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

Wytse Oortwijn(✉) and Marieke Huisman(✉)

University of Twente, Enschede, The Netherlands
{w.h.m.oortwijn,m.huisman}@utwente.nl

Abstract. Over the last decades, significant progress has been made on formal techniques for software verification. However, despite this progress, these techniques are not yet structurally applied in industry. To reduce the well-known industry–academia gap, industrial case studies are much-needed, to demonstrate that formal methods are now mature enough to help increase the reliability of industrial software. Moreover, case studies also help researchers to get better insight into industrial needs.

This paper contributes such a case study, concerning the formal verification of an industrial, safety-critical traffic tunnel control system that is currently employed in Dutch traffic. We made a formal, process-algebraic model of the informal design of the tunnel system, and analysed it using mCRL2. Additionally, we deductively verified that the implementation adheres to its intended behaviour, by proving that the code refines our mCRL2 model, using VerCors. By doing so, we detected undesired behaviour: an internal deadlock due to an intricate, unlucky combination of timing and events. Even though the developers were already aware of this, and deliberately provided us with an older version of their code, we demonstrate that formal methods can indeed help to detect undesired behaviours within reasonable time, that would otherwise be hard to find.

1 Introduction

Despite tremendous progress over the last decades on both the theory and practice of formal techniques for software verification [13], these techniques are not yet structurally applied in industrial practice, not even in the case of safety-critical software. Even though formal methods have shown to be able to increase software reliability [6,8,10], their application is often time consuming and may additionally require expert knowledge. Nevertheless, especially in the case of safety-critical software where reliability demands are high, industry can benefit *greatly* from the current state-of-the-art in formal verification research.

To make this apparent, industrial case studies are needed, that show industry and society that formal methods are now ready to help increase software dependability in practice. In turn, such industrial case studies also help researchers and

W. Ahrendt and S. L. Tapia Tarifa (Eds.): IFM 2019, LNCS 11918, pp. 418–436, 2019.
https://doi.org/10.1007/978-3-030-34968-4_23

developers of verification tools to get insight into the needs of industry. By doing so, researchers can improve and adapt their techniques to industrial needs, and thereby reduce the well-known gap between academia and industry.

This paper discusses such an industrial case study. It elaborates on our experiences and results of the formal verification of a safety-critical component of a *control system for a traffic tunnel* that is currently in use in the Netherlands. This particular software component is responsible for handling emergencies. When a fire breaks out inside the tunnel, or a traffic accident occurs, it should start an emergency procedure that evacuates the tunnel, starts the fans to blow away any smoke, turns on the emergency lights to guide people out, and so on. Naturally, the Dutch government imposes very high reliability demands on the traffic tunnel control software, and in particular on this emergency component, which are specified in a document of requirements that is over 500 pages in length [16].

The tunnel control software is developed by Technolution [22], a Dutch software and hardware development company located in Gouda. Technolution has hands-on experience in developing safety-critical, industrial software[1]. The development process of the traffic tunnel control system came together with a very elaborate process of quality assurance/control, to satisfy the high demands on reliability. Significant time and energy has been spent on software design and specification, code inspection, peer reviewing, unit and integration testing, etc.

In particular, during the design phase, the intended behaviour of the tunnel control software has been worked out in great detail: all system behaviours have been specified in pseudo code beforehand. Moreover, these pseudo code descriptions together have been structured further into a finite state machine, whose transitions describe how the different software behaviours change the internal state of the system. Nevertheless, both the pseudo code and this finite state machine have been specified *informally*, and do not have a precise, checkable formal semantics. Throughout the software development process, no formal methods or techniques have been used to assist in the major effort of quality control.

In this case study, we investigate how formal methods can help Technolution to find potential problems in their specification and (Java) implementation, with realistic effort, and preferably at an early stage of development. Technolution is above all interested in establishing whether: (**1**) the specification is *itself* consistent, by not being able to reach problematic states, e.g., deadlocks in the finite state machine; and (**2**) whether the Java code implementation is written correctly with respect to the pseudo code specification of the intended behaviour.

To address both these properties, we use a combination of existing verification techniques, to deal with their different nature. More specifically, for (**1**) we construct a *formal model* of the pseudo code specification and the underlying finite state machine. This model is specified as a process algebra with data, using the mCRL2 modelling language. After that, we use the mCRL2 *model checker* to verify whether the model adheres to certain requirements (e.g., deadlock freedom and strong connectivity), which we formalise in the modal μ-calculus.

[1] To illustrate, Technolution also delivers commercial software written in Rust.

For (**2**), we use VerCors [3] to *deductively* verify whether the control system is correctly implemented with respect to the pseudo code specification. This is done by proving that the implementation is a *refinement* of our mCRL2 model, using our earlier work on model-based deductive verification [17,18].

Our verification effort actually led to the detection of undesired behaviour: the system can potentially reach an internal state in which the calamity procedure is not invoked when an emergency has occurred, due to an intricate, unlucky combination of timing and events. Even though Technolution was well-aware of this—they deliberately provided us with an older version of their specification and implementation—we demonstrate that formal methods *can* indeed help to find such undesired behaviours at an early stage of development. We also demonstrate that formal techniques are able to provide results within reasonable time, that are otherwise hard to find manually. To illustrate, this undesired behaviour was found within approximately 7 working days.

