On the Industrial Application of Critical Software Verification with VerCors

Marieke Huisman(✉) and Raúl E. Monti(✉)

University of Twente, Enschede, The Netherlands
{m.huisman,r.e.monti}@utwente.nl

Abstract. Although software verification is evolving fast in both theoretical and practical aspects, it still remains absent from the actual industrial production cycle. Case studies can help to encourage these integrations. We report on our experiences applying software verification in several projects with industry. In particular, we report on two projects on the verification of tunnel control software at Technolution, where we go from a high-level design to concrete code. These case studies show the power of combining model checking (using mCRL2) and deductive verification (using VerCors) as complementary approaches. We also report on a project with Thales, where we looked at antenna bearing control software, and specified this based on their requirements documents. For all cases, we report on lessons learned and on directions for future work to improve both our tool and the industrial methodology for ensuring software correctness. Notably, our second case study involves the modelling and verification of critical software by a team of engineers from Technolution. This case study is an ongoing project; we describe our experience on the team's learning curve for this experiment and present the preliminary conclusions on the case study.

1 Introduction

Over the last years, software has become omnipresent in our daily lives, in a very wide range of different applications, such as games, business software, and embedded control software [29]. With the omnipresence of software, also the potential consequences of software failures have increased, while at the same time we see that all software contains bugs [15,26]. Thus, there is an urgent need for tools and techniques that can be used to easily identify and prevent bugs.

In this paper, we focus in particular on critical embedded control software, and we investigate how formal verification techniques can be used to improve the reliability of such control software. For this kind of software, reliability and the absence of bugs is even more important than for other applications, as the consequences of software errors can be very serious. To investigate how we can improve the reliability of such software, we consider several case studies of industrial control software, for tunnel and antenna bearing control, respectively, and we discuss how we analysed those using techniques supported by the VerCors

T. Margaria and B. Steffen (Eds.): ISoLA 2020, LNCS 12478, pp. 273–292, 2020.
https://doi.org/10.1007/978-3-030-61467-6_18

verifier for the verification of concurrent software [5]. The verification work on these case studies has been done in close collaboration with our industrial partners, Technolution and Thales. These companies have an elaborated software development process, involving meticulous pair review, enormous test suits for unit and integration testing, and a careful architectural design. However, the preliminary results of our case studies show that despite this careful process, we are still able to identify software errors using the VerCors verification technology.

The VerCors verification technology is mainly based on deductive verification, i.e. a user annotates the source code with suitable pre- and postconditions, and then from the specifications and the source code, the VerCors verifier generates proof obligations (using the intermediate language Viper [20]), which are suitable for automated first-order reasoning, using e.g. Z3 [12]. In addition, VerCors also provides techniques to reason about behavioural models of (concurrent) programs [23,24]. Concretely, the user defines a model and then uses a model checker (currently mCRL2 [16]) to reason about the model, while deductive verification is used to show that the source code implementation is a refinement of the model.

The goal of the investigations described in this paper is three-fold. First of all, we would like to demonstrate that it is indeed possible to use formal verification techniques on industrial software, and that formal verification is able to find errors in the software. Second, we would like to investigate what is needed to make formal verification part of the main-stream development process, i.e. what is needed to ensure that formal verification techniques become usable for non-experts. The outcomes w.r.t. this goal can be divided into two parts. Part of it is technical and related to the question how can we make formal verification techniques easier to use. We will discuss some ideas in this direction in this paper (see Sect. 4.3). Part of the outcome is also psychological, i.e. it has to do with how to *convince* our industrial partners that it is worthwhile to invest in the use of formal verification techniques. This is a long-term process, which requires to build up a trust relation. We started this process by inviting some of our industrial contacts for the *VerCors advisory board*. However in the mean time we realised it takes much more than just talking about what we are able to do. We also need to demonstrate this on examples that are of interest to *them*. Moreover, it involves identifying bugs and problems in the code they have developed, and which cannot easily be identified with the quality-control cycle that is already in place, i.e., the use of formal verification techniques really need to have *additional value*. Third, we use these case studies to identify new directions of work for the VerCors verifier, which need to be addressed to make sure that the verification techniques can handle the size and complexity of modern industrial software.

Notice that the VerCors verifier has originally been developed for the verification of concurrent software. In the case studies described in this paper, concurrency is not relevant. However as one of the goals of the VerCors project is to make verification of concurrent software accessible to non-verification experts, we feel that the case studies presented in this paper are an important first step in this direction. Moreover, the software components that we investigated here

might be part of larger applications, where concurrency is used. Therefore, we believe that it is important that formal verification technology combines ease of use with support for a large range of language features, including concurrency.

Concretely, this paper describes three different case studies. The first case study (Sect. 3) describes work we did for Technolution on detecting a deadlock in tunnel control software. This case study has been described in detail elsewhere [25], but we include a high-level description here because it illustrates our point that incorporating formal verification in industrial software development is a long-term process. This first case study was done on existing software with a known bug. As we were able to identify the bug much faster, and in a systematic way, this led to the second case study described in Sect. 4, where we use formal verification techniques *in parallel* to the standard software development, and *in direct collaboration* with some Technolution employees. In both these case studies, we use our technique to specify a behavioural model, following Technolution's design documents, and to analyse these models using mCRL2, while using VerCors to show that their implementation adheres to the model. The third case study (Sect. 5) addresses the verification from a different perspective, as we encode a requirements document directly into pre- and postconditions, and then verify whether the code respects these pre- and postconditions.

