




On Deductive Verification of an Industrial Concurrent Software Component with VerCors

Raúl E. Monti^(✉), Robert Rubbens^(✉), and Marieke Huisman

Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{r.e.monti,r.b.rubbens,m.huisman}@utwente.nl

Abstract. This paper presents a case study where a concurrent module of a tunnel control system written in Java is verified for memory safety and data race freedom using VerCors, a software verification tool. This case study was carried out in close collaboration with our industrial partner Technolution, which is in charge of developing the tunnel control software. First, we describe the process of preparing the code for verification, and how we make use of the different capabilities of VerCors to successfully verify the module. The concurrent module has gone through a rigorous process of design, code reviewing and unit and integration testing. Despite this careful approach, VerCors found two memory related bugs. We describe these bugs, and show how VerCors could have found them during the development process. Second, we wanted to communicate back our results and verification process to the engineers of Technolution. We discuss how we prepared our presentation, and the explanation we settled on. Third, we present interesting feedback points from this presentation. We use this feedback to determine future work directions with the goal to improve our tool support, and to bridge the gap between formal methods and industry.

Keywords: Case study · Verification · Concurrency

1 Introduction

Software components for critical infrastructure should be kept to the highest standards of safety and correctness. Traditional methods for acquiring high safety standards include code reviewing and testing. These improve the reliability of software, but do not and cannot guarantee the absence of bugs. Software is also becoming more concurrent every year. The number of execution scenarios in concurrent software is even greater than in classical sequential software, due to interleaving and timing aspects. This makes code reviewing and testing even less effective. Specifically, there are too many interleavings of multiple threads, causing problematic interleavings to easily be missed during code reviewing and testing. Furthermore, concurrency related bugs such as data races and race conditions are intrinsically difficult to analyse with testing, since their effects are

platform dependent. For example, changing the OS of the system could cause previously passing tests to fail, due to different scheduling policies. It is also hard to test for specific interleavings. Hence, methods besides testing and code reviewing are needed to achieve the highest standards of safety and correctness in concurrent software.

To complement classical methods in the context of concurrent and critical infrastructure software, we believe formal methods must be considered. In particular, formal methods that can deal with the concurrent context must be used. In contrast to code review and testing, formal verification is exhaustive and can formally guarantee the absence of bugs in different stages of the software development cycle. Moreover, formal verification uses a standard semantics of the language in question, which guarantees consistent behaviour across platforms. Whenever platforms disagree, formal verification ensures that this difference is accounted for in the code. These properties of formal verification makes software more predictable, and hence safer.

Despite recent advances in software verification capabilities, the use of formal methods in industry is still limited. We think that case studies that show the successful application of formal methods will greatly contribute towards further adoption of formal methods in industry in two ways. First, it showcases the advances and capabilities of software verification tools to our industrial partners. Second, it generates valuable feedback, with which we can improve our tools and further adapt them to the software production cycle. This work discusses such a case study, where we verify a safety critical software for tunnel traffic control using our software verification tool VerCors.

VerCors is a deductive verifier, specialised in the verification of *concurrent* software [4]. It supports Java, C, OpenCL, and a custom input language called PVL. VerCors can prove several useful generic properties about programs, such as memory safety and absence of data races. Additionally, VerCors can prove functional correctness properties, such as “the sum of all integers in the array is computed”. To verify programs with VerCors, the programs must be annotated by the user, following a Design by Contract like approach. Annotations are pre- and post-conditions of methods, specifying permissions to access memory locations and functional properties about the program state. VerCors processes the program and the annotations, and verifies if the program adheres to the annotations by applying a deductive program logic optimised for reasoning about concurrent programs. VerCors has been applied to concurrent algorithms [19, 21, 22], and also to industrial code in earlier case studies [11, 18].

This paper is the result of a close collaboration with Technolution [25], a Dutch software and hardware development company located in Gouda, the Netherlands, with a recorded experience in developing safety-critical industrial software. It is also the next part in a series of papers to investigate the feasibility of applying formal methods within the design and production process of Technolution. For more information on the earlier parts, we refer the reader to [11, 18]. Finally, this paper is also an attempt to approach industrial partners to collaborate on the broader goal of making deductive verification, and specifi-

cally VerCors, available for industrial practitioners. This collaboration is carried out in the context of the *VerCors Industrial Advisory Board*, with the goal to learn how to introduce deductive verification into the production cycle of software, and to improve the tool and make it easier to use. Another important goal of the VerCors Industrial Advisory Board is to make industrial partners aware of the guarantees of formal verification, in contrast to the weaker guarantees of testing based quality assurance. This paper discusses our efforts to communicate our results and explain the verification process to the engineers at Technolution, as well as their feedback on our approach.

