AALTITOAD
A Tick Tock Automata Verification Engine

Asger Gitz-Johansen
Department of Computer Science at Aalborg University
Denmark
Email: agitzj16@student.aau.dk

Abstract: *We present an extension of a Model Based Development (MBD) pipeline, used and developed in collaboration with the tanker truck production company HMK Bilcon A/S for their SafeCon III product production. We introduce an associated industrial modeling formalism called Tick Tock Automata (TTA) and the Computation Tree Logic (CTL) satisfaction relation, so model checking can be performed. We implement a model checking tool called AALTITOAD that performs reachability analysis on TTAs. We compare the performance with other tools compatible with TTA models and show that the performance of AALTITOAD is drastically superior to compared solutions. The tool is carefully developed to have trace equivalence with the SafeCon III product.*

I. Introduction

With the rising power of embedded computers, software used in industrial safety-critical scenarios are getting more and more complex. This increasing complexity can provide room for faults and exotic errors. These can hide deeply within the source code, which can lead to everything from inconveniencing to potentially devastating failures. The famous example of ESAs (the European Space Agency) Ariane 5 rocket exploding due to a programming error [1], and the United Kingdom's automatic ambulance dispatch service spiraling out of control [2], are but a few examples. We believe that utilizing more obvious humanly readable representations, such as a graphical model of the system, will aid the discovery of such errors. Combined with formal methods and verification and validation through traditional unit and user testing, the confidense in the correctness of a software-intensive system should increase drastically. Modeling languages are not a new concept to most developers. Languages such as UML (Unified Modeling Language) [3] has been an industry standard for modeling use cases, functional specifications and systems interaction ever since version 1.1 was introduced in 1997 [4]. More formal ways of modeling systems and their behaviour are also used, such as Petri Nets [5, 6] (PN) or Timed Automata (TA) [7, 8, 9].

Model Based Engineering (MBE) is the practice of using humanly understandable models as a base for easier communications between departments as well as individual developers. In this paper, we will refer to Model Based Development (MBD) as the act of applying the principles of MBE only in the context of software development. This distinction is made because, as Brian Selic explains in [10], the software development industry still has cultural, social as well as technical barriers to break down before mass recognition of MBE. In order to enable a MBD pipeline, a set of tools that can facilitate the modeling, code generation, testing and formal verification activites, needs to be developed. This paper is a continuation of a previous collaboration with tanker truck production company HMK Bilcon A/S [11], where we laid the theoretical foundations for such a toolchain to be developed. In this work we enhance their development pipeline with formal methods and automatic verification and validation, by developing a model checker tool called AALTITOAD (**Aal**borg **Ti**ck **To**ck **A**utomata vali**d**ator).

The paper is structured as follows: Section II introduces the case with HMK Bilcon A/S, their SafeCon III product, MBD pipeline and goes into detail about how such pipelines can be enhanced with formal methods. In Section III we explore the various tools that can enable enhanced MBD pipelines. We then introduce the notion of Tick Tock Automata (TTA), their semantics and how to perform model checking on them in Section IV. We then present how the AALTITOAD tool functions in Section V. In Section VI we introduce competing TTA analysis tools, and in Section VIII we compare them with AALTITOAD. We then conclude on our findings in Section IX, and discuss future work in Section X.

II. SafeCon III & HAWK

HMK Bilcon A/S is a tanker truck production company. Their trucks are equipped with a complex piping and pump system for transferring product between tank compartments and for loading/unloading. This system is controlled electronically via an onboard computer system called SafeCon III [12]. The system is developed in a MBD pipeline. This means that the

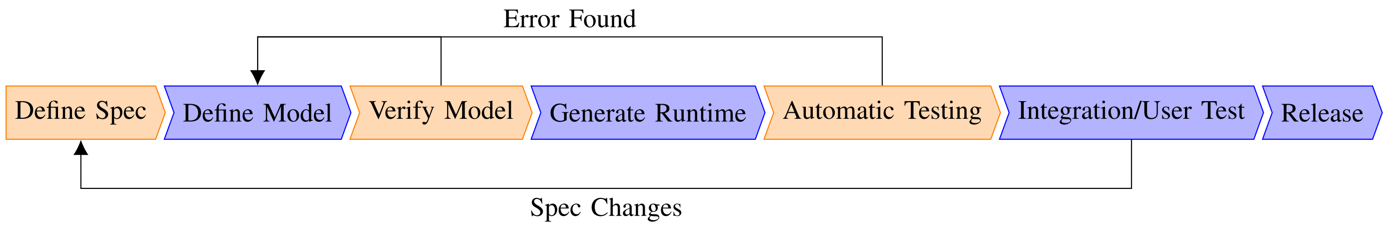

Figure 1. Process Flow Chart for a Verification Enhanced MBD pipeline

system's features are modelled by non-programmer domain experts in a custom built user-friendly graphical modelling language, which is then compiled into an executable. In the case of SafeCon III, the model is translated into a state-machine format, that is executed by an environment called HAWK (Hardware Abstraction With Knowledge). Throughout this paper, we will refer to the model itself as the SafeCon III model and the runtime that hosts and executes the model as HAWK.

A. *Enhancing MBD pipelines with verification*

MBD pipelines are environments of tools, where software engineers do not write the code that dictates the behavior of the product directly. Instead they model the system in a formal, usually preferably graphical, modelling language and then use a generator module to create a shippable runtime. As previously mentioned, HMK Bilcon A/S has employed such a development pipeline in the development of their SafeCon III product. Such pipelines are ripe to be enhanced with verification and automatic testing techniques with minimal to no intervention of the every-day development. The process flow chart in Figure 1 shows an overview of how such MBD pipelines can be enhanced. The blue segments are the regular activities of an MBD pipeline and the orange segments are the new activities proposed in this paper.

The *Automatic Testing* segment refers to an activity, where a tool, in the style of TRON [13], will simulate the model in parallel with the runtime. This approach is also known as Model Driven Testing (MDT). With the knowledge of how the model is supposed to react and function, the tool can provide automatic input and verify the expected outputs. If a fault is found during this step, but not in the verification step, it indicates that the generator module is not implemented correctly. We will not be exploring this part of the pipeline in depth in this paper, but it can provide vital feedback for both the generator and model developers.

