

# Tick Tock Automata

## a Modelling Formalism for Real World Industrial Systems

Asger Gitz-Johansen

Department of Computer Science at Aalborg University  
Denmark

Email: agitzj16@student.aau.dk

**Abstract:** *Many software intense industry systems are safety critical, meaning even small errors can lead to catastrophic scenarios. Such systems are ripe for reaping the benefits of model checking and verification techniques. We present a real-world inspired reactionary semantics in the form of an automata based theory that we call Tick Tock Automata, supporting verification techniques such as reachability analysis. The theory is based on a specific use case, in collaboration with the truck production company HMK Bilcon A/S, where a domain specific modeling language have been developed to support a Model Based Development (MBD) pipeline.*

### I. INTRODUCTION

Industrial software is typically safety critical, meaning that even minor errors can result in catastrophic scenarios. But software development is a human craft, and humans make mistakes and hence errors will inevitably propagate. The industry already have a lot of ways to mitigate the introduction of errors: Static code analysis tools, clever compilers, unit testing, mutation testing etc. However, in spite of all these steps preventable errors still occur. Many of these errors can be detected before they get deployed if formal verification is adopted into the production pipeline [1]. However, adoption of verification techniques is typically done in a theory-first direction. Where a base theory is developed into a theoretical tool, called a model checking tool, that software engineers are forced to use and adhere to. Whilst many of the tools are free to use and open source, this direction of adoption can be harmful to the efficiency of the pipeline. Engineers have to learn even more standards on top of the ones they already know, opening the possibility of errors in the verification integration. This direction of adoption is not by choice. If a company wishes to integrate verification into their software production pipeline they have to either; tediously model their system in the chosen modeling language and hope they did not make any mistakes (Model Driven Development) or they would have to restructure their entire pipeline and existing toolchain to originate from a modeling language that might not even be expressive enough for their purposes (Model Based

Development). These factors have attributed to the minimal adoption of verification in industry, even if the engineers of the system already know about the potential benefits it can bring.

Some of the only successful applications of formal verification techniques in the production of industrial software is either in avionics [2] where safety is extremely important or in large companies that can have internal support in building the needed infrastructure [3, 4].

The goal of this project is to go in the opposite direction. Using an existing real world system, derive a theory that describes the behaviour and supports verification. We will be focusing on a specific case from petrol tanker truck manufacturer HMK Bilcon, that is already using a Model Based Development pipeline to produce embedded software for their trucks, with a domain specific modeling language.

The rest of the paper is structured as follows: In Section II we go over the differences between Model Driven Development and Model Based Development and argue which one is more suitable for integration of verification. In Section III we survey the various model checking tools available for industrial usage. In Section IV we go into detail about the HMK Bilcon case and in Section V we compare their domain specific modelling language with traditional modelling languages. Section VI and Section VII goes into detail about what a Tick Tock Automata is and how it models the system from the use case. We then provide some techniques to avoid state space explosion when verifying in Section VIII. Then in Section IX and Section X we discuss possible implementations of the theory and draw conclusions on our findings.

### II. MODEL BASED DEVELOPMENT VS MODEL DRIVEN DEVELOPMENT

Although they initially sound incredibly similar, there is a very crucial difference between Model Based Development (MBD) and Model Driven Development (MDD). These development

methods are two different ways of integrating a formal way of specifying system features into software development.

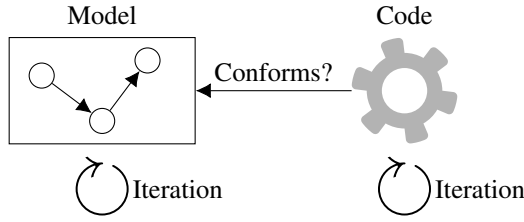


Figure 1. Model Driven Development diagram

Figure 1 illustrates how an MDD pipeline typically works. Developers iterate on the system via the code i.e. traditional development where you write code and add new components etc. Then once a feature is developed, a domain expert has to update the Model of how the feature is expected to behave. Typically the order in which these iterations are done does not matter and both tasks can be done by separate people in parallel. Then when both the model and the code is updated, you would use a Model Driven Testing (MDT) approach to ensure that the expected behaviour and actual behaviour agree.

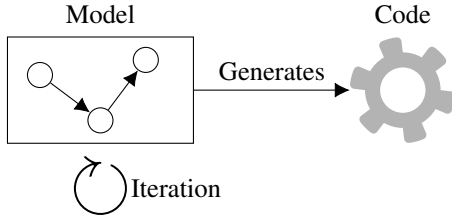


Figure 2. Model Based Development

Figure 2 illustrates how an MBD pipeline would typically work. Here iteration only happens on the model. This is because the model *is* the system. Once a system feature is requested, either a domain expert or a developer then models that feature, presses a button, and then the modelling software generates the code or executable that adheres to that behaviour. The main issues with MBD are that developers now have to be domain experts and domain experts have to be developers. This makes it difficult for companies to adopt an MBD pipeline, unless it is done from the very beginning of the project. There is also put a lot of trust into the modelling software itself, which may have fundamental flaws and may generate incomplete, broken or even worse, wrong code. Whereas a MDD based pipeline can be very easy to integrate into an already existing development pipeline. All you need is a domain expert to model the system features. However, because of the double iteration, there is a lot of manual maintenance

associated with MDD. You might not even be able to hit every single edge case when doing testing of the model and code.

In terms of integrating formal verification into a pipeline, we argue that it is most beneficial in the long run to adopt a Model Based Development pipeline. The verification step can be performed during code generation, or if the verification takes a long time, at selective times. If the code generation is sound and complete according to the modelling formalism used, then the guarantees offered by the verification would also hold for the actual code generated.

### III. MODELLING AND VERIFICATION TOOL SURVEY

In the world of model checkers, a lot of different tools have been developed. Many of them have been used in industrial cases but most tend to be born from academic interests. This chapter will illuminate a set of model checking tools, that is at the time of writing either available for licensing, purchase or free. For the sake of brevity, we have also chosen to omit abandoned projects or projects that are not released yet.

#### A. UPPAAL

The UPPAAL toolbox is a model checker based on the Network of Timed Automata formalism [5]. The tool is developed through a joint effort from Uppsala University in Sweden and Aalborg University in Denmark, hence the name UPPAAL. The tool has been used in many industrial cases. From audio protocols to verifying the correctness of gearbox controllers [6]. UPPAAL can only be used commercially with a proprietary license, which is a common criticism of the otherwise very successful tool. The UPPAAL name has a lot of different spinoff model checkers. Some of which has discontinued development, but some are still relevant and are receiving updates. We have listed the ones that are most relevant to the HMK Bilcon case study.

1) **TIGA**: UPPAAL **T**imed **G**ames, is a version that uses timed game automata to solve reachability and safety properties. This version can be useful in systems with user interaction, or non-predictable hardware sensors.

2) **CORA**: Uppaal **C**ORA (**C**ost **O**ptimal **R**eachability **A**nalys) is a version, that uses an extension of timed automata called Linearly Priced Timed Automata (LPTA) which allows the designer to annotate costs to the model. This cost specification can be useful when finding the most efficient path through some problem.

3) **PORT**: Uppaal **P**ORT is designed with a focus on verifying embedded and real-time systems. It is integrated

with the Eclipse plugin, SAVE IDE (Integrated Development Environment). Modelling in this tool is done via a component-based version of Timed Automata, where system setup is specified in an Architecture Modelling Language.

4) *SMC*: Uppaal SMC (Statistical Model Checking) uses many randomly independent traces of the model to statistically develop a confidence level of certain properties. This approach makes the tool very fast compared to the other tools, however the statistical approach can technically miss some very crucial edge cases. Uppaal SMC is integrated into the main distribution of the Uppaal toolbox.

### B. ECDAR

The ECDAR (Environment for Compositional Design and Analysis of Real time Systems) tool is an open-source effort to provide an automata theory based backend, similar to Uppaal. By using an extended formalism called Timed Input/Output Automata and with an extended theory to accomodate it [7]. The toolchain has its own Graphical User Interface (GUI<sup>1</sup>), that enables composition of automata in a parallel manner, which enables design of small individual components that focus on solving a single task. There are, at the time of writing, three versions of the Ecdar toolchain: Ecdar 0.11 that is implemented in C++ but uses closed source components, Ecdar 2.2 which is licensed under the GPL3 license and is mostly just an alternative GUI for Ecdar 0.11, and Jecdar which is a limited backend reimplementaion done in Java as the name suggests.