For all case studies, we describe the lessons that we learned from the case study, and we also identify directions for future research. As mentioned above, future research is aimed at different directions: making verification technology easier to use for non-experts, and improving our verification technology to make it applicable to a larger range of applications.

Finally, this paper concludes with a discussion of related work in Sect. 6, where we compare our experiences with other experience reports discussing the use of formal verification in an industrial setting, and then concludes in Sect. 7 with the most important lessons that we learned from the case studies described in this paper.

2 Background

2.1 VerCors

VerCors [5] is a static verification tool that focuses on the verification of (concurrent) programs written in high level programming languages such as Java, OpenCL, OpenMP for C and its own prototypal verification language PVL. VerCors allows reasoning about data race freedom, memory safety, and functional properties of (possibly non-terminating) concurrent programs. Static verification in VerCors follows a design by contract approach: the user needs to specify the code with program-annotations in the form of pre- and post-conditions, following the style of JML [18]. VerCors then takes the program and its annotations and translates it into a problem for the intermediate language verifier Viper [20].

VerCors implements permission-based Separation Logic (PBSL) [1,6] to reason about different concurrency models, notably heterogeneous concurrency (e.g. Java programs) and homogeneous concurrency (e.g. GPU kernels). For this, the

program specification needs to explicitly express heap ownership in the form of permission annotations.

Figure 1 shows an example of program specification for verification with VerCors. The lines prepended with `//@` present VerCors specifications. Keyword `Perm` is used to indicate heap ownership. For method `inc` we require that the thread executing this method should have write permission to variable `x` in order to change its value. This is done at line 5, where value `1` means full (write) permission. The logic of VerCors verifies that the sum of all granted permissions to a same heap location never exceeds 1, which ensures absence of data races in a verified program. Then, we can ensure that the final value of `x` will actually be the expected one (line 7). At line 6 we only ensure to return half (`1\2`) of the permissions we obtained for `x`, i.e. a read permission. Retaining half of the permission to `x` will actually cause the verification to fail at line 17, since we will not be able to provide the write permission required by the second call to `inc`.

```
1   class Foo{
2
3     int x;
4
5     //@ requires Perm(x,1);
6     //@ ensures Perm(x,1\2);
7     //@ ensures x == \old(x)+1;
8     void inc(){
9       x = x + 1;
10    }
11
12    //@ requires Perm(x,1);
13    //@ ensures Perm(x,1);
14    //@ ensures x == \old(x)+2;
15    void incx2(){
16      inc();
17      inc();
18    }
19
20  }
```

Fig. 1. VerCors verification example.

Most interesting for the first two case studies in this work, VerCors features a model-based verification technique [23,24]. This uses a process algebra language to capture the behaviour of a software system by abstracting its access to shared memory by means of actions. The model specifies the acceptable executions of the system, and this can be verified to fulfil behavioural properties by an external model checker. Our technique then allows to connect blocks of code from the implementation to the actions in the abstract model. VerCors is then capable of verifying if the model is actually an abstraction of the code, or equivalently, if the implementation refines the model.

A central aspect of this technique is a formally proven deduction system which allows to link the abstract modelled behaviour of the software and its actual implementation. With this, we fill the usual gap between the model and the implementation: (safety) properties that are proven valid in the model are by refinement also true for the code.

The VerCors Advisory Board. The VerCors Advisory Board consists of members of the Thales, BetterBe, Technolution, Rosen, and PolderValley, companies. It is intended to be a place to exchange interests and experience with the industrial side of software production. The members of the Advisory Board were selected based on former interactions with them, who approached the VerCors team

presenting some interest in formal verification. It was also intended that they would represent an wide and diverse spectrum in the industrial application field for formal verification. The Advisory Board is intended to meet with the VerCors team twice a year. During the meetings we present our advances in support for formal verification and we get feedback, new ideas and case study proposals from the industrial side.

2.2 mCRL2

mCRL2 [8] is a state of the art tool set for model checking which offers an ACP-style process algebra as modelling language and allows to verify properties specified in modal μ-calculus with data. The tool set offers around sixty different tools to describe, manipulate and analyse the models.

Figure 2a shows an mCRL2 model of a producer/consumer protocol, where the producer generates messages of type `A` and `B`, queues them, and sends them to a consumer. Line 1 defines a new type `T` with constructors `A` and `B`. Line 3 defines several actions, parametrised by the type `T`. Lines 5 and 9 define process `P` and `C` respectively representing the producer and the consumer respectively. `P` is parametrised by an mCRL2 native type `List`, to queue up to two generated messages. In line 12, `init` indicates the initial setting of the system: `allow` defines the visible actions (opposite to usual hiding of actions) and `comm` renames the multi-action `snd|rcv` to `com` to indicate that `P` and `C` will communicate by both executing `snd` respectively `rcv` at the same time. Finally, line 16 defines that `P` and `C` should execute in parallel.

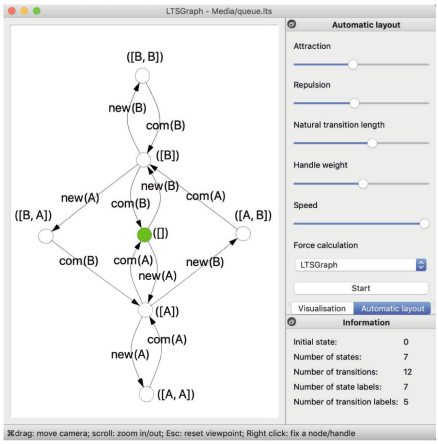
Figure 2b shows the labeled transition system of the producer/consumer model as presented by the `ltsgraph` tool from the mCRL2 tool chain. This tool allows to visually inspect the model described by the algebra. mCRL2 also allows to minimise the models against several bisimulation notions, to reduce the size of the generated state space.