Compared to traditional development pipelines where engineers develop an application, ships it and then supports it by changing the source code, having automatic proofs of proper product behavior on every release can be a potential selling point for the product. Especially if the product is deployed in a safety-critical environment, such as the case is with the SafeCon III system. We therefore explore the possibility of enhancing HMK Bilcon's development pipeline with a verification engine.

III. Related Work

In this section we present a subset of the various industrial development tools that can enable MBD pipelines. Ranked from most relevant to the SafeCon III case to the least, we will explore their available modeling tools and some use cases where an enhanced MBD pipeline have been applied.

A. *PLC Verification*

Programmable Logic Controllers' (PLC) way of reasoning and reacting to external stimuli is very similar to that of SafeCon III. This means that the SafeCon III model might be translatable to a PLC language and then analysed from there. With this posibility in mind we explore the various model checkers available for PLC systems.

Frey and Litz [5] translate PLC control logic to a specialized Petri Net (PN) format, that are similar in structure to workflow nets [14], Weng and Litz [15] extends this by presenting a verification approach using LTL (Linear Temporal Logic) to specify expected behavior and analyse PN representations of the control logic. The nets from this approach can be verified via a PN-based model checker, such as TAPAAL [16] or ITS-Tools [17] (Model Checker tools that accept PNs compete on a yearly basis in the Model Checking Contest (MCC) [18]). However, this approach does not provide any out-of-the-box tool for supporting the automatic testing step.

B. *jABC*

Developed at the technical university at Dortmund, the Java Application Building Center [19, 20] (jABC) is an integrated development environment (IDE) that offers a modeling based approach to systems programming. The models are written in a UML (Unified Modeling Language) dialect. The IDE can be extended with an array of plugins which includes a game-based model checker called GEAR [21, 22], and a code generation module called Genesys [23]. With both of these plugins installed, jABC can enable a complete enhanced MBD pipeline. Steffen et. al shows how to use jABC in a Model Driven Development (MDD) manner in [24]. Switching to an MBD pipeline should be relatively easy since the main differentiating factor between MDD and MBD is the ability to perform code generation. However jABC is completely closed development environment, meaning that if HMK Bilcon were to replace their existsing toolchain with jABC, they would have to reimplement most if not all of their lower-level driver code. Not to mention the models should be translated into a format accepted by jABC, effectively forcing the team to start over again.

C. Esterel

Esterel is a textual programming language designed for development of reactive systems [25]. Defined in 1998 by Gerard Berry [26] the language is compiled into finite state machines (FSM), which provides well understood theoretical semantics and enables support for formal verification techniques. With little translation, verification tools based on Binary Decision Diagrams (BDDs) such as NuSMV [27], PRISM [28] or the Esterel specific verifier: Xeve [29], can perform model checking analysis on the FSM results of the Esterel compiler. With this, the technological stage on which enhanced MBD pipelines can live is set. As an example the European Space Agency (ESA) used Esterel together with a verification tool called FORMID [30, 31] for their ExoMars mission. The Esterel code was translated into executable instructions for the robot, and the FORMID tool analyzed the FSM accordingly. The model is verified even more by performing automatic TRON style testing as described in Subsection II-A. Since Esterel is a textual language, it can be intimidating for non-programmers to use and therefore it is not suitable in the case of HMK Bilcon.

D. IAR Visual State

IAR Visual State [32] is a proprietary State Charts [33]-based IDE, featuring automatic code generation and a BDD based model checking module. The tool boasts an integrated debugger, that can help finding runtime faults, but no automatic testing facilities are available. This means that this IDE can only partially support an enhanced MBD pipeline.

E. MBSE or MBD

Model Based Systems Engineering [34] (MBSE)[1] is the practice of designing, implementing, verifying and validating systems through a model based approach. This sound very similar to MBD, however as we describe it MBD is specifically for software development and the models are not neccesarily abstract enough to be meaningful to people other than system experts. In contrast MBSE affects everyone in the process of creating the product(s). Everyone from user experience designers to corporate executives, even including software engineers. NASAs Jet Propulsion Laboratory (JPL) uses a modeling language called SysML [35] to enable an MBSE workflow [36, 37]. This minimizes their manual documentation labor and assists in better communications between the various teams.

F. Subconclusion

The most promising of the presented tools is jABC. It supports every step in the enhanced MBD pipeline. In fact, if used correctly jABC can be used to enable the much broader MBSE pipelines. But as the SafeCon III project would have to be mostly, if not completely, reimplemented, only marginal gains will be obtained. We will therefore focus on implementing a model checking engine that can enhance their existing MBD pipeline with verification and automatic testing.

IV. Tick Tock Automata

Tick Tock Automata (TTA) is an automata based modelling formalism that is derived from a custom language in an industrial use case. Most of this subsection is derived from the original paper introducing TTAs [11]. As a preliminary, we define the following sets and functions:

- $\mathbb{B} = \{tt, ff\}$ is the boolean value domain,
- $\mathbb{V} = \mathbb{R} \cup \mathbb{B}$ is the set of variable values,
- A clock valuation $\mu : C \to \mathbb{R}$ which is a mapping from clocks to reals, and
- A variable valuation $v : V \cup \Omega \to \mathbb{V}$ is a mapping from variables (internal as well as external) to variable-values.

We denote by $\mathbb{R}^C$ the set of all clock valuations, i.e. $\mu \in \mathbb{R}^C$, and we denote by Λ the set of all variable valuations, i.e. $v \in \Lambda$.