### C. H-UPPAAL

(Hierarchical Uppaal) Is an open source GUI that can draw automata in a hierarchical manner. It is designed to use the UPPAAL backend to verify flattened versions of its models, but it does not offer any verification by itself and it cannot generate executable code either.

### D. TAPAAL

TAPAAL is a Free and Open Source (FOSS) model checking toolbox, focusing on the Timed Arc Petrinet formalism[8]. Developed at Aalborg Univesity, the tool supports two variations of verification engines, one that uses continuous time licensed under the GPL v2 license [9], and one using discrete time steps with the BSD2 License [10]. Each of these backends have their individual strengths and weaknesses in terms of speed and performance. TAPAAL does not support

automatic code generation, but tools like *stverif* [11] goes the other way, targeting TAPAAL as a verification engine for verifying already written PLC (Programmable Logic Controller) programs written in the *structured text* language.

### E. SPIN

Developed by Bell Labs in the early 80s, Spin (Simple Promela Interpreter) is focused on concurrent and distributed system verification, and is one of the most popular model checking tools. It has been used for verification of everything from phone calling systems [12] to space exploration probe Cassini's handoff algorithms [13]. The tool uses a textual modelling language called PROMELA (**PRO**cess **ME**ta **L**anguage), which is used to describe the models to be verified. Queries are expressed in Linear Temporal Logic (LTL), enabling developers to ask questions about linear properties of their programs. Perhaps as a result of being so old, Spin is available directly in the stable release of Debian Linux and Ubuntu, and fits very well together with other much used programs such as gcc and yacc. This makes it a lot easier for companies to integrate Spin into their already existing pipelines.

### F. Other Modelling Tools

A lot of industry born modelling tools have the ability to generate code. Such functionality enables more direct control of how the system should function, and can provide a more practical perspective on how to design models.

1) *IBM Rational Rhapsody*: Originally implemented by i-Logix in 1997 [14], Rational Rhapsody offers a modelling formalism called State Charts and supports automatic code generation through the STATEMATE tool [15]. The newer IBM versions support testing profiles being specified via a UML formalism, so developers can perform Model Based Testing (MBT) on their systems. However, no automatic formal model verification is supported in Rational Rhapsody.

2) *Sparx Enterprise Architect*: Owned and developed by Sparx Systems, this tool offers a UML based development and code generation from such models [16]. Developers using this tool are able to automatically generate test cases based on high level concepts such as Use Case models, ensuring proper compatibility with the feature requirements. The tool also provides strong documentation utilities, such that feature-implementation-reports will mostly write themselves. However, there are no utilities for formally verifying your models.

<sup>1</sup>Also sometimes called a *front-end*.

3) *IAR Visual State*: Owned by IAR Systems, IAR Visual State is a modelling tool using a formalism based on David Harel's theory of State Charts [17], it can synthesize executable C code that can be programmed onto an embedded chip. The tool is utilizing a hierarchical modelling syntax [18] and has a built-in verification engine, that can detect potential problems such as two transitions editing the same variable at the same time, effectively producing a race condition. The tool is using a Binary Decision Diagram (BDD) based approach for reachability analysis of such race condition states and if such a state is reachable, the model is considered invalid and the tool spits out an error and a trace for the system designer to handle. Like a traditional Model Checker tool.

#### G. Tool Summary

Table I gives an overview of the presented tools. There are no tools that offer a free license, code generation *and* verification features at the same time. The IAR Visual State toolbox is the tool that comes closest to that description, but the fact that the license is restrictive can prohibit small and upcoming companies to adopt a model based development pipeline. The H-Uppaal tool does not natively provide code generation or verification, because it is really just meant to be a GUI for drawing models that can pass those models on to underlying engines.

Tool	License(s)	Formalism	CodeGen	Verification
Uppaal	Proprietary / Free for Academics	Net of Timed Automata	No	Yes
Tapaal	GPLv2 / BSD2	Timed Arc Petrinets	No	Yes
SPIN	BSD3	PROMELA	No	Yes
Ecdar	MIT	Input/Output Automata	No	Yes
H-Uppaal	MIT	Input/Output Automata	No	No
IBM RR	Proprietary	State Charts	Yes	No
Sparx EA	Proprietary	UML	Yes	No
IAR VS	Proprietary	State Charts	Yes	Yes

Table I  
TOOLS OVERVIEW

If a company such as HMK Bilcon would want to use a Model Based Development pipeline, they would have to rely on products such as IAR Visual State for tool support. This would limit them to a specific formalism and code output format. These restrictions also mean that if any errors or obvious optimizations was found in the tool itself, they would have to wait for IAR Systems to patch it, even if the fix was relatively easy to do. Because of this, we argue that a custom solution extending and/or combining the freely licensed tools would be ideal and incidentally this is exactly what HMK

Bilcon has done.

#### IV. CASE WITH HMK BILCON A/S

HMK Bilcon is a Danish industrial truck producer that assembles custom petrol tanker trucks. The trucks can be customized to the individual buyer's desires as much as possible, so they treat each truck order as an individual project instead of a mass production product. Most of these trucks need a system that automates the sequences of actions needed to perform a product delivery (typically petrol, but also other types of wet products), and this is where HMK Bilcon have developed the SafeCon line of systems. These systems help the truck driver control his trucks' piping systems and various lights from a number of panels.

The newest SafeCon version, SafeCon III, is the system of interest for this project. In Figure 4 the physical architecture of the system is shown. SafeCon III is a flexible system operated by a traditional x86\_64 industrial-use rated control computer, dictating what the system should do, and the physical input and output ports are controlled by a set of custom PCB prints (IO Boards). These boards are communicated to and from via a specialized router that acts as a bridge between the serial connection and the Wired Local Area Network (WLAN). The router also provides an Internet connection, enabling Over The Air (OTA) updates and remote truck support if the truck driver calls in for assistance.

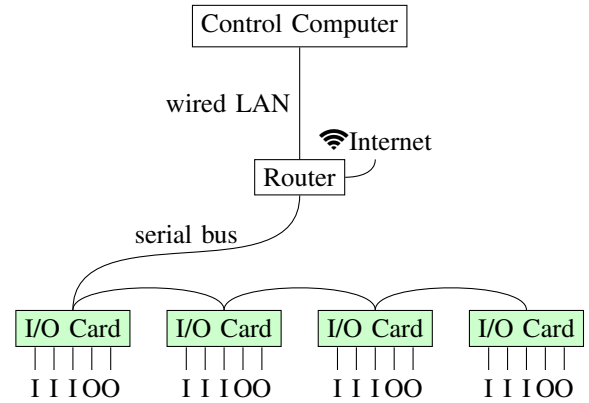


Figure 4. SafeCon III system physical setup

The logic executed on the control computer is dictated by a state machine model, written in a custom modelling language reminiscent of Networks of Timed Automata theory. Then state machine code is generated from these models, which is in turn executed on the control computer. This pipeline is clearly an MBD pipeline and is therefore ripe for exploiting

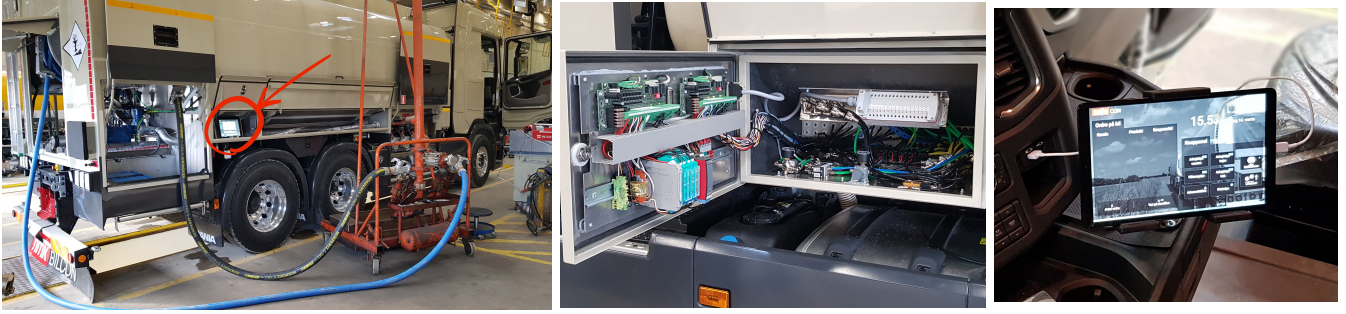


Figure 3. SafeCon III Installed on a truck. Highlighted on the left photo is the touch interface attached to the truck. The control computer is behind the screen. The photo on the middle is the custom PCB I/O Boards and the right-most photo is of the tablet interface located inside the driver's cabin. Photos provided by HMK Bilcon

the benefits that verification techniques can bring. Figure 5 is a more detailed diagram of this MBD pipeline.