In particular, we discuss the verification of software for a tunnel on a road called the Blankenburgverbinding [3]. This tunnel and the hard- and software supporting it will be responsible for funnelling thousands of cars every day. The control software of this tunnel monitors and controls almost every aspect of the tunnel, in both normal and calamity situations. Thus, in order to give some safety guarantees to its daily users, it is highly important that the software conforms to the requirements of the national regulations and to the specifications provided by the engineers who design and develop it. To demonstrate how formal methods can help here, we applied our software verification tool VerCors to a submodule of the control software of the tunnel, in particular to analyse concurrency related issues such as data races and memory safety.

To develop the tunnel software, Technolution followed an iterative V-model approach. The customer handed in the requirements in form of a BSTTI document [16] and LTS specifications. These were used to derive actual software requirements via system decomposition and design. In addition to the custom validation flow of the V-model, the customer also imposed requirements on the development process. These included, but were not limited to, units being inspected to verify that they implement their requirements via Fagan inspection performed by a developer not involved in creation and review of the code, requirements-based testing at software module level (i.e. higher integration level than units) using the MC/DC coverage approach, and UI-design based testing at a software chain level (i.e. integration of multiple systems) with a process flow approach.

This rigorous approach to software development resulted in them spotting some unexpected behaviour in their tunnel software, where a certain condition over the state snapshot of a component was evaluated differently at two spots throughout which the snapshot should remain unchanged. Nevertheless, later they could not reproduce this behaviour and, by the time we were given the code to analyse, they had not been able to spot a bug that might explain this behaviour. The code we received was already in testing phase. As can be seen in this paper, we discovered concurrency related bugs in this code, which we think were likely the cause of the unexpected behaviour. We show that VerCors can effectively catch this kind of bugs, in production phase, by using simple code annotations in the form of methods pre and post-conditions specifying the memory access pattern of such methods.

The goal of this particular study is three-fold. First we want to investigate how much we can support the verification of industrial Java software with VerCors. Second, we want to focus this time on a *concurrent* piece of software and on concurrency issues such as data races, for which our tool is specialised. Notice that to exploit modern architectures, modern software is often concurrent, and not many deductive verification tools can deal with this. Third, we want to investigate how our verification procedure can be improved for industrial adoption. For this, we are particularly interested in the feedback from the Technolution team with respect to the verification procedure we followed.

Contributions. In this paper we discuss the following:

- Details of the tunnel verification case study, such as the analysis workflow and the problems we discovered.
- The process of communicating our results to Technolution.
- The feedback from Technolution and its engineers regarding our analysis and our presentation of it.
- Future plans for VerCors and the analysis of concurrent industrial software.

Outline. Section 2 presents the background for this research, i.e., we describe the tunnel software and architecture. We also explain how to use VerCors for concurrent software verification. In Sect. 3 we discuss our process of verification of the concurrent data manager module, we explain the bugs we spotted, and we show how applying VerCors would have avoided these bugs. Section 4 describes the experience of explaining our procedure and reporting our results to the engineers at Technolution, and their feedback. Section 5 describes our own reflection and future directions towards our goal of improving VerCors for industrial application. Additionally, we mention some broader goals for the formal methods community. Finally, Sect. 6 summarises and concludes.

2 Background

In this section we describe the two technologies relevant to this paper. First, we describe the system architecture of the tunnel software. Then, we describe VerCors, the tool used for verification of Java code.

2.1 Tunnel System Architecture

In the Netherlands, the architecture of software for tunnels is regulated by the Basic Specification of Technical Installations for Tunnels (BSTTI¹). The BSTTI [16] specifies that the architecture for tunnel software is strictly hierarchical. The system is summarised in Fig. 1.

At the top layer of this hierarchy are the human operators that operate the system. These operators give commands to the system, and inspect the values

¹ *In Dutch:* BasisSpecificatie Tunnel-Technische Installatie.

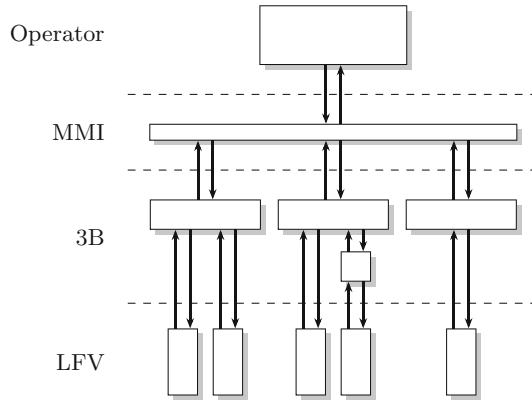


Fig. 1. Informal overview of the architecture specified by the BSTTI.

of various sensors in the system, using the Human-Machine Interface (MMI²) layer. The MMI processes these commands and forwards these to components of the Control, Instruct, Guard (3B³) layer. The 3B layer and its components are responsible for the high-level control of the physical subsystems of the tunnel. Examples of 3B components are water drainage, lighting, and electricity systems. As 3B components can be responsible for controlling entire subsystems, they also have a degree of autonomy. The individual 3B elements communicate with components in the Logical Function Fulfiler (LFV⁴) layer. Components in the LFV layer abstract the communication with the sensors and actuators of the tunnel to check and control them. Examples of these sensors and actuators are the smoke sensors and fans, the lights, or the entrance barriers. They can be located at various places in the tunnel and are connected to their LFV counterparts over various kinds of network connections following different protocols.