1.1 Contributions and Outline

This paper contributes a successful industrial verification case study that concerns real-world, safety-critical code, and discusses our verification effort and results. The contributions of the case study itself are:

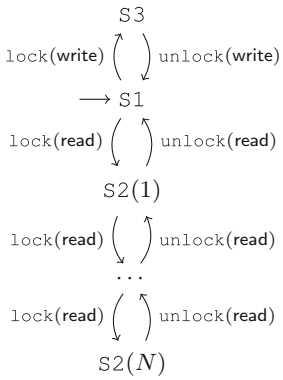
- A formal process-algebraic model of the informal pseudo code description of the tunnel control software, that is defined using mCRL2.
- An analysis of this mCRL2 model, via state-space exploration, and by checking desired μ-calculus properties on the model, like deadlock-freedom.
- A machine-checked proof that the (Java) implementation adheres to the pseudo code specification, by proving that the program refines our mCRL2 model. This refinement proof is done using the automated verifier VerCors.

Here we should note that the actual Java implementation of the tunnel control system is confidential, as well as the documents from the design phase, and therewith also the mCRL2 model and VerCors files that we produced. We therefore sometimes slightly simplify their presentation for the purpose of this paper, for example by using different variable/method/transition names. Nevertheless, the presentation of the case study does not deviate very much from the original, so this paper still gives an accurate overview of our approach and results.

Outline. The remainder of this paper is organised as follows. Section 2 gives preliminaries on the use of mCRL2 and VerCors. Then, Sect. 3 gives more detail on how the tunnel control system is informally specified by Technolution, by discussing the structure of the pseudo code and the finite state machine. Section 4 explains how we modelled this informal specification in mCRL2, after which Sect. 5 discusses its analysis. Section 6 explains how VerCors is used to deductively prove that the tunnel control system correctly implements our mCRL2 model. Section 7 relates our work to existing approaches and industrial case studies, before Sect. 8 concludes.

```
1  sort AccessType ≜ read | write;
2
3  act lock, unlock : AccessType;
4
5  proc
6    S1 ≜ lock(read)·S2(1) + lock(write)·S3;
7    S2(n : ℕ>0) ≜
8      n < N → lock(read) · S2(n + 1) +
9      1 < n → unlock(read) · S2(n − 1) +
10     n = 1 → unlock(read) · S1;
11   S3 ≜ unlock(write) · S1;
12
13 init S1;
```

(a) The mCRL2 model of a RW lock.



(b) The underlying state machine.

Fig. 1. An mCRL2 specification of a RW lock, and the underlying state space.

2 Preliminaries

This section gives preliminaries on the use of mCRL2 (Sect. 2.1) and VerCors (Sect. 2.2).

2.1 Modelling and Analysis with mCRL2

During the case study, we modelled the (informal) tunnel control software specification as a process algebra with support for data. This was done using the specification language of mCRL2 [11]. mCRL2 is a toolset that comes with an ACP-style process-algebraic modelling language, and contains more than sixty tools to support visualisation, simulation, minimisation, state-space generation and model checking of these mCRL2 processes [4]. The model checking back-end takes as input an mCRL2 model, together with a temporal property specified in the modal μ-calculus, and determines whether the model satisfies this property. By default this is done via exhaustive (symbolic) state space analysis.

We further illustrate how this modelling and analysis works by means of a small example, that is presented in Fig. 1. This example demonstrates how a simple read–write (RW) lock can be specified and verified with mCRL2. A RW lock can be acquired multiple times for read-only purposes, but can also provide exclusive write access for a single client: a multiple-reader/single writer lock.

Figure 1b shows the corresponding state machine. Initially the RW lock is unlocked (`S1`). From here the lock can be acquired once for the purpose of writing (`S3`), via a `lock`(write) action, and can subsequently be released again via `unlock`(write). Similarly, from `S1`, the lock can be acquired/released multiple times for reading purposes. The state `S2`(n) represents a read lock that has been acquired n times, where n is bounded to some constant threshold N.

Figure 1a presents the mCRL2 encoding of this locking protocol. The specification language of mCRL2 has various built-in data types (like positive numbers;

see line 7), but also allows defining custom abstract data types, as **sort**'s. Line 1 defines a sort that enumerates the different kinds of accesses that can be granted by the RW lock: read-only (read) access, and read/write (write) access.

Line 3 defines the *actions* for the locking protocol, which represent the basic, observable behaviours of the system. In this example, there are only two observable events, namely `locking` and `unlocking`. In mCRL2, actions can be parameterised by data. In this case, both actions are parameterised by *AccessType*.

These two actions can be composed into *processes* (lines 5–11). This example defines three processes, corresponding to the three locking states: `S1` (unlocked), `S2` (locked for reading purposes), and `S3` (locked for read/write purposes).

Processes are of the following form, where e is an expression:

$$P, Q ::= \varepsilon \mid \delta \mid a(\overline{e}) \mid \tau \mid P \cdot Q \mid P + Q \mid b \to P \mid X(\overline{e}) \qquad \text{(processes)}$$

Of course, the mCRL2 modelling language is actually much richer than the above language [11], for example by supporting parallel compositions. Yet this is the fraction of the mCRL2 language that we will use throughout the paper.

Clarifying the constructs: ε is the *empty* process, without behaviour, whereas δ is the *deadlocked* process, which neither progresses nor terminates. The process $a(\overline{e})$ is an *action invocation*, with $\overline{e}$ a sequence of arguments, while τ is a special, reserved action that models internal, *un*observable system events. $P \cdot Q$ is the sequential composition of P and Q, whereas $P + Q$ is their non-deterministic choice. The process $b \to P$ is the *conditional* process, that behaves as P if b is a Boolean expression that evaluates to *true*, and behaves as δ if b evaluates to *false*. Finally, $X(\overline{e})$ is the invocation of a process named X, with input arguments $\overline{e}$.