Definition 1.1. *Tick Tock Automata (TTA)*
A Tick Tock Automata A is a tuple: $A = (L, l_0, C, V, \Omega, E, v_0, \tau)$ where:

[1]Also sometimes reffered to as Model Based Systems Development (MBSD)

- L is a set of locations,
- $l_0 \in L$ is the initial location,
- C is a set of clocks,
- V is a set of internal variables,
- Ω is a set of external variables,
- E is a set of edges of the form $E = L \times G \times U \times 2^C \times L$,
- $v_0 \in \Lambda$ is the initial variable valuation,
- $c_0 \in \mathbb{R}^C$ is the initial clock valuation, where $c_0(x) = 0$ for all $x \in C$, and
- $\tau : V \cup \Omega \to 2^{\mathbb{V}}$ is the type function denoting the domains of variables.

For simplicity, we write $l \xrightarrow{g,u,r} l'$ if there exists an edge $\langle l, g, u, r, l' \rangle \in E$, where $g : \Lambda \times \mathbb{R}^C \to \mathbb{B}$ is a boolean guard function, $u : \Lambda \to \Lambda$ is an update function and $r : \mathbb{R}^C \to \mathbb{R}^C$ is a clock reset function. We say that a variable valuation function $v \in \Lambda$ is well typed if and only if $v(x) \in \tau(x)$ for all $x \in V \cup \Omega$.

The semantics of a TTA is split into two steps: A Tick-step and a Tock-step. The Tick-step semantics is defined as a transition system $\langle S, s_0, \to_{tick} \rangle$, where $S = L \times \Lambda \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, v_0, c_0)$ is the initial state and $\to_{tick} \subseteq S \times S$ is the transition relation. A transition $(l, v, c) \to_{tick} (l', v', c')$ is valid iff: $\langle l, g, u, r, l' \rangle \in E$ and $g(v, c) = tt$, $v' = u(v)$ and $c' = r(c)$. In other words the guard g is has to be satisfied, the update function should result in the new variable valuation and the clock reset function should result in the clock valuation c'. Additionally, we say that if no edges are valid (i.e. no guard is satisfied) in the current state s, no state change will happen but the Tick-step is still performed i.e. In state $s = (l, v, c)$, if for all $e = \langle l, g, u, r, l' \rangle \in E$ are not satisfied $g(v, c) = \mathit{ff}$, then $s \to_{tick} s$ is valid.

The Tock-step semantics is defined as a sequence of two functions $\Gamma : \Lambda \to 2^\Lambda$ and $\gamma : \mathbb{R}^C \to 2^{\mathbb{R}^C}$. It does not change the location of a TTA, but it influences the variables and clocks respectively:

$$\Gamma(v) = \{v' \in \Lambda \mid \text{if } x \in \Omega\text{: } v'(x) \in \tau(x) \quad \text{otherwise: } v'(x) = v(x) \text{ s.t. } v'(x) \in \tau(x)\} \tag{1}$$

The Γ function has the possibility of changing the value of all external variables Ω to any arbitary variable-value. Picking the new valuation is then just a non-deterministic choice. This behavior is supposed to model the unpredictability of the external world, where an external variable can change to any value on a whim.

$$\gamma(c) = \{c' \in \mathbb{R}^C \mid c'(x) = c(x) + d \text{ for all } x \in C, \text{ and for some } d \in \mathbb{R}_0\} \tag{2}$$

The γ function delays all the clocks in the system by some real value d. Just as with the Γ funciton, picking the new clock valuation is a non-deterministic choice. Picking a valuation from these two functions together in succession gives us the semantics for Tock-step: A Tock-step is a transition relation $\to_{tock} \subseteq S \times S$, where $(s, s') \in \to_{tock}$ if and only if s is of the form $s = (l, v, c)$ then $s' = (l, v', c')$ where $v' \in \Gamma(v)$ and $c' \in \gamma(c)$.

Additionally, we will denote $s \to_{tt} s'$ to mean either a Tick-step transition or a Tock-step transition from a state s to state s'. The overall semantics of a TTA is a simple cycle of Tick-steps and Tock-steps starting with a Tock-step. This means that the system cannot take two same-step transitions directly after another i.e. $s \to_{tick} s' \not\to_{tick} s''$ or $s \to_{tock} s' \not\to_{tock} s''$. This may seem restrictive, but considering that any Tock-step transition is valid (even $s \to_{tock} s'$, where the state does not change, i.e. $s = s'$), and that $s \to_{tick} s'$, where $s = s'$ is also a valid transition, this will ensure no deadlocks. Intuitively, this can be seen as the system gets a *chance* to change the state in a Tick-step, and a *chance* to change the state in a Tock-step. However, we say that if no edges are valid in the current state s, and no Tock-step can enable any new edges, we say that the TTA has reached a functional deadlock. Put more formally, we say that a state $s = \langle l, v, c \rangle$ is deadlocked $\langle l, v, c \rangle \models \textit{deadlock}$ if for all $\langle l, v', c' \rangle \in \{s' \mid s \to_{tock} s'\}$ we have that for all $\langle l, g, u, r, l' \rangle \in E$, $g(v', c') = \mathit{ff}$.

The definiton of a TTA makes an explicit difference between *internal* and *external* variables, this is due to the industry-roots of which the theory originates. External variables represent the sensory input values that the control computer can read from but never explicitly override, and the Γ function simulates the unpredictable behavior of the external world via non-deterministic choice. Note that the semantics of a TTA does not permit time delay in the Tick-step semantics, and that the only way to delay time is in a Tock-step. This is designed to disencourage developers to wait long periods of time in the Tick-step. Most, if not all, real world problems that require waiting for some amount of time are dependent on external events, and therefore the system should not wait to check the external variables.

Definition 1.2. *Semantics of Networks of TTAs*
Having a single TTA running is often not satisfactory for complex use cases. Therefore we also introduce the notion of *networks* of TTAs, that are composed in parallel and interact with eachother through variables:

Let $N = \{A_0, \ldots, A_n\}$ with $A_i = (L_i, l_0^i, C, V, E_i, \Omega, v_0, c_0, \tau)$ be a network of n Tick Tock Automata. Let $\bar{l_0} = \langle l_0^0, l_0^1, \ldots, l_0^n \rangle$ be the initial location vector.

