Forside

Eksamensinformationer
DDD880012P - d807f20 - project

Besvarelsen afleveres af

Thomas Pedersen
tpede16@student.aau.dk

Emil Ernstsen
eernst16@student.aau.dk

Alexander Bilgram Kristensen
abkr16@student.aau.dk

Peter Haahr Taankvist
ptaank16@student.aau.dk

Peter Borch Greve
pgreve16@student.aau.dk

Administration
Ulla Øland
ulla@cs.aau.dk
+4599408854

Bedømmere
Ulrik Nyman
Eksaminator
ulrik@cs.aau.dk
+4599409985

Besvarelsesinformationer

Titel: HMKAAL: Formal Verification and Detection of Non-determinism and Data Races in Industrial Systems
Titel, engelsk: HMKAAL: Formal Verification and Detection of Non-determinism and Data Races in Industrial Systems
Tro og love-erklæring: Ja
Indeholder besvarelsen fortroligt materiale: Publication approved by the student group and HMK Bilcon 2021-04-09

Aalborg University student project
Spring semester 2020

HMKAAL: Formal Verification and Detection of Non-determinism and Data Races in Industrial Systems

Alexander Bilgram* | Emil Ernstsen | Peter B. Greve | Peter H. Taankvist | Thomas Pedersen

Computer Science, Aalborg University, Denmark

Correspondence
*Alexander Bilgram Email: abkr16@student.aau.dk

Abstract

In recent years the use of model checking in the industry has increased. The contribution of this paper is to extend the current knowledge of model checking with an industrial case. We therefore explore model checking possibilities for the system, SafeCon III, provided by the Danish company HMK Bilcon. Furthermore, this paper presents a formal semantics of SafeCon III, in which the system is described as a network of components. In order to check for the common model checking property, reachability, this paper introduces a branching temporal logic called Reduced Computation Tree Logic that is able to express reachability properties for a network of components. During implementation we find that the state space of SafeCon III is much too large to realistically model check even with the use of certain reduction techniques. Therefore, we decide to use a random depth first simulation of the system instead, where we detect data races and non-determinism. This approach has the drawback that we are only able to tell if a property *is* reachable, but not whether it *is not* reachable. We provide the simulator, HMKAAL, for HMK Bilcon, and their use of the system show that HMKAAL is able to provide significant value for HMK Bilcon already having discovered several errors in SafeCon III.

KEYWORDS:
Formal verification, Simulation, Data Race Detection, Non-determinism detection

1 | INTRODUCTION

The use of modelling and verification in the industry has slowly increased throughout recent years[1]. The goal of this paper is to further look into how modelling and verification can be utilized in another scenario. More specifically, we look into how to construct a case specific verification tool for the system SafeCon III[2], made by HMK Bilcon(HMK).

HMK is a provider of tank truck solutions for delivery of oil and gas, military solutions for the Danish Defense Department, mobile solutions for the health sector and different events as well as grain- and animal feed solutions[3]. In short they provide the container part of a truck trailer as well as different software functionalities regarding the container. HMK applies model-based development via an integrated development environment for model checking called H-UPPAAL[4] in order to produce software. HMK uses model-based development, in the sense that they compile the models made in H-UPPAAL into functioning Java code, allowing for non-programmers to create functioning code. Originally, H-UPPAAL applied the verification engine of UPPAAL[5] for verifying different queries on the models modelled in H-UPPAAL. However, UPPAAL is closed-source and according to HMK the license is too expensive for a small company, such as HMK, which means that they cannot verify the models made in H-UPPAAL. We introduce a new verification tool, HMKAAL, made specifically for the GUI of H-UPPAAL and how HMK uses this GUI. Contrary to many other popular model checkers[6] HMKAAL will be written in the programming language Rust[7].

During implementation of HMKAAL, we find that it is not realistic to do an exhaustive search of the state space due to the size of SafeCon III. Therefore, we instead check properties using a random depth first simulation, having the drawback of not being able to tell if a property *is not* reachable. Even though the simulation may not be conclusive, HMK may be able to draw conclusions based upon the runtime and the trace.

Related Work:

There exist many different model checkers[6]. These model checkers use different formalisms, e.g. the model checker, UPPAAL, seen in [5][8] which is a model checker for timed automata. Another popular model checker is called SPIN[9], which is a verification system for models of distributed software systems represented as finite automata.

There are many ways to use modelling and verification techniques in industrial solutions. Although still not widespread there has been an increased interest in model checking in the industry. As such, different papers have researched how to apply model checking in an industrial context. In [10] we see an example of this, as the paper looks at the potential industrial application of model checking in safety-related software. The authors in [1] researched the application of model checking in the Finnish nuclear industry and showed that model checking is a viable option for detecting flaws in these types of systems.

Concurrently with our project, the author in [11] represents Tick Tock Automata which is a modelling formalism for a real world industrial system in corporation with HMK Bilcon, which we use for inspiration for the semantics introduced in this paper.

As seen in related research, it is possible to apply model checking in the industry. Therefore, we propose using techniques from the modelling and verification field in an industrial case with HMK Bilcon.

Outline:

The paper is organized as follows: Section 2 covers HMK's proposed problem and the different challenges related to model checking SafeCon III. In Section 3, we present a formal semantics of SafeCon III presented as a network of components(NOC). In Section 4, we introduce a branching temporal logic able to express different reachability properties on networks of components. In Section 5, we describe the implementation of HMKAAL. Firstly, we look into different reduction techniques and discuss the effects of these in regards to the HMK specific case. Secondly, we discuss the change to a simulation based approach and the implications of this change. Lastly, we present an algorithm for simulating a NOC. In Section 6, we experiment with HMKAAL by benchmark testing the system and delivering the system to HMK to be tested in production. Lastly, in Section 7, we conclude on the findings and describe concepts for further work.

2 | PROBLEM DEFINITION

HMK has made a compiler that generates Java code from models made in H-UPPAAL. However, these models cannot be model checked as there exists no open source model checker for H-UPPAAL yet. Furthermore, because of HMK's inhouse compiler, the semantics of how HMK uses H-UPPAAL and H-UPPAAL's intended use, are different. One possibility would be to translate to an existing model checker such as SPIN. However, the expectation is that a custom verification tool can increase performance in terms of computation and usefulness.

