

AALTITOAD Version 1.x.x

An Extendable Verification Engine and Simulator for Tick Tock Automata

Asger Gitz-Johansen

Denmark

Email: asger.gitz@hotmail.com

Abstract: We present the syntax and semantics of a novel modelling formalism called Tick Tock Automata (TTA), designed for industrial usage, intuitiveness and practical applicability in mind. Enabling the usage of this formalism, we have implemented the AALTITOAD (**A**alborg **T**ick **T**ock **A**utomata **V**alidator) tool suite, which supplies a verifier and a simulator/runtime which can be extended by third parties in a plugin-style manner. We present the architecture and associated algorithms required for formal verification of TTA models and compare with the previous release of the suite with promising results. The TTA formalism have been applied in an Model Based Development (MBD) pipeline at local tanker truck production company HMK Bilcon A/S in their SafeCon III product and the AALTITOAD toolchain is free and open source and is available at github.

I. INTRODUCTION

When programming embedded systems that are supposed to control some machine, traditionally, a team of software engineers would take some specified requirements and translate it into a series of logics that a computer can understand and execute. This process relies heavily on the communication skills of the customer, machine-designers and product owner as well as the level of skill of the software developers. This is very prone to miscommunication/misunderstandings and therefore incorrect implementations which can become very expensive if mismanaged. In a better framework, the concerns should be separated and let each expert concentrate on their strenghts, i.e. the programmers focus on creating solid and generic software packages and the customer or domain experts define the business logic directly through a common human-friendly language. This is the essence of the Model Based Development philosophy, where we let the domain experts define and support the business logic and the software engineers build and support the underlying platform instead of amending issues related to miscommunications.

The AALTITOAD (**A**alborg **T**ick **T**ock **A**utomata **V**alidator) tool suite aims to provide all the underlying software tools required for such a workflow to exist. At the current release,

we provide a model language specification, extendable runtime implementation, formal methods verification engine and a general-purpose library for future creative developers to implement e.g. static analysis tools. The tool-suite and MBD philosophy has been applied industrially at HMK Bilcon A/S, a local tanker truck production company. Everything is released as an GPL3-licensed open source git repository and is available on github (github.com/sillydan1/aaltitoad).

In this paper, we introduce the syntax and semantics of the Tick Tock Automata modelling formalism, tool-suite architecture, associated verification algorithms and provide a comparisson with our previous alpha version of the suite, which was developed through several student projects [1, 2, 3, 4] in association with HMK Bilcon A/S and Aalborg University (AAU).

Starting in Section II where we formally explore the syntax and semantics of networks of TTAs and define the new maximal step heuristic for more robust state traversal. In Section III we provide a loose specification of a syntactic sugar over networks of TTAs that should make it easier for domain experts to segmentize their logics and keep an overview. In Section IV we outline the general architecture of the runtime tool and how it provides the opportunity for developers to integrate custom parsers, ticker and tocker services dynamically. In Section V we describe our experimental setup for benchmarking the AALTITOAD verifier and in Section VI we go over the results of the experiments. We provide our conclusions in Section VII and finally we provide a list of potential future works for the tool suite in Section VIII.

II. THEORY

A. Preliminaries

Let \mathbb{B} be the boolean value domain containing the values tt (true) and ff (false):

$$\mathbb{B} = \{tt, ff\}$$

Let \mathbb{V} be the variable value domain containing all reals and the boolean values:

$$\mathbb{V} = \mathbb{R} \cup \mathbb{B}$$

Given a set of variables V , let Λ be the set of variable valuation functions such that a given function $v \in \Lambda$ is of the form:

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

Given a set of clocks symbols C , let \mathbb{R}^C be the set of clock valuation functions such that a given function $c \in \mathbb{R}^C$ is of the form:

$$c : C \rightarrow \mathbb{R}$$

Throughout the paper, we will be using element substitution when modifying select values from valuation functions i.e. given variable valuation functions $\{v, v'\} \subseteq \Lambda$, we say that an expression of the form $v'' = v[v(x)/v'(x)]$ substitute the entry of the variable x in v with the entry in v' but all other valuations remain.

Additionally, we define the operator for function composition over sets by the symbol \circ i.e. given a list of functions $F = [f, g, h]$ we say that:

$$f \circ g \circ h = \bigcirc_{\alpha \in F} \alpha$$

Note that the order of the list may be important depending on the functions.

B. Syntax and Semantics

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

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

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

The set of guards $g \in G$ are guarding boolean functions over a variable and clock valuation: $g : \Lambda \times \mathbb{R}^C \rightarrow \mathbb{B}$, and the set of updates $u \in U$ are variable valuation modifier functions $u : \Lambda \rightarrow \Lambda$, typically defined through an assignment expression syntax e.g. $\alpha := \beta + 5$ where $\{\alpha, \beta\} \subseteq V$. An edge $e \in E$ is then a tuple from a source location $s \in L$ to a target location $t \in L$ annotated with a guard, update and a set of clocks $r \in 2^C$ that are to be reset to zero when the edge is chosen.