According to the BSTTI [16], the system must follow these general principles:

- Control must flow from the human operator level to the LFV level.
- Communication must take place along the parent-child hierarchy outlined in Fig. 1. Specifically, sideways communication between neighbouring 3B/LFV components, or between 3B components and LFV components that have no parent-child relation, must not take place.

These principles were prescribed because they make actions taken by the system traceable. If the physical system takes a certain action, the strict hierarchy allows tracing back to which component or decision caused the action. Note that it might not always be a human who caused the action. Since 3B components can have a degree of autonomy, it is possible that an autonomous action causes a physical action to take place.

² Man-Machine Interface.

³ Besturing, Bediening, Bewaking.

⁴ Logische Functie Vervuller.

2.2 VerCors

VerCors [4] is a deductive verifier for concurrent programs. It supports Java, C, OpenCL, and the Prototypal Verification Language (PVL). To make verification of concurrent programs tractable, VerCors uses permission-based separation logic [10]. This version of separation logic uses permissions to decide whether threads can read or write shared data. A complete permission allows a thread to write, but does not obligate it to. A fraction of a permission only allows a thread to read. This ensures that at any given moment, there can only be one writer, or many readers, but not both at the same time. We refer to a write permission as “**write**” while we refer to a fractional read permission as “**read**”. Fractions of permissions can be distributed between threads. When a thread no longer needs a permission, fractions of a permission can be recombined into a complete permission. In particular, permissions are never duplicated, ensuring that there can only be at most one write permission.

Alternatively, permission fractions can also be represented as concrete numbers. In this case, a “**write**” permissions corresponds to a fraction of **1**. A “**read**” permission corresponds to any fraction between **0** and **1**, such as $\frac{1}{2}$ or $\frac{3}{4}$. Operationally, having a permission fraction bigger than **0** and smaller than **1** allows a thread to read a memory location, but not write to it. In this work, we mostly use the terms **read** and **write**, but in more complicated contracts, sometimes the numerical form is required. For a more thorough introduction to permission-based separation logic, we refer the reader to “Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic” by Clément Hurlin [12].

To prepare methods for analysis with VerCors, they need to be annotated with pre- and post-conditions. Pre- and post-conditions are sometimes also referred to as the contract of a method. Pre-conditions describe the permissions a method needs from the caller, as well as any functional properties that must hold when the method is called. Post-conditions indicate the permissions that a method returns to the caller, as well as any functional properties that may be assumed. For example, when an object is constructed, permissions to access the fields of the object are returned by the constructor of the object. As a functional property, the constructor can guarantee that all fields are zero-initialized.

In Listing 1 an example is shown of such permission annotations in Java. The annotations are placed in comments and are highlighted. In the annotations, the expression after the **requires** keyword specifies the pre-condition of the method. In this case, permission is required for the **total** field of class **C**. The **ensures** keyword indicates the postcondition. In this case, the permissions from the pre-condition are returned to the caller of the **add** method, as well as the functional property that **total** is incremented by **x**.

When verifying an individual method, methods that are called are not re-verified, instead their pre-condition is established and their post-condition assumed. This is called modular verification, and it ensures that the correctness of a method does not depend on the implementation of other methods. This means that it is easy to update method implementations without break-

```
1 class C {
2     int total;
3
4     //@ requires Perm(total, write);
5     //@ ensures Perm(total, write) ** total == \old(total) + x;
6     void add(int x) {
7         total += x;
8     }
9 }
```

Listing 1. Example usage of permissions in Java. Write permission is required in the pre-condition of method `add`. Because of this, `add` can only be called when the caller has write permission for `total`. Then, the field `total` is incremented by the value `x`. Finally, the `write` permissions are returned via the post-condition, as well as the functional property that `total` is incremented by `x`.

ing the correctness of other methods. If necessary, implementations of methods can also temporarily be omitted, which can be useful for describing and enforcing interfaces between independent teams, or when implementations are not yet available.

3 Verification of the Concurrent Data Manager Module

When discussing our plans for collaboration with Technolution, the engineers suggested as a case study their new control software for the Baak tunnel. For this tunnel, they have developed a system that is responsible for controlling and reporting on all critical and non-critical components, such as escape doors, fire-prevention measures, water drainage systems, lighting, ventilation, etcetera. In order to reduce the time spent in spotting a concurrent candidate module to analyse, we agreed to meet a first time with an engineer, experienced with the tunnel software, who could guide us through it.