As such we have two main problems in order to provide a useful verification tool for HMK:

1. Specify the semantics of HMK's use of H-UPPAAL
2. Identify which options from the modelling and verification field are most useful for HMK

Due to the specific use of H-UPPAAL we ignore many of the functionalities that H-UPPAAL offers, which may result in optimizations that could not have otherwise been made. However, HMK's way of using H-UPPAAL also creates problems as they use the software in conjunction with hardware. The hardware has the option to, at an unknown point in time, update certain variables to unknown values. This means we have variables that theoretically can be any value in a given range. As such, we have to correctly represent the communication between the models and the hardware.

As such there are also several sub-problems for the verification tool itself which we seek to solve:

1. Are there any optimization techniques that can be applied to the HMK specific semantics?

2. How do we represent the communication between the hardware and the models?
3. How do we represent the relatively random points in time at which the hardware variables are updated?

2.1 | Formal Verification for HMK

Before we begin the specification of HMK's semantics and implementation, we questioned HMK about what features would be most useful for them. This is done to prioritize our time right in order to produce a useful system for HMK. It is also important for the options we have in regards to verifying their models, as their specific wishes may open up for certain optimizations or alternative ways of verifying. Furthermore, we may need to take other information into consideration that is not normally considered in a model checker.

There are three key features that HMK would like the tool to have:

1. Given that HMK's models are relatively large, HMK would like to be able to tell if a location is reachable.
2. Additionally, they would like functionality for checking whether two locations are mutually exclusive. As a continuation of this they would like to check for data races between models i.e. whether two models are writing to the same variable at the same time. This also includes reachability of variables i.e. whether a certain variable can gain a certain value.
3. SafeCon III should be deterministic, therefore HMK would like to be able to detect non-determinism i.e. if two or more edges are active from the same location at the same point in time.

3 | SAFECON III

Although H-UPPAAL supports hierarchical timed automata[4][12], HMK does not use this functionality. As such the notion of *time* in SafeCon III, is modified from hierarchical timed automata. SafeCon III instead implements a *Tick Tock semantics*[11].

The Tick Tock semantics consists of a *Tick*, in which every component takes a single transition if possible, and a *Tock* in which the system reads input values from the hardware layer. The system executes n Tick steps, where $n \geq 10$, until a Tock step is encountered and the input values are read. The reason why the system executes at least ten Tick steps before a Tock is allowed, is to ensure meaningful progression of the H-UPPAAL components. The intuition of the Tick Tock semantics is illustrated in Figure 1.

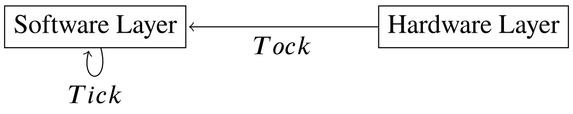

FIGURE 1 Illustration of the Tick Tock intuition

A bi-product of this semantics is that in each Tick step, every component is concurrently taking a transition with no synchronisations because no components should be able to modify the same variables within a single step, i.e. no data races. In fact, the Tick Tock semantics does not utilize synchronisations at all, although the hierarchical timed automata formalism supports them.

3.1 | A formal semantics of networks of components

In this section we describe SafeCon III as a network of components (NOC) inspired by the notion of networks of Tick Tock Automata[11]. NOC is a more general formalism, while networks of Tick Tock Automata are made specifically for HMK. Firstly, we formalize a single component.

Let $\mathbb{V} = \mathbb{R} \cup \mathbb{B}$ be the variable domain where $\mathbb{R}$ is the set of real values and $\mathbb{B}$ is the set of boolean values. We say Ω is the set of all possible variable valuations. SafeCon III is comprised of a number of *Components*, where a single component, C, is defined as a tuple $C = (\mathcal{L}, \ell_0, \mathcal{V}, \mathcal{K}, E)$, such that:

- $\mathcal{L}$ is a finite set of locations,
- $\ell_0 \in \mathcal{L}$ is the initial location,
- $\mathcal{V}$ is a set of modifiable variables,
- $\mathcal{K}$ is a set of unmodifiable variables, and
- $E \subseteq \mathcal{L} \times \mathcal{U} \times \mathcal{B}(\mathcal{V} \cup \mathcal{K}) \times \mathcal{L}$ is a finite set of edges.

Where $u \in \mathcal{U}$ is an update function, $u : \Omega \rightarrow \Omega$ and $\mathcal{B}(\mathcal{V} \cup \mathcal{K})$ is the set of constrains over the modifiable and unmodifiable variables given by the abstract syntax:

$$g ::= x \bowtie n \mid x_1 \bowtie x_2 \mid g_1 \wedge g_2 \mid g_1 \vee g_2$$

where $n \in \mathbb{V}$, $x_1, x_2 \in \mathcal{V} \cup \mathcal{K}$ and $\bowtie \in \{\leq, <, \neq, =, >, \geq\}$.

Let $(\mathcal{L}, \ell_0, \mathcal{V}, \mathcal{K}, E)$ be a component and $v \in \Omega$ be a valuation such that $v : \mathcal{V} \cup \mathcal{K} \rightarrow \mathbb{V}$. The semantics of a component is defined as a transition system $\langle S, s_0, \rightarrow \rangle$ where $S \subseteq \mathcal{L} \times \Omega$ is the set of states, $s_0 = (\ell_0, v_0)$ is the initial state where v_0 is the initial valuation and $\rightarrow \subseteq S \times S$ is the transition relation defined as $(\ell, v) \rightarrow (\ell', v')$ iff. $(\ell, u, g, \ell') \in E$ such that $u(v) = v'$ and $v \models g$ where $\models$ is the satisfaction relation. We denote such a legal transition as $\ell \xrightarrow{u,g} \ell'$ and if no such legal transition exists we denote it $\ell \overset{u,g}{\not\longrightarrow}$.