With the syntactic elements defined we can now define the semantics, of which we split into two parts: The tick-step and tock-step semantics (hence the name of the formalism). Loosely described, the tick-step is where we traverse the control flow graph that the location and edges-set define and the tock-step will modify the external variables based off of external input.

Definition 1.2. *tock semantics*

Formally, a tock-step is defined intentionally very vaguely by the two variable valuation modifier functions Γ and γ as defined in Equation 1 and Equation 2 respectively

First, we non-deterministically choose a value for each external variable. This non-determinism emulates that the external environment of an input/output system is entirely unpredictable and could potentially change to any value.

$$\Gamma(v) = v[v(x)/v'(x)], \forall x \in \Omega \text{ for some } v' \quad (1)$$

Then we non-deterministically choose a value to delay all clocks by. This emulates that it might take a while for a physical system to read the inputs.

$$\gamma(c) = \text{for some } d \geq 0, c[c(x)/c'(x)]. \\ \forall x \in C \text{ s.t. } c'(x) = c(x) + d \quad (2)$$

The full tock-step is then performed via function composition $\gamma \circ \Gamma$

The tick-step semantics for a single TTA is intended to be fairly intuitive. We provide the semantics in the form of Structured Operational Semantics (SOS).

Definition 1.3. *SOS rules for a single TTA*

To perform a tock-step, we can choose between two rules: **(Tock-2)** and **(Tock-1)** whereas the first rule refers to a tock-step that happens right after another tick-step, and the second tock rule starts from the *initial* location.

$$\frac{}{\langle l_0, v_0, c_0 \rangle \xrightarrow{\text{tock}} \langle l_0, v', c' \rangle} \text{ where } \begin{array}{l} v' \in \Gamma(v) \wedge \\ c' \in \gamma(c) \end{array} \quad (\text{Tock-1})$$

$$\frac{\langle l'', v'', c'' \rangle \xrightarrow{\text{tick}} \langle l, v, c \rangle}{\langle l, v, c \rangle \xrightarrow{\text{tock}} \langle l, v', c' \rangle} \text{ where } \begin{array}{l} v' \in \Gamma(v) \wedge \\ c' \in \gamma(c) \end{array} \quad (\text{Tock-2})$$

To perform a tick-step there must be a tock-step before it, where the final state would be the result of taking an enabled transition $e \in E$.

$$\frac{\langle l, v'', c'' \rangle \xrightarrow{\text{tock}} \langle l, v, c \rangle}{\langle l, v, c \rangle \xrightarrow{\text{tick}} \langle l', v', c' \rangle} \text{ where } \begin{array}{l} \exists e = \langle l, g, u, r, l' \rangle \in E \text{ s.t.} \\ g(v, c) = \text{true} \wedge \\ r(c) = c' \wedge \\ u(v) = v' \end{array} \quad (\text{Tick})$$

To see the SOS rules in action, consider the example TTA in Figure 2 consisting of three locations $l1, l2$ and $l3$ connected by the edges $E = \{t1, t2, t3\}$ and starts at location $l1$. In Figure 1 we apply the SOS rules to reach a state where i is incremented once. This is far from the entire reachable state space, which in fact is infinite since there's an infinite loop of tick/tock steps between $l1$ and $l2$, but it illustrates how to apply the rules.

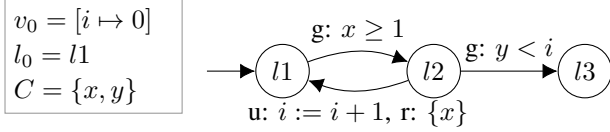


Figure 2. Example TTA with location $l3$ being unreachable

C. Parallel TTAs

Being limited to a single TTA can be challenging when designing an automation system. So in this section we will provide the syntax and semantics of executing multiple instances of interconnected TTAs in parallel.

Definition 1.4. *Network of Tick Tock Automata (NTTA) syntax*
The syntax of an NTTA is fairly simple:

Let N be a network of n TTAs where $A^i = \langle L^i, V, \Omega, C, E^i, l_0^i, v_0, c_0 \rangle$ are TTAs for all $n \geq i \geq 0$.

Note that the network shares the same variable sets and valuations across the TTAs.

1) NTTA semantics: Since the TTAs in a network all share the same variable and clock sets, the tock-step functions Γ and γ aren't changed. However, the tick-step semantics are a bit more complex than previously described, which is mostly due to the increased amount of non-determinism that parallelism introduces. The amount of non-deterministic decisions increases in the case where there are multiple enabled edges across the network as expected, but there is increased complexity in the fact that applying updates is not necessarily order-independent.