3 Case Study 1: Tunnel Emergency Control Software

This section gives a high level overview of the first case study study [25] carried out between our group and Technolution [27], a Dutch software and hardware development company located in Gouda, with a recorded experience in developing safety-critical industrial software. The aim of this case study was to understand to what extent our formal verification tools could be applied in the context of Technolution's software development projects. The case study analysed an emergency control module from a traffic tunnel system, by (1) formalising and verifying the design by means of the mCRL2 model checker and (2) by demonstrating that the implementation is a refinement of the design by using our VerCors tool. The main goal of the case study was to explore if the mentioned combination of model checking and refinement by deductive verification could really be applied to a real-world software product, and to what extent this would be beneficial for the company involved, i.e. Technolution. We refer the curious readers to [25] and [23] for more details about this case study.

```
1   sort T = struct A | B;
2
3   act snd, rcv, new, com: T;
4
5   proc P(q:List(T)) =
6     (#q<2) -> sum x: T . new(x) . P(q<|x)
            +
7     (#q>0) -> snd(head(q)) . P(tail(q));
8
9   proc C =
10    sum x: T . rcv(x) . C;
11
12  init
13      allow({new, com},
14        comm({snd|rcv->com},
15          P([]) || C() ));
```

(a) mCRL2 model of prod/cons.



(b) `ltsgraph` visualisation of prod/cons.

Fig. 2. mCRL2 model of a producer/consumer and its state space visualisation in the `ltsgraph` tool.

3.1 Formal Verification of an Industrial Safety-Critical Traffic Tunnel Control System

Technolution provided us with the specification and implementation of an already deployed *emergency control software* from a *traffic tunnel*. This control system is in charge of ensuring that the right measures are taken when any (possible) calamity is detected in the tunnel. These measures could be, for instance, enabling fire extinguishers, turning on visual notification for people to know how to get to a safe place, turning the fans in the right direction to expel the smoke out of the tunnel, etc. This software is considered to be highly critical and for this reason the Dutch government imposes very high reliability demands on it, which are specified in a document of requirements that is over 500 pages in length [21].

As expected, the development process of the traffic tunnel control system was executed as an elaborate process of quality assurance/control, to satisfy the high demands on reliability imposed by the Dutch government. Significant time and energy has been spent on software design and specification, code inspection, peer reviewing, unit and integration testing, etc. In particular, we were given a precise design of the system, assisted by pseudocode definitions for each functionality (see Fig. 3 for an excerpt of [21]) and detailed state machines indicating the possible state changes after executing these functionalities (see [25] for more details). However, even though precise, the specification was informal and could not be formally verified.

Our goal was to demonstrate how formal methods could aid the verification of this control software by answering two main questions: (1) Is the informal specification of the tunnel control system consistent, meaning that it does not

```
1  Evacueer()
2    Evacueer de aangegeven verkeersbuis.
3    (Overgang 6)
4
5    Conditie: #substate = calamiteit_volledig
6    Acties: #substate := calamiteit_evacuatie
7            NaarEvacuatie()
```

Fig. 3. Pseudo-code design of the `Evacueer()` function from the National Tunnel Standard [21].

reach undesired states such as deadlocks or present any dangerous behaviour? and (2) does the actual implemented code follow the pseudocode specification?

Our approach to address the verification of these properties followed a combination of verification techniques (see Fig. 4 for a graphical representation of the approach). To answer question (1), we used model checking, by developing a formal model from the informal specification and verifying appropriate properties on this model. Particularly, we used the model checker mCRL2 for this task. To answer question (2), we used the refinement technique developed for VerCors [23,24], mentioned above, to formally prove that the code implementing the controller followed the behaviour described by the model.

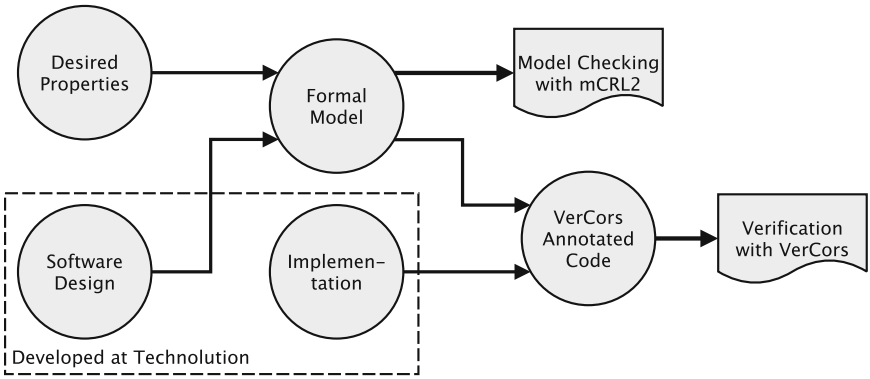


Fig. 4. Scheme of our model based deductive verification approach to code verification.

3.2 Model Checking the Design

To build the model for the control system, we followed the pseudocode descriptions of the intended functionalities and state machine diagrams describing the state jumps triggered by these functionalities. We also took into account interesting parts of the execution model, which slightly restricted the order of execution of the functionalities.

Although the initial model was too big to analyse, a series of abstraction iterations allowed us to arrive to a sufficiently expressive and concise model, small enough to verify. A couple dozen of properties elaborated by us after consulting the team of engineers from Technolution were verified on the model. During the verification we were able to identify a deadlock in the model, resulting from an intricate combination of times and events, which is very difficult to discover by methods traditionally implemented in industry. In fact, Technolution had discovered this error by chance, and had intensionally given us a faulty design to test our approach.