The Tick-step semantics of a network of TTAs are defined similarly to just a single TTA, with the location now being a location vector instead i.e. it is defined as a transtion system $\langle S, s_0, \rightarrow\rangle$ where $S = (L_0 \times \cdots \times L_n) \times \Lambda \times C$ is the set of states, s_0 is the initial state and $\rightarrow \subseteq S \times S$ is the transition relation. But because the clocks, external and internal variables are shared across the network, the transition relation formation rules are more complicated in order to disallow race conditions and ensure atomicity on parallel update evaluation. For the sake of brevity, we refer to the original paper [11] for a more detailed description. The Tock-step semantics of a network of TTAs are identical to that of a single TTA.

A. RCTL Satisfiability

In this section we will define a subset of the Computation Tree Logic (CTL) language, and define the satisfiability mapping for networks of TTAs.

Definition 1.3. *Reduced Computation Tree Logic (RCTL) for TTAs*

CTL is a branching logic that describes conditions over branches of computation in some system. Here we define a subset of the logic in a grammar focusing on safety and reachability.

$$\varphi ::= \forall\Box P \mid \exists\diamond P \tag{3}$$

$$\begin{aligned} P ::= tt \mid ff \mid l \mid e_1 \bowtie e_2 \mid (\neg P) \mid (P_1 \wedge P_2) \\ \mid (P_1 \vee P_2) \mid (P_1 \Rightarrow P_2) \mid (P_1 \Leftrightarrow P_2) \end{aligned} \tag{4}$$

$$e ::= V \mid C \mid \mathbb{V} \tag{5}$$

Where $\bowtie = \{<, \leq, =, \neq, >, \geq\}$ is the various comparison operators and $\mathbb{V}$ is the variable value domain.

We define the expression evaluation function $E : V \cup C \cup \mathbb{R} \cup \mathbb{V} \rightarrow \mathbb{B} \cup \mathbb{V}$ as a mapping of variables, clocks, real values and variable values to boolean or variable values. Given a state in a network of TTAs N with the current variable valuation v and clock valuation c, E is defined as follows:

E Input	Result
$x \in V$	$v(x)$
$x \in C$	$c(x)$
$x \in \mathbb{R}$	x
$x \in \mathbb{V}$	x
$x \in \mathbb{B}$	x

We assume predicates to be well-typed, i.e. if $a \bowtie b$ then $E(a)$ and $E(b)$ belongs to the same domain.

For clarity, we call the φ rules for quantifiers, and the P rules for predicates. We say that, given a state s in a network of TTAs, an RCTL query φ is satisfied $s \models \varphi$ if and only if:

φ Query	Satisfaction Condition
$s \models tt$	tt
$s \models ff$	ff
$s \models l$	$s = (\bar{l}, v, c)$ s.t. $\bar{l} = \langle l^0, \ldots, l^n\rangle$ where $l^i = l$ for $i \in \{0, \ldots, n\}$
$s \models e_1 \bowtie e_2$	Given s, $E(e_1) \bowtie E(e_2) = tt$
$s \models (\varphi_1 \wedge \varphi_2)$	$s \models \varphi_1 \wedge s \models \varphi_2$
$s \models (\varphi_1 \vee \varphi_2)$	$s \models \varphi_1 \vee s \models \varphi_2$
$s \models (\varphi_1 \Rightarrow \varphi_2)$	$s \models \varphi_1 \Rightarrow s \models \varphi_2$
$s \models (\varphi_1 \Leftrightarrow \varphi_2)$	$s \models \varphi_1 \Leftrightarrow s \models \varphi_2$
$s \models \neg\varphi$	$s \not\models \varphi$
$s \models \forall\Box\varphi$	$s \models \neg\exists\diamond\neg\varphi$
$s \models \exists\diamond\varphi$	$\exists s \rightarrow_{tt} \cdots \rightarrow_{tt} s'$ where $s' \models \varphi$

We say that for the $e_1 \bowtie e_2$ rule, $s = \langle \bar{l}, v, c\rangle$ is satisfied if the rule evaluates to tt. Note that the safety quantifier ($\forall\Box$) is the negated version of the reachability quantifier ($\exists\diamond$). Equation 1.3 provides a visual intuition of how the query quantifiers describe predicates across branching paths of execution.

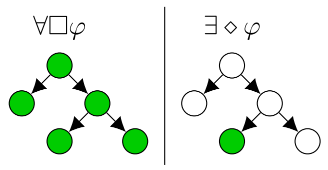

Figure 2. Query quantifiers and how they describe paths to satisfaction. The filled states are those in which the predicate φ is satisfied, and the queries holds for their respective graphs top state.

As an example, consider the network of a single TTA in Figure 3. The predicate $a \geq 3$ would evaluate to ff in the initial state $s = \langle \bar{l}_0, v_0, c_0\rangle$ since the variable valuation $v_0(a) = 0$, but taking the $l1 \rightarrow l2 \rightarrow l1$ cycle three times, the evaluation of $v_3(a) = 3$, making the predicate $a \geq 3$ evaluate to tt. If we were to ask the query in a reachability manner: $\exists\diamond a \geq 3$, it would then be satisfied in the initial state s_0 as we just demonstrated that there does exist a path from s_0 leading to a state where $a \geq 3$ is true.

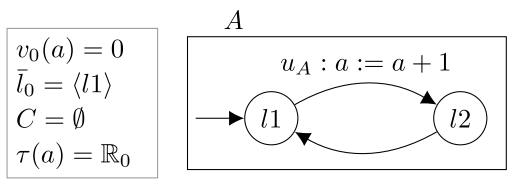

Figure 3. Network of a single TTA A with infinite state space

For simplicity, we write $N \models \varphi$ for some network of TTAs N and a query φ, to mean that N's initial state s_0 satisfies φ i.e. $s_0 \models \varphi$.

V. AALTITOAD Approach

In this section, we discuss the approach of AALTITOAD. Our main algorithm is shown in Algorithm 1. We use a forward reachability search algorithm to answer reachability queries. It operates on a waiting W and a passed P list. On Line 3 the algorithm picks the next state element to look at. Initially this will be the initial state s_0. This line, or Line 6 is where you would implement the search strategy i.e. Breadth First Search (BFS) or Depth First Search (DFS). If the query is satisfied in the picked state, we have found a reachable state and returns true. If not, we add all subsequent states that s leads to. At Line 8 we move the state from the waiting list to the passed list. If there are no more states to pick, we have proven un-reachability and therefore return false.