As an example, consider a network of two TTAs, each with one enabled edge a and b . They both change the variable

x , but to differing values. If we were to apply both updates together, there would be differing answers on the result of x 's value depending on the order in which we applied the updates. Instead, there is simply two choices: Take a or take b individually, and we disallow the simultaneous ab transition. We call this situation an *overlapping non-idempotent update conflict*, and the tick-step for NTTAs must be able to handle such situations.

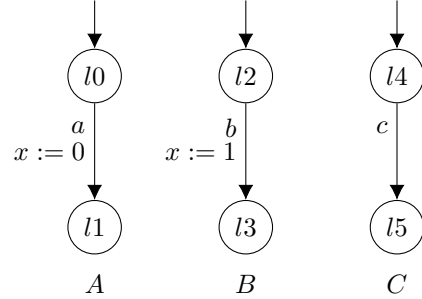


Figure 3. Example network with one update conflict and a third independent edge, each with an empty guard expression (shorthand for tt .)

Additionally, we want the tick-step semantics to be maximal in the sense that if a TTA in the network have a non-conflicting enabled edge, it must be included in the tick-step. As an example, consider the network of three TTAs as displayed in Figure 3 each with an enabled edge a, b and c . Simply taking a or b by themselves should not be permitted by the semantics, because c is enabled and in no conflict with any of the other edges, and therefore *must* be taken as well. Note that c cannot be taken by itself either, the semantics *must* also choose between a or b . So in the example, the valid choices are: $\langle a, c \rangle$ or $\langle b, c \rangle$.

To define an algorithm that fulfills the needs of such semantics, we first provide some helper definitions. $\mathbb{C} = \Lambda \times \Lambda \times 2^V \rightarrow 2^V$ defined in Equation 3 provides the set of differing valuations/changes by comparing the two provided variable valuation functions.

$$\mathbb{C}(v, v', V) = \{a | a \in V \text{ where } v[a] \neq v'[a]\} \quad (3)$$

In order to figure out whether there are conflicting changes

$$\begin{array}{ll} \text{(Tock-1)} & \overline{(l1, [i \mapsto 0], [x \mapsto 0, y \mapsto 0]) \xrightarrow{\text{tock}} (l1, [i \mapsto 0], [x \mapsto 1, y \mapsto 1])} \quad (\text{delay clocks by 1}) \\ \text{(Tick)} & \overline{(l1, [i \mapsto 0], [x \mapsto 1, y \mapsto 1]) \xrightarrow{\text{tick}} (l2, [i \mapsto 0], [x \mapsto 1, y \mapsto 1])} \quad (t1) \\ \text{(Tock-2)} & \overline{(l2, [i \mapsto 0], [x \mapsto 1, y \mapsto 1]) \xrightarrow{\text{tock}} (l2, [i \mapsto 0], [x \mapsto 2, y \mapsto 2])} \quad (\text{delay clocks by 1}) \\ \text{(Tick)} & \overline{(l2, [i \mapsto 0], [x \mapsto 2, y \mapsto 2]) \xrightarrow{\text{tick}} (l1, [i \mapsto 1], [x \mapsto 0, y \mapsto 2])} \quad (t2) \end{array}$$

Figure 1. Example application of the SOS rules to trace $t1.t2$. Note that the variable valuation only changes in $t2$

from one valuation to two new valuation choices, we provide $I = \Lambda \times \Lambda \times \Lambda \times 2^V \rightarrow \mathbb{B}$ in Equation 4.

$$I(v, v', v'', V) = \exists a \in \mathbb{C}(v, v', V) \cap \mathbb{C}(v, v'', V) \quad \text{s.t. } v'[a] \neq v''[a] \quad (4)$$

The function $X = E \times E \rightarrow \mathbb{B}$ (defined in Equation 5) is then an abstraction over \mathbb{C} and I , so we can check if there are any conflicting changes between two edges.

$$X(\langle s^1, g^1, u^1, r^1, t^1 \rangle, \langle s^2, g^2, u^2, r^2, t^2 \rangle) = s^1 = s^2 \vee I(v, u^1(v), u^2(v), V) \quad (5)$$

To finish off the preliminaries, we define the postset of a node in a graph $n \bullet$ as the set of nodes that have an incoming edge from the node n in Equation 6.

$$n \bullet = \{x \mid \forall \langle n, x \rangle \in E\} \quad (6)$$

In Algorithm 1 we define an algorithm that can create a dependency graph from a given NTTA state. To avoid confusion we will refer to the elements in this dependency graph as *nodes* and *connections*. Not to be confused with *locations* and *edges* which are TTA syntax elements. Each node represents a TTA edge, and each connection represents an overlapping update conflict or TTA choice.