Moving back to the example, the `S1` process can either perform a `lock`(read) action, followed by the process invocation `S2`(1), or can do a `lock`(write), after which `S3` is invoked (see line 6). Here the $\cdot$ connective has the highest precedence, followed by $\to$, and then $+$. Process `S3` (line 11) is only able to release the write lock and therewith to continue as `S1`. Finally, `S2`(n) allows to (re)acquire/release read locks, depending on n being small/large enough, respectively, on lines 8–10.

For the actual case study, we modelled the tunnel control system in a similar way: by studying the state machine specification, and encoding it into mCRL2.

Modal μ-calculus. After having constructed a model, mCRL2 allows to analyse it, by checking whether it satisfies a given temporal specification. These specifications are written in the *modal μ-calculus*, a powerful formalism that allows to specify properties about sequences of actions, i.e., traces of the input model.

Properties in the modal μ-calculus are defined by the following language.

$$\alpha, \beta ::= \mathit{true} \mid a(\overline{e}) \mid \neg\alpha \mid \alpha \cdot \beta \mid \alpha + \beta \mid \alpha^* \qquad \text{(action formulae)}$$
$$\phi, \psi ::= b \mid \neg\phi \mid \phi \wedge \psi \mid \langle\alpha\rangle\phi \mid [\alpha]\phi \mid \mu x.\phi \mid \nu x.\phi \qquad \text{(state formulae)}$$

The actual specification language of mCRL2 is again much richer; we refer to [11] for a complete overview and a more detailed description.

Properties in the modal μ-calculus are defined in terms of *action* and *state* formulae. Action formulae α describe sequences of actions $a(\overline{e})$, where *true* stands

for any action. Such descriptions are negatable: $\neg\alpha$ expresses any sequence except for α. Action formulae can also sequentially be composed, $\alpha_1 \cdot \alpha_2$, or alternatively be composed, $\alpha_1 + \alpha_2$, and α^* is the *repetition* (Kleene iteration) of α.

State formulae ϕ, ψ express properties that should hold in the current state. These properties may for example be built from pure Boolean expressions b, but may also contain *modalities*, $\langle\alpha\rangle\phi$ and $[\alpha]\phi$, to express that ϕ must hold after a certain sequence of actions α has been observed. More specifically, $\langle\alpha\rangle\phi$ is the *may modality*, which expresses that, from the current state, the model is able to perform a sequence of actions complying with α, after which ϕ directly holds. Its dual is the *must modality*, $[\alpha]\phi$, which expresses that, from the current state, after the performance of any action sequence α, the property ϕ directly holds.

State formulae may also contain *fixpoint operators*, μ and ν, to specify infinite system behaviour. Here $\mu x.\phi$ is the *least fixpoint* of ϕ, i.e., the smallest reachable set of states satisfying ϕ, where x is the fixpoint variable. These are used to express liveness properties. Its dual is the *greatest fixpoint*, $\nu x.\phi$, used to express safety properties, representing the largest reachable set of states satisfying ϕ.

Below we give three example properties that hold for our RW lock model:

$$[\texttt{unlock}(\mathsf{read})]\mathit{false} \tag{1}$$

$$\nu x.(\,\langle \mathit{true}^* \cdot \texttt{lock}(\mathsf{write})\rangle \mathit{true} \wedge [\mathit{true}^*]x\,) \tag{2}$$

$$\nu x.(\,[(\neg\texttt{unlock}(\mathsf{write}))^* \cdot \texttt{lock}(\mathsf{read})]\mathit{false} \wedge [\mathit{true}^* \cdot \texttt{lock}(\mathsf{write})]x\,) \tag{3}$$

Property (1) states that read locks cannot be released from the initial state, as initially no locks have been acquired. Furthermore, (2) expresses that it always remains possible to acquire the write lock. Finally, (3) states that, when holding the write lock, no read lock can be obtained until the write lock is released.

2.2 Deductive Verification with VerCors

Besides modelling the tunnel software specification in mCRL2, we also used deductive techniques to automatically prove that the code implementation adheres to this specification. This is done using VerCors [3], an automated deductive verifier that targets programs written in high-level languages, like Java and (subsets of) C, that are annotated with JML-like (pre/postcondition) specifications.

VerCors actually specialises in concurrency verification, and is, among other things, able to reason about: fork/join concurrency in Java, GPU kernels in the context of OpenCL, and OpenMP directives for loop parallelisation. Nevertheless, we use VerCors in a purely sequential setting for the purpose of this case study, as the tunnel software does not use any concurrency.

In particular, we use our earlier work on model-based verification [17,18], to mechanically establish that the code implementation of the tunnel control system correctly implements (*refines*) our mCRL2 specification. This resolves a well-known problem in model checking, known as the *abstraction problem*: is the model a *sound* behavioural abstraction of the modelled system?

```
1 shared int r_count;
2
3 modifies r_count;
4 guard 0 < r_count;
5 effect r_count = \old(r_count)−1;
6 action unlock(read);
7
```

(a) The action contract for unlock(read).

```
8  requires Proc(unlock(read) · P + Q);
9  ensures Proc(P);
10 void releaseRead() {
11   action unlock(read) {
12     r_count := r_count − 1;
13   }
14 }
```

(b) Simple read lock release method.

Fig. 2. A simple example of our model-based verification approach, showing the action contract of the unlock(read) action (a), and a code implementation (b).