When verifying safety queries (e.g. $\forall\Box\varphi$), we search for the negated predicate $\neg\varphi$ and then invert the result. i.e. If Algorithm 1 returns false, we have searched through the entire statespace without finding a state that satisfies $\neg\varphi$. Therefore we have proven that φ is invariantly true for all states. If it returns true, we have found a state where φ is not satisfied, and therefore $\forall\Box\varphi$ does not hold, as shown in the semantics.

Algorithm 1 Pseudo code for search algorithm for reachability queries

```
1:  procedure FORWARD REACHABILITY SEARCH($S$,$\varphi$)
2:      $P = \emptyset$ ; $W = \{s_0\}$
3:      pick $s \in W$ do
4:          if $s \models \varphi$ then return $tt$
5:          for all $s \rightarrow_{tt} s'$ do
6:              $W = W \cup \{s'\}$
7:          end for
8:          $W = W \setminus \{s\}$ ; $P = P \cup \{s\}$
9:          if $W = \emptyset$ then return $ff$
10:     end
11: end procedure
```

A. State Space Explosion and Mitigation

As is common in formal verification, when searching through the statespace of networks of TTAs the size of the space tends to explode exponentially. Consider the previous example in Figure 3. The statespace is infinite because each cycle in the TTA will result in a new state where the evaluation of variable a is plus one bigger than the previous state. If we wanted to search for a state where the query $\exists \diamond a < 0$ is satisfied, we would have to search forever, because we will never find a state where variable a is less than its initial value, which in this case is zero.

Another way that networks of TTAs explode the statespace is via the Tock-step semantics. For each external variable in the system N with the domain size of M, the statespace explodes to M^N. Consider the network of three TTAs in Figure 4. If we assume that the domain of the external input variables a, b and c is that of an eight-bit integer, the statespace explodes into $255^3 = 1.65 \times 10^7$. This can be somewhat mitigated by representing the states by considering what the guards actually test for. So, instead of enumerating all 255 possible values of variable a, we would just explore the states where $a \geq 0$ and $a < 0$. This effectively reduces the statespace to $2^3 = 8$.

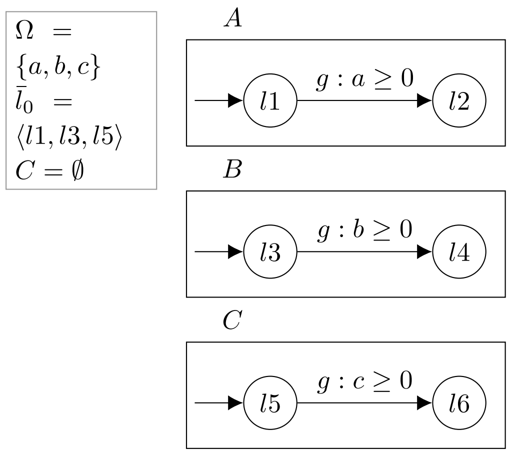

Figure 4. Network of three TTAs A, B and C

Additionally, to further reduce the search space we do not explore those states that are inconsequential to the progress of the individual TTAs. During the statespace search, we enumerate only those states that progress or halt the guards. We notion these guards and the variables they test as *interesting*. To help define this, we introduce a helper function for extracting the set of variables and clocks, that a guard is testing:

$$T : G \rightarrow 2^C \cup 2^\Omega \cup 2^V$$

As a preliminary, we also define a simplified boolean logic grammar for guards:

$$g ::= C \mid \Omega \mid V \mid \mathbb{V} \mid (g_1) \mid g_1 \wedge g_2 \mid g_1 \vee g_1 \mid g_1 \bowtie g_2 \mid \neg g_1$$

Where $\bowtie = \{<, \leq, =, \neq, \geq, >\}$. We then define the function as follows:

T Input	Result
$x \in C$	$\{x\}$
$x \in \Omega$	$\{x\}$
$x \in V$	$\{x\}$
$x \in \mathbb{V}$	$\emptyset$
(g_1)	$T(g_1)$
$g_1 \land g_2$	$T(g_1) \cup T(g_2)$
$g_1 \lor g_2$	$T(g_1) \cup T(g_2)$
$g_1 \bowtie g_2$	$T(g_1) \cup T(g_2)$
$\neg g_1$	$T(g_1)$

Consider the guard $g = x > 10 \land y = z$, testing whether the value of clock x is greater than 10, and that variables y and z are of the same value. We then say that all the variables tested by guard g is: $T(g) = \{x, y, z\}$. By executing the algorithm shown in Algorithm 2 on all locations in the TTA (and subsequently all locations in all TTAs in a network) we find the set of interesting variables for those locations. For networks of TTA, we define $\Pi : S \rightarrow 2^{\Omega}$ as the function for calculating interesting variables. Given a state $s = (\bar{l}, v, c)$ in a network of n TTAs:

$$\Pi((\langle l^0, \dots, l^n \rangle, v, c)) = \bigcup_{i=0}^{n} InterestingVars(l^i, E^i, \Omega)$$

And for singular TTAs in state $s = \langle l, v, c \rangle$:

$$\Pi((l, v, c)) = InterestingVars(l, E, \Omega)$$

We reduce the Γ function to $\hat{\Gamma} : S \rightarrow 2^{\Lambda}$ to accommodate these sets, like so:

$$\hat{\Gamma}(s) = \{v' \in \Lambda \mid \text{if } x \in \Pi(s)\text{: } v'(x) \in \tau(x) \\ \text{otherwise: } v'(x) = v(x) \text{ s.t. } v'(x) \in \tau(x)\} \quad (6)$$

This limits the Tock-step to only changing those variables that have any consequence on the location graph, effectively pruning away all the paths that leads to nowhere. In AALTITOAD, these interesting variables are found, and cached at parse-time and then referred to during exploration.