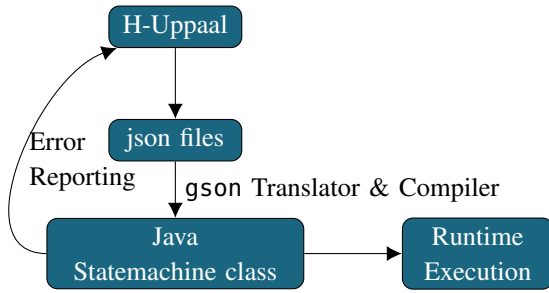


Figure 5. SafeCon III Model Based development pipeline, from model drawing to executable code - Figure provided by HMK Bilcon

At first, an engineer defines the desired logic of the system in the H-Uppaal tool. Once a feature has been modelled, a custom compilation backend makes sure that there are no obvious errors in the expressions. The backend also makes sure to reject some of the features that H-Uppaal supports, but the SafeCon system does not, such as Invariants. If any errors are found by the compiler, it reports it to the engineer via the error window in H-Uppaal, so that he may fix it. If no errors are found, the state machine is considered valid and can be executed on the SafeCon III hardware. The H-Uppaal tool was selected because of its user friendly interface and association with verification theory, providing a platform for verification integration.

## V. HARDWARE ABSTRACTION WITH KNOWLEDGE

The logic used is internally called by the acronym HAWK (Hardware Abstraction With Knowledge) and as the name suggests, it was originally intended as a convenient abstraction for modeling behaviour of hardware, whilst containing some idea of the current and past state of the system. The logic that they ended up with have some key deviations from established theories such as Networks of Timed Automata:

- Variables are globally accessible
- Variables are used instead of channels for inter-process communications
- Unknown variables (Input ports change on a whim)
- Variable value domains are finite
- Custom domain specific runtime semantics

These deviations are essentially a byproduct of the way the language was designed. The use of variables instead of channels was primarily born from a production speed perspective, where variables were implemented first and then exploited to do other things. When the notion of implementing channels came to light, the most financially viable choice was to just keep using variables instead.

The fact that the system has some variables which will always be unknown is really just a consequence of the real world execution requirement for the language. Input ports and sensory data are just completely out of the software engineers' hands to determine before the system is running. However, a nice feature born from this execution-need is the fact that variable value domains simply cannot be infinite. 64bit integers have large domains, yes, but it is not infinite and SafeCon III mostly uses 8bit integers for input values and booleans for internal variables.

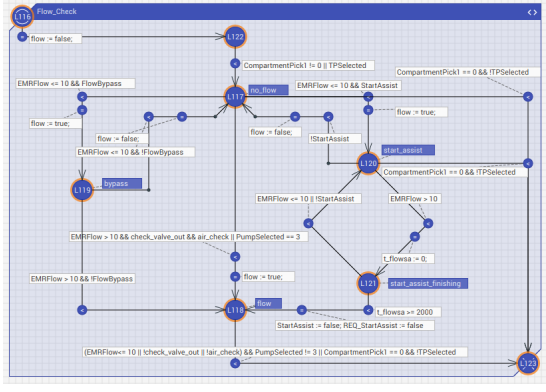


Figure 6. Example of a state machine component in HAWK

HAWK clearly has a big focus on manipulating variables. This focus is primarily born from an engineering standpoint, since the logic should be able to actually run whilst still being legible, understandable, easy to use, fast and efficient to produce new features in. HMK Bilcon does not wish to adopt a new formalism, since that would mean they have to throw out their existing code base and feature-set, just for a more mathematically sound formalism. It is just not financially viable. Therefore, we will in the rest of this paper focus on formulating what HMK Bilcon already have produced in a mathematical manner, such that verification techniques can be eventually integrated into the MBD production pipeline.

## VI. MATHEMATICAL PRELIMINARIES

$\mathbb{B}$  is the boolean value domain with the values *tt* (true) and *ff* (false).

$$\mathbb{B} = \{tt, ff\} \quad (1)$$

We define the union of the boolean value domain and real numbers to be the *variable-value domain*. We do not include the set of natural numbers, since it is already included in the set of real numbers.

$$\mathbb{V} = \mathbb{B} \cup \mathbb{R} \quad (2)$$

We define function mutation by element replacement of some function  $f : C \rightarrow D$  with some other function  $g : C \rightarrow D$  as a function  $h : ((C \rightarrow D) \times (C \rightarrow D) \times 2^C) \rightarrow (C \rightarrow D)$ :

$$h = f[f(x)/g(x)] \text{ for } \forall x \in C', \text{ where } C' \subseteq C \quad (3)$$

$$\parallel$$

$$h(f,g,C')(x) = \begin{cases} g(x) & \text{if } x \in C', \\ f(x) & \text{else} \end{cases}$$

We then say that the set  $C'$  of such a mutation is the *element influence* of function  $h$ . The element influence can also be written as:

$$\mathbf{Infl}(h) = C' \quad (4)$$

For brevity, we sometimes write  $h(f,g,C')$  as  $h_{C'}(f,g)$ . We call functions that modify other functions in this manner *function modifier functions* or FMFs for short. The set of all FMFs is  $\mathbf{F}$  with  $h \in \mathbf{F}$ . We also define the  $\oplus$  relation as a composition of FMFs. We creatively call this relation the FMF composition relation, and it has the type  $\oplus : \mathbf{F} \times \mathbf{F} \rightarrow \mathbf{F}$  and is defined as follows:

If we have two FMFs  $f_A, f_B$  modifying the  $f$  function with functions  $f'$  and  $f''$  respectively:

$$\begin{aligned} f_A(f, f') &= f[f(x)/f'(x)] \text{ for } \forall x \in A \\ f_B(f, f'') &= f[f(x)/f''(x)] \text{ for } \forall x \in B \end{aligned} \quad (5)$$

If  $A \cap B = \emptyset$  then we have that the composition of  $f_A, f_B$  is<sup>2</sup>:

$$\begin{aligned} \oplus(f_A, f_B)(f) &= f[f(a)/f'(a), f(b)/f''(b)] \text{ for} \\ &\forall a \in A \text{ and } \forall b \in B \end{aligned} \quad (6)$$

The FMF composition can also be written as  $f_A \oplus f_B$ . Given a set  $F = \{f_1, f_2, \dots, f_k\}$ , we also say that  $\bigoplus_{f \in F}^F f$  is the same as  $f_1 \oplus f_2 \oplus \dots \oplus f_k$ . The element influence of such a composed function would be  $\mathbf{Infl}(f_1 \oplus f_2 \oplus \dots \oplus f_k) = \mathbf{Infl}(f_1) \cup \mathbf{Infl}(f_2) \cup \dots \cup \mathbf{Infl}(f_k)$ .

To avoid confusion later on, we also define the function composition operator for sets of functions. The composition ( $\circ$ ) of function  $f$  and  $g$  is defined as:

$$(f \circ g)(x) = f(g(x)) \quad (7)$$

If we then say that the set containing functions  $f$  and  $g$  is  $T_{fg} = \{f, g\}$ , we then say that:

$$f(g(x)) = (\bigcirc_{r \in T_{fg}}^{T_{fg}} r)(x) \quad (8)$$

Note that the law of transitivity does not necessarily apply for the function composition operator. It depends on the functions  $f$  and  $g$  themselves.

## VII. TICK TOCK AUTOMATA

In this section we will introduce the notion of a new type of automata, called a Tick Tock Automata, that has a primary focus on being as close to the SafeCon III systems functionality as possible. Figure 7 illustrates the ticking and tocking, that the system does. A tick represents the time-frame of which the logical part of the system operates, then some output to the world is set. During the tock time-frame, the system reads input values from all the sensors and informs

<sup>2</sup>If  $A \cap B \neq \emptyset$ , then  $\oplus(f_A, f_B)$  is undefined



the system of what changes has occurred and then another tick can happen, making the system run in tick-tock cycles.

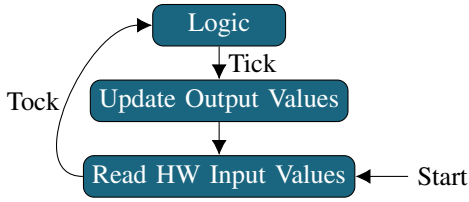


Figure 7. Tick Tock intuition flow chart

In a more theoretical sense, this means that the statemachine is only allowed to update its state by following transitions during a *Tick* cycle. As is evident from Definition 1.1, this formalism is heavily tied to unknowable external inputs.