This refinement approach considers process-algebraic models to be abstract descriptions of shared-memory behaviour. Any (say, Java) implementation of the RW lock would use shared memory to implement the locking functionality, for example by maintaining a shared integer field r_count to administer how many times a read lock has currently been acquired. These shared-memory behaviours are specified in terms of *action* and *process contracts*, which are essentially pre/-postcondition extensions to the mCRL2 language. Figure 2a shows a possible action contract for the unlock(read) action, that consists of: a **modifies** clause that determines which shared fields are modified; a **guard** clause that determines the condition under which the action is allowed to be performed; and an **effect** clause that logically describes the effect on shared memory of performing the action. In this case, performing unlock(read) has the effect of decreasing r_count by one in the implementation, given that it was positive beforehand.

These actions and their contracts can be related to program code by means of code annotations. Figure 2b shows the (simplified) annotations that would be required for a simple implementation of releasing read locks. The program and process are related via *process ownership predicates*, Proc(P), which express that the remaining program is allowed to behave as specified by (the action sequences of) P, with respect to shared memory behaviour. As a precondition, a process of the form unlock(read) $\cdot\, P + Q$ is required by releaseRead, for some P and Q. This implies that the method has the choice to behave as unlock(read), and the remaining program to execute according to P (as is ensured on line 9).

Furthermore, *action blocks* are used to relate process-algebraic actions to program code. Lines 11–13 show an action block specification that links the performance of unlock(read) on process level, to the decrement of r_count in program code. VerCors checks that all modifications to r_count are made within such action blocks, and also automatically verifies whether the decrement of r_count on line 12 is actually allowed according to the process-algebraic specification. These checks enable VerCors to prove that a process-algebraic model is a sound abstraction of the shared-memory behaviour of the modelled program.

For the actual case study, we enriched the actions of our mCRL2 model of the tunnel system with action contracts in a similar way, which we derived from the informal pseudo code. Moreover, we added annotations to the program code, and used VerCors to verify that the implementation adheres to the model.

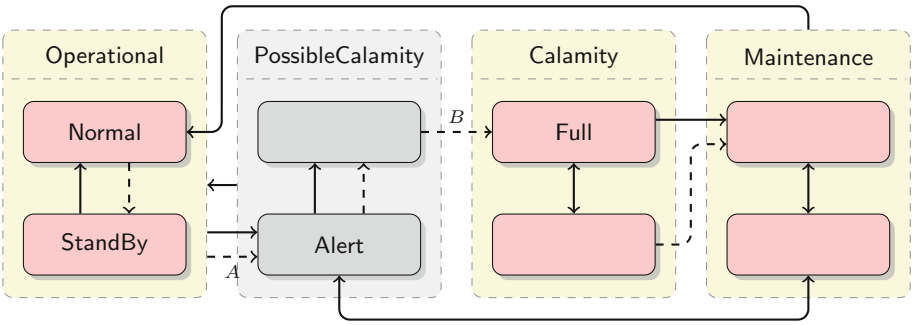


Fig. 3. A simplified visual representation of the FSM. The two transitions that are later written-out as pseudo code in Fig. 4 are labelled A and B.

3 Informal Tunnel Software Specification

Before detailing how mCRL2 and VerCors are applied on the actual case study, let us first discuss the informal specification of the traffic tunnel control system.

Technolution invested significantly in an extensive design phase, to ensure the quality of the control system and to cope with the high reliability demands. During this phase, the intended behaviour of the control software was written-out in pseudo code, together with domain experts. These pseudo code specifications were further structured into a finite state machine (FSM). The states of this FSM are the operational states of the tunnel system (e.g., operating normally, under repair, evacuating, etc.), while the transitions are the pseudo code descriptions of the system behaviour. The FSM thus illustrates how the different behaviours/events of the tunnel system should change its operational state.

Moreover, during the development phase, significant time and effort were invested in ensuring that the code was correctly implemented with respect to this specification. This was done primarily via unit testing and code reviewing.

This section gives more detail on how the tunnel control software was (informally) specified. Section 3.1 discusses the structure of the FSM, after which Sect. 3.2 elaborates on the pseudo code specification, i.e., the transitions of the FSM.

3.1 Structure of the FSM

Figure 3 illustrates the structure of the FSM specification of the tunnel control system. This illustration is simplified for confidentiality reasons: the actual FSM contains many more states and transitions. Nevertheless, the overall structure and the described behaviour are close to the original FSM specification.

The operational states are organised in a 2-layer hierarchy. For example, the composite state Operational contains two sub-states: Normal and StandBy. Transitions come in two flavours. Solid transitions (→) represent *manual interactions*, made by human operators through control panels. Dashed transitions (-->) are

automatic events that are taken autonomously by the control system itself, for example to react to time-outs or sensor input. Any transition whose source is a composite state can also be taken by any of the underlying substates. Moreover, the composite PossibleCalamity state (displayed in grey) is a *ghost state*. Ghost states are special, in the sense that the system can be in a ghost state while also being in a non-ghost state (e.g., to specify that a GUI dialog is being displayed). For example, the tunnel system can be in Alert and Normal simultaneously.

The functional meaning of the specification is roughly as follows. Being in Normal means that the system is in the normal operating state. From Normal the system may autonomously go in StandBy state, as result of, e.g., smoke or heat in the traffic tunnel that is detected via sensor reading. If the system finds enough reason to suspect a real calamity, it may autonomously decide to go from Operational to the Alert state. The Alert state can also manually be entered, when a human operator presses the emergency button on a control panel. The PossibleCalamity composite ghost state starts a timer upon entering. While in this state, if a human operator does not intervene in time by manually cancelling the alert status (thereby going back to the Operational state), the system will automatically launch the Calamity programme, for example to evacuate the traffic tunnel. Such calamities can be recovered from via Maintenance: manually repairing or resolving the calamity's cause. By doing so, the system can manually be brought back to the Normal operating state. However, it may also re-enter Alert in case new potential calamities are detected during maintenance.