3.3 Code Verification by Refinement

Next, we used VerCors to prove deductively that the code developed by Technolution followed the formal specification of the tunnel, i.e. it refines the actual model. An important property of our refinement technique is that it preserves safety properties, such as deadlock freedom. The mCRL2 model is first translated to an intended equivalent model in the specification language of VerCors. The actions of the model can then logically be attached to sets of commands in the corresponding code by annotating blocks of code. The proof rules that VerCors implements allow to formally verify that the behaviour of the code follows the behaviour of the model [23,24]. Actually, we did not verify the Java implementation, but we translated it into equivalent code in our prototypal verification language (PVL), since at the moment this has a better support for the refinement verification. Although there is no formal proof of the equivalence between the original mCRL2 model and the VerCors model, the relation becomes quite evident from the proximity of the model structure. During the verification refinement we were able to conclude that the code followed the behaviour specified by the model.

3.4 Lessons Learned

During this first case study, in roughly 7 working days, with a single PhD student assigned to the project, we constructed a formal model of the informal specification of the tunnel control system, analysed it using mCRL2, and used VerCors to deductively prove that the implementation adheres to the specification. This resulted in the detection of undesired behaviour, preventing the control system from automatically starting the calamity procedure after an emergency has been detected. Even though Technolution was already aware of this behaviour, they only found it coincidentally, while we demonstrated that formal methods can indeed help to find such undesired behaviours in a structural manner, and within realistic time. It is our intention to continue investigating this technique in further case studies with Technolution and other companies. We believe that the success of this first step in the case study was highly influenced by the quality of the tunnel specification which, despite being informal, was well-structured, and therefore had the potential to be formalised within reasonable time.

Note that we did not find any discrepancies between the code and the specification by means of our refinement technique. We believe that this could be because we verified an implementation that was already deployed and thoroughly tested. This partly jeopardised our goal of demonstrating the usability of the technique, as the company only saw the extra effort, without any gain in the form of bugs found. Nevertheless they became aware of the potential of the technique and opened to new collaborations.

We also mentioned that we did not verify the actual Java implementation but an intended equivalent PVL implementation. On future case studies we would like to extend our tool support for the refinement technique to the rest of our front end languages, in order to verify the actual code. This is in fact a compulsory step for the systematic application of our technique in industry.

Finally, the experiences with this case study led to an idea to investigate if the pseudocode specification language can be formalised into a domain specific language (DSL), that can be automatically translated to mCRL2. We suspect that the specification of the pseudocode description in terms of this language would be a mechanical and straightforward activity. As a consequence, we expect that this language will greatly reduce the effort of adopting our technique for further verification at Technolution, in contrast to the steep learning curve necessary to use the mCRL2 modelling language.

4 Case Study 2: A New Tunnel Emergency Control Software

After the promising results of the first case study, we are currently working on a follow up project with Technolution. In contrast to our earlier collaboration, in this project we are working on the formal specification and verification *during* the development process of a new tunnel project. Moreover, a team of engineers from Technolution will be in charge of the formal modelling and model checking verification steps, while we advise them and work on the verification of code.

A main goal of this follow up project is to understand how much effort it will take the engineers at Technolution to learn formal verification and to what extent they can use this knowledge to verify their software. Furthermore, we want to obtain new insights and ideas on how to ease this process. Although not formally a goal of the project, we expect to be able to shape the characteristics of the DSL language as we mentioned as an insight from our previous case study with Technolution. Another main goal of this case study is to showcase our refinement verification with VerCors. We hope to increase the chances of finding bugs this time, by targeting the implementation in the earlier stages of development.

Figure 5 shows the steps we agreed upon before the start of the project: a team of two or three engineers from Technolution will use our help and expertise to learn the modelling principles surrounding mCRL2. They will try to model an interesting part of the tunnel specification by themselves, while only seeking for our help if really needed. Once the first version of the code for the selected components will become available to us, we will start the refinement verification against the mCRL2 model from the Technolution team using the VerCors tool.

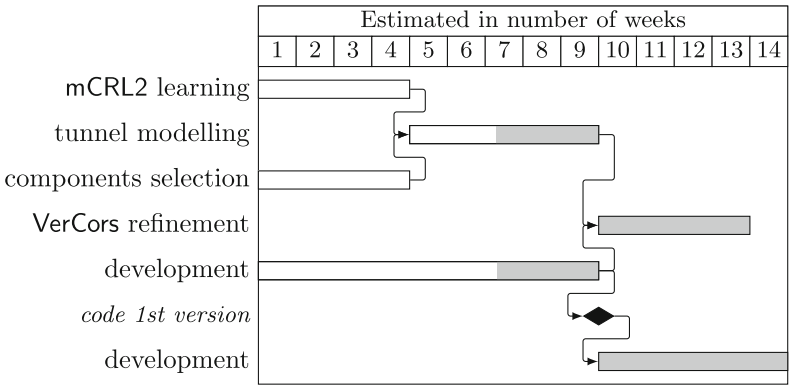


Fig. 5. Gantt chart of the steps of the project. White colour inside bars indicates the progress already made. (Color figure online)

4.1 Progress of the Project

We started the project by teaching mCRL2 modelling to the Technolution team. For this, we developed a series of slides presenting the main concepts of modelling, focusing on their applicability. We also used the mCRL2 web tutorial [19] for putting the learned concepts into practice. The other main standard source to learn mCRL2 is the book "Modeling and Analysis of Communicating Systems" [16]. Although the engineers working in the project initially dedicated considerable time to reading this book and understanding its concepts, they finally concluded that it was ineffective for their purpose. They claimed that the book assumes a thorough understanding of many formal mathematical concepts and notations, not common for outsiders in formal verification, and that the return on investment for understanding these concepts is not directly applicable to the modelling; they experienced quite some distance between the learned concepts and their application using the modelling language. They expect to have sources better aimed towards the application of the verification methodologies.