Algorithm 2 Algorithm to find the interesting external input variables.

```
1: procedure INTERESTINGVARS($l$,$E$,$\Omega$)
2:     $I = \emptyset$
3:     for each $\langle l, g, u, r, l' \rangle \in E$ do
4:         $I = I \cup T(g)$
5:     end for
6:     Return $I \cap \Omega$
7: end procedure
```

B. Trace Equivalence With HAWK

The AALTITOAD tool also provides a simulation mode, where we do not remember which states we have visited, and just execute the model. This mode does not handle the Γ-part of the Tock-step, but it can take in timing information and delay timers accordingly. This mode was developed with the purpose of ensuring that the implementation of the semantics is completely the same as what the HAWK tool is. At the time of writing, we have compared traces with timing information up to five hundred Tick-steps on the provided SafeCon III models, and no deviation has been found.

VI. Existing TTA Tools

AALTITOAD is not the first attempt at enhancing HMK Bilcon's development pipeline with verification. In this section we will briefly introduce the two other attempts. In Section VIII we will compare the performance and accuracy of these tools to AALTITOAD.

A. HMKAAL

Implemented in Rust and developed by Bilgram et al. [38], HMKAAL (**HMK** Bilcon and **Aal**borg University) is a collaboration with a group of Aalborg University Students and HMK Bilcon to develop a verification tool for their SafeCon III models. However, due to the explosiveness of the SafeCon III Models, the HMKAAL tool is only able to provide positive answers to dataraces, non-determinism and reachability queries by simulating the runtime in a random search manner, like AALTITOADs simulation mode.

B. NuSMV

Intregrated directly into the HAWK tool is a translation routine, that translates the input model to a NuSMV [27] model and performs formal verification on that. However, this tool does not provide guarantees towards trace-equivalence with the HAWK runtime.

In the same vein as with the integrated NuSMV solution, it would is possible to simulate a TTA model in another, more mature modelling language such as Timed Automata [7] or Stopwatch Automata [39]. And then perform model checking on those translations with the respective verification engines. There is, however, one caveat to this approach: You would need to be absolutely sure that the translation is semantically correct, and represents the runtime implementation before trusting any of the verification results. Like we do by ensuring trace equivalence with the HAWK system.

VII. SafeCon III Analysis

During development of AALTITOAD, SafeCon III was used as a target model for analysis. By comparing traces with the HAWK system, we discovered implementation errors in how HAWK syncronizes clocks and evaluates guards containing string-typed variables. These bugs were inconsequential for the behavior of the specific SafeCon III model, but were they left unfixed they have potential to result in major timing errors and erroneous behavior. The faults were corrected and integrated immediately after discovery.

The purpose of a model checker is to analyse satisfiability of queries and conformance to expected behavior, and HMK Bilcon A/S have provided some queries for us to check. However, with 150 TTAs running in parallel and 321 external variables the model is too big to realistically perform complex model checking with the current version of AALTITOAD. In fact when attempting to check for deadlocks on the model, after four iterations of the forward reachability algorithm the state space explodes to 9×10^{15} states. This happens even after applying the Tock-step statespace reductions. When investigating the cause, the explosion can be attributed to situations where many TTAs are duplicated and perform guard checks on an array of external variables. Specifically this results in a state where 53 unique external variables are tested and therefore the statespace explodes into 2^{53} different permutations of those guards. The conclusion is that further reduction techniques are required to perform model checking on the SafeCon III model. An obvious candidate is to implement an abstract symbolic representation of the search, perhaps in the form of Trace Abstraction Refinement (TAR) as described in [40] or Partial Order Reduction (POR) as described in [41], but defined for TTAs instead of Petri Nets.

VIII. Experiments

In this section we present and discuss our experimental setup and findings in comparing AALTITOAD with the tools presented in Section VI.

A. NuSMV Setup

For comparing the verification engine of AALTITOAD and the NuSMV translations, we modelled a test set of four versions of Fischer's mutual exclusion algorithm. Figure 5 shows the base TTA used where x is the only timer in the system, pid is the process identifier, k is a constant set to 2, id is a control variable, and ctr is a counting variable used to determine how many parallel TTAs are in $l4$ at the same time. This TTA was then duplicated two, five and ten times and the pid was updated to match the processes' number. We then asked the verification engines to ensure that no more than one process is allowed to be in the critical section at once (in this case $l4$ is the critical section). If the query $\forall\Box ctr \leq 1$ is satisfied, this property of the network is preserved.

$$\begin{tikzcd}
{} \arrow[r] & l1 \arrow[r, "{g: id = -1,\ r: \{x\}}"] & l2 \arrow[d, bend left, "{g: x \leq k \quad u: id = pid}"] \\
& l4 \arrow[u, "{u: id = 0;\ ctr = ctr - 1}"] & l3 \arrow[u, "{g: x > k \wedge id \neq pid \quad r: \{x\}}"'] \arrow[l, "{g: id = pid \wedge ctr = 0 \quad u: ctr = ctr + 1 \quad r: \{x\}}"]
\end{tikzcd}$$

Figure 5. Fischer's mutual exclusion algorithm as a Tick Tock Automata

B. HMKAAL Setup

For comparing the simulation mode of AALTITOAD with HMKAAL, we repurposed a proof of concept demo project provided by HMK Bilcon to be accepted by both tools. The project is just a simple demonstration that pumps product from a chosen tanker compartment where the trucks pump system starts and stops depending on a user button press and some security pins. The demo consists of five TTAs in parallel: One for controlling the pumps, one for controlling the user interface variables and the last three TTAs contains logic for opening and closing the tanker compartments.