3.2 Pseudo Code Specification

Figure 4 gives an idea of the structure of the pseudo code specification of the tunnel system. These pseudo code descriptions were provided by the Dutch Ministry of Infrastructure and Water Management, as part of a national standard on traffic tunnels [16]. The figure highlights two transitions of Fig. 3, labelled as A and B, that describe interesting, important key system behaviours. Transition A specifies how the control system should autonomously request a calamity status when it suspects the traffic tunnel to be in an emergency situation. This will cause the system to go into the Alert ghost state, and therewith start the timer. Transition B specifies what should happen when this timer expires.

Elaborating on the textual format, all autonomous/manual system behaviours are specified in pseudo code style. Any such system behaviour corresponds to a transition in the FSM (denoted by **`transition`**) and is given a unique identifier (**`name`**). The internal state of the system is determined by the values of a set of *pseudo-variables*, which are prefixed with a `#` in the figure. The **`effect`** clauses exactly describe how the transition changes the internal state. The **`condition`** clauses specify under which conditions these state changes are allowed.

Transition A is able to request the calamity procedure to be initiated, by setting `#request_calamity` to `true`, given that `#possible_calamity_detected` has been set to `true` by some other system behaviour, e.g., as result of sensor reading. Such a request will also configure a timer, named

```
1  transition: A (autonomous)
2  name: `ProceedToAlertStatus`
3  condition:
4    #possible_calamity_detected = true &&
5    #request_calamity = false &&
6    #state = Operational;
7  effect:
8    #request_calamity := true;
9    #calamity_timeout := now() + __calamity_timeout_frame;
10
11 transition: B (autonomous)
12 name: `StartCalamityProgrammeAfterTimeout`
13 condition:
14   #state != Calamity &&
15   #request_calamity = true &&
16   now() > #calamity_timeout;
17 effect:
18   #request_calamity := false;
19   #state := Calamity::Full;
20   invoke CalamityProgramme();
```

Fig. 4. The format of the textual specification of the tunnel system.

`#calamity_timeout`, for cancelling the request. Transition B specifies what should happen when this timer expires: in that case the system should enter the operational state Calamity (if not already in there) and start the `CalamityProgramme()`.

The control software of every Dutch traffic tunnel is required to comply with these specifications. This is checked by an external code review committee.

4 Modelling the Tunnel Control System Using mCRL2

Even though the tunnel control software has been specified extensively, prior to our work there had been no formal, structural effort to establish whether the specification *itself* obeys the desired properties. For Technolution, the main properties of interest concern *reliability* and *recoverability*: does the system always go into the Calamity state in real emergency situations? And is it always possible to recover from calamities, and thereby go back to the Normal operational state?

To automatically check for such desired properties, we modelled the pseudo code specifications and the underlying FSM as a process in the mCRL2 language. Figure 5 shows the main structure of our mCRL2 model. This is again a simplified representation; the actual model consists of roughly 700 lines of code.

mCRL2 allows new data types to be defined using the **sort** keyword. We use data sorts to explicitly model the different operational states that the tunnel system might be in, as the structured sort *State*, defined on line 2. Also explicitly

```
1  sort
2    State ≜ struct Normal | StandBy | Alert | Full | ···;
3    Var ≜ struct possibleCalamityDetected | requestCalamity | ···;
4    Val ≜ struct true | false | unknown | ···;
5
6  act enter : State;
7
8  proc
9    %% Encoding of transition A (autonomous)
10   ProceedToAlertStatus(state : State, σ : Var → Val, phase : Nat) ≜
11     σ(possibleCalamityDetected) ∧ ¬σ(requestCalamity) ∧
12       isInOperational(state) →
13         enter(Alert) · System(state, σ[requestCalamity := true], phase);
14
15   %% Encoding of transition B (autonomous)
16   StartCalamityProgrammeAfterTimeout(state, σ, phase) ≜
17     ¬isInCalamity(state) ∧ σ(requestCalamity) →
18       enter(Full) · System(Full, σ[requestCalamity := false], phase);
19
20   %% Encoding of the top-level specification
21   System(state : State, σ : Var → Val, phase : Nat) ≜
22     %% First phase: handling GUI input
23     (phase = 1) →
24       (CancelPossibleCalamity(state, σ, phase) +
25       omitted + τ · System(state, σ, 2)) +
26     %% Second phase: handling internal/external controls and function calls
27     (phase = 2) → ( omitted + τ · System(state, σ, 3)) +
28     %% Third phase: processing autonomous system behaviour
29     (phase = 3) →
30       (ProceedToAlertStatus(state, σ, phase) +
31       StartCalamityProgrammeAfterTimeout(state, σ, phase) +
32       omitted + τ · System(state, σ, 4)) +
33     %% Fourth phase: processing sensor data and update all variables
34     (phase = 4) → (τ · System(state, updateVars(σ), 1));
35
36 init System(Maintenance, σ_init, 1);
```

Fig. 5. The main structure of the mCRL2 formalisation of the specification.

modelled are the various "pseudo-variables" that are used in the textual specification (defined on line 3), together with a domain of values for these variables (on line 4). These three data types are used to model the internal state of the tunnel control system.

Line 6 covers the definition of actions, which model the basic, observable units of computation. One of the main challenges was to determine which observable behaviours of the tunnel system to model explicitly. We experienced that mod-

elling too many behaviours leads to state space explosions, while modelling too few hampers analysis. As the main properties of interest are properties of operational state reachability, the most important observable events to model are the transitions between the operational states. These are modelled as `enter(`s`)` actions, where $s \in \mathit{State}$ is the operational state that is being entered.