Let f and g be functions on the same domain and range and let $F_{f,g}$ be the set containing f and g, $F_{f,g} = \{f, g\}$. We define function composition over sets of functions as:

$$f(g(x)) = \circ F_{f,g}(x)$$

The function composition operator over sets is only applicable if the set of functions is symmetric i.e. $f(g(x)) = g(f(x))$, meaning the order of function application does not matter. The implication of this ensures data races cannot occur.

We define *networks of components*. Let $C_i = (\mathcal{L}_i, \ell_i^0, \mathcal{V}, \mathcal{K}, E_i)$ be a network of n components, where $1 \leq i \leq n$. We denote a location vector as $\overline{\ell} = (\ell_1, ..., \ell_n)$ and a location replacement as $\overline{\ell}[\ell_i'/\ell_i]$ meaning the ith location ℓ_i of $\overline{\ell}$ is replaced by ℓ_i'. Furthermore, we define a notion of multiple location replacements as $\overline{\ell}[\overline{\ell''}/\overline{\ell'}]$, resulting in $\overline{\ell}$ where all locations in the location vector $\overline{\ell'}$ are pairwise replaced with locations in $\overline{\ell''}$.

The Semantics of the NOC is defined as a transition system $\langle S, s_0, \rightarrow \rangle$, where $S = (\mathcal{L}_1 \times \cdots \times \mathcal{L}_n) \times \Omega$ is the set of states, $s_0 = (\overline{\ell}_0, v_0)$ is the initial state, where $\overline{\ell}_0 = (\ell_1^0, ..., \ell_n^0)$ is the initial location vector and v_0 is the initial valuation, and $\rightarrow \subseteq S \times S$ is the transition relation defined as the following three transition rules:

$$(\overline{\ell}, v) \rightarrow (\overline{\ell'}, v') \iff \forall \ell_i' \in \overline{\ell'}. \exists \ell_i \xrightarrow{u,g} \ell_i' \text{ where } \ell_i \in \overline{\ell} \text{ and } v' = \circ U(v)$$
$$\text{where } U = \{u \mid \forall \ell_i' \in \overline{\ell'}. \exists \ell_i \in \overline{\ell}. \exists (\ell_i, u, g, \ell_i') \in E_i\} \cup \gamma$$

The above transition describes every component in the NOC taking a legal transition, where $\gamma : \Omega \rightarrow \Omega$ is a function updating transition independent variables in the valuations, such as timers.

$$(\overline{\ell}, v) \rightarrow (\overline{\ell}[\overline{\ell''}/\overline{\ell'}], v') \iff \forall . \ell_i'' \in \overline{\ell''}. \exists . \ell_i' \xrightarrow{u,g} \ell_i'' \text{ where } \ell_i' \in \overline{\ell'}, \forall \ell_i''' \in \overline{\ell}. \ell_i''' \notin \overline{\ell'} \Rightarrow \ell_i''' \overset{u,g}{\not\longrightarrow}$$
$$\text{and } v' = \circ U(v) \text{ where } U = \{u \mid \forall \ell_i'' \in \overline{\ell''}. \exists \ell_i' \in \overline{\ell'}. \exists (\ell_i', u, g, \ell_i'') \in E_i\} \cup \gamma$$

The above transition describes only some components in the NOC taking a legal transition.

$$(\overline{\ell}, v) \rightarrow (\overline{\ell}, v') \iff \forall \ell_i \in \overline{\ell}. \ell_i \overset{u,g}{\not\longrightarrow} \text{ and } v' = \gamma(v)$$

The above transition describes no component taking a transition. This transition relation describes the runtime semantics for a Tick in a NOC.

Let $C_i = (S_i, s_i^0, \mathcal{V}, \mathcal{K}, E_i)$ be a NOC. A Tock is described as writing changes to the unmodifiable variables, $\mathcal{K}$, defined by the following function, $\mathcal{T} : \Omega \rightarrow \Omega$:

$$\mathcal{T}(v) = v' \text{ where } v'(x) = \begin{cases} \mu(x) & \textit{if } x \in \mathcal{K} \\ v(x) & \textit{otherwise} \end{cases}$$

where μ is some function describing the input values from the hardware layer.

4 | REDUCED COMPUTATION TREE LOGIC

In order to test for reachability we introduce a branching temporal logic, Reduced Computation Tree Logic (RCTL), that can express reachability properties for networks of components. The logic is a subset of Computation Tree Logic[13], with similar constructs and a more restricted grammar.

4.1 | Syntax

The syntax of RCTL consists of formulas over locations and variable constraints. The formulas are formed according to the following grammar:

$$\psi ::= \exists\Diamond\varphi \qquad \varphi ::= \ell \mid x \bowtie n \mid x_1 \bowtie x_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \varphi_1 \implies \varphi_2$$

Where $\ell \in \mathcal{L}$ is a location in a component, $x, x_1, x_2 \in \mathcal{V} \cup \mathcal{K}$ are variables in the NOC, $n \in \mathbb{R} \cup \mathbb{B}$ and $\bowtie \in \{\leq, <, \neq, =, >, \geq\}$. Intuitively, $\exists\Diamond\varphi$ expresses that a state in which φ is satisfied, is reachable from the initial state.

4.2 | Semantics

Let $C_i = (S_i, s_i^0, \mathcal{V}, \mathcal{K}, E_i)$ be a NOC. The semantics of RCTL is defined over transition systems, $\langle S, s_0, \rightarrow \rangle$, where $S = (\mathcal{L}_1 \times \cdots \times \mathcal{L}_n) \times \Omega$ is the set of states, $s_0 = (\overline{\ell}_0, v_0)$ is the initial state and $\rightarrow$ is the Tick transition relation described in Section 3.1. We define a path as a sequence of states reachable from some state, $s = (\overline{\ell}_i, v_i)$ where $s \in S$:

$$path(s) = ((\overline{\ell}_i, v_i), (\overline{\ell}_{i+1}, v_{i+1}), \ldots) \text{ where } \forall j \geq i.(\overline{\ell}_j, v_j) \rightarrow (\overline{\ell}_{j+1}, v_{j+1})$$