In between the teaching meetings, we had several extra modelling meetings, where we investigated the new tunnel specification and attempted to both figure out the differences with the former project and to choose interesting new aspects to model and verify. This turned to be beneficial in the sense that it also helped the engineers to analyse how the different concepts of modelling could match the different aspects of the system design.

When a good level of understanding of the modelling and verification concepts was reached, the Technolution team was ready to select the target modules to verify. The new tunnel differs from the former tunnel by being split into two consecutive sections. It consists of two parallel tubes both split in half by a joining lane (see Fig. 6 for clarification). This involves introducing new control units and protocols to coordinate each segment of the tubes. The engineering team decided to target these new modules for their verification.

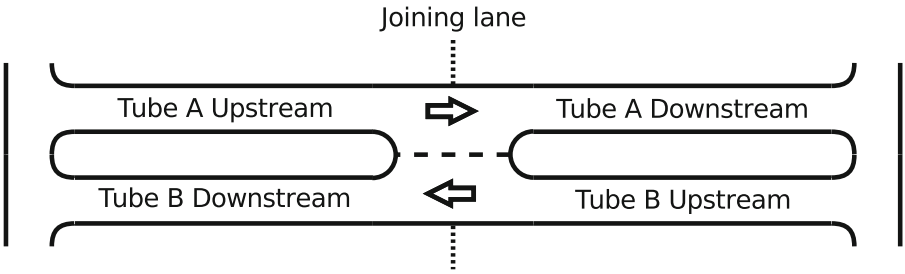


Fig. 6. The new tunnel topology.

In the current stage of the project, the Technolution team is attempting to model the selected modules. They are also defining the properties to verify and use these to decide upon the level of abstraction they should apply during the modelling. We expect the first version of the code to be available by August 2020. Then we will be able to start the verification by refinement with VerCors.

4.2 Lessons Learned Up to Now

Even though the project is still ongoing, we can already report on some important observations, mostly involving the learning curve of the modelling language.

We found that the official material, i.e. the web tutorial [19] and book [16], alone is not enough to learn the mCRL2 modelling language with an approach on application: the tutorial examples helped to get started but they focus mostly on covering each of the characteristics of the language, and miss a focus on real case studies concerning for instance of distributed systems and communication protocols. The book discusses the mathematical concepts behind the tool, and lacks the practical approach desired by the engineers. For the same reason, its language differs from the one used in practice and thus misguides the engineers when trying to apply what they learn.

We also realised that the modelling language is too low level and general for the usual purpose of the engineer. As an example, during some proof-of-concept designs that we developed with the Technolution team, we came up with many different solutions to the problem of communicating the state of a process to another process. Some of these solutions turned out to be more efficient than others, i.e. they generated a smaller state space for the same problem. Some were actually found to be wrong solutions. Finding and understanding which one is viable and more efficient could have been spared from us if this mechanism had been offered at a higher level of abstraction.

Furthermore, we experienced that the complexity of μ-calculus is highly unpractical for our purpose. Of course the property specification language is not necessary meant to be practical, but powerful. However, our experience was that explaining the initial concepts of the logic, without yet involving the fix-point modalities, already took a substantial amount of time and practice. The

fixpoint modalities were presented from an application point of view as a way to express finite/infinite recursions on the propositional formulas, or liveness and safety properties respectively. Nevertheless the concept remains to be unclear, and the typical working mode right now is to use pre-built templates and modify them when needed to express a new property.

On a positive side, we experienced that with the current set-up for the case, the attitude of the engineers involved is much more proactive in comparison to our previous general experiences with students at university level. This makes it possible to compensate for the experienced difficulties, such as not completely understanding the intricacies of the modelling language.

4.3 Sketches on Future Directions

As mentioned before, we found that the complexity of the μ-calculus formulas are one of the weakest points in adapting our formal verification techniques to an industrial setting. As a workaround, it was discussed several times during our meetings that it would be convenient to prepare more template formulas, to avoid reworking them each time a similar property has to be defined. An important property of such patterns is that they can be tool independent, since they can be translated into different logics for property specification. Furthermore, if defined simple and intuitive, such patterns can reduce the learning curve for the use of formal logics for specification.

As an example, in the context of emergency control systems, it is very common to require properties about recoverability: *It is always possible to recover from a calamity state.* That is, we should always be able to take our system back to a normal execution after a calamity has been resolved. A common way of specifying such property by a μ-calculus formula would look like:

$$\begin{array}{l}[\mathit{true}^*.\mathit{enterCalamity}]\ \nu.X(\\ \quad [\neg \mathit{exitCalamity}^*]\langle \mathit{exitCalamity}\rangle \mathtt{true} \wedge \\ \quad\quad [\mathit{true}^*.\mathit{enterCalamity}]X)\end{array} \tag{1}$$

which is neither direct to understand nor to remember. Ideally, we would like to use a declarative version such as

$$\mathtt{recover}(\mathit{enterCalamity}, \mathit{exitCalamity})$$

which is automatically translated into the formula above.