The traffic tunnel control system is modelled as the `System(`$\mathit{state}, \sigma, \mathit{phase}$`)` process (lines 21–34), whose arguments determine the internal state of the tunnel. In particular, *state* determines its operational state, whereas σ provides a valuation for all pseudo-variables. The third argument, *phase*, is maintained for technical reasons. This is because the overall system is specified and implemented as a (busy) working loop, that continuously cycles through four different phases, to (1) handle GUI input, (2) process internal requests, (3) autonomously make decisions, and (4) read from sensors and update all variables accordingly. These phases have been made explicit in our model, using *phase*. Every phase has the non-deterministic choice to advance to the next phase, as an internal τ action.

The earlier highlighted transitions A and B both describe autonomous behaviour, and thus are both handled in phase 3 (lines 30–31). Their behaviours are modelled on lines 9–18, and closely follow the pseudo code specification.

Finally, line 36 specifies the initial state of the control system. The system initialises in Maintenance state, and starts by handling phase 1 events. The mapping σ_{init} is a constant that holds the initial valuation of pseudo-variables.

5 Analysing the Tunnel Control System with mCRL2

Now that we have a formal model of the tunnel system specification in mCRL2, we can study its state space, and determine whether it satisfies desired properties, formulated in the modal μ-calculus, with relatively little effort. Technolution was primarily interested in verifying these properties: (i) Deadlock freedom and strong connectivity: are all operational states reachable at any point during execution? (ii) Reliability: does the system automatically go to Full after detecting an emergency, unless this is manually cancelled? (iii) Recoverability: can calamities always be recovered from, by getting the system back to Normal?

A major challenge during analysis was to keep the model's state space small enough to be able to analyse it in a reasonable time. In particular, we needed to improve our mCRL2 model various times, as earlier versions suffered from state space explosions resulting from the explicit modelling of time. Recall that the informal specification includes software behaviours that depend on time, for example the timers that are maintained by the PossibleCalamity ghost state. In earlier versions of our model, these timers were modelled explicitly, as discrete values: natural numbers that were bounded by some threshold. However, their analysis was only feasible with thresholds no larger than three time units, which is insufficient. We later solved this scalability issue by modelling time *implicitly*, by constructing the model in such a way that certain actions must happen before others. More specifically, instead of having certain actions depend on timers or timeouts to happen before others, we let them happen non-deterministically, but in such a way that the original order of action occurrences is preserved.

Our latest model has an underlying state space of roughly 4.200 states and 25.400 transitions, which takes about 4 min to generate[2]. This clearly shows that the tunnel system specification comprises far too many behaviours for software designers/developers to comprehend, without the help of automated tooling. In fact, mCRL2 helped us further, by allowing to minimise this state space modulo branching bisimilarity [12], leaving only 27 states and 98 transitions. This reduction gave us better insight into the system's behaviour.

Together with Technolution, we formulated several dozens of desired properties as μ-calculus formulae, and checked these on the reduced mCRL2 model. An example of such a formula is given below, expressing that the StandBy state can only ever be reached via the Normal state of operation.

$$\nu x.\big(\ [(\neg \texttt{enter}(\mathsf{Normal}))^* \cdot \texttt{enter}(\mathsf{StandBy})]\mathit{false} \wedge \tag{4a}$$

$$[\mathit{true}^* \cdot \texttt{enter}(\mathsf{StandBy})]x\ \big) \tag{4b}$$

More precisely, this greatest fixed-point formula expresses that StandBy cannot be reached via any path of non-"enter(Normal)" actions (by 4a), and that this reachability property remains preserved each time StandBy is entered (4b).

In addition to checking these properties, we also inspected the state space of the minimised model and discussed its structure with Technolution. Ultimately, our verification exposed an intricate violation of the requirement of reliability. We found that the control system can reach a potentially dangerous situation, in which the Calamity state cannot automatically be entered after having detected a potential emergency (unless a human operator manually interferes), due to an intricate, unlucky combination of timing and events. The following reliability property exposes this behaviour, by stating that, while being in the Alert ghost state, it must always be possible to *directly* enter Full, unless the alert status is manually cancelled. This property does not hold for our mCRL2 model.

$$[\mathit{true}^* \cdot \texttt{enter}(\mathsf{Alert})]\,\nu x.\big(\tag{5a}$$

$$[\neg(\texttt{cancel} + \texttt{enter}(\mathsf{Full}))^*]\langle \texttt{enter}(\mathsf{Full})\rangle \mathit{true} \wedge \tag{5b}$$

$$[\mathit{true}^* \cdot \texttt{enter}(\mathsf{Alert})]x\ \big) \tag{5c}$$

Nevertheless, this is precisely the violating behaviour that Technolution hoped we would find. This is because they already found it, by chance, and deliberately provided us with an older version of their specification and implementation. Our case study therefore shows that formal techniques can indeed help to find such problematic behaviours in a more reliable and structural manner, and at an early stage of development, within reasonable time: we found it within 7 working days.

[2] On a Macbook with an Intel Core i5 CPU with 2.9 GHz, and 8Gb internal memory.

```
1  shared bool possibleCalamityDetected, requestCalamity;
2  shared State state;
3
4  modifies state;
5  effect state = s;
6  action enter(State s);
7
8  // The encoding of transition A, as a single action
9  accesses possibleCalamityDetected, state;
10 modifies requestCalamity;
11 guard possibleCalamityDetected ∧ ¬requestCalamity;
12 guard isInOperational(state);
13 effect requestCalamity;
14 action ProceedToAlertStatus;
15
16 accesses state;
17 modifies requestCalamity;
18 guard ¬isInCalamity(state) ∧ requestCalamity;
19 effect ¬requestCalamity;
20 action flipCalamityRequest;
21
22 // The encoding of transition B, as a sequential composition of two actions
23 process StartCalamityProgrammeAfterTimeout ≜
24   flipCalamityRequest · enter(Full);
```

Fig. 6. The VerCors encoding of transitions A and B, as processes with contracts.