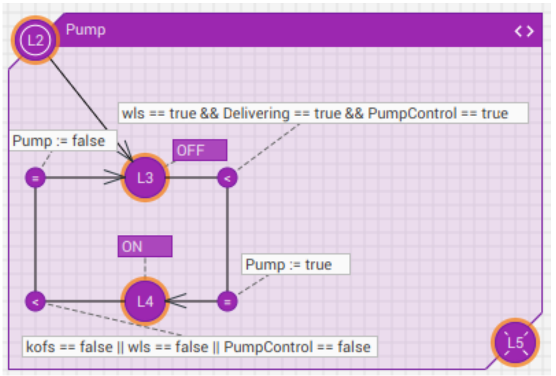
Pump
L2
wls == true && Delivering == true && PumpControl == true
Pump := false
OFF
L3
ON
Pump := true
L4
kofs == false || wls == false || PumpControl == false
L5

Figure 6. Screenshot of the SafeCon III demo TTA in the H-UPPAAL tool, containing the pump logic.

Figure 6 shows the TTA that controls the pump. The `Pump` variable is a variable controlling the actuators necessary for activating the attached pump.

C. Results

All experiments were performed on a PC equipped with an AMD FX-8350 4GHz CPU, 16GiB of memory and running Debian 10 64-bit, kernel version 4.19.0-13-amd64. Figure 7 shows the time it took AALTITOAD and the NuSMV solution to find an answer to the safety query $\forall\Box ctr \leq 1$. Evidently, the AALTITOAD tool is significantly faster in finding the result in all cases. Memory usage never reached the limit of 16GiB, except for the fischer-10 test on NuSMV, where it used up all the memory and was aborted after 20 hours of runtime.

Test Set	AALTITOAD	NuSMV
fischer-2	0.008 seconds	13.14 seconds
fischer-5	0.019 seconds	6 minutes 43 seconds
fischer-10	0.067 seconds	Out Of Memory

Figure 7. Verification engine experimental results

It should be noted, that AALTITOAD is not capable of verifying full CTL properties, whereas NuSMV is. But for reachability search for TTAs it is evidently preferable to use AALTITOAD. With some extra verbosity flags, both tools output how many states they visited during the search and as can be seen in Figure 8, because of AALTITOADs symbolic and interestingness reductions it needed to visit fewer states to reach an answer compared to the NuSMV solution.

Test Set	AALTITOAD	NuSMV
fischer-2	27 states	647 states
fischer-5	43 states	1487 states
fischer-10	93 states	Out Of Memory

Figure 8. Verification engine experimental results. Amount of visited states needed to provide answers.

This indicates that the reason for NuSMVs poor performance in these tests are due to the way that models are translated and represented and not necessarily to the performance of NuSMV itself.

Figure 9 compares the time it takes for the HMKAAL tool and AALTITOAD to explore the SafeCon III demo project for ten, twenty, fifty and sixty five thousand ticks. No tocks have been evaluated, or more formally every Tick-step has been proceeded with a Tock-step $s \rightarrow_{tock} s'$ where the state have not changed $s = s'$. Note that the times include writing a trace file to disk. Because of that we have included Figure 10, which shows the size of those trace files.

Tick Amount	AALTITOAD	HMKAAL
10,000	1.6 seconds	5.6 seconds
20,000	3.37 seconds	10.98 seconds
50,000	8.3 seconds	27.14 seconds
65,000	10.5 seconds	34.7 seconds

Figure 9. Tick evaluation speed results

Tick Amount	AALTITOAD	HMKAAL
10,000	16 MiB	15 MiB
20,000	32 MiB	29 MiB
50,000	79 MiB	73 MiB
65,000	103 MiB	95 MiB

Figure 10. Tick trace file size results

Evaluating ticks mostly boils down to how fast the guard and update expressions are evaluated. At the time of writing AALTITOAD uses a library called "cparse" [42] to evaluate all guards and right-hand-side expressions of updates. As evident, AALTITOAD is generally faster to evaluate ticks than HMKAAL, but the trace files are slightly larger. This is mostly due to AALTITOAD's attempt to keep the traces in a humanly readable format whereas HMKAAL does not.

IX. Conclusion

In this paper we formalized and implemented the semantics of Tick Tock Automata into a verification tool called AALTITOAD. We compared the tool with existing Tick Tock Automata formal method solutions and showed signifigant improvements in calculation speed. During implementation we made sure that the tool is trace equivalent with HMK Bilcon's HAWK system and with that, we found and fixed hidden faults in the HAWK runtime. With AALTITOAD the developers at HMK Bilcon are now, to a limited extent, able to enhance their MBD pipeline. The full SafeCon III model is too large for the current version of AALTITOAD to effectively analyse.

AALTITOAD is licensed with an open source license and is available at www.github.com/sillydan1/AALTITOAD. The tool is designed to be integrated directly into HMK Bilcon's Continuous Integration (CI) solutions, but can still be executed directly from command line.

X. Future Work

As is almost traditional within the field of formal model checking, the main proplem that AALTITOAD faces is the statespace explosion problem. This could conceivably be mitigated by introducing more symbolic representations of the states, effectively batching states into clumps. Trace Abstraction Refinement (TAR) using a SMT solver, such as the Z3 tool [43] is an obvious candidate and we will likely be exploring that avenue as future work. Exclusion of parallel TTAs based on their variable influence could also help reduce the search time significantly, i.e. if a component is in a verifiably seperate graph, we could ignore all graph irrelevant to the query.

At the time of writing, due to time constraints AALTITOAD can only perform reachability and safety analysis and no counter examples are able to be output. We would like to extend the tool to support full CTL and counter example output capability.

Currently the automatic testing facilities i.e. the simulator mode does not run in parallel with the HAWK system. Instead it analyses trace files. Making the tool able to run continuously would provide HMK Bilcon with better feedback on their code generation module and increase confidense in the SafeCon III product.

References

[1] Bashar Nuseibeh. Ariane 5: who dunnit? *IEEE Software*, 14(3):15, 1997.

[2] Anthony Finkelstein and John Dowell. A comedy of errors: the london ambulance service case study. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 2–4. IEEE, 1996.

[3] Object Managent Group. Unified modeling language v2.5.1. https://www.omg.org/spec/UML/2.5.1/, 2017.

[4] Object Managent Group. Unified modeling language v1,1. https://www.omg.org/spec/UML/1.1, 2017.