Of course, ideas for adapting the logics for property specification into more suitable languages for a systematic use in industry are not new. For example, the Bandera tool for Model Checking Java code [10], defines the Bandera temporal specification language (BSL) [11]. The language consists of a set of common specification patterns corresponding to common classes of temporal requirements such as response, precedence or absence. For instance, a *response* property requires that the occurrence of a designated state or event is followed by another designated state or event in the execution. BSL also defines pattern scopes which

restrict the fragments of execution where the property patterns are validated. For instance a *between* scope may define that a certain pattern is expected to be valid in between the occurrence of some pair of designated states or actions. For example, property (1) can be expressed in BSL by using a `leads to` pattern and a `globally` scope as follows:

$$exitCalamity \texttt{ leads to } enterCalamity \texttt{ globally}$$

The EVALUATOR3.5 tool from the CADP [14] project, defines a translation from BSL to μ-calculus [7].

Also SUGAR [4] presents an interesting approach to ease the specification of properties in temporal formalisms by defining syntactic sugar on top of CTL. As an example, a *next_event* operator, defined in terms of the weak until operator, refers to the next time an event may occur, in contrast to `AX` which refers to the next state. The language also defines a *within* operator to define properties to be expected between two states or actions, and syntactic sugar to introduce counters into the formulas. A similar approach is followed by the SALT [3] language, where several syntactic sugars are introduced inspired by relevant case studies on real-world specifications.

As an output of our case study, we plan to produce a list of interesting properties and patterns. We would like to analyse if some of the mentioned specification languages effectively covers this list, or if we would need some extension to them. An alternative direction would be defining a more specific language tailored to emergency control systems in general. In any case, we consider that it is important to still allow the use of the underlaying formalism for property specification (μ-calculus in the case of mCRL2), for a bigger expressive power whenever needed.

5 Case Study 3: Antenna Bearing Controller in C

Next we discuss a third case study, provided by the Thales group [28]. Thales is a world-scale software and hardware company specialised in defence technology, digital identity and security, and transportation among others.

The goal of this case study is to understand how VerCors can improve the quality of Thales' software development. A parallel goal is to investigate what needs to be improved in VerCors to be able to verify Thales' software projects in the future.

As part of our VerCors Advisory Board, a representative of Thales attended a presentation about the type of verification we target with our tool. As a result, they proposed us to verify a critical component of the control software for a radar. This component is in charge of validating the messages obtained from the sensors that measure the current position of the radar bearing.

5.1 Case Study Settings

The settings for this case study considerably differ from the two case studies discussed above. The code provided by Thales is completely written in C and instead of a design document a requirements document was provided. We investigated how can we use VerCors C support to validate the code against the requirements, and how to improve our support (if needed).

Structure of the Requirements. The bearing validation component can be divided into several smaller modules, which are each in charge of the validation of specific data included in the messages received from the sensors. For each of these modules, the requirement document defines a *general* requirement enumerating which *sub*-requirements have to be fulfilled in order to validate a specific property of the message assigned to this module (See Fig. 7 for clarification). The smaller requirements define conditions to be fulfilled by specific values in the message. These conditions are usually comparisons against predefined constants, or values in former messages, aiming to ensure the consistency of the newly arrived messages. The structure of messages and their fields are defined in a separate document and used in the conditions of the requirements.

Structure of the Code. On the implementation side, the messages from the sensors are represented by structures with field names that closely follow the ones defined in the requirements document. Other fields inside these same data structures allow the code to keep track of the validity verdict for the message. In particular, a `valid` field in the message structure indicates if the message is considered to be valid, i.e. if it is considered to fulfil all the validity conditions from the radar validation component. A few other fields are used as `flag`s to indicate the cause of invalidation of the message.

A single C-file implements the code to validate each requirement from the validation component. The code follows the structure of the requirements document. There are *general* functions which validate the *general* requirements by

```
REQ_MODULE_0:
  MODULE_0 declares that
       message M is valid only
       if SUB_REQ_0 to SUB_REQ_N
       are validated.
```

(a) A general requirement.

```
SUB_REQ_0:
  The validation component
       shall verify that the
       value of field F from the
       message M lays between
       constants C and C'.
...
SUB_REQ_N:
    The validation component
         shall verify that ...
```

(b) Sub-requirements for (a).

Fig. 7. Structure of the requirements for the validation component. (Concrete names from the case study have been anonymised for confidentiality reasons).

```
void REQ_MODULE_0(struct Message *msg){
  SUB_REQ_0(msg);
  SUB_REQ_N(msg);
  ...
}

void SUB_REQ_0(struct Message *msg){
// set msg->flag_0 to the result of validating SUB_REQ_0
// from the requirements document, and update msg->valid.
...
}
...
void SUB_REQ_N(struct Message *msg){
// set msg->flag_N to the result of validating SUB_REQ_N
// from the requirements document, and update msg->valid.
...
}
```

Fig. 8. Structure of implementation code for 7

calling *sub*-functions which validate the *sub*-requirements. The message data structure travels through the function calls as a parameter and the verdict of the validations are written into it using the `valid` and `flag`s fields (see Fig. 8 for clarification).

5.2 Approach to Verification with VerCors

To verify the implementation of the validation component, we translated each sub-requirement from the specification document to a postcondition for the function in charge of validating the corresponding sub-requirement. Figure 9 shows an example specification for `SUB_REQ_0` from Fig. 7b.

```
\*@
  ...
  ensures (msg->valid && msg->flag_sub_req_0) ==
            (C <= \old(msg->F) && \old(msg->F) <= C');
@*\
void sub_req_0(struct Message *msg){
...
}
```

Fig. 9. Example of contract for verification of requirements.