In this first meeting the engineers from Technolution not only suggested a set of modules to verify, but also pointed out a problem that they would like us to consider, since it was most likely a concurrency issue. The system they had built at that point was functional, behaved properly, and passed all tests. However, sometimes, according to their data logs, certain status data would unexpectedly change during execution of the system. These unexpected changes were never problematic in realistic scenarios, so therefore they considered it benign. However, it was still unexpected, and they would like to understand why this happens.

As a first step, we decided to go through the code base and try to understand the structure. We used a couple of meetings with the Technolution team to get some guidance around the code. Also, several times we asked for further code to inspect, such as supporting libraries of the system.

We found that the components of a tunnel can be quite diverse, and to cope with that diversity, several layers of abstraction and interfacing code had been

built into the tunnel software, which made it non-trivial to understand for us. Nevertheless this was not a problem for our verification approach, as it is modular at the level of methods, and annotating the code was straightforward. We ran VerCors on the fly, while annotating the code, and even mocked some library calls, by means of abstract methods and ghost code. We did have problems with VerCors lacking support for some frequently used Java features, such as inheritance and generics. Once we decided on the module to verify, the effort to abstract from unsupported Java features and annotate the code was very little; the annotations were trivial to us, and it took just an afternoon to reach the conclusions. Moreover, due to the simplicity of the specification, VerCors was able to verify the code in just a couple of seconds.

3.1 Event Loop Analysis

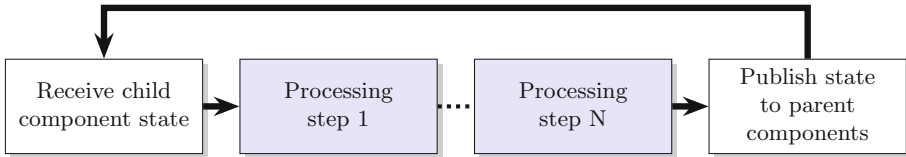


Fig. 2. The 3B function processing event loop.

We inspected the event loop of the main module, because most of the concurrent behaviour happens there. This involved peeling off the abstraction layers of the event loop framework, which is responsible for receiving and dispatching messages and executing each step of the processing loop of 3B components. This process repeats until the main module is shut down. An illustration of the typical processing loop of a 3B function can be found in Fig. 2. A processing loop starts by obtaining the state of all the child components for this 3B function (first rectangle in the figure). This state is then used to take control decisions along several processing steps inside the loop (shaded rectangles in the figure). It is here that the Technolution engineers were suspecting that something is wrong. In particular, during this control decision period, this state *must not change*. The suspicion was that somehow the state *was being changed*.

Continuing the explanation of Fig. 2, at the end of the loop our own state is prepared and made available to the upper 3B functions in the hierarchy (see Fig. 1 for clarification). In general, 3B functions and LFV components work asynchronously. The communication of the state between LFVs and 3B functions is managed by specialised data managers which need to synchronise the state of these asynchronous elements at the start and end of the 3B function processing loop. In a generic *data manager* of the event loop framework, we were able to spot two problems through manual inspection of the source code.

Problem 1: Forbidden Data Sharing. The first problem was related to aliasing between references to data structures representing the status of the child components of a 3B function. To better understand this, let us look at

Fig. 3: each 3B function uses two copies of the data structure representing the status of its child LFVs and 3B functions. One of these copies, the “internal” copy, represents the internal knowledge that a 3B function has of its children. It is used by the 3B function processor to make control decisions and should remain unchanged during the time of a processing loop iteration, this is, along the shaded steps in Fig. 2. On the other hand, the “dynamic” copy is updated each time a status update message from a child component is received. These messages arrive at any point during a processing loop iteration and the updates are asynchronously applied. At the end of an event iteration, the dynamic copy is used by the *data manager* to update the internal copy.

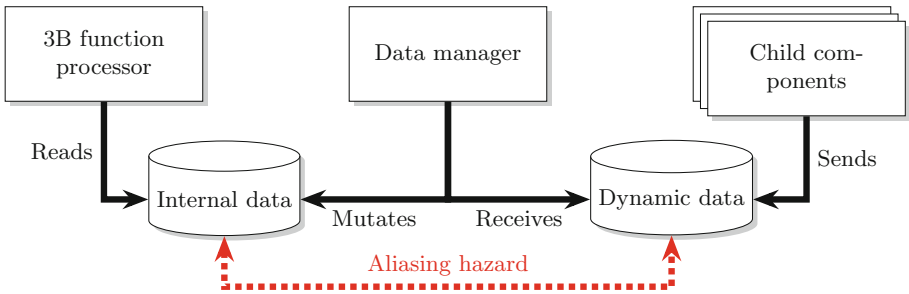


Fig. 3. Shared data snapshots.