We denote a path as π and use the notation $\pi[i]$ to access the ith state in the path. The set of all paths from some s, $Paths(s)$, is defined as:

$$Paths(s) = \{\pi \mid \pi[0] = s\}$$

The satisfaction of an RCTL formulae is defined as the relation $\models$. The satisfaction relation for φ is described as:

$$\begin{aligned}
s \models \ell &\iff \ell \in \overline{\ell} \text{ where } s = (\overline{\ell}, v) \\
s \models x \bowtie n &\iff v(x) \bowtie n \text{ where } s = (\overline{\ell}, v) \\
s \models x_1 \bowtie x_2 &\iff v(x_1) \bowtie v(x_2) \text{ where } s = (\overline{\ell}, v) \\
s \models \neg\varphi &\iff s \not\models \varphi \\
s \models \varphi_1 \implies \varphi_2 &\iff \text{if } s \models \varphi_1 \text{ then } s \models \varphi_2 \text{ else } \mathit{true} \\
s \models \varphi_1 \wedge \varphi_2 &\iff s \models \varphi_1 \text{ and } s \models \varphi_2 \\
s \models \varphi_1 \vee \varphi_2 &\iff s \models \varphi_1 \text{ or } s \models \varphi_2
\end{aligned}$$

The satisfaction relation for ψ is described as:

$$s \models \exists\Diamond\varphi \iff \exists\pi \in Paths(s).\exists i \geq 0.\pi[i] \models \varphi$$

We say $sat(\psi)$ is the satisfaction set over ψ, defined as: $sat(\psi) = \{s \mid s \in S \wedge s \models \psi\}$. A NOC is said to satisfy formula ψ if the formula is satisfied in its initial state:

$$NOC \models \psi \iff s_0 \models \psi$$

In other words $s_0 \in sat(\psi)$. Essentially meaning there exists a path from the initial state to a state satisfying the RCTL formula.

5 | IMPLEMENTATION

In this section we describe the implementation of the verification tool as well as the considerations behind it. Firstly, we discuss the plausibility of model checking in the traditional sense as well as reduction techniques to reduce the state space of SafeCon

III. Secondly, we describe alternative reduction techniques from the modelling and verification field that could have been a possibility. Lastly, we argue for the decision to create a simulator instead of a model checker and discuss the benefits and shortcomings of this approach.

5.1 | Plausibility of Model checking

Traditionally, model checking exhaustively searches the state space to check whether a given property is satisfied. However, this can result in state space explosion, which can be reduced by applying different reduction techniques. These techniques involve the use of data structures such as binary decision diagrams[14] to collapse states into symbolic representations, difference bound matrices[15] which reduces an infinite state space to a finite one, as well as techniques to reduce the state space that is necessary to search, such as partial order reduction[16].

5.2 | SafeCon III state space and reduction

Because of the Tock in SafeCon III, model checking results in state space explosion. This is the case as the function μ, describing the Tock input from the hardware layer, can change all hardware variables to any value. As an example let the set of unmodifiable variables, $\mathcal{K} = \{k_1, ..., k_n\}$, contain both analog and digital input values, where analog inputs are numbers and digital inputs are booleans. Assuming $n = 20$ with 10 analog and 10 digital input values. The state space would then explode to $2^{10} + 255^{10} = 1.16 \cdot 10^{24}$ states, as the analog input has 255 possible values. This exponential behaviour of the state space means model checking without applying efficient reduction techniques is unfeasible. The following sections present different reduction techniques that are applicable to SafeCon III.

5.2.1 | Implicit partial order reduction

The Tick transition relation presented in Section 3.1 ensures that the sequence of transitions in components is inconsequential. This is the case as the function composition operator on the valuation, $\circ U(v)$, is symmetric, meaning the appliance of updates on the modifiable variable domain, $\mathcal{V}$, is symmetric. Essentially, the semantics of SafeCon III implements an implicit partial order reduction within the Tick transition relation, i.e. the order of component transitions within a single Tick transition does not matter. If this was not the case, every combination of component transitions would further add to the size of the state space. In fact, if we observe this being violated during execution, we detect it as a data race, described in Section 2.1, and report a warning. This is not possible according to the semantics presented in Section 3.1. Why this choice was made and how we handle data races will be elaborated further in Section 5.7.

5.2.2 | Decompositional model checking

Another option for reducing the state space of SafeCon III is the notion of decomposing the NOC. The resulting NOC would then only contain the components which either are affected by the query to be checked or components that affect components to be checked. However, SafeCon III is highly interconnected meaning this reduction technique is very dependent on the property to be checked.

5.2.3 | Symbolic representation

Lastly, we attempt to reduce the number of states by representing states symbolically. For example, we introduce ranges for each variable $x \in \mathcal{V} \cup \mathcal{K}$ that represents all the values that x can evaluate to, while satisfying a given formula. As such, the range for a given variable is restricted during the run as it is used in comparisons and assignments.

Formally, this is introduced by modifying our valuation with ranges such that $v : \mathcal{V} \cup \mathcal{K} \rightarrow \mathbb{V} \times \mathbb{V}$. Let $\Upsilon : \mathcal{B}(\mathcal{V} \cup \mathcal{K}) \times \Omega \rightarrow \Omega$ be a restriction function. Informally we update a valuation from a guard and a valuation, based on the restrictions the guard imposes on the variables. We can then modify the transition relation for components: $\rightarrow \subseteq S \times S$ to instead be defined as $(\ell, v) \rightarrow (\ell', v')$ iff. $(\ell, u, g, \ell') \in E$ such that $\Upsilon(g, v) = v''$, $v'' \models g$ and $u(v'') = v'$.

For example, given a state $s = (\ell, v)$ where $v(x) = [0, 255]$ taking the transition $\ell \xrightarrow{x<20} \ell'$ results in two new states, $s' = (\ell', v')$ where $v'(x) = [0, 19]$ and $s'' = (\ell, v'')$ where $v''(x) = [20, 255]$. Note that the new range is always as large as possible.