To validate a general requirement, such as the one in Fig. 7a, we specified its corresponding function with the following postcondition:

$$\texttt{ensures msg->valid} \wedge !\texttt{msg->flag}_0 \wedge \cdots \wedge !\texttt{msg->flag}_N \Leftrightarrow \texttt{cond}_0 \wedge \cdots \wedge \texttt{cond}_N;$$

where $\texttt{cond}_i$ encodes the conditions specified by sub-requirement i.

To showcase the approach we validated a single general requirement (and its sub-requirements) that was representative of the rest of them. We did not find any implementation errors during the validation. However, while inspecting the code, we found annotations in the form of comments indicating assumptions on the use of certain functions, such as an argument being positive, and we turned them into requirements of the alleged functions. During verification, we realised that one of these assumption was not met. Fortunately, the documented assumption was not considered in the implementation of the function and thus it did not evolve into an error in the code. However, the engineers from the Thales group considered this a useful discovery, because they claimed that most of the times developers would blindly follow the assumption and not implement a check inside the function in order to keep the code simple. Notice that these specifications in the form of comments are completely overlooked by the test suites which only involve testing the implementation code.

5.3 Lessons Learned

On the positive side, we discovered that even in relatively simple and well structured code, requirement inconsistencies can be overlooked by traditional testing techniques, while they are easily spotted by our tool.

On the other hand, a first limitation we found while working on this case study, is our poor support for the C language. For instance, support for structures and floating point numbers was missing. While we were able to add support for structures (in a limited fashion) quite easy, we decided in the meanwhile to abstract from floating point calculations by rounding every value to a close integer; the amount of work that it would take to add support for floats would have stalled the case study.

In this same direction, it was interesting to learn the discrepancies between the characteristics of the code used in industry and the one we imagine should be targeted while developing VerCors. In fact we are more used to focus on intricate algorithms with complex concurrency models, while this is usually the kind of code that industrial critical software tries to avoid. The risks of producing this kind of software, which is difficult to analyse, is too high. At the same time, we as developers of a verification tool, overlook other aspects such as the importance of having complete support of front end languages.

Another caveat we found in our tool is the amount of annotations needed to verify relatively simple properties. Actually, most of the annotations belong to permission-related constraints, inherent to VerCors' separation logic based verification. This problem was strengthened due the presence of large structures with many fields for which the precise access permissions have to be defined at each function contract.

5.4 Directions for Improving VerCors

A first direction of improvement we observe from this case study is to broaden our support for C. Some of this support can be added in a straightforward way, or by just applying some (substantial) engineering effort. Some other aspects, such as reasoning about floating point numbers will be a bigger challenge. We can find inspiration for this in other verification tools which already support these features, such as Why3 [30] or Frama-C [2].

We should also work on automating the annotations of contracts for verification. The amount of annotations the developer needs to manually introduce is sometimes overwhelming, and it makes the code difficult to read and maintain. During this case study the rate reached up to around 10 lines of specification code for each line of implementation code. A possible way to reduce this rate would be to extend our C frontend by defining predicates to encapsulate annotations, which are already present in our Java and PVL frontends. However, we believe that adding this support may not be sufficient by itself, since we have also found this problem in the other frontends before. A complementary solution would be to work on fully automated, or semi-automated user-assisted procedures to annotate permissions constraints. This solution will imply research work to develop and to figure out to what extent we can be apply these techniques.

6 Related Work

The lessons learned and the various discoveries during or case studies are of course not completely new to us, and we can find similar experiences in other works. In [22] the authors present several case studies on critical software, one of which is on deductive verification of a C program. They happened to find the same problem we have with floating point numbers which are pervasive in this kind of code. They also discus on the complexity of the verification languages and the struggles that non-expert personnel have to go through to come up with the right specifications to verify. They admit that even after 8 years of conducting case studies at the same company, they have not yet managed to introduce formal verification in a larger scale.

Dwyer et al. [13] hypothesise that a main cause of the difficulty of transitioning into industrial application of formal methods for verification is that practitioners are unfamiliar with specification processes and notations. They propose a pattern-based approach to property specification. We also discuss this, since it has actually been a concurrent request from the engineers involved in our case studies.

Larsen et al. [17] analyses 20 years of development of their model checking tool UPPAAL. They emphasise the importance of industrial case studies and collaborations, which they claim to have guided the construction of their tool. From the lessons learned in this process, they highlight that reaching an industrial impact of formal methods requires several iterations on collaboration with industrial partners and a coordinated evolution of both the tool and the

industrial methods. They also emphasise that, although formal, the specification languages should be engineer-friendly in order to increase the chances of impact.

Cok [9] states that engaging the software developers responsible for code development directly in the specification and verification process is a current challenge. In this sense, we believe our work is making a valuable contribution, since we are presenting an actual experience where we have managed to involve the design engineering team of Technolution (as described in Sect. 4) in the tasks of modelling, specifying and verifying their software product. In fact, it is not easy to find similar case studies; in the big majority of cases, the experts in formal verification are in charge of the experimentation.

7 Conclusion

A first conclusion from our work is that it takes time to build a relation with industry that may result in the eventual adaption of formal verification into an industrial environment and its application as a successful way of verifying critical software. We noticed that our VerCors Advisory Board has been very helpful to generate this relation; it allows us to communicate what we can do to industrial partners, and to get proposals for case studies from them.