6 Specification Refinement Using VerCors

As a next step, we use VerCors to deductively verify that the code implementation adheres to the FSM and pseudo code specification. This is done by proving that the code correctly implements (refines) our mCRL2 model, using our earlier work on model-based verification. Such a proof also adds value to the model, as it establishes that the model is a sound abstraction of the program's behaviour.

As explained in Sect. 2.2, the process algebra language that VerCors uses is an extension of mCRL2, in which all process and action definitions are enriched with pre/postcondition-style contracts. These contracts are used to connect/link processes and actions to program code: they logically describe how the performance of an action corresponds to an update to shared memory, very much like the **`effect`** and **`condition`** clauses used in the pseudo code specification. With these contracts we can mechanically prove with VerCors that every execution of the program corresponds to an action trace (a run) in the mCRL2 model. These links between programs and models preserve safety properties (i.e., $\nu x.\phi$).

For this project, we manually encoded our mCRL2 model into the process algebra language of VerCors[3]. Figure 6 shows an excerpt of this encoding, in

[3] Both these languages can be translated into one another, and we are actively working on mechanising these translations.

which transitions A and B are again highlighted. The VerCors encoding consists of a large number of action declarations, corresponding to the FSM transitions, with contracts that closely follow the pseudo code specifications. Moreover, this version does not use a valuation σ for the pseudo variables, like in Sect. 4, but rather connects to the actual shared fields in the program code (e.g., lines 1–2). The variables *state* and *phase* have been translated likewise. Our VerCors encoding is intended, but not (yet) proven, to be equivalent to the mCRL2 version.

Line 6 defines the `enter(`s`)` action, whose performance has the effect of modifying the shared variable *state*, by assigning s to it. Lines 9–14 give the specification of transition A, as a single action, with a contract that closely follows the corresponding textual specification. Transition B is defined to be composed out of two actions: `flipCalamityRequest` for setting the *requestCalamity* flag to *false*, and `enter` for changing the operating state of the tunnel to Full.

Program Annotations. The next step is to deductively prove that the implementation adheres to the VerCors encoding of the specification, as explained in Sect. 2.2. The actual tunnel control system is implemented in Java. However, we converted this implementation to PVL—an object-oriented toy input language of VerCors—since our model-based verification approach is currently best supported by the PVL front-end (we are currently improving its support for Java).

Figure 7 shows and highlights the annotations of the PVL code implementations of transitions A (lines 2–16) and B (lines 19–35). The **yields bool** *branch* annotations on lines 2 and 19 indicate that *branch* is an extra *output* parameter that only exists for the sake of specification. In the figure, *branch* represents which branch has been executed by the program, and is used in the postconditions to ensure the matching, corresponding process-algebraic choice.

The contract of proceedToAlertStatus states that it will execute as prescribed by the process `ProceedToAlertStatus`$\cdot P+Q$ for some P and Q (line 3), and depending on the execution branch taken, is left with either P (line 4) or with Q (line 5) upon termination. The contract of startCalamityProgrammeAfterTimeout follows the same specification pattern, as well as most of the other methods. Since this model-based verification approach is compositional, we could use it to verify that the entire implementation complies with the process-algebraic specification.

Our deductive verification effort did not directly reveal any problems or violations in the implementation: all methods comply with their specified behaviour. This is expected, as the implementation has been unit tested and code reviewed very rigorously. Nevertheless, this compliance between specification and implementation is now confirmed, by means of a machine-checked proof.

However, this verification did help us, as tool developers, to better understand the needs from industry, and to identify weaknesses in our approach and tooling. To give an example, for future use, Technolution finds it important that our model-based verification technique is applicable on Java code, instead of PVL, and in a more automated manner. We are now actively working on this.

```
1  // The annotated code implementation of Transition A
2  yields bool branch;
3  requires Proc(ProceedToAlertStatus · P + Q);
4  ensures branch ⇒ Proc(P);
5  ensures ¬branch ⇒ Proc(Q);
6  void proceedToAlertStatus() {
7    branch := false;
8    if (possibleCalamityDetected ∧ ¬requestCalamity ∧
9          state = Normal ∨ state = StandBy) {
10     action ProceedToAlertStatus {
11       requestCalamity := true;
12     }
13   }
14   calamityTimeout := now() + __calamity_timeout_frame();
15   branch := true;
16 }
17
18 // The annotated code implementation of Transition B
19 yields bool branch;
20 requires Proc(StartCalamityProgrammeAfterTimeout · P + Q);
21 ensures branch ⇒ Proc(P);
22 ensures ¬branch ⇒ Proc(Q);
23 void startCalamityProgrammeAfterTimeout() {
24   branch := false;
25   if ("state is in calamity" ∧ requestCalamity) {
26     action flipCalamityRequest {
27       requestCalamity := false;
28     }
29     action enter(Full) {
30       state := Full;
31       calamityProgramme();
32     }
33     branch := true;
34   }
35 }
```

Fig. 7. Relating the tunnel specification to the implementation using VerCors.