It turns out that the data manager accidentally aliased both the internal and the dynamic copies. This is a simple but common mistake, and in line with the expectations of the Technolution engineers. The internal copy would then change midway through a processing loop iteration whenever the dynamic copy would receive an update.

As a verification exercise, we decided to annotate the data manager module in order to demonstrate how we could have avoided this mistake by using VerCors. Actually, we simplified the module for the sake of focusing on the interesting aspects, and to avoid incompatibilities with our current support of the Java language. We further discuss this in Sect. 5. As we expected, it turned out to be straightforward to rule out this mistake. List. 2 shows a simplification of the actual aliasing bug and the annotations we used. Lines 9 and 10 are the preconditions specifying that we need permissions to *write* on `internal` and its field `value` while we need to be able to read `dynamic` and its field `value`. Our postconditions, at lines 11 and 12, specify that these permissions should also be returned to the caller. We specify permissions to each of them separately, using the separation conjunction (`**`), since they should correspond to two different data structures.

At line 16, `dynamic` is assigned to `internal`. Therefore, `internal.value` and `dynamic.value` represent the same memory location. At this point VerCors complains about our postcondition. List. 3 shows the VerCors output for this

```

1  class Data{
2    int value;
3  }
4
5  class Manager{
6    Data internal;
7    Data dynamic;
8
9    /*@ requires Perm(internal, write) ** Perm(dynamic, read);
10   /*@ requires Perm(internal.value, write) ** Perm(dynamic.value, write);
11   /*@ ensures Perm(internal, write) ** Perm(dynamic, read);
12   /*@ ensures Perm(internal.value, write) ** Perm(dynamic.value, write);
13   void sync() {
14     internal.value = dynamic.value;
15     ...
16     internal = dynamic;
17   }
18 }

```

Listing 2. Ruling out aliasing with VerCors

faulty case. The error message “PostConditionFailed:InsufficientPermission” at line 11 indicates that we are missing permissions to access a memory location. The brackets and dashes at lines 5 and 7 indicate where the problem lies: we do not possess the amount of permission we want to ensure in the second half of line 12 of our code. In fact, we already gave up all the permission we had on this memory location through its alias, in the first half of the same postcondition line. After VerCors indicates something is wrong, the user must find out why this is the case and spot the undesired aliasing.

After analysing this bug with the engineers involved in our case study, we concluded that this aliasing would likely have been the reason of the unexpected behaviour they had detected. It apparently had not affected the overall behaviour of the system, but the reason why such a bug did not extend into a serious fault was not clear. The enormous amount of execution scenarios due to interleaving and timing aspects also makes it difficult to reproduce the immediate effects of this bug. The bug should be fixed since we cannot exclude that it may, under certain circumstances, trigger a major fault in the tunnel control system.

Problem 2: Internal Data Leakage. A second bug was spotted while annotating this module for verification with VerCors. Another method of the module was leaking a reference to a private field of the class. List. 4 illustrates this case. This is not harmful on its own, but it is usually considered bad practice. This may unintentionally allow a user of this class to concurrently access the field without following its synchronisation regime, which may result in a data race. Permission annotations in VerCors will not disallow acquiring the reference, but the annotations will ensure that there is no way to access any fields of this reference without holding the necessary permissions. This restriction rules out any data races.

```

1 Errors! (1)
2 == Manager.java ==
3   /*@ requires Perm(internal.value, write) ** Perm(dynamic.value, write);
4   /*@ ensures Perm(internal, write) ** Perm(dynamic, read);
5   [-----]
6   /*@ ensures Perm(internal.value, write) ** Perm(dynamic.value, read);
7   [-----]
8   void sync() {
9     internal.value = dynamic.value;
10  [-----]
11  PostConditionFailed:InsufficientPermission
12  =====
13  == Manager.java ==
14  /*@ requires Perm(internal.value, write) ** Perm(dynamic.value, write);
15  /*@ ensures Perm(internal, write) ** Perm(dynamic, read);
16  [-----]
17  /*@ ensures Perm(internal.value, write) ** Perm(dynamic.value, read);
18  [-----]
19  void sync() {
20    internal.value = dynamic.value;
21  [-----]
22  caused by
23  =====
24  The final verdict is Fail

```

Listing 3. VerCors output for alias spotting

```

1 class Manager{
2   private Data internal; // protected_by(this)
3
4   synchronized Data get_internal() {
5     return internal;
6   }
7 }

```

Listing 4. Reference to private data leakage

3.2 Discussion on the Discovered Bugs

The two bugs we found are typically overlooked by testing and manual inspection: their effects are triggered by very specific combinations of timings and interleaving that are too complicated to cover by test cases. A manual inspection may mistakenly consider these usages to be safe, or overlook them while searching for functional behaviour bugs instead of memory safety.