We create two examples with the new notion of ranges and symbolic states; one to show how the state space is reduced and another to show how there still is a problem with state space explosion. Given two hardware variables x_1 and x_2, both with ranges from 0 to 255, we show how comparisons create new symbolic states.

$$\begin{tikzcd}
 & \langle l, x_1 = [0,255],\ x_2 = [0,255]\rangle \arrow[dl] \arrow[dr, "x_1 \leqslant 50"] & \\
\langle l', x_1 = [0,50],\ x_2 = [0,255]\rangle & & \langle l, x_1 = [51,255],\ x_2 = [0,255]\rangle
\end{tikzcd}$$

FIGURE 2 Comparison of ranges to an integer value

As seen in Figure 2 we create only two new symbolic states, compared to the 255^2 new states that would be created without symbolic values.

For comparisons between concrete values or expressions, which can be evaluated to a concrete value, and symbolic ranges will yield two new states with the comparators $\{\leq, <, >, \geq\}$ (over and under) and three new states with $\{\neq, =\}$ (over,under and equals). However, a problem occurs when comparing two hardware variables with overlapping ranges as seen in Figure 3. To

$$\begin{tikzcd}[column sep=small]
 & & & & \langle l, x_1 = [20,35],\ x_2 = [25,50]\rangle \arrow[dllll] \arrow[dlll] \arrow[dl] \arrow[d, "x_1 < x_2"] \arrow[dr] \arrow[drrr] & & & \\
\begin{matrix} x_1 = [20,24] \\ x_2 = [25,50] \end{matrix} & \begin{matrix} x_1 = [20,25] \\ x_2 = [26,50] \end{matrix} & \cdots & \begin{matrix} x_1 = [20,35] \\ x_2 = [36,50] \end{matrix} & \begin{matrix} x_1 = [35,35] \\ x_2 = [25,35] \end{matrix} & \begin{matrix} x_1 = [34,35] \\ x_2 = [25,34] \end{matrix} & \cdots & \begin{matrix} x_1 = [25,35] \\ x_2 = [25,25] \end{matrix}
\end{tikzcd}$$

FIGURE 3 Comparisons of two ranges. Locations omitted for clarity.

calculate how many new states are produced from such a comparison we introduce the functions; *glb(l,u)*, which returns the greatest lower bound of the two ranges *l* and *u* and *lub(l,u)*, which returns the least upper bound of the two ranges *l* and *u*. The amount of newly created symbolic states, given two variables x_1, with range $r_1 := v(x_1)$, and x_2, with range $r_2 := v(x_2)$, would then be

$$max((lub(r_1, r_2) - glb(r_1, r_2)) \cdot 2, 1)$$

for $\{\leq, <, >, \geq\}$ and

$$max((lub(r_1, r_2) - glb(r_1, r_2)) \cdot 3, 1)$$

for $\{\neq, =\}$.

This quickly increases the size of the state space. The max function is used in case the two ranges do not overlap, in which case we only get one new symbolic state. For example [50,255] > [0,49] will produce only one new state, where no ranges are changed. Additionally, both these examples were only with one comparison between two ranges. If a guard included more comparisons between multiple different ranges the number of states increases exponentially in the number of comparisons.

To further extend the problem, given a Tick transition in a NOC where a number of components, *n*, each takes a transition where hardware variables are present in the guard, the number of new states are then exponential in *n*. This is the case as we need to account for every combination of newly created states. For example if we have three components, that during a Tick produces 15 new states each, we get 15^3 new states that need to be checked.

For a highly interconnected NOC such as SafeCon III that is also highly dependent on hardware values, it is not uncommon that we have 30 components taking a transition in a single tick, with more than half being dependent on hardware variables. As such, it is clear that we still experience state space explosion very early in the model checking process, making exhaustive model checking of a system such as SafeCon III unfeasible with the presented reduction techniques.

5.3 | Alternative reduction techniques

As the above presented techniques are not sufficient to realistically model check SafeCon III, there are some alternative reduction techniques that are worth looking into. Though these alternatives are promising, time constraints on the project cause us to not pursue them any further.

Binary Decision Diagrams

To reduce the state space, many modern model checkers rely on Binary Decision Diagrams (BDDs)[14] as their primary state space representation technique. BDDs are a data structure used to represent boolean functions expressed in an *if-then-else* form. These are powerful for symbolic model checking of systems, but are restricted by the fact that the states have to be binary. However, there exists modifications such as Interval Decision Diagrams (IDDs)[17], which could be promising for model checking of SafeCon III. IDDs allow child nodes of the diagram to instead be associated with an interval, very similar to the notion of ranges on variables presented above.

The use of BDDs or IDDs in HMKAAL could have a significant effect on the state space, but we made the design decision to not use these data structures early in the process, as we believed a customized system would give be more useful, albeit slower, to HMK. Furthermore, we believed detection of data races and non-determinism would pose a problem using BDDs or IDDs.

Difference Bound Matrices

Difference Bound Matrices (DBMs) is a well studied approach for handling constraint systems, such as the clock constraints of UPPAAL[18]. The matrices represent constraints by having a row and a column for each constraint variable. The row then stores the variables lower bounds for the differences to all other variables, while the column stores the upper bounds[15]. For HMKAAL, the application of this approach would be to implement the ranges described in Section 5.2.3. However, in Rust there is no library which supports the DBM data structure. As such, if we wanted to make use of DBMs we would have to implement the data structure ourselves which is out of scope for this project. Furthermore, as we do not have any formal clocks we expect that the effect of DBMs would likely not be worth the implementation time.

5.4 | Simulation

Due to the issues described in Section 5.2, performing an exhaustive search of SafeCon III's state space becomes infeasible, without the use of possible reduction techniques such as BDDs or IDDs mentioned in Section 5.3. As such we switch to a simulation based approach, and will in this section describe what this means for the usefulness and computation speed of our system and describe certain techniques used to further increase the usefulness and speed.