**Definition 1.1.** *Tick Tock Automata (TTA)*

A Tick Tock Automata  $A$  is a tuple:  $A = (L, l_0, C, V, E, \Omega, v_0, c_0)$  where:

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

The set  $\Omega$  is a set of variables, that represent the physical sensors and actuators that a TTA is attached to. The  $\Omega$  set can be split into two mutually disjoint sets:  $\Omega_o$  for *external output variables* and  $\Omega_I$  for *external input variables*. Additionally, we define the set of all variables as  $Z = \Omega \cup V$ . With the set of all variables, we can also define what is called the *Tick-writeable variables* set as  $Z_O = Z \setminus \Omega_I$ . This set will come in handy later. Edges  $e \in E$  are defined as vectors of elements:  $e = \langle l, g, u, r, l' \rangle$ , where:

- $l \in L$  is a start location,
- $g \in G$  is a guard,
- $u \in U$  is an update function,
- $r \subseteq C$  is a set of clocks to be reset, and
- $l' \in L$  is an end location

We also write  $l \xrightarrow{g, u, r} l'$  if  $\exists e \in E$  such that  $e = \langle l, g, u, r, l' \rangle$ . If the edge has no guard ( $g = \varepsilon$ ), update ( $u = \varepsilon$ ) or clock reset-set ( $r = \emptyset$ ), we simply omit them from the notation. e.g.  $l \xrightarrow{u} l'$  means that  $\exists e \in E$  such that  $e = \langle l, \varepsilon, u, \emptyset, l' \rangle$ .

Let  $\Lambda$  be the set of all possible variable valuations s.t.  $v \in \Lambda$  is of the form

$$v : Z \rightarrow \mathbb{V}$$

A clock valuation  $c \in \mathbb{R}^C$  function is defined as:

$$c : C \rightarrow \mathbb{R}_{\geq 0}$$

A guard  $g \in G$  is defined as a function, that maps a variable or clock valuation to either a true-value or a false-value.

$$g : \Lambda \cup \mathbb{R}^C \rightarrow \mathbb{B}$$

An update  $u \in U$  is defined as a function

$$u : \Lambda \rightarrow \Lambda$$

Additionally, we define the type function  $\tau$ , that maps variables to domain sets:

$$\tau : Z \rightarrow 2^{\mathbb{V}}$$

We abuse the notation  $r(c) \rightarrow c'$  to mean  $c' = c[c(x)/c''(x)]. \forall x \in r$  s.t.  $c''(x) = 0$ . If for some edge  $e = \langle l, g, u, r, l' \rangle$ , there is no clocks to reset  $r = \emptyset$ , then we say that  $r(c) = c$  for any  $c \in \mathbb{R}^C$ .

□

TTAs also have a graphical notation that is almost identical to Timed Automata. Locations are represented with circles and edges are represented with arrows going from one location to another. Edges are annotated with the guards, updates and clock reset-sets and the initial location is simply indicated by an arrow pointing to it.

A requirement set by HMK Bilcon is that the statemachine should be able to take multiple transitions before a hardware input read cycle is started. This is primarily due to ease of development, but it also allows them to do some clever logic as illustrated in Figure 8. In this figure we have a statemachine that starts in location  $l1$ , and has three different guards:  $g1, g2$  and  $g3$ .

This logic is similar to fall-through in traditional C-style switch cases, such that the updates  $u1, u2$ , and  $u3$  will always be executed in immediate sequence of eachother and the guards select at which place in the update sequence the execution should start. This update sequence is required to be completed before any external input values are read.

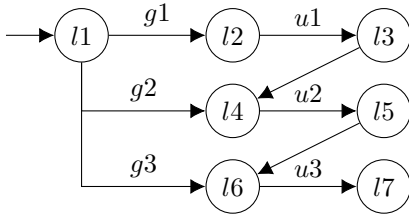


Figure 8. Multiple transitions should be possible in selective parts of the logic

**Definition 1.2. TTA Semantics**

Let  $A = (L, l_0, C, V, E, \Omega, v_0, c_0)$  be a TTA. The semantics is defined as a Transition system  $\langle S, s_0, \rightarrow \rangle$ , where  $S = L \times \Lambda \times C$  is the set of states,  $s_0 = (l_0, v_0, c_0)$  is the initial state, and  $\rightarrow \subseteq S \times S$  is the transition relation defined by:

A Tick-transition  $(l, v, c) \rightarrow (l', v', c')$  is valid iff.

- $l \xrightarrow{g, u, r} l'$  such that:
- $g(v) = tt$  and
- $g(c) = tt$  and
- $u(v) = v'$  and
- $r(c) \rightarrow c'$

A transition is said to be enabled if there exists an edge from  $l$  to  $l'$ , where the guard is satisfied within the  $v$  variable valuation function and clock valuation function  $c$ , and that the associated update function  $u$  results in the new  $v'$  variable valuation function.

A Tock-transition is a two step process, where we first “read” the input values from hardware. This is described with the  $\Gamma$  function. Defined as a function  $\Gamma : \Lambda \rightarrow \Lambda$ , where:

$$\Gamma(v) = v[v(x)/v'(x)]. \forall x \in \Omega_I \text{ s.t. } v'(x) \in \tau(x) \quad (9)$$

This function replaces all values associated with the external input variables, in a type-safe manner. The selection of the replacement function  $v'$  is non-deterministic. This is analogous with a completely unknown external environment.

As the second step of a Tock-transition, we define the clock-delay step as the  $\gamma$  function:

$$\gamma(x) = \text{for some } d \geq 0, c[c(x)/c'(x)]. \quad \forall x \in C \text{ s.t. } c'(x) = c(x) + d \quad (10)$$

Essentially, all we do here is delay all clocks by some non-deterministically selected value  $d \geq 0$ . This is analogous to the fact that reading hardware values is not an instantaneous process, and can even take forever (timeouts).

We can now define the Tock-transition as two sequential statements:

$$v = \Gamma(v) ; c = \gamma(c) \quad (11)$$

For the TTA semantic to support the behaviour described in Figure 8 we will introduce the notion of *Immediacy*. Note that Immediacy is different from Timed Automata’s (TA) Urgency, which refers to time progression and not tock actionability. However, immediacy somewhat resembles the notion of Committedness, but we choose to make the explicit distinction, because of the fundamental differences TTAs have compared to traditional Timed Automata (TA). We say that a location  $l$  is annotated as immediate if it is in the set of immediate locations  $I \in L$ . If the current state  $s = (l, v, t)$  is in an immediate location  $l \in I$ , then we will take another Tick action instead of a Tock action. To make the update sequence described in Figure 8 happen before any Tock actions are taken, we simply annotate the locations  $l2, l3, l4, l5$  and  $l6$  as immediate. In Figure 9, these annotations are illustrated as squares.

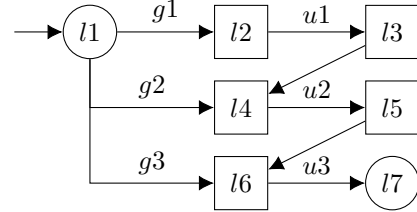


Figure 9. Immediacy enables fine grain execution noodle control

The full Tick-Tock behaviour with immediacy is illustrated in Figure 10 as a flow chart. Note that a Tick-step transition cannot perform a time delay, but it is possible in a Tock-step to delay forever (discretely).

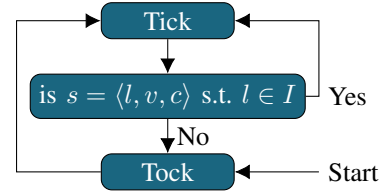


Figure 10. Tick Tock flow chart

We say that a TTA  $A$  is *Tock-independent* if all reachable locations in a given TTA  $A$  are marked as immediate  $l \in I$ . Such a TTA will never be able to execute a Tock-transition. □

With a Tick Tock Automata and its semantics defined, we can define a network of parallelly composed TTAs.