The effects of these bugs in a deployed system might be dangerous as it is hard to claim they do not cause incorrect behaviour. To prove that they do not cause incorrect behaviour, one would have to consider all possible interleavings of the processes of the system. The difficulty of this inspection increases exponentially when the number of concurrent processes and timing factors increases. In other words, proving that the system is not affected by the bugs by manual inspection is untractable.

Fortunately, the memory bugs we found are detectable with VerCors, by annotating methods in a straightforward manner with the permissions they require/ensure for the fields that they read/modify. An example of this can

be found in the the pre- and post-conditions of List. 2. These annotations are made compulsory by the tool, meaning that if they are not there the tool will terminate with an error. If verification succeeds, then VerCors guarantees that there is no data race in the code.

4 Results Presentation

In this section we describe our preparation process and presentation of the results to the bigger team of engineers at Technolution, which included a broader group than just those involved in the case study. We also describe our impressions of the final presentation and discuss the most interesting feedback points from the audience.

4.1 Presentation Design Process

After the case study was analysed by hand and translated to VerCors, we wanted to present our findings to a bigger audience of engineers at Technolution. However, we had experienced in former meetings with the Technolution team that we had not been able to effectively explain what VerCors checks, and how to annotate programs for VerCors. Therefore, we agreed to be careful and first present the results only to the Technolution team involved in the case study.

It turned out the initial presentation had several shortcomings, which we discuss here, because we think they provide important general insights.

First, the initial presentation tried to explain several useful verification concepts. For example, it discussed the benefits of fractional permissions, compared to non-splittable ownership tickets. It also discussed the difference between annotating only for memory safety, and annotating for functional properties as well. This was done to show how we use VerCors. However, without a formal methods background, the explanation of these tradeoffs is hard to follow. Additionally, most of these concepts are not necessary in order to explain the basis of our approach to verification of memory safety. The solution was to *only* focus on this basis, which is: annotating code with permissions.

Second, the examples used in the initial presentation combined orthogonal concepts to make the examples non-trivial. While engaging for experts, we found out that this is bad for teaching how an approach works. This is especially relevant in the context of a presentation, where the audience needs to understand the slides quickly and explanations need to be short. The solution is to make the examples more targeted. Even when discussing the fundamental basis of our verification approach, each example should only highlight the one relevant aspect of it. For instance, our final presentation contained a code example that had exactly *one* error. The code example on the next slide added exactly *one* annotation, consisting of only one permission, to resolve the error. Additionally, examples from the initial presentation were split up such that each sub-example fit on one screen with a large font. With each example presented in isolation and using as few lines of code as possible, they were also easier to understand.

Third, some of the examples in the initial presentation contained concerns unrelated to verifying concurrency, such as division by zero and rounding. The solution was to ensure that no concerns appear in the example that are unrelated to concurrency or memory safety, since we experienced that this would deviate the attention of the audience to topics we are not interested in discussing.

For the particular case of Technolution, we found out that it was useful to compare our approach with the Rust language, which was familiar to them [17]. This was actually suggested by the Technolution side during our presentation preparation meetings. We also took care with how we phrased certain concepts. Since there might be a difference between what we regard as a permission and what an engineer regards as a permission, we had to ensure this was not a problem from the beginning.

Finally, we made sure to clarify that we do not execute the code, but logically analyse it. For this, we compared it to making a pen and paper proof. This is needed to step away from the usual runtime verification approach of unit and integration testing.

To summarise, we learned that a “good” formal methods presentation to a non-formal audience should have at least the following properties:

- Introduce only key concepts of the formalism in question that are actually needed to understanding the basic idea of the formalism.
- Examples should present only one new concept at a time. Combining orthogonal concepts into one example is not helpful.
- Examples must be short, to ensure they fit on one slide, can be interpreted quickly by the audience, and also be explained quickly by the presenters.
- Examples must not contain unrelated concerns. The domain of the audience might introduce concerns the presenters are not aware of. Therefore, experts in the domain of the audience should be asked beforehand.
- Determine concepts the audience is already familiar with, and draw parallels between those and the concepts in the presentation. However: take care that the audience does not take this analogy too far, to avoid misunderstanding. Avoiding reuse of terms from the audience domain can help.

Additionally, our overall approach consisted of several iterations of refining the presentation using feedback of the smaller group. We think this helped us to narrow down what the Technolution engineers would most likely be interested in, what information would benefit them, and what information could be safely discarded from the presentation. It also helped us to agree on the proper language to transfer this knowledge. The drawback of this approach is that it is time consuming, because the presentation had to be presented twice before the final presentation. Furthermore, the feedback had to be documented by Technolution, and also had to be processed by us. Nevertheless, we think that this process will be quicker next time, due to reusing lessons learned in this case study.