Algorithm 1 Generate dependency graph, where nodes are enabled TTA edges. Connections are conflicting updates or differing choices in TTA

```

1: procedure EDGEDEPGRAPH( $E, V, \langle \bar{l}, v, c \rangle$ )
2:    $N = Q = \emptyset$ 
3:   for each  $l \in \bar{l}$  do
4:     for each  $x \in \{e \mid e = \langle l, g, u, r, l' \rangle \in E \text{ where } g(v, c) = tt\}$  do
5:       for each  $n \in N$  do
6:         if  $X(x, n)$  then
7:            $Q = Q \cup \{\langle x, n \rangle\}$ 
8:        $N = N \cup x$ 
9:   return  $\langle N, Q \rangle$ 

```

Consider the example network of three TTAs in Figure 4. In the initial state there are four enabled edges a, b, c, d and e and if we find the resulting edge dependency graph we would get the graph pictured in Figure 5.

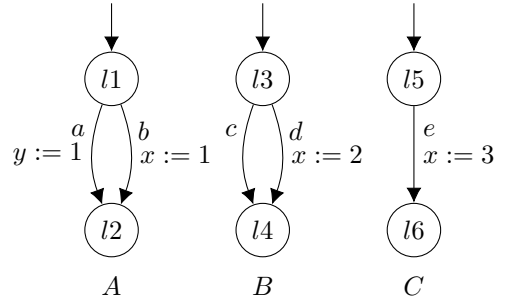


Figure 4. Multiple transitions should be possible in selective parts of the logic

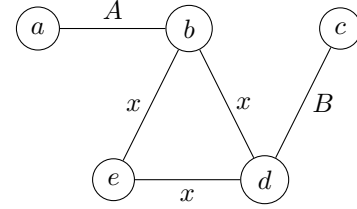


Figure 5. Dependency graph with three types of dependency connections. Connection labels refer to network instances (A, B) and variables (x) from Figure 4, similarly, node labels refer to the edge names (a, b, c, d, e)

With a dependency graph, we can now find all the maximal combinations of edge choices that are not in conflict, in this example the available maximal solutions are: $\langle a, c, e \rangle$, $\langle a, d \rangle$ and $\langle b, c \rangle$. The algorithms for finding these maximal solutions are provided in Algorithm 2 and Algorithm 3.

Algorithm 2 Accumulate maximal solutions iteratively

```

1: procedure MAXNOTCONNECTED( $G = \langle N, E \rangle$ )
2:    $S = \emptyset$ 
3:   for each  $n \in N$  do
4:     FINDSOLUTION( $S, \{n\}, N$ )
5:   return  $S$ 

```

Algorithm 3 Recursively find maximal set of non-connected nodes in a graph and insert them into the S set

```

1: procedure FINDSOLUTION( $S, \alpha, N$ )
2:   if  $\exists s \in S$  where  $\alpha \subseteq s$  then
3:     return
4:    $K = N \setminus (\alpha \cup \bigcup_{x \in \alpha} x \bullet)$ 
5:   if  $K = \emptyset$  then
6:      $S = S \cup \alpha$ 
7:   for each  $k \in K$  do
8:     FINDSOLUTION( $S, \alpha \cup \{k\}, N$ )

```

For convenience, we combine the algorithms for finding the dependency graph and maximal non-connected sets into one shorthand function called TS (tick-step solutions). Such a

function would return a set of sets of TTA edges representing network choice options, or more formally: $TS : 2^E \times 2^V \times S \rightarrow 2^{2^E}$. We define the function in Equation 7 below.

$$TS(E, V, s = \langle \bar{l}, v, c \rangle) = \text{MAXNOTCONNECTED}(\text{EDGEDEPGRAPH}(E, V, s)) \quad (7)$$

Definition 1.5. *SOS rules for a network of TTAs (NTTA)*

We can now define the full NTTA semantics via the SOS rules **NTick**, **NTock-2** and **NTock-1**. The tock-step rules are very similar to what we defined in definition 1.3 because NTTAs share variables and clocks across the network and Equation 1 and Equation 2 don't affect the location space. **NTick** on the other hand has changed quite a lot.

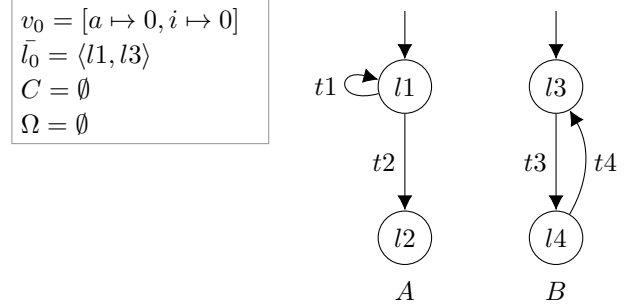
$$\frac{}{\langle \bar{l}_0, v_0, c_0 \rangle \xrightarrow{\text{tock}} \langle \bar{l}_0, v', c' \rangle} \text{ where } \begin{array}{l} v' \in \Gamma(v) \wedge \\ c' \in \gamma(c) \end{array} \quad (\text{NTock-1})$$