**Definition 1.3. Network of Tick Tock Automata (NTTA)**

Let  $A_i = (L_i, l_0^i, C, V, E_i, \Omega, v_0, c_0)$  be a network of  $n$  Tick Tock Automata. Let  $\bar{l}_0 = \langle l_0^0, l_0^1, \dots, l_0^n \rangle$  be the initial location



vector. The semantics of an NTTA is defined as a transition system  $\langle S, s_0, \rightarrow \rangle$ , where  $S = (L_0 \times \dots \times L_n) \times \Lambda \times C$  is the set of states,  $s_0 = \langle \bar{l}_0, v_0, c_0 \rangle$  is the initial state and  $\rightarrow \subseteq S \times S$  is the transition relation defined by:

A Tick-transition  $(\bar{l}, v, c) \rightarrow (\bar{l}', v', c')$  is valid iff:

- 1) Either  $\bar{l} \neq \bar{l}'$  or  $v \neq v'$  or  $c \neq c'$ , and
- 2) for all  $\forall l' \in \bar{l}'$ , we have that
  - a) If  $\exists e = \langle l, g, u, r, l' \rangle \in E_i$  for any  $0 \leq i \leq n$  where  $l \in \bar{l}$  and
  - b)  $g(v) = g(c) = tt$
  - c) Then for any  $0 \leq j \leq n$ ,  $\nexists e' = \langle p, g', u', r', p' \rangle \in E_j$  s.t.  $p \in \bar{l}$ ,  $p' \in \bar{l}'$  and  $e' \neq e$ , where:
    - i)  $g'(v) = g'(c) = tt$ , and
    - ii)  $\text{Infl}(u) \cap \text{Infl}(u') \neq \emptyset$
  - d) Else,  $l'$  also has to be an element in  $l' \in \bar{l}$ .
- 3)  $v' = (\bigoplus_{u \in \text{Tu}(\bar{l}, \bar{l}')} \text{Tu}(\bar{l}, \bar{l}'))(v)$ , where:
  - a)  $\text{Tu}(\bar{l}, \bar{l}') = \{ u \mid \forall l' \in \bar{l}' \text{ s.t. } \exists e = \langle l, g, u, r, l' \rangle \in E_i \text{ for any } 0 \leq i \leq n, \text{ where } l \in \bar{l} \text{ and } g(v) = g(c) = tt \}$  is the set of *taken transition updates*
- 4)  $c' = (\bigcirc_{r \in \text{Tr}(\bar{l}, \bar{l}')} \text{Tr}(\bar{l}, \bar{l}'))(c)$ , where:
  - a)  $\text{Tr}(\bar{l}, \bar{l}') = \{ r \mid \forall l' \in \bar{l}' \text{ s.t. } \exists e = \langle l, g, u, r, l' \rangle \in E_i \text{ for any } 0 \leq i \leq n, \text{ where } l \in \bar{l} \text{ and } g(v) = g(c) = tt \}$  is the set of *taken transition clock reset-sets*

For a given location vector e.g.  $\bar{l} = \langle l1, l2, l3 \rangle$ , we say that the location  $l2$  is an element in the vector written as  $l2 \in \bar{l}$ . Since networked TTAs share all variables, internal as well as external, a Tock-transition of an NTTA is identical to having just a single TTA. However, the condition for immediacy to re-issue a Tick-transition has to be slightly modified. In Figure 11 we show the updated immediacy condition.

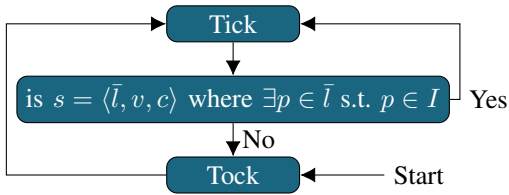


Figure 11. Tick Tock flow chart for network of Tick Tock Automata

This means that, as long as a single TTA in the network is in an immediate location, no Tock-transition can be taken. If the network never allows a Tock-transition, we call it *Tock-independent*.

The strict rules for when a transition is valid or not, avoid some of the nasty situations encountered when modelling concurrent systems that share variables. Consider the scenarios illustrated in Figure 12, Figure 13, Figure 14 and Figure 15.

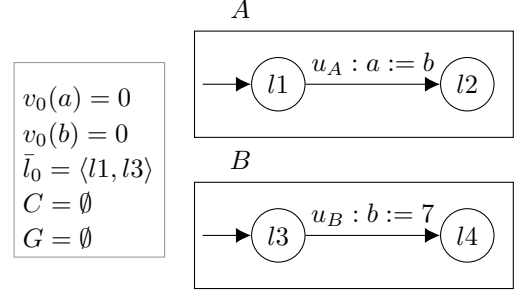


Figure 12. Network of two TTAs A and B, posing a potential variable updating issue

The case of Figure 12 shows the potential race condition. If we take the transition  $\langle \langle l1, l3 \rangle, v_0, c_0 \rangle \rightarrow \langle \langle l2, l4 \rangle, v', c' \rangle$ , what value does  $v'(a)$  evaluate to?

Given a network of Tick Tock Automata, such as the one described in Figure 12, the transition from the initial location vector  $\langle l1, l3 \rangle$  to  $\langle l2, l4 \rangle$  results in a variable valuation  $v'$ , where  $v'(a) = 0$  because rule 3 is defined by the FMF composition of the updates on the edges  $l1 \xrightarrow{u_A} l2$  and  $l3 \xrightarrow{u_B} l4$ , where

$$u_A(v) = v[v(a)/v''(a)] \text{ where } v''(a) = v(b) \text{ and}$$

$$u_B(v) = v[v(b)/v'''(b)] \text{ where } v'''(b) = 7.$$

In other words:  $v'$  is defined as  $u_A \oplus u_B(v_0) = v'$ , which is equal to:

$$u_A \oplus u_B(v_0) = v_0[v_0(a)/v''(a), v_0(b)/v'''(b)] \text{ where } v''(a) = v_0(b) \text{ and } v'''(b) = 7 \quad (12)$$

And since  $v_0(b) = 0$  and  $v''(a) = v_0(b)$  then  $v''(a)$  must be  $v''(a) = 0$ , which results in  $v'(a) = 0$ . Even though  $v'(b) = v'''(b) = 7$ .

The intuition behind this behaviour is that all variable assignments are atomic, and all right-hand-side expressions are evaluated before any variables are assigned.

The example illustrated in Figure 13 shows why it is very important to have such strict Tick-transition validity rules as we do. If we take the transition from  $\langle \langle l1, l3 \rangle, v_0, c_0 \rangle \rightarrow \langle \langle l2, l3 \rangle, v', c' \rangle$ , what value does  $v'(a)$  evaluate to?

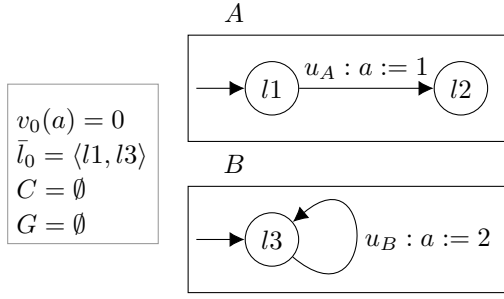


Figure 13. Network of two TTAs  $A$  and  $B$ , potentially giving a race condition

In the transition  $\langle \langle l1, l3 \rangle, v_0, c_0 \rangle \rightarrow \langle \langle l2, l3 \rangle, v', c' \rangle$ , there exists an edge  $e = \langle l1, \varepsilon, u_A, \emptyset, l2 \rangle$ , but there also exists another edge  $e' = \langle l3, \varepsilon, u_B, \emptyset, l3 \rangle$ , where the influence of the update functions  $u_A$  and  $u_B$  clash, i.e.  $\text{Infl}(u_A) \cap \text{Infl}(u_B) = \{a\} \neq \emptyset$ . Therefore, since we have a violation of the 2.c.ii rule there is no valid transition from the location vector  $\langle l1, l3 \rangle$  to  $\langle l2, l3 \rangle$ . However, there is a transition from  $\langle \langle l1, l3 \rangle, v_0, c_0 \rangle \rightarrow \langle \langle l1, l3 \rangle, v', c' \rangle$ , where  $v'(a) = 2$ . In fact, there are no valid Tick-transitions from the initial state that ends up in a state with location  $l2$  in their location vector.

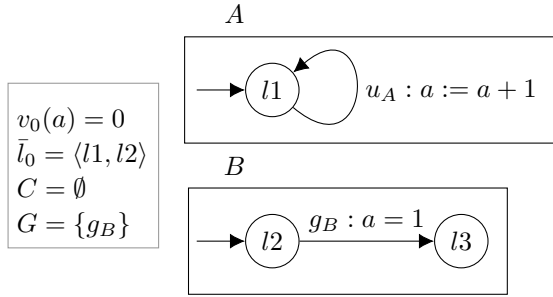


Figure 14. Seemingly a Tick-deadlock, however there does exist a chain of transitions that can get  $B$  to  $l3$ .

