*, ser. SAT'12, 2012, pp. 157–171.

[16] A. Bouali and B. Dion, "Formal Verification for Model-Based Development," in *SAE Technical Paper 2005-01-0781*, 2005.

[17] J.-L. Colaço, B. Pagano, and M. Pouzet, "Scade 6: A Formal Language for Embedded Critical Software Development," in *TASE 2017 - 11th International Symposium on Theoretical Aspects of Software Engineering*, Nice, France, Sep. 2017, pp. 1–10. [Online]. Available: https://hal.inria.fr/hal-01666470

[18] L. Zou, N. Zhan, S. Wang, and M. Fränzle, "Formal verification of simulink/stateflow diagrams," in *Automated Technology for Verification and Analysis, ATVA 2015*. Springer, pp. 464–481. [Online]. Available: https://www.springer.com/gp/book/9783319249520

[19] B. Marre and B. Blanc, "Test Selection Strategies for Lustre Descriptions in GATeL," *Electron. Notes Theor. Comp. Sci.*, vol. 111, no. C, pp. 93–111, Jan. 2005.

[20] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli, "The Kind 2 Model Checker," in *CAV 2016, Toronto, Canada*, 2016, pp. 510–517.

[21] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *(CAV'11)*, ser. LNCS, vol. 6806. Springer, Jul. 2011, pp. 171–177.

[22] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin: Springer, 2013, pp. 93–107.

[23] J. Christ, J. Hoenicke, and A. Nutz, "SMTInterpol: An Interpolating SMT Solver," in *Model Checking Software*, Berlin, 2012, pp. 248–254.

[24] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds., Cham, 2014, pp. 737–744.

[25] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proc. of the Theory and Practice of Software*, ser. ETAPS'08, Berlin, 2008, pp. 337–340.

[26] C. Barrett, A. Stump, C. Tinelli, S. Boehme, D. Cok, D. Deharbe, B. Dutertre, P. Fontaine, V. Ganesh, A. Griggio, J. Grundy, P. Jackson, A. Oliveras, S. Krstić, M. Moskal, L. D. Moura, R. Sebastiani, T. D. Cok, and J. Hoenicke, "The SMT-LIB Standard: Version 2.0," Tech. Rep., 2010.

[27] T. Kahsai, Y. Ge, and C. Tinelli, "Instantiation-based Invariant Discovery," in *Proceedings of the Third International Conference on NASA Formal Methods*, ser. NFM'11, Berlin, 2011, pp. 192–206.

[28] E. Ghassabani, A. Gacek, and M. W. Whalen, "Efficient Generation of Inductive Validity Cores for Safety Properties," in *Proc. of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York: ACM, 2016, pp. 314–325.

[29] G. E. Hagen and C. Tinelli, "Scaling Up the Formal Verification of Lustre Programs with SMT-Based Techniques," in *FMCAD 2008*.