$$\frac{\langle \bar{l}'', v'', c'' \rangle \xrightarrow{\text{tick}} \langle \bar{l}, v, c \rangle}{\langle \bar{l}, v, c \rangle \xrightarrow{\text{tock}} \langle \bar{l}, v', c' \rangle} \text{ where } \begin{array}{l} v' \in \Gamma(v) \wedge \\ c' \in \gamma(c) \end{array} \quad (\text{NTock-2})$$

$$\frac{\langle \bar{l}'', v'', c'' \rangle \xrightarrow{\text{tock}} \langle \bar{l}, v, c \rangle}{\langle \bar{l}, v, c \rangle \xrightarrow{\text{tick}} \langle \bar{l}', v', c' \rangle} \text{ where } \begin{array}{l} \exists \alpha \in TS(E, V, \langle \bar{l}, v, c \rangle) \text{ s.t.} \\ \forall \langle l, g, u, r, l' \rangle \in \alpha. l \in \bar{l} \wedge l' \in \bar{l}' \wedge \\ v' = (\bigcirc_{\langle l, g, u, r, l' \rangle \in \alpha} u)(v) \wedge \\ c' = (\bigcirc_{\langle l, g, u, r, l' \rangle \in \alpha} r)(c) \end{array} \quad (\text{NTick})$$

D. Example NTTA SOS Usage

When introducing multiple TTAs parallelly, the complexity and therefore the reachable statespace have the potential to increase dramatically. Consider the NTTA depicted in Figure 6 with two TTAs *A* and *B*, where in isolation each TTA is rather boring. *A* would either take *t1* infinitely, or take *t2* and *B* would be stuck forever, since the guard of *t3* would never be true. However, when composing them together in parallel, and make them share the variable pool *B* can reach *l4* and will add another infinite amount of reachable states by cycling *t3, t4*, even if *A* is stuck in *l2*.



Edge	Guard	Update	Clock Resets
<i>t1</i>	<i>tt</i>	<i>a</i> := <i>a</i> + 1	\emptyset
<i>t2</i>	<i>a</i> > 0	<i>i</i> := <i>i</i> + 1	\emptyset
<i>t3</i>	<i>a</i> > 0	<i>i</i> := <i>i</i> + 1	\emptyset
<i>t4</i>	<i>tt</i>	ϵ	\emptyset

Figure 6. text

Figure 7 shows an example application of the SOS rules, where the location vector $\langle l2, l4 \rangle$ is reached by taking the series of edges $\langle t1 \rangle \rightarrow \langle t3, t1 \rangle \rightarrow \langle t2 \rangle$. Since $\Omega = \emptyset$ and $C = \emptyset$, no tock-step transition can change the state, so nothing happens during that step. Note that the transition sequence $\langle t1 \rangle \rightarrow \langle t2, t3 \rangle$ is also a valid way of reaching $\langle l2, l4 \rangle$. We encourage the reader to apply the SOS rules themselves to verify this.

E. Reachability

For the sake of brevity, we provide the notion of the tick-postset \bullet (states reachable in one tick-step from *s*) and the tock-postset \circ (states reachable in one tock-step from *s*) in Equation 8 and Equation 9 respectively.

$$\begin{aligned} \bullet(\langle \bar{l}, v, c \rangle) &= \{s' \mid s' = \langle \bar{l}', v', c' \rangle \text{ s.t.} \\ &\quad \forall \alpha \in TS(E, V, \langle \bar{l}, v, c \rangle) \text{ where} \\ &\quad \forall \langle l, g, u, r, l' \rangle \in \alpha. l \in \bar{l} \wedge l' \in \bar{l}' \wedge \\ &\quad v' = (\bigcirc_{\langle l, g, u, r, l' \rangle \in \alpha} u)(v) \wedge \\ &\quad c' = (\bigcirc_{\langle l, g, u, r, l' \rangle \in \alpha} r)(c)\} \end{aligned} \quad (8)$$

$$\begin{aligned} \circ(\langle \bar{l}, v, c \rangle) &= \{s' \mid s' = \langle \bar{l}, v', c' \rangle \text{ s.t.} \\ &\quad v' \in \Gamma(v) \wedge c' \in \gamma(c)\} \end{aligned} \quad (9)$$

We write $s\bullet$ as a shorthand for $\bullet(s)$ and $s\circ$ as shorthand for $\circ(s)$.

To perform reachability analysis on NTTA, we provide a slightly altered version of the forward reachability search

algorithm [5] in Algorithm 4. The alternations are at Line 8 and Line 10 which ensures that a for every reachable tick-state, we must also take a tick-step.