Figure 14 illustrates an example of how a TTA can be dependent on other TTAs in a network. In isolation, the TTA  $B$  will never be able to enter  $l3$ , because of the guard  $g_B$ . But if  $A$  fires its transition, updating the  $a$  variable to evaluate to 1, then  $B$  can proceed to  $l3$ . Or more formally:<sup>3</sup>

$$\begin{array}{l|l} \langle \langle l1, l2 \rangle, v_0, c_0 \rangle \rightarrow & v_0(a) = 0 \\ \langle \langle l1, l2 \rangle, v', c' \rangle \rightarrow & v'(a) = 1 \\ \langle \langle l1, l3 \rangle, v'', c'' \rangle & v''(a) = 2 \end{array}$$

For situations where there is a choice between two edges, the resulting transition system simply branches into nondeterministic choice. Figure 15 shows a network containing a single automata, that produces such a transition system.

<sup>3</sup>Note that  $c_0 = c' = c''$ , so we do not really care about it in this example

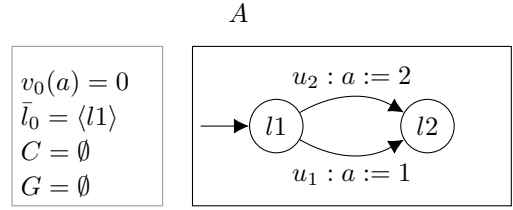


Figure 15. Network of Tick Tock Automata with only one automata, showing nondeterministic choice of two edges, both manipulating the evaluation of variable  $a$ .

taking $u_1$	$\langle \langle l1 \rangle, v_0, c_0 \rangle \rightarrow$	$v_0(a) = 0$
	$\langle \langle l2 \rangle, v', c' \rangle$	$v'(a) = 1$
taking $u_2$	$\langle \langle l1 \rangle, v_0, c_0 \rangle \rightarrow$	$v_0(a) = 0$
	$\langle \langle l2 \rangle, v'', c' \rangle$	$v''(a) = 2$

Nondeterministic behaviour may become an issue in real-world applications such as the SafeCon III system. However, they are not necessarily a critical issue. Once a nondeterministic choice is encountered, the system could note it as a warning and then simply pick one. If a critically wrong path is chosen, the developer will have that warning as a guide.

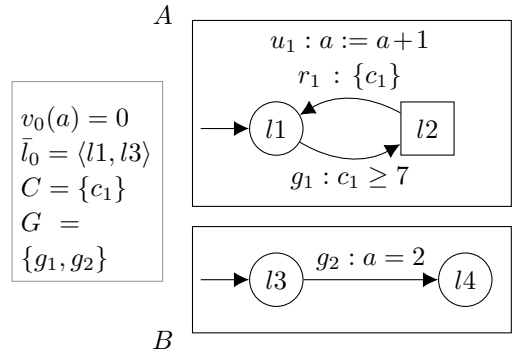


Figure 16. Network of TTAs that require at least one Tock-transition to proceed past  $\bar{l}_0$

Figure 16 shows an example TTA network that uses both immediacy ( $l2 \in I$ ) and clocks ( $c_1$ ). The network consists of a TTA ( $A$ ) that increments variable  $a$  by one for every seven seconds that has passed, and another TTA ( $B$ ) that has the opportunity to go from  $l3$  to  $l4$  when variable  $a$  is exactly two. To ensure that the incrementation of  $a$  is immediate,  $l2$  is marked as an immediate location. Intuitively this network looks like it can go down two different paths: One where the  $B$  TTA passes up on the opportunity to go into  $l4$ , and one where it takes it. All whilst the  $A$  TTA counts variable  $a$  up towards infinity.

In Figure 17 the reachable state space of Figure 16 is presented as a single transition system, that is also containing Tock-transitions. Each state is represented as a location vector, a constraint for the valuation of variable  $a$ , and a constraint

of the clock valuation of clock  $c_1$ . This way of representing the states is somewhat inspired by the *zones* concept used in the Uppaal verification engine [19]. The transitions marked green are the ones that can produce an infinite sequence of steps, where the  $l4$  location is within the location vector, and the transitions marked red are those that can produce an infinite sequence of steps, where  $l4$  is not in the location vector. This means that if we were to ask the query: “Will we always eventually reach  $l4$ ?”, the answer would be no. But, interestingly the SafeCon III system has a specific condition for issuing a Tock-transition that we have not integrated into the semantics yet. The SafeCon III system guarantees that the system will get *at least* ten chances to take a Tick-transition before another Tock-transition can be taken. As marked by the light blue box, this extra condition will result in a drastically reduced state-space and makes the answer to the previous query a *yes*. Whether or not this is a desired property of the system is a decision for the engineer designing the system to make. Adding this condition can be done fairly easily, by just adding it to the conditions of taking a Tock as can be seen in Figure 18.

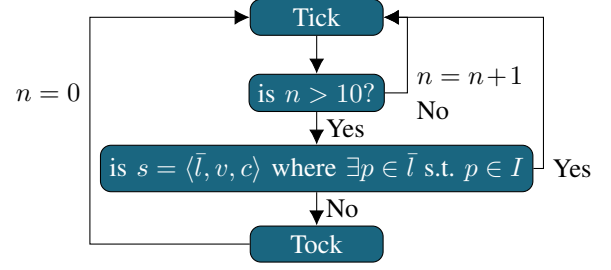


Figure 18. Tick Tock flow chart for network of Tick Tock Automata updated for SafeCon III conformance

The minimum amount of Tick-transition opportunities can be changed in accordance with whatever specific properties the system should have. For getting the desired state space in the light blue box in Figure 17, we only need to compare  $n > 2$ .

### VIII. AVOIDING STATESPACE EXPLOSION

The definition of Tock-transitions is very open ended and can lead to some very difficult situations when verification is attempted. The fact that all input variables can change from

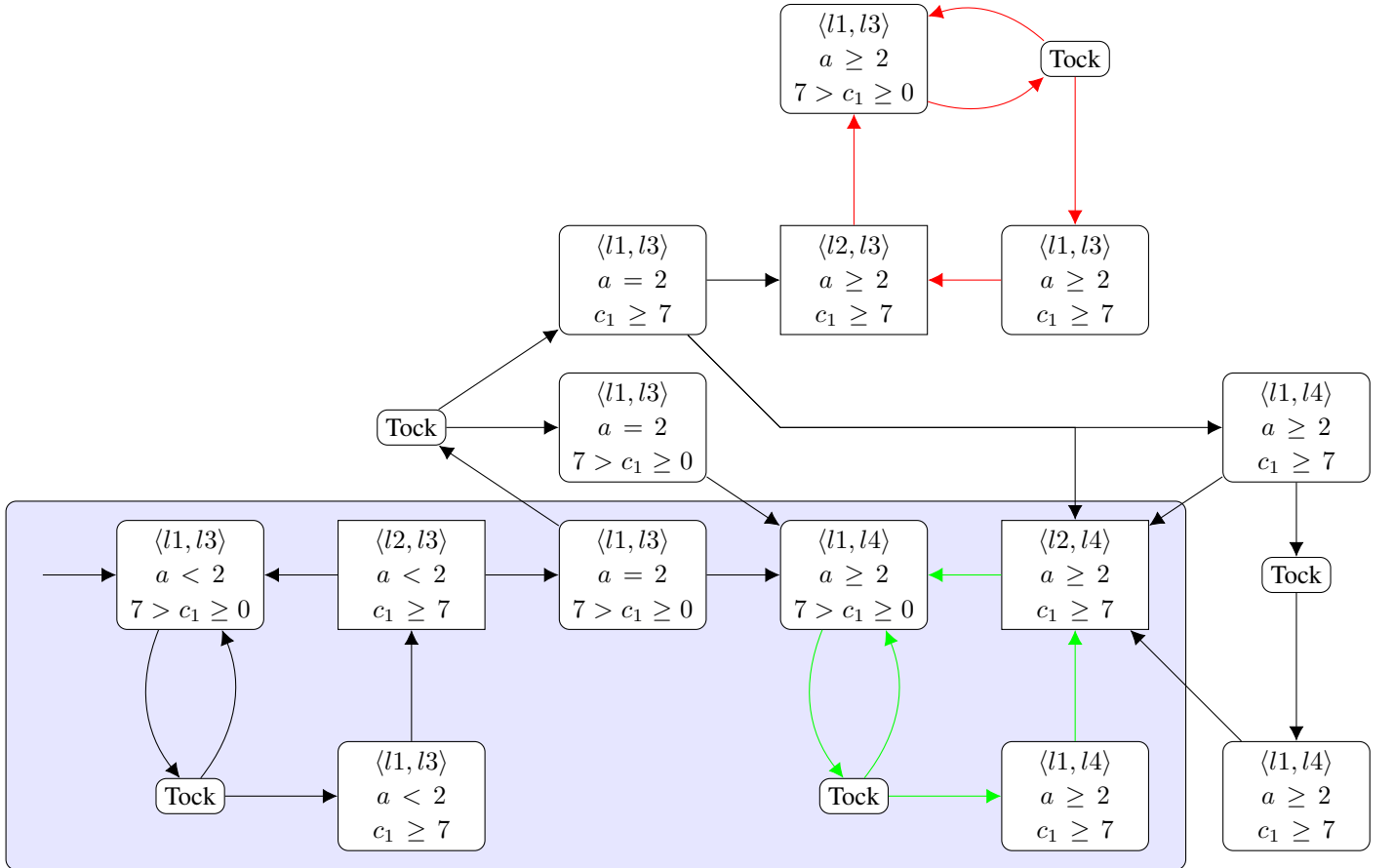


Figure 17. Reachable state space for the NTTA described in Figure 16. States with an immediate location in them are annotated by a rectangle with sharp corners (non-rounded corners).