4.2 Presentation Conclusions

During and after the final presentation, several questions were asked and comments were made, both by the presenters and the audience. We have collected the most insightful and applicable ones below.

Testing Exceeds Verification in Short term Gains. During the presentation it was mentioned that some teams do not even use testing to its fullest. We agree with the observation that it is more beneficial for most projects to first test 80% of their code base, before starting to consider formal verification. Additionally, there are formal methods to enhance and/or multiply the testing effort. Some examples are generation of test cases, mutation testing and QuickCheck-like testing [6, 13, 26].

Annotation & Specification Culture. Speaking from the experience of the Technolution engineers, it is impossible to ask engineers to write the annotations needed to use VerCors, or formal verification tools in general. Engineers do not even write comments that you would like to have in the general case. Therefore, there is a big gap between the annotations engineers are willing to write, and what verification tools require. This can be improved upon by the formal verification tools, by having smarter tools, generating some annotations, having design shorthands, and setting effective defaults. But, the difference is so big, that to adopt formal verification tools widely, there also needs to be a culture shift about commenting and annotating code.

Similarities to Rust. Most engineers have heard of or worked with Rust. Verification tools can exploit this to lower the barrier for using verification tools, and make them more easily understandable and adoptable.

Optimise for the Common Case. Related to Rust, an approach that the engineers thought could be useful to verification tools is the “optimise for the common case” approach. In this approach, tools optimise for the use case that is most common in practice. For exceptional or unsafe use cases, alternative syntaxes and escape hatches are added. Usually, these alternative syntaxes are also more verbose, making non-standard code also visually distinct. Furthermore, the general case should be safe and hard to get wrong. If applied successfully, we expect that the usage of this approach could reduce the amount of annotation needed for verification, improve the readability and decrease the unwillingness of the programmer to follow the verification path.

Library Calls. Some engineers expressed concerns about not having contracts for libraries that a team uses. It is true that if a library has no contracts, someone needs to write them. However, it is not a problem that the source of the library is not available due to modular verification (explained in Sect. 2.2). Additionally,

there are ways to reduce friction caused by these missing contracts. For example, it is possible to create a central database of library contracts. For cases where the specification for a library is not in the database, the specification language could offer syntax for defining contracts for a library separately.

Why not Use Automatic Static Analysis Tools Instead? An engineer pointed out that he had some experience with various static and automatic analysis tools. They raised the valid question of why code should be annotated for VerCors, when there are tools that can spot memory bugs without annotations. Some examples of such tools are Klocwork [14], FindBugs [9], Coverity [8] and SonarQube [24]. Our answer to this question is that these kind of static analysis tools are not verification tools. Instead, they do a “best effort” analysis to find patterns that *may* relate to bugs. This means such tools are not exhaustive, and can report false positives and warnings which have to be manually inspected. Verification tools, in contrast, give strong formal guarantees on the validity of the queried property over the analysed system. In other words, given a specification that faithfully models the desired behaviour, false positives are rare.

Additionally, code analysis tools often can be used in tandem. Therefore, we think it can be beneficial for teams to use tools with different purposes at different stages of development, or even simultaneously. This way the quality of the final product can be maximised.

5 Future Research

Future research for the VerCors team will go in several directions.

One direction of research is to reduce the number of annotations required before VerCors can be used. Currently, if there are no annotations in the code, VerCors cannot make any assumptions about the code. However, for industrial code, simple assumptions are often correct. For example, two fields on one object usually do not contain the same reference. We expect that it will cost less effort to annotate for the exceptions of the previous rule, than to annotate wherever it applies. Additionally, research is already being done to see if some of the required annotations can be generated instead.

Another direction of research is to improve the support of VerCors for Java features such as inheritance and generics. Currently a manual translation to a subset of Java is necessary to verify industrial Java code with VerCors, however efforts are being made to improve the support [20, 23].

Finally, we think future research of the formal methods community as a whole should be about designing simpler specification languages which are closer to the concepts and models of software development teams. We found that the semantics of our specification terminology is distant from the intuition of the engineers. The understandability of specification languages is a common problem in the formal verification community. For example, consider Linear or Branching time logics [15], μ -calculus [5] and other algebras commonly used to specify designs in model checking. The learning curve of these languages is too steep

and they become impractical for the daily use of a software engineer. Therefore the next step has to be one of effective reduction: reducing the expressive power of these languages down to a level where they can be easily understood, while retaining enough power to check properties that are of interest to the engineers. Progress is already being made on this with languages such as SALT [1] and Sugar [2], and all the work surrounding the Bandera Specification Language (BSL) [7].