[5] Georg Frey and Lothar Litz. Verification and validation of control algorithms by coupling of interpreted petri nets. In *SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218)*, volume 1, pages 7–12. IEEE, 1998.

[6] Ajay Kattepur. Workflow composition and analysis in industry 4.0 warehouse automation. *IET Collaborative Intelligent Manufacturing*, 1(3):78–89, 2019.

[7] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[8] Fabio Martinelli, Francesco Mercaldo, Antonella Santone, Christina Tavolato-Wötzl, and Paul Tavolato. Timed automata networks for scada attacks real-time mitigation.

[9] Klaus Havelund, Arne Skou, Kim Guldstrand Larsen, and Kristian Lund. Formal modeling and analysis of an audio/video protocol: An industrial case study using uppaal. In *Proceedings Real-Time Systems Symposium*, pages 2–13. IEEE, 1997.

[10] Bran Selic. What will it take? a view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.

[11] Asger Gitz-Johansen. Tick tock automata.

[12] Safecon system. https://hmkbilcon.com/en/fuel-tank-solutions/safecon-system/. Accessed: 2020-21-12.

[13] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Testing real-time embedded software using uppaal-tron: an industrial case study. In *the 5th ACM international conference on Embedded software*, pages 299 – 306. ACM Press New York, NY, USA, September 18–22 2005. URL http://doi.acm.org/10.1145/1086228.1086283.

[14] Jose Antonio Mateo, Jiří Srba, and Mathias Grund Sørensen. Soundness of timed-arc workflow nets in discrete and continuous-time semantics. *Fundamenta Informaticae*, 140(1):89–121, 2015.

[15] Xiying Weng and Lothar Litz. Verification of logic control design using sipn and model checking: methods and case study. In *Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No. 00CH36334)*, volume 6, pages 4072–4076. IEEE, 2000.

[16] A. David, L. Jacobsen, M. Jacobsen, K.Y. Jørgensen, M.H. Møller, and J. Srba. TAPAAL 2.0: integrated development environment for timed-arc Petri nets. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, volume 7214 of *LNCS*, page 492–497. Springer-Verlag, 2012.

[17] Yann Thierry-Mieg. Symbolic model-checking using its-tools. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 231–237, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46681-0.

[18] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, E. Amparore, M. Beccuti, B. Berthomieu, G. Ciardo, S. Dal Zilio, T. Liebke, S. Li, J. Meijer, A. Miner, J. Srba, Y. Thierry-Mieg, J. van de Pol, T. van Dirk, and K. Wolf. Complete Results for the 2019 Edition of the Model

Checking Contest. http://mcc.lip6.fr/2019/results.php, April 2019.

[19] TU Dortmund. Java application building center. http://ls5-www.cs.tu-dortmund.de/projects/jabc/index.php, 2020.

[20] Ralf Nagel. *Technische Herausforderungen modellgetriebener Beherrschung von Prozesslebenszyklen aus der Fachperspektive: von der Anforderungsanalyse zur Realisierung.* PhD thesis, Citeseer, 2009.

[21] Marco Bakera, Tiziana Margaria, Clemens D Renner, and Bernhard Steffen. Model checking with the jabc framework.

[22] Marco Bakera, Tiziana Margaria, C Renner, and Bernhard Steffen. Property-driven functional healing: Playing against undesired behavior. *10th CONQUEST*, 2007.

[23] Sven Jörges, Tiziana Margaria, and Bernhard Steffen. Genesys: service-oriented construction of property conform code generators. *Innovations in Systems and Software Engineering*, 4(4):361–384, 2008.

[24] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. Model-driven development with the jabc. In *Haifa verification conference*, pages 92–108. Springer, 2006.

[25] Gérard Berry. The constructive semantics of pure esterel. 1999.

[26] G erard Berry. The foundations of esterel, 1998.

[27] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[28] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[29] Amar Bouali. Xeve, an esterel verification environment. In *International Conference on Computer Aided Verification*, pages 500–504. Springer, 1998.

[30] K. Kapellos. Muroco-ii : Formal robotic mission inspection and debugging executive summary. 2005.

[31] Alberto Medina. Ergo technology review (ergo_d1.1). ERGO Consortium, 2016.

[32] Andrzej Wąsowski and Peter Sestoft. On the formal semantics of visualstate statecharts. 2002.

[33] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[34] INCOSE. Model based systems engineering (mbse). https://www.nasa.gov/consortium/ModelBasedSystems, 2020.

[35] Matthew Hause et al. The sysml modelling language. In *Fifteenth European Systems Engineering Conference*, volume 9, pages 1–12, 2006.

[36] Todd J Bayer, Matthew Bennett, Christopher L Delp, Daniel Dvorak, J Steven Jenkins, and Sanda Mandutianu. Update-concept of operations for integrated model-centric engineering at jpl. In *2011 Aerospace Conference*, pages 1–15. IEEE, 2011.

[37] Bjorn Cole and J Steven Jenkins. Connecting requirements to architecture and analysis via model-based systems engineering. In *AIAA Infotech@ Aerospace*, page 1116. 2015.

[38] Alexander Bilgram, Emil Ernststen, Peter B. Greve, Peter H. Taankvist, and Thomas Pedersen. Hmkaal: Formal verification and detection of non-determinism and data races in industrial systems. 2020.

[39] Franck Cassez and Kim Larsen. The impressive power of stopwatches. In *International Conference on Concurrency Theory*, pages 138–152. Springer, 2000.

[40] Franck Cassez, Peter Gjøl Jensen, and Kim Guldstrand Larsen. Verification and parameter synthesis for real-time programs using refinement of trace abstraction. *arXiv preprint arXiv:2007.10539*, 2020.

[41] Frederik M Bønneland, Jakob Dyhr, Peter G Jensen, Mads Johannsen, and Jiří Srba. Stubborn versus structural reductions for petri nets. *Journal of Logical and Algebraic Methods in Programming*, 102:46–63, 2019.

[42] Brandon Amos Vinícius Garcia. cparse. https://github.com/cparse/cparse, 2020.

[43] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.