one Tock to another is not only unrealistic, but also explodes the reachable state space exponentially. As an example consider a simple TTA with just a single location  $l1$ ,  $l_0 = l1$  with a single looping edge  $E = \{\langle l1, \varepsilon, \varepsilon, l1 \rangle\}$ , and the set of external input variables  $\Omega_I = \{i_1, \dots, i_k\}$ . Let us assume that all external input variables are boolean typed, then the amount of reachable states would explode to  $2^k$ . If we say that the system has fifty external input variables ( $k = 50$ ), which is not an unimaginable scenario, the state space would grow into roughly  $1.1 \times 10^{15}$  different states. All reachable via a single Tock-transition. For the sake of effective reachability analysis, this explosion behaviour will obviously have to be mitigated or avoided as much as possible. Clocks can also make the state space explode exponentially, but by representing states as zones as described in [19] and shown in Figure 17, clock-induced statespace explosion can be effectively minimized. In this section we will focus on variable-induced statespace explosion mitigation. We are taking a two step approach to reducing this problem; Restricting the definition of the Tock-transition itself and providing the system designer with a way of manually defining domain-guarantees for the external input variables.

#### A. Interesting-ness

In order to reduce statespace explosion, we introduce the set of *interesting external input variables* (written as  $\Pi_I$ ). An interesting external input variable is one that might be used in the next series of Tick-transitions after a Tock has occurred. Since a Tick-transition is unable to override the external input variables, we are only interested in looking at the guards of the edges when searching for  $\Pi_I$ . Therefore we introduce the *testing* function  $T$ , which maps edges to a set of variables:

$$T : E \rightarrow 2^Z \quad (13)$$

As an example, consider the guard  $g = a < 3$  and update  $u = x = a$  on edge  $e$ , respectively testing if the variable  $a$  is lower than three and setting variable  $x$  to the value of  $a$ . We then define  $T(e) = \{a, x\}$ . If we want to know if an edge  $e$  is testing an external input variable, we can simply mask the internal variables and external output variables  $T(e) \setminus (V \cup \Omega_O) \subseteq \Omega_I$ . Algorithm 1 shows an algorithm for finding the set of interesting variables given a location and the set of edges in the associated TTA. Note that this is essentially a recursive tree search, where we are recursing if an immediate location is potentially reachable from the given location vector. For networks of TTAs, *InterestingVarSearch* would simply be run on each location and associated edge set in the location vector:

$$\Pi_I(\bar{l}, E_0, \dots, E_k) = \bigcup_{i=0}^{i=k} \text{InterestingVarSearch}(l_i, E_i)$$

We say that for some network of  $k$  TTAs in a state  $\bar{l}$  that  $\Pi_I = \Pi_I(\bar{l}, E_0, \dots, E_k)$ . Using the set of interesting external input variables, we can redefine the variable changing part of the Tock-transition ( $\Gamma$ ) to only change the interesting variables of the current state  $\bar{l}$ :

$$\Gamma(v) = v[v(x)/v'(x)]. \forall x \in \Pi_I \text{ s.t. } v'(x) \in \tau(x) \quad (14)$$

**Algorithm 1** Algorithm to find the interesting external input variables. This algorithm ignores the addition to the flowchart in Figure 18.

---

```

1: procedure INTERESTINGVARSEARCH( $l, E$ )
2:    $\Pi_I = \emptyset$ 
3:   for each  $e = \langle l', g, u, r, l'' \rangle \in E$  do
4:     if ( $l' = l$ ) then
5:       if  $T(e) \setminus (V \cup \Omega_O) \subseteq \Omega_I$  then
6:          $\Pi_I = \Pi_I \cup (T(e) \setminus (V \cup \Omega_O))$ 
7:       end if
8:       if  $l'' \in I$  then
9:          $\Pi_I = \Pi_I \cup \text{InterestingVarSearch}(l'', E \setminus \{e\})$ 
10:      end if
11:    end if
12:  end for
13:  Return  $\Pi_I$ 
14: end procedure

```

---

This notion of interestingness is not restricting in any way meaningful to the functionality of the NTTA. It only prunes away the states that are uninteresting in terms of verification and can be seen as analogous to lazy evaluation of variables. It can also be extremely effective in reducing the reachable statespace. Consider the example TTA from before, with a single location  $l1$ ,  $l_0 = l1$  and a single edge  $E = \{\langle l1, \varepsilon, \varepsilon, \varepsilon, l1 \rangle\}$  and the fifty boolean-valued external input variables. In this example, none of the external input variables have any effect on the behaviour of the system and so they are all deemed uninteresting, reducing the size of the reachable statespace from  $1.1 \times 10^{15}$  to just two states; One for the Tock-transition and one with the location vector  $\langle l1 \rangle$ .

Interestingness reduces the exponent part of the explosion problem, but if the base is already big then reducing just the exponent may not be enough i.e.  $256^{50} = 2.6 \times 10^{120}$  (fifty byte-valued variables<sup>4</sup>) is a tremendously huge number, and if we reduce the exponent to let's say 4 interesting variables, we still have that  $256^4 = 4.2 \times 10^9$  different states, which may still be too large for verification. Therefore we will introduce *guarantee flagging*.

<sup>4</sup>In the SafeCon III system, they typically have between three to four analogue inputs, which are stored in a byte [0-255]

## B. Guarantee Flagging

Systems developed in a Model Based Development manner, such as SafeCon III, are typically designed by a domain expert. During development these experts can typically provide some guarantees that certain values will either increase, decrease, stay the same or change to something within a range. These guarantees are also usually directly dependent on some assumptions, such as the current state of the system or the environment. As an example: In SafeCon III it is very unlikely that the truck will accelerate if the handbrake pulled. With an error tolerance of 2 km/h this can be described as a predicate:


$$\rho_{hb} = \text{handbrake} \implies \text{current\_acc} \leq 2\text{km/h} \quad (15)$$

However, a guarantee might not always be true; it depends on the current state of the system. So we need to be able to define for which states that a certain predicate holds true. One way of doing this is by introducing the syntactic idea of *flags*, where the system designer/domain expert flags certain locations of the TTAs with different *colored* flags. These flags represent certain predicates that are expected to be satisfied for when the system is in such a flagged location.

Formally, we define it as a mapping of locations to predicates  $F(l) = \rho$ , where  $\rho$  is a boolean predicate describing the current assumptions and guarantees of the system. We say that a predicate  $\rho$  is *testing* ( $T(\rho)$ ) a variable, if that variable occurs in the predicate. As an example the predicate in Equation 15 is testing the variables  $T(\rho_{hb}) = \{\text{handbrake}, \text{current\_acc}\}$ . If no manual flag is set for a certain location  $l$ , we simply say that  $F(l) = tt$ .

This Assumption/Guarantee idea is inspired by stateful A/G Interfaces described by L. de Alfaro and T. A. Henzinger in [20], however we will not be defining a formal interface theory for TTAs, as we deem it out of scope for this paper.

For a network of TTAs, guarantee predicates would be defined as the logical conjunction of all flags in the current location vector  $F(\vec{l}) = \bigwedge_{l \in \vec{l}} F(l)$ .