5.4.1 | Simulation vs. Model Checking

We switch to a random depth first simulation of the state space. We do this by randomizing the value of each unmodifiable variable per Tock in its given range, i.e. this is the description of μ in a Tock step. Furthermore, we let timers grow with a fixed rate of 1 millisecond per Tick, in order to simplify the notion of time in the simulation. In case of non-determinism we choose a random edge. In case of data races we describe our policy in Section 5.7. This still allows us to check for reachability, data races and non-determinism. As such, we still comply with two out of three of the features mentioned in Section 2.1, sacrificing detection of mutual exclusion. We also sacrifice other functionality that could have proved useful. We can no longer verify whether a location or a variable valuation is *not* reachable. We also sacrifice the functionality to guarantee that there are no data races and no non-determinism. Additionally, if we wanted to extend the verification tool to e.g. be able to guarantee deadlock freedom, this is no longer a possibility.

5.4.2 | Architecture

The architecture of HMKAAL is shown in Figure 4. HMKAAL expects a NOC in the JSON file format, which it converts into a set of structs for traversing. It also expects a query of one or more RCTL formulas in JSON format. Once the NOC and formulas are obtained, the simulator traverses the state space checking whether the formulas are satisfied while also analyzing the NOC for faults as described in 2.1. HMKAAL completes by producing a result, which reports whether the query is satisfied or inconclusive. Furthermore, a trace and warnings such as data races and non-determinism are reported.

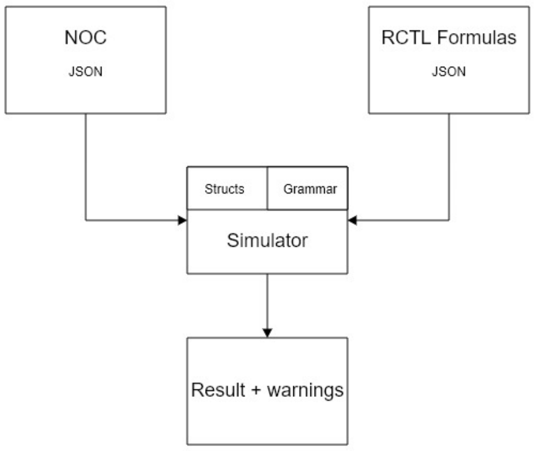

FIGURE 4 architecture of HMKAAL

5.4.3 | Algorithm

The algorithm for simulating a NOC is seen in Algorithm 1. The algorithm presents the simulation of a NOC, i.e. SafeCon III, and the property checking along the way.

If the algorithm returns *true* it shows that the $NOC \models \psi$ assuming ψ is the RCTL formula to be satisfied. However, if the algorithm does not return *true* it does not necessarily show that the $NOC \not\models \psi$, as the algorithm only simulates a single run as compared to a full exploration of the state space.

5.5 | Algorithm modifications

In addition to the NOC and Queries in JSON format we provide multiple CLI options to make the simulator as useful as possible for HMK. These options include the possibility to change the number of Ticks and Tocks, as well as different quality of life options such as specifying a number of hours to simulate and setting initial random values. Additionally, we add an option for specifying the seed to be used in any randomly generated information in the simulation for reproducibility.

Some of these options would change the algorithm slightly, but are left out for simplicity. In addition to the options we have added a file in which ranges for given hardware variables can be specified. For example, any variable in SafeCon III with type analog input will have range [0,255], but may in reality only receive a value in range [0,2]. This can greatly increase the probability of traversing all edges, thereby exploring more of the state space per simulation and increasing the probability that we can verify the given queries. We will expand on the effect of this in Section 6.

Additionally, we allow for multiple queries. This will obviously change the algorithm, as there is thus an array of ψ instead of a single ψ. The modification to the algorithm would then be to have an array of bits that has a length corresponding to the number of ψ. We would then flip the bits accordingly, as the formulas are satisfied and in the end return the array of bits.

Algorithm 1 Pseudocode for simulation of some NOC, N, as described in Section 3.1

```
Let $\langle S, s_0, \rightarrow \rangle$ be a transition system for some N
Let $\exists\Diamond\varphi$ be a formula to be satisfied
Let $s = s_0$ be the current state
Let $Tick\_interval$ be an interval of Ticks that decide an amount of Tick transitions before a Tock. [100,200] as an example.
Let $Ticks$ be a random number in $Tick\_interval$
Let $Tick\_counter = 1$
Let $Tocks$ be an amount of Tocks to occur before stopping the simulation
Let $Tock\_counter = 1$

while $Tock\_counter <= Tocks$ do
    if $s \models \varphi$ then
        return true
    end if
    if $Tick\_counter == Ticks$ then
        Let $v = \mathcal{T}(v)$ in $s = (\overline{\ell}, v)$                ▷ Tock step
        $Tick\_counter = 0$
        $Tock\_counter = Tock\_counter + 1$
        $Ticks$ = random number in $Tick\_interval$
    end if
    Let $s = (\overline{\ell'}, v')$ according to the Tick transition relation in Section 3.1
    $Tick\_counter = Tick\_counter + 1$
end while
```

5.6 | Output trace

In addition to the array of bits, we also output a file containing the trace that the simulator executed. More specifically, the trace consists of a vector of simulations each containing; a seed, a result, Ticks and Tocks. The result will contain whether the formula was satisfied or not. A Tick contains the following:

- Vector of component information which contains the following:
  - Current state
  - Next state
  - Local variable values
  - Information about non-determinism in the transition
- Data race which contains two elements:
  - Variable being written to
  - Vector of components that writes to the variable

Whereas a Tock consists of all input modifications to the unmodifiable variables. As such, it is possible to see all values and the state of the components at any given Tick/Tock combination. Additionally, with the seed it is possible to recreate a simulation, and thus be used to perform a form of regression testing to test if new functionality changes the outcome of the simulation. This output also satisfies some of the requirements set in Section 2.1, such as detection of non-determinism and data races.

5.7 | Data race detection in HMKAAL

As mentioned in Section 5.2.1, the handling of data races in HMKAAL does not correlate with the semantics presented in Section 3.1. The semantics presented in Section 3.1 describes *correct* runtime semantics of a NOC i.e. how a NOC *should* behave. However, SafeCon III does not halt on these types of errors and HMKAAL emulates the execution of SafeCon III. As such, we instead simply detect the data race, report a warning and continue execution. We handle the data races by ensuring that all reads use the valuation of the given variable *before* the Tick transition i.e. read before write, and for multiple writes we assign the latest write value to the variable.