Algorithm 4 Forward reachability search algorithm for networks of TTAs. Finds a state that satisfies the reachability query φ together with a back-tracable graph.

```

1: procedure FORWARDREACHABILITYNTTA( $s_0, \varphi$ )
2:    $W = \{\langle \epsilon, s_0 \rangle\} \cup \{\langle s_0, s' \rangle \mid s' \in s_0 \circ\}$ 
3:    $P = \emptyset$ 
4:   while  $W \neq \emptyset$  do
5:      $s = W.pop()$ 
6:      $P = P \cup \{s\}$ 
7:     if  $s \models \varphi$  then return  $\langle tt, s \rangle$ 
8:     for each  $s' \in s \bullet \cap P$  do
9:        $W = W \cup \{\langle s, s' \rangle\}$ 
10:    for each  $s'' \in s' \circ \cap P$  do
11:       $W = W \cup \{\langle s', s'' \rangle\}$ 
12:  return  $\langle ff, \epsilon \rangle$ 

```

F. Interestingness

Note that the available tick-step set can be quite large, in fact, there are as many tick reachable states from any state s as there are possible clock valuations times the amount of possible variable valuations of the external variables $|s \circ| = |2^C| \times |2^{\Lambda_\Omega}|$. If the sets Ω and C are both empty or even suitably small then there shouldn't be much of a problem, but the reachable state space will explode exponentially if left unchecked. However, many (if not all) of these states will not have any effect for finding a result of a given reachability query, so we introduce the notion of interestingness to the tick-step.

To determine if a given state s has any reachable interesting tick-steps, we look at all outgoing edges from s . If one of those edges' guard expression references a variable in the Ω -set or a clock in the C -set, we say that state s can be affected by tick-step manipulation. To determine specifically what assignments of those variables/clocks result

in a satisfied/unsatisfied guard, we utilize a modern SAT solver such as Z3 [6] that can provide variable assignment values for SAT/UNSAT if available. As a demonstration of this approach, consider the example depicted in Figure 8 with three parallel TTAs and four tick-manipulatable guard expressions available in s_0 .

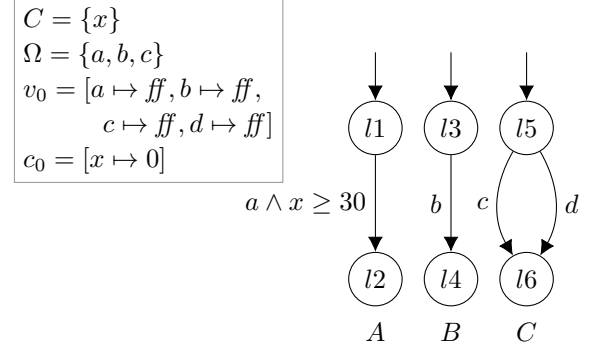


Figure 8. Example NTTA demonstrating external variable comparisons in guards

Considering only the clock x , the reachable tick-statespace is already infinitely big. However, we don't want to explore the states $x \mapsto 0$, $x \mapsto 1$, $x \mapsto 2$ etc. We are actually only interested in the variable- and clock-assignments that can enable a guard expression or force it to be disabled. To do this, we have to modify the guard expression a little bit by adding a delta (δ) value to all clock references with the *Mod* function $Mod : G \rightarrow G$:

$Mod(\phi)$ Input	Result
tt	tt
ff	ff
(ϕ_1)	$Mod(\phi_1)$
$\phi_1 \bowtie \phi_2$	$Mod(\phi_1) \bowtie Mod(\phi_2)$
$\neg \phi_1$	$\neg Mod(\phi_1)$
α	$(\alpha + \delta)$ if $\alpha \in C$, otherwise α

Where $\bowtie = \{<, \leq, =, \neq, \geq, >, \wedge, \vee, -, +, \times, /\}$ is the set of binary operators available in a guard expression. Additionally,

(NTock-1)	$\langle \langle l1, l3 \rangle, [a \mapsto 0, i \mapsto 0], \epsilon \rangle \xrightarrow{Tick} \langle \langle l1, l3 \rangle, [a \mapsto 0, i \mapsto 0], \epsilon \rangle$	(empty tick)
(NTick)	$\langle \langle l1, l3 \rangle, [a \mapsto 0, i \mapsto 0], \epsilon \rangle \xrightarrow{Tick} \langle \langle l1, l3 \rangle, [a \mapsto 1, i \mapsto 0], \epsilon \rangle$	($\langle t1 \rangle$)
(NTock-2)	$\langle \langle l1, l3 \rangle, [a \mapsto 1, i \mapsto 0], \epsilon \rangle \xrightarrow{Tick} \langle \langle l1, l3 \rangle, [a \mapsto 1, i \mapsto 0], \epsilon \rangle$	(empty tick)
(NTick)	$\langle \langle l1, l3 \rangle, [a \mapsto 1, i \mapsto 0], \epsilon \rangle \xrightarrow{Tick} \langle \langle l1, l4 \rangle, [a \mapsto 2, i \mapsto 1], \epsilon \rangle$	($\langle t3, t1 \rangle$ non-det. tick)
(NTock-2)	$\langle \langle l1, l4 \rangle, [a \mapsto 2, i \mapsto 1], \epsilon \rangle \xrightarrow{Tick} \langle \langle l1, l4 \rangle, [a \mapsto 2, i \mapsto 1], \epsilon \rangle$	(empty tick)
(NTick)	$\langle \langle l1, l4 \rangle, [a \mapsto 2, i \mapsto 1], \epsilon \rangle \xrightarrow{Tick} \langle \langle l2, l4 \rangle, [a \mapsto 2, i \mapsto 2], \epsilon \rangle$	($\langle t2 \rangle$ non-det. tick)