6 Conclusion

We have applied VerCors to a submodule of tunnel control software. This software contained a known benign but unexpected runtime behaviour, which lacked an explanation. Through manual analysis, a bug and a weakness were found, one of which is a possible explanation of the unexpected runtime behaviour. We have communicated our results to the Technolution team and the Technolution engineers through a carefully prepared presentation that underwent multiple feedback rounds from the Technolution team. This allowed us to focus on the information that is most useful to the engineers, and leave out the information that is not directly necessary.

The results of this presentation are suggestions and insights from the engineers of Technolution. For example, it was suggested that there are similarities between Rust and our annotations which VerCors can exploit. It was also suggested that there should be support for easily modelling contracts of software libraries. Another observation we have made is that there is a large gap between the annotations that must be written to apply VerCors, and the maximum amount of annotations engineers are typically willing to write. There was also an observation from an engineer that, in the short term, proper testing practices yield more benefits than formal verification does in the short term.

Finally, we have discussed future directions for our work, such as implementing assumptions about the typical structure of industrial Java code in VerCors, as well as adding more extensive support for the Java language in VerCors.

Acknowledgements. We thank Technolution for the opportunity to analyse their code, their guidance and support.

References

1. 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
2. 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
3. Welkom bij de Blankenburgverbinding. <https://www.blankenburgverbinding.nl/>. Accessed 21 Jan 2022

4. 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
5. Bradfield, J., Walukiewicz, I.: The mu-calculus and model checking. In: Handbook of Model Checking, pp. 871–919. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_26
6. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. SIGPLAN Not. **35**(9), 268–279 (2000). <https://doi.org/10.1145/357766.351266>
7. 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.) SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30–September 1, 2000, Proceedings. LNCS, vol. 1885, pp. 205–223. Springer, Cham (2000). https://doi.org/10.1007/10722468_13
8. Coverity. <https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html>. Accessed 18 May 2022
9. Findbugs. <https://findbugs.sourceforge.net/>. Accessed 18 May 2022
10. Haack, C., Huisman, M., Hurlin, C., Amighi, A.: Permission-based separation logic for multithreaded java programs. Log. Methods Comput. Sci. **11**(1) (2015). [https://doi.org/10.2168/LMCS-11\(1:2\)2015](https://doi.org/10.2168/LMCS-11(1:2)2015), <https://lmcs.episciences.org/998>
11. Huisman, M., Monti, R.E.: On the industrial application of critical software verification with VerCors. In: Margaria, T., Steffen, B. (eds.) ISO/IEC JTC1/SC22/WG2 N1540, pp. 12478, pp. 273–292. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61467-6_18
12. Hurlin, C.: Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic. Ph.D. thesis, Université Nice Sophia Antipolis (09 2009)
13. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011). <https://doi.org/10.1109/TSE.2010.62>
14. Klocwork. <https://www.perforce.com/products/klocwork>. Accessed 18 May 2022
15. Kropf, T.: Introduction to Formal Hardware Verification, 1st edn. Springer, Heidelberg (1999). <https://doi.org/10.1007/978-3-662-03809-3>
16. Landelijke Tunnelstandaard (National Tunnel Standard). <http://publicaties.minienm.nl/documenten/landelijke-tunnelstandaard>. Accessed Apr 2022
17. Matsakis, N.D., Klock II, F.S.: The rust language. In: ACM SIGAda Ada Letters, vol. 34, pp. 103–104. ACM (2014)
18. 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
19. Oortwijn, W., Huisman, M., Joosten, S.J.C., van de Pol, J.: Automated verification of parallel nested DFS. In: Biere, A., Parker, D. (eds.) TACAS 2020. LNCS, vol. 12078, pp. 247–265. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_14
20. Rubbens, R.: Improving Support for Java Exceptions and Inheritance in VerCors, May 2020. <http://essay.utwente.nl/81338/>

21. Safari, M., Huisman, M.: Formal verification of parallel prefix sum and stream compaction algorithms in CUDA. *Theoret. Comput. Sci.* (2022). <https://doi.org/10.1016/j.tcs.2022.02.027>
22. Safari, M., Oortwijn, W., Joosten, S., Huisman, M.: Formal verification of parallel prefix sum. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) *NFM 2020*. LNCS, vol. 12229, pp. 170–186. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_10
23. Şakar, O.: Extending support for Axiomatic Data Types in VerCors, April 2020. <https://essay.utwente.nl/80892/>
24. Sonarqube. <https://www.sonarqube.org/>. Accessed 18 May 2022
25. Technolution. <https://www.technolution.eu>. Accessed Apr 2022
26. Timmer, M., Brinksma, H., Stoelinga, M.: Model-based testing, NATO science for peace and security series D: Information and Communication Security, vol. 30, pp. 1–32. IOS Press (2011). <https://doi.org/10.3233/978-1-60750-711-6-1>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