6 | EXPERIMENTS

In this section we perform benchmark tests on different parts of the system to determine performance. Specifically, we measure the time of a Tick and the time of evaluating a query known to be satisfied to show how the randomness in the simulator can have a large effect on execution time. We leave out the time of a Tock as the time it takes for a single Tock is less than one millisecond. First, we calculate the time it takes to do a single Tick. The result can be seen in Table 1 and are calculated from 250 Ticks.

In the following experiments we run with 10 Ticks pr. Tock and 25 Tocks pr. reset. Query1 is $\exists\Diamond L2157$ and Query2 is $\exists\Diamond L2157 \wedge L171$ (The query in HMKAAL would be: E<> L2157 && Woc_18_7.L171, where Woc_18_7.L171 is a location in an component instance). These are actual locations in the SafeCon III system. Both of these are satisfiable and are dependent on hardware variables to be true. The experiments are performed on a machine with an AMD Ryzen 5 3600 6-core processor, 16 GB RAM and an INTEL SSDPEKNW512GB harddisk.

For Query1 and Query2 we perform 200 runs for each query through a python script. The use of a Python script may result in some overhead, however, it is probable that this is how HMKAAL would be used in an actual case. The computation times can be seen in Table 2. The computation times shown includes time for preprocessing and time spent writing to files.

	Tick (ms)
Min	7.08
Avg	7.71
Max	8.24

TABLE 1 Computation time for a single Tick

	Test Query1 (s)	Test Query2 (s)
Min	5.07	5.16
Avg	5.36	6.01
Max	7.23	10.09

TABLE 2 Computation time for query evaluation

In all experiments the time of preprocessing is around 4.7 seconds, without much variance. From the results we see that a single Tick is reasonably fast and the time for a single Tick is relatively constant across many Ticks. For the query results it can be seen that the randomness in the simulator plays a large role in the computation time. This is the case as the average time in the two queries differs, but the minimum time is almost the same. It also seems that adding multiple requirements for a query to be satisfied, will take longer on average which is to be expected.

Effect of ranges file

In Section 5.5 we stated that we would show the effect of the ranges file on the searched state space. We conducted experiments for the amount of locations visited without ranges, with ranges and with ranges and random initial values. For each configuration we made 20 runs with the following options on an unsatisfiable query: 10 Ticks per Tock, 25 Tocks in total and run for 30 seconds. The results can be seen in Table 3. As seen the ranges file can have a positive effect on how many locations are visited

	Visited w.o. ranges	Visited w. ranges
Min	297	373
Avg	344	391
Max	367	404

TABLE 3 Effects of using ranges file

in a given amount of time, which is to be expected. Of course given enough time, simulating both with and without ranges could visit every reachable location.

Lastly, we want to test the difference between running different instances of HMKAAL each evaluating a single query and a single instance of HMKAAL evaluating all queries at once. We have the following seven queries:

- $\exists\Diamond L1265 \land CD\_flowmeterrequired \neq REQ\_CD\_flowmeterrequired$
- $\exists\Diamond L2157 \land CD\_tankname = "Invalid"$
- $\exists\Diamond L32 \land hrgs2\_in.1 = true$
- $\exists\Diamond L171$
- $\exists\Diamond L114$
- $\exists\Diamond L21$
- $\exists\Diamond L83$

These are all actual queries for SafeCon III i.e. all locations and variables exist in SafeCon III. The results can be seen in Table 4. In Table 4 we see that running multiple queries on the same thread or different threads, makes little difference. We theorize,

	Parallel single query (s)	Multi query single thread (s)
Min	7.99	6.63
Avg	26.75	26.82
Max	109.08	124.46

TABLE 4 Comparison of Parallel single query and Multi query single thread

however, that given even more queries the most optimal solution would be to evaluate multiple queries on several threads. E.g. given 64 queries and an 8 core processor the best solution would be to make 8 threads each evaluating 8 queries.

An even better solution would be to let the threads communicate i.e. start multiple threads each evaluating all the queries, but they would share the bit array mentioned in Section 5.5. As such, each query would only need to be evaluated by a single thread instance.

6.1 | Impact and HMK Response

As HMKAAL is a tool made for HMK, we have had regular meetings with HMK. In these meetings we discussed problems that arose during implementation of HMKAAL. This led to the discovery of several occasions of non-determinism and data races in SafeCon III while testing HMKAAL. Furthermore, during development we found an error in HMKs compiler, which was an unintended way of providing value to HMK. This shows that not only formal verification, but also the process of implementing a verification tool, can give a deeper overall understanding of the system, thus providing value to development of software.

When HMKAAL was ready for use, HMK was then given the possibility to try the verification tool and gave following response regarding their use and the future of HMKAAL in HMK:

"As the tool currently only support executing batches of queries via a terminal, we aim to create our own frameworks surrounding HMKAAL to improve the usability and effectiveness of the tool.
First, we aim to integrate HMKAAL directly into H-UPPAAL, such that our developers can create, run and view queries while programming on their own machine. This we expect to greatly improve our effectiveness in debugging issues and to allow automatic testing of new functionality by requiring queries to return specific results.
Second, we aim to integrate this form of testing into our quality assurance process. Each new feature developed in H-UPPAAL must pass all of the developed tests in addition to be code reviewed by another developer. These tests would be run on an external server in parallel, allowing timely feedback. If done properly, this makes it possible to have automatic regression testing of several core system functionality as part of our development, given tests are developed."