Figure 7. Application of SOS rules for networks of Tick Tock Automata. Notice the non-determinism ticks

we define the function T , that finds all variables that a guard expression tests $T : G \rightarrow (V \cup C)$:

$T(\phi)$ Input	Result
tt	\emptyset
ff	\emptyset
(ϕ_1)	$T(\phi_1)$
$\phi_1 \bowtie \phi_2$	$T(\phi_1) \cup T(\phi_2)$
$\neg \phi_1$	$T(\phi_1)$
α	$\{\alpha\}$

Combining these functions, we can then generate the set of guards from state s that are *interesting*, modified with the Mod function and negated (see Equation 10).

$$Y(s = \langle \bar{l}, v, c \rangle) = \{(m, \neg(m)) \mid m = Mod(g) \\ \forall e = \langle l, g, u, r, l' \rangle \in E : l \in \bar{l} \wedge T(g) \subseteq (\Omega \cup C)\} \quad (10)$$

We need this set to explore all interesting assignments and delays that can force some, but not necessarily all of the guards. We can then feed the cross product of these modified guard expressions to the Z3 SAT solver (For the sake of brevity, we simplify the Z3 function just to the type: $Z3 : G \rightarrow \Lambda \times 2^C$ and will refer to [7, 8] for implementation details). Combining it all we get the new, interesting tick-step function depicted in Equation 11.

$$\circ(s) = \{Z3(x_1 \wedge x_2 \wedge \dots \wedge x_n) \mid \\ \forall (x_1, x_2, \dots, x_n) \in \bigtimes_{Y(s)}\} \quad (11)$$

III. HAWK

The somewhat concise nature of the syntax of an NTTA is very good for verification and makes it notationally nice to work with. But it does not scale very well when applied in a practical setting, where potentially hundreds of TTAs have to be defined and maintained. We will therefore provide a layer of syntactic sugar called HAWK (Hardware Abstraction With Knowledge) that provides all of the practicalities required by a Model Based Development pipeline such as: templated composition, hardware integration metadata and TTA instance parameterization. HAWK syntax should be directly translatable into NTTA, which can then be executed and analyzed by the AALTITOAD tool.

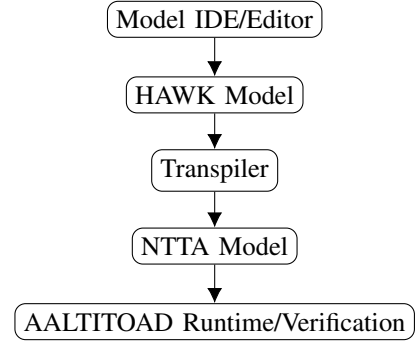


Figure 9. The HAWK transpilation pipeline

In the following subsections we will provide a general description of the transpilation subroutines required to transpile HAWK to NTTA (Keep in mind that these transpilation routines are subject to change as the HAWK language might be extended and/or modified in the future).

A. Templated Composition and Parameterization

Similar to classes or structs in object-oriented programming languages, HAWK supports a structured way of defining and instantiating classes of functionality. In HAWK, a class is called a *template* and instantiations are called simply *instances*. Figure 10 illustrates an example of what elements a template consists of. A template will always contain two locations: the initial and final location. The initial location is restricted to only allow outgoing edges, and the final location is similarly restricted to only allow incoming edges. Every instance starts in their respective initial location. Since a TTA is unable to "exit" we stitch the final and initial locations together by making them the same location, making the instance automatically loop. This looping behavior helps designers to structure their code in looping logics that will tend not to stall/deadlock and provides a bit more structure to the design of the control logics. By convention, the initial and final location will have different names even though they will be compiled into one location during de-sugaring.

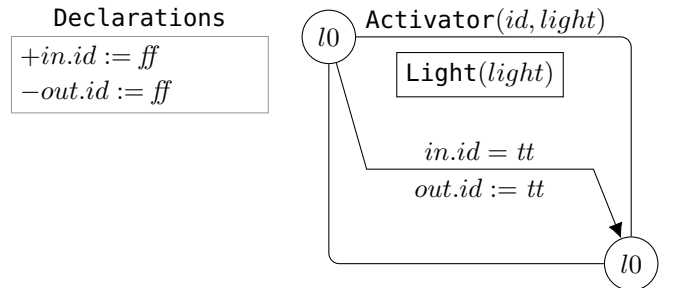


Figure 10. Simple example of the anatomy of a HAWK template instantiation

Any template can declare parallel instances of any other template which are then flattened into one big list of TTA instances, in this example, any instantiation of `Activator` will also instantiate one `Light` template parameterized with the argument `light`.