In our experience, case studies have been very useful. They helped us to understand what are the usual problems that industry faces when developing critical software, and they helped us to improve our tool to make it suitable to solve such problems. We noticed that even some small success in a case study can open further collaborations and experiments with the industrial partners, since it showcases for them that there exists a real possibility of applying formal methods in their industrial processes. We find that in general, they are interested in finding techniques that can help them to improve software quality, as long as there is a good trade-off between invested time and results.

From the case studies analysed in this work, we have derived several points for improvement of our tool. These involve broadening the support for our frontend languages by, for instance, supporting floating point reasoning, as well as structures in C. We would also like to investigate patterns and DSL languages for our model based verification, in order to ease its adoption, which is now quite limited by the learning curve of the mCRL2 tool. Finally it is worth to investigate automating the permissions specifications, which is currently an error prone unpleasant job for the developer.

In the future we would like to continue with these case studies and spin-offs that may emerge from them. We are looking forward to the results of our ongoing tunnel control software verification with Technolution, and we will look for new case studies on which to showcase our planned upgrades to the C frontend.

References

1. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. Log. Methods Comput. Sci. **11**(1), 1–66 (2015)

2. Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 127–141. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_11
3. Bauer, A., Leucker, M., Streit, J.: SALT—structured assertion language for temporal logic. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 757–775. Springer, Heidelberg (2006). https://doi.org/10.1007/11901433_41
4. Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., Rodeh, Y.: The temporal logic sugar. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 363–367. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_33
5. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
6. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, 12–14 January 2005, pp. 259–270. ACM (2005)
7. The BSL to MU-calculus webpage. http://cadp.inria.fr/resources/evaluator/rafmc.html. Accessed June 2020
8. Bunte, O., et al.: The mCRL2 toolset for analysing concurrent systems. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 21–39. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_2
9. Cok, D.R.: Java automated deductive verification in practice: lessons from industrial proof-based projects. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 176–193. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_16
10. Corbett, J.C., et al.: Extracting finite-state models from Java source code. In: Ghezzi, C., Jazayeri, M., Wolf, A.L. (eds.) Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, 4–11 June 2000, pp. 439–448. ACM (2000)
11. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby: A language framework for expressing checkable properties of dynamic software. In: Havelund, K., Penix, J., Visser, W. (eds.) Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification, Stanford, CA, USA, 30 August – 1 September 2000. LNCS vol. 1885, pp. 205–223. Springer, Heidelberg (2000). https://doi.org/10.1007/10722468_13
12. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
13. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Ardis. M.A., Atlee, J.M. (eds.) Proceedings of the Second Workshop on Formal Methods in Software Practice, 4–5 March 1998, Clearwater Beach, Florida, USA, pp. 7–15. ACM (1998)
14. Fernandez, J.-C., Garavel, H., Kerbrat, A., Mounier, L., Mateescu, R., Sighireanu, M.: CADP - a protocol validation and verification toolbox. In: Alur, R., Henzinger, T.A. (eds.) Proceedings of the 8th International Conference Computer Aided Verification, CAV 1996. LNCS, New Brunswick, NJ, USA, 31 July – 3 August 1996, vol. 1102, pp. 437–440. Springer (1996). https://doi.org/10.1007/3-540-61474-5_97

15. Ganapathi, A., Patterson, D.A.: Crash data collection: a windows case study. In: Dependable Systems and Networks (DSN), pp. 280–285. IEEE Computer Society (2005)
16. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
17. Guldstrand Larsen, K., Lorber, F., Nielsen, B.: 20 years of *real* real time model validation. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 22–36. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_2
18. Leavens, G., Baker, A., Ruby, C.: JML: a notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188. Springer, Boston (1999). https://doi.org/10.1007/978-1-4615-5229-1_12
19. mCRL2–Tutorials. https://www.mcrl2.org/web/user_manual/tutorial/tutorial.html. Accessed May 2020
20. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Pretschner, A., Peled, D., Hutzelmann, T. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 50, pp. 104–125. IOS Press (2017)
21. Landelijke Tunnelstandaard (National Tunnel Standard). http://publicaties.minienm.nl/documenten/landelijke-tunnelstandaard. Accessed May 2020
22. Nyberg, M., Gurov, D., Lidström, C., Rasmusson, A., Westman, J.: Formal verification in automotive industry: enablers and obstacles. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 139–158. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_14
23. Oortwijn, W.: Deductive techniques for model-based concurrency verification. Ph.D. thesis, University of Twente, Netherlands (2019)
24. Oortwijn, W., Gurov, D., Huisman, M.: Practical abstractions for automated verification of shared-memory concurrency. In: Beyer, D., Zufferey, D. (eds.) Proceedings of the 21st International Conference Verification, Model Checking, and Abstract Interpretation, VMCAI 2020. LNCS, New Orleans, LA, USA, 16–21 January 2020, volume 11990, pp. 401–425. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39322-9_19
25. Oortwijn, W., Huisman, M.: Formal verification of an industrial safety-critical traffic tunnel control system. In: Ahrendt, W., Tapia Tarifa, S.L. (eds.) IFM 2019. LNCS, vol. 11918, pp. 418–436. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34968-4_23
26. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Where the bugs are. In: 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISTTA), pp. 86–96. ACM (2004)
27. The Technolution webpage. https://www.technolution.eu. Accessed May 2020
28. The Thales webpage. https://www.thalesgroup.com/en. Accessed May 2020
29. van Genuchten, M., Hatton, L.: Metrics with impact. IEEE Soft. **30**, 99–101 (2013)
30. Why3 Floating point axiomatisation. http://why3.lri.fr/stdlib/floating_point.html. Accessed June 2020