7 | CONCLUSION

In this paper, we implemented a verification tool for the industrial case provided by HMK, the SafeCon III system. We found that SafeCon III can be formalized as a network of components, in order to check for reachability. However, if SafeCon III without modifications was to be model checked it would result in a state space explosion. We researched different options for reducing the state space, but came to the conclusion that the options would either require too much time or would not reduce the state space enough, thus still resulting in state space explosion. We therefore decided to construct a simulator to still provide some value for HMK. With the simulator it is still possible to verify that a location or variable value *is* reachable, however, it is not possible to verify that it *is not* reachable. Furthermore, simulation provides the ability to detect data races and non-determinism in the system, however, it cannot be guaranteed that all such problems are found. The simulator also has the limitation that it cannot guarantee mutual exclusion, which was one of HMKs wishes. As such, two of the three features found in cooperation with HMK were satisfied. The report from HMK showed that HMKAAL is a useful tool for HMK. HMK has reported that they will work to integrate HMKAAL directly into H-UPPAAL in order to increase usability of HMKAAL. Furthermore, they expect this will greatly increase the effectiveness in debugging issues and it will allow automatic testing of new functionality by requiring specific results from queries. They expect the use of HMKAAL in their pipeline will reduce the amount of errors and unexpected behaviour in SafeCon III.

Already during the testing of HMKAAL errors were found in SafeCon III in the forms of non-determinism and data races. Additionally, an error was found in HMKs compiler during implementation. This proves that formal verification, as well as implementing a verification tool, can give value in terms of awareness about the system and overall understanding of the system.

7.1 | Future Work

This paper implements a simulation of the SafeCon III system, provided by HMK. There were multiple ways to approach this industrial case besides the approach we chose. An approach could be to use the data structures BDDs and IDDs mentioned in 5.3. It has been shown that using BDDs, compositionality and dependency analysis, it is possible to model check a state space with the size of 10^{476}[19]. This could potentially reduce the state space such that model checking would be possible instead of simulation. If this was the case, the grammar could also be extended to handle the operator $\forall$ and the path quantifier $\square$. The path quantifier $\square$ would allow testing if a property holds in every state in a path. The operator $\forall$ allows to test if a property holds in every path. Including these operators would make it possible to answer queries such as whether a variable is always true: $\forall\square var = true$ or if a variable eventually becomes greater than 4: $\forall\Diamond var > 4$.

As the possibility of model checking is uncertain, an option to make the simulator more efficient in terms of searching the state space would be guided or targeted search. One such solution would be to adjust the value of some hardware variables such that they satisfy guards on edges that are rarely traversed, thus increasing the probability of traversing a larger part of the state space.

Another potential shortcoming is in the way we model SafeCon III as a NOC. In the NOC semantics we do not introduce formal clocks, as seen in many similar formalisms such as Tick Tock Automata[11] or timed automata[20]. Both of these make use of formal clocks to represent time in the system. We decided against this way of representing time due to time constraints of the project as well as the limited use of timers in SafeCon III. The timers in SafeCon III are mostly used as a kind of timer to reflect the time it takes for the hardware to perform different tasks. This does, however, mean that the timers in HMKAAL are deterministic in nature, and thus grow with a fixed time of 1 millisecond per Tick transition. While it is unlikely, this could mean some locations go unvisited.

If clocks were to be represented formally, a data structure such as Difference Bound Matrices (DBM) would be promising. This could be further extended to also incorporate the notion of ranges described in Section 5.2.3, as these representations are alike.

References

1. Pakonen A, Tahvonen T, Hartikainen M, Pihlanko M. Practical applications of model checking in the Finnish nuclear industry. *10th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies* 2017: 12.
2. HMKBilcon . SafeCon. https://hmkbilcon.com/da/tankbilsloesninger/safecon-system/; 2020.
3. HMKBilcon . Vores Historie. https://hmkbilcon.com/da/historien; 2020.
4. Mouritzen NK, Jensen RH. Introducing Hierachies to Networks of Timed Automata: H-UPPAAL a New Integrated Development Environment for Model Checking. http://people.cs.aau.dk/~ulrik/HUppaal/H_UPPAAL-9sem.pdf; 2016.
5. Larsen K, Pettersson W, Yi W. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer* 1997; 1.
6. Wikipedia . List of model checking tools. https://en.wikipedia.org/wiki/List_of_model_checking_tools; 2020.
7. Klabnik S, Nichols C. The Rust Programming Language. https://doc.rust-lang.org/stable/book/#the-rust-programming-language; 2020.
8. Behrmann G, David A, Larsen KG. A Tutorial on Uppaal. *Formal Methods for the Design of Real-Time Systems* 2004; 3185: 38.
9. Holzmann GJ. The Model Checker SPIN. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* 1997; 23(5): 17.
10. Fantechi A, Gnesi S. On the Adoption of Model Checking in Safety-Related Software Industry. *Lecture Notes in Computer Science* 2011; 6894: 14.
11. Gitz-Johansen A. Tick Tock Automata - a Modelling Formalism for Real World Industrial Systems. 2020.
12. David A. *Hierarchical Modeling and Analysis of Timed Systems*. PhD thesis. Department of Information Technology, Uppsala University, 2003.
13. Baier C, Katoen JP. *Principles of Model Checking*. Cambridge, Massachusetts: MIT Press . 2008.
14. McMillan KL. Symbolic Model Checking. *Verification of Digital and Hybrid Systems* 2000; 170: 117-137.
15. Bengtsson J. *Clocks, DBMs and States in Timed Systems*. PhD thesis. Department of Information Technology, Uppsala University, 2002.
16. Alur R, Brayton RK, Henzinger TA, Qadeer S, Rajamani SK. Partial-order reduction in symbolic state space exploration. *Computer Aided Verification* 1997; 1254: 340-351.
17. Strehl K, Thiele L. Interval Diagram Techniques and Their Applications. *International Workshop on Post-Binary ULSI Systems* 1999: 23-24.
18. Behrmann G, Bengtsson J, David A, Larsen KG, Pettersson P, Yi W. UPPAAL Implementation Secrets. *Lecture Notes in Computer Science* 2002; 2469: 3-22.
19. Lind-Nielsen J, Andersen H, Hulgaard H, Behrmann G, Kristoffersen K, Larsen K. Verification of Large State/Event Systems Using Compositionality and Dependency Analysis. *Formal Methods in System Design* 2001; 18: 5–23.
20. Alur R, Dill D. A Theory of Timed Automata. *Theoretical computer science* 1995; 126(2): 183-235.

How to cite this article: A. Bilgram, E. Ernstsen, P.B. Greve, P.H. Taankvist, and T. Pedersen (2020), HMKAAL