7 Related Work

Various earlier successes have been reported in the use of model checking in industrial case studies. mCRL2, for example, maintains a gallery of industrial showcases online [15], which includes, among others, the modelling and analysis of firmware for a pacemaker [23], as well as control software used for experiments at the Large Hadron Collider at CERN [14]. Glabbeek et al. formalised the AODV wireless routing protocol in AWN (Algebra for Wireless Networks) [9]—a process algebra for modelling mobile ad-hoc networks—and used it to reason about safety-critical routing properties. Ruijters et al. [20] uses statistical

model checking to study different maintenance strategies for railway joint, in collaboration with ProRail—a Dutch national railway infrastructure manager. Moreover, [1] reports on the experiences of the use of TLA+ at Intel.

In the context of deductive verification, in 2015, de Gouw et al. [10] successfully detected an intricate bug in the standard implementation of OpenJDK's TimSort algorithm, which is used daily by billions of users worldwide. Another successful application of deductive verification is the use of Infer at Facebook [5], to detect potential regressions during continuous integration testing. In [2], a formal verification of a cloud hypervisor is reported, using Frama-C. Also OpenJML has been used successfully for the verification of industrial code; [7] discusses several observations and experiences. Moreover, [19] discusses four industrial case studies that have been performed with VeriFast: two Java Card smart card applets, a Linux networking component, and a Linux device driver.

Regarding combinations of deductive verification and model checking, in [21], CBMC and Frama-C have been used to verify embedded software for satellite launching. But apart from this work, we are not aware of any other industrial applications of model checking combined with deductive verification.

8 Conclusion

During our case study, we found that, even though the specification of the tunnel control system is informal, it *is* well-structured, and therefore has the potential to be formalised within reasonable time. In roughly 7 working days, we constructed a formal model of the informal specification, analysed it using mCRL2, and used VerCors to deductively prove that the code implementation adheres to it. 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 found it coincidentally. We demonstrate that formal methods can indeed help to find such undesired behaviours more structurally, and within realistic time.

As a follow-up, we will continue to collaborate with Technolution, by being involved in an upcoming project in late 2019, concerning safety-critical software. In this project, we will attempt to apply formal methods *during* the software design and development process, rather than after the deployment phase.

This case study also helped us to learn about the needs from industry, and the shortcomings of our tooling, which we will work on before starting the follow-up project. More specifically, we will improve VerCors's support for Java, and work on automated translations between mCRL2 and the process algebra language of VerCors. We will also investigate if the pseudo code specification language can be formalised into a DSL, that is automatically translatable to mCRL2.

Acknowledgements. This work is partially supported by the NWO VICI 639.023.710 Mercedes project and by the NWO TOP 612.001.403 VerDi project.

References

1. Beers, R.: Pre-RTL formal verification: an intel experience. In: DAC, pp. 806–811 (2008). https://doi.org/10.1145/1391469.1391675
2. Blanchard, A., Kosmatov, N., Lemerre, M., Loulergue, F.: A case study on formal verification of the anaxagoros hypervisor paging system with Frama-C. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 15–30. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19458-5_2
3. 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
4. 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
5. Calcagno, C., et al.: Moving fast with software verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 3–11. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_1
6. Clarke, E.M.: The birth of model checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 1–26. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0_1
7. 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
8. Filliâtre, J.: Deductive software verification. STTT **13**(5), 397–403 (2011). https://doi.org/10.1007/s10009-011-0211-0
9. van Glabbeek, R., Höfner, P., Portmann, M., Tan, W.: Modelling and verifying the AODV routing protocol. Distrib. Comput. **29**(4), 279–315 (2016). https://doi.org/10.1007/s00446-015-0262-7
10. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
11. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
12. Groote, J.F., Wijs, A.: An $O(m \log n)$ algorithm for stuttering equivalence and branching bisimulation. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 607–624. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_40
13. Huisman, M., Joosten, S.J.C.: Towards reliable concurrent software. Principled Software Development, pp. 129–146. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98047-8_9
14. Hwong, Y., Keiren, J., Kusters, V., Leemans, S., Willemse, T.: Formalising and analysing the control software of the compact muon solenoid experiment at the large hadron collider. SCP **78**(12), 2435–2452 (2013). https://doi.org/10.1007/978-3-642-29320-7_12
15. mCRL2—Showcases. https://www.mcrl2.org/web/user_manual/showcases.html. Accessed July 2019

16. Landelijke Tunnelstandaard (National Tunnel Standard). http://publicaties.minienm.nl/documenten/landelijke-tunnelstandaard. Accessed June 2019
17. Oortwijn, W., Blom, S., Gurov, D., Huisman, M., Zaharieva-Stojanovski, M.: An abstraction technique for describing concurrent program behaviour. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 191–209. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_12
18. Oortwijn, W., Blom, S., Huisman, M.: Future-based static analysis of message passing programs. In: PLACES, pp. 65–72 (2016). https://doi.org/10.4204/EPTCS.211.7
19. Philippaerts, P., Mühlberg, J., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software verification with verifast: industrial case studies. SCP **82**, 77–97 (2014). https://doi.org/10.1016/j.scico.2013.01.006
20. Ruijters, E., Guck, D., van Noort, M., Stoelinga, M.: Reliability-centered maintenance of the electrically insulated railway joint via fault tree analysis: a practical experience report. In: DSN, pp. 662–669. IEEE Computer Society (2016). https://doi.org/10.1109/DSN.2016.67
21. Silva, R., de Oliveira, J., Pinto, J.: A case study on model checking and deductive verification techniques of safety-critical software. In: SBMF, Federal University of Campina Grande (2012)
22. The Technolution. https://www.technolution.eu. Accessed June 2019
23. Wiggelinkhuizen, J.: Feasibility of formal model checking in the Vitatron environment. Master's thesis, Eindhoven University of Technology (2007)