$\Omega_I$	$\{start, pmpspd, pmpindc, cancel\}$
$\Omega_O$	$\{pmpgo\}$
$V$	$\{x\}$
$\tau(pmpspd)$	$\{0, \dots, 255\}$
$u_1$	$pmpgo := tt$
$u_2$	$x := 0$
$u_3$	$x := x + pmpspd$
$g_1$	$start = tt$
$g_2$	$cancel = tt \vee x \geq 400$
$g_3$	$pmpindc = tt \wedge pmpspd \geq 5$
	$(pmpindc = tt \wedge 20 \leq pmpspd \leq 40) \vee (pmpindc = ff \wedge pmpspd \leq 5)$

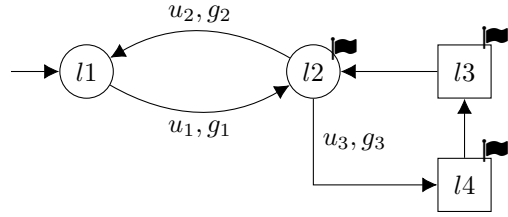



Figure 19. Example TTA that uses guarantee flag () to shrink the domain of *pmpspd* variable within a subsection of the TTA

Figure 19 shows a small example of a TTA, that accumulates values from input variable *pmpspd* into local variable  $x$  until it is cancelled by input variable *cancel*. It is flagged with the predicate that analogue variable *pmpspd* (pumpspeed) will only be within  $20 \leq pmpspd \leq 40$  whenever the *pmpindc* (pumping indicator) input variable is true and  $pmpspd \leq 5$  if it is false. This lowers the amount of possible reachable states from  $255^1 + 2^4 = 272$  to  $20^1 + 2^3 = 28$  whenever  $pmpindc = tt$  and  $5^1 + 2^3 = 13$  when  $pmpindc = ff$ . This may seem like an insignificant saving in this example, but consider the TTA in Figure 19 be duplicated five times and composed into a network<sup>5</sup>. Then the savings would go from  $255^5 + 2^4 = 1.07 \times 10^{12}$  states to  $20^5 + 2^3 = 3.2\text{million}$  when  $pmpindc = tt$  and  $5^5 + 2^3 = 3133$  states when  $pmpindc = ff$ . Combined with *interestingness*, this technique can be quite effective.

Since guarantee flag predicates are joined in conjunction in networks of Tick Tock Automata, it might be interesting to check if they are satisfiable in conjunction with each other or not. If some of them aren't the model checking engine could then do a reachability analysis of whether or not a state with such a predicate conjunction is reachable. Such a state would obviously be an error. The model checker could then warn the system designer when they add new predicates to the system that are in conflict with each other. The question of satisfiability is an NP-Complete problem, but strategies such as Binary Decision Diagrams (BDD) can be quite efficient in solving that. If done at design time rather than verification time (i.e. making the predicate satisfiability check part of the IDE (Integrated Development Environment) itself) the satisfiability checks can be cached by saving them in the project file.

## IX. DISCUSSION

The Tick Tock Automata theory is mostly derived from the way that the SafeCon III system operates, but the theory is not limited to describing only that particular system. Since the pipeline of execution is so similar, TTA theory can

<sup>5</sup>Only the parts concerning analogue variables are duplicated

also be applied to PLC (Programmable Logic Controllers) based systems. However, the world of PLC programming is very different from traditional PC (Personal Computer) programming, and translating the model to an executable state machine would have to be reimplemented for each system. We choose to see this as an extra strong argument for adopting Model Based Development with integrated verification pipelines. Since developers only have to change the model to state machine translation layer, assuming that the new translation routine is implemented correctly in terms of the modelling theory, the model and all the verified queries associated with it still hold. Potentially no extra iteration is needed to be done on the model making it hardware agnostic and potentially saving many development hours that would have been spent on implementing the same functionality as before.

It would be beneficial to check for guarantee flag violations during runtime. If any violations occur, the system state should be logged so that engineers can fine tune their predicates so that the verification results more accurately represent how the real world operates.

The algorithm `InterestingVarSearch` can be run at design time on non-immediate locations and then embed the results into the project files, that way no unnecessary searching is done at verification time. The guarantee flagging tool could then potentially use this to highlight to the designer what variables are interesting at the location they are flagging, and in turn assist the designer to make more efficient predicates.

Throughout this paper, we have been considering traditional state based verification for reachability analysis. However, perhaps a more efficient approach would be to use a BDD based approach, where states are represented as predicates over variables and reachability is computed by ROBDD (Reduced and Ordered Binary Decision Diagram) computation as described in [21]. Such an approach may also benefit from limiting the set of variables considered in a particular state with `InterestingVarSearch`.

For a company to implement an MBD pipeline, there needs to exist user friendly, efficient and reliable tools that can either produce runtime executable code, or provide an interface for developers to implement their runtime statemachine execution framework and then produce statemachines. At the time of writing, the only notable tool that provides this is IAR VisualState, but no FOSS (Free and Open-source Software) solutions that can provide everything in one package currently exists. We hope that this TTA theory will serve as the theoretical platform for such tooling to be created.

## X. CONCLUSION

In this paper, we defined an automata-based syntax and semantics based off of HMK Bilcon's SafeCon III system, that we named Tick Tock Automata (TTA). We defined the semantics of parallel composition of such automata and extended the semantics to adhere to SafeCon III's actual runtime behaviour by enforcing a minimum of ten ticks to be taken. We defined two approaches to avoid statespace explosion for efficient verification: Interestingness of external input variables and manual guarantee flagging of locations. The theory introduced in this paper can serve as a basis for a general MBD tool chain that targets systems with execution pipelines similar to that of PLCs, such as the SafeCon III system. Enabling HMK Bilcon to introduce verification when such a tool has been developed.

## XI. ACKNOWLEDGEMENTS

We would like to express our gratitude to HMK Bilcon for providing information, source code and work space for this collaboration, and we hope that the theories here are usable and applicable for the SafeCon III project. We would also like to thank our supervisor Ulrik for a nice and pressure-free collaboration.

## REFERENCES

- [1] John Harrison. *Formal Verification in Industry*. <https://www.cl.cam.ac.uk/~jrh13/slides/anu-06dec02/slides.pdf>.
- [2] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, pages 532–546, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-05089-3.
- [3] Peter W O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 13–25, 2018.
- [4] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, March 2015. ISSN 0001-0782. doi: 10.1145/2699417. URL <https://doi.org/10.1145/2699417>.
- [5] 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.
- [6] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller. In *Proc.*



- of the 4th Workshop on Tools and Algorithms for the Construction and Analysis of Systems, number 1384 in Lecture Notes in Computer Science, pages 281–297. Springer-Verlag, March 1998.
- [7] Alexandre David, Kim Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Ecdar: An environment for compositional design and analysis of real time systems. pages 365–370, 09 2010. doi: 10.1007/978-3-642-15643-4\_29.
  - [8] 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, pages 492–497. Springer-Verlag, 2012.
  - [9] Gnu general public license. URL <https://www.gnu.org/licenses/old-licenses/gpl-2.0.html>.
  - [10] The 2-clause bsd license. URL <https://opensource.org/licenses/BSD-2-Clause>.
  - [11] Asger Gitz-Johansen, Dan Kristiansen, and Morten Konggaard Schou. stverif. <https://github.com/skgaal/stverif>, 2019.
  - [12] Gerard J Holzmann and Margaret H Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.
  - [13] Francis Schneider, Steve M Easterbrook, John R Callahan, and Gerard J Holzmann. Validating requirements for fault tolerant systems using model checking. In *Proceedings of IEEE International Symposium on Requirements Engineering: RE’98*, pages 4–13. IEEE, 1998.
  - [14] David Harel and Eran Gery. Executable object modeling with statecharts. In *Proceedings of IEEE 18th International Conference on Software Engineering*, pages 246–257. IEEE, 1996.
  - [15] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on software engineering*, 16(4):403–414, 1990.
  - [16] Sparx Systems LLC. Sparx enterprise architect user manual, 2020. URL [https://sparxsystems.com/enterprise\\_architect\\_user\\_guide/](https://sparxsystems.com/enterprise_architect_user_guide/).
  - [17] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.
  - [18] Andrzej Wąsowski and Peter Sestoft. On the formal semantics of visualstate statecharts. 2002.
  - [19] Gerd Behrmann, Patricia Bouyer, Kim G Larsen, and Radek Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. *International Journal on Software Tools for Technology Transfer*, 8(3):204–215, 2006.
  - [20] Luca De Alfaro and Thomas A Henzinger. Interface theories for component-based design. In *International Workshop on Embedded Software*, pages 148–165. Springer, 2001.
  - [21] Henrik Reif Andersen. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen*, page 5, 1997.