Each template can provide a list of variable declarations. These are simply added to the global list of variables, but they might be marked as "private" (by using the minus symbol in this example¹) which limits them to be only mentionable in instances of the declaring template or its children declarations. Public variables will be registered only once into the global symbol list, no matter how many times a template is instantiated.

A template can be parameterized with formal parameters that can be used in local expressions and declarations of partial variable identifiers. In our example we have the parameter `id` that is used in the partial variable identifiers `in.id`² and `out.id`. As an example, consider two `Activator` instantiations: `Activator(0,light0)` and `Activator(1,light1)`, there will be registered $\{in.0, out.0, in.1, out.1\}$ to the V -set.

Since instantiation has to start somewhere, the template name `Main` is reserved and must not be instantiated by any other templates. Additionally, there must not be any recursive or circular instantiation, since what would effectively mean you would have to create an infinite amount of TTAs in your network. This restriction can be solved with a simple dependency graph analysis using e.g. Tarjan's SCC algorithm [9].

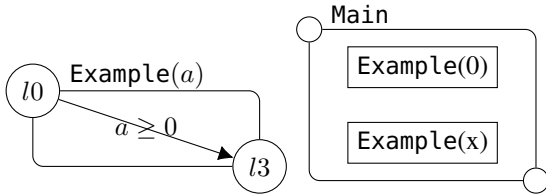


Figure 11. Example instantiation of two `Example` templates, one parameterized with 0 and one with x .

B. External

In order to register a variable in the Ω -set, HAWK provides a registered keyword: `extern`. In practice, this is mainly relevant for verification engines, but it can also provide a nice limitation on which variables are modifiable by a tocker program.

¹in AALTITOAD, we use `private/public` instead of plus and minus signs

²Variable identifiers are allowed to contain dots. It's just part of the name, and makes it easier to understand what is going on in large programs

C. Persistence

On embedded systems, in case of a sudden unexpected power loss, sometimes you want to store some value to the hard disk. This is enabled by HAWK by adding a `persistent` keyword to the declaration. Any update to a persistent variable should be saved to disk before the next tick-step is issued, but it is up to the runtime implementation to determine when and how to store the value. All persistent symbols should be loaded from disk at startup before any ticks or tocks have been issued. When verifying models with persistent symbols, we can over-approximate the state space by placing them in the Ω set.

D. Hardware Integration Metadata

For some variables, it is important to keep track of which ones relate to the external input and output ports and which piece of hardware they are actually meant to refer to. To make this easier, HAWK provides a way of declaring a collection of metadata associated with each variable. This metadata can then be looked up by tocker integration programs to filter for types of variables that are associated with their specific hardware integration. For regular non-hardware variables, the metadata entry is simply not provided. An example of how such metadata might be added in a json encoding style is depicted in Figure 12. Since metadata is designed for runtime tocker integration programs, it will be ignored by the verification engine.

```
{
  "symbols": [
    {
      "id": "a",
      "type": "int",
      "val": 0
    },
    {
      "id": "b",
      "type": "int",
      "val": 0,
      "metadata": {
        "type": "digital_input",
        "address": 31
      }
    }
  ]
}
```

Figure 12. Example symbol declaration json encoding with integration metadata. Only the `b` symbol provides metadata

IV. APPROACH

The AALTITOAD project provides a couple of front-end command-line interfaces: `simulator` and `verifier` for executing and verifying your models respectively. If you wish to write your own front-ends, the library `libaaltitoad` provides the core functionality you need. In this section we will provide an overview of the architecture of an AALTITOAD front-end, and we hope that after reading this section, you will have a preliminary understanding of how to

design your own integrations or front-end. For a more in-depth and up-to-date experience, please check out the git repository.

A. Tickers And Tockers

Integrating with hardware requires a piece of programming that can modify the variable list. There are two types of integration programs, affectionately called tickers and tockers. A ticker-program is executed (ideally in parallel with all other ticker programs) once every tick-step is performed, and vice-versa for tocker-programs and the tock-step. In fact, the verification front-end provides a special tocker-program that calculates all the interesting tock-choices available.

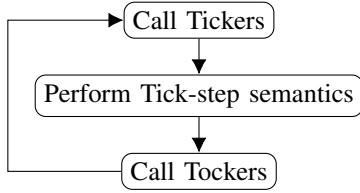


Figure 13. Sequence of events performed by the AALTITOAD runtime

There are two general ways of implementing a tocker program: blocking or parallelly executing. As an example, consider a simple tocker program that needs to fetch some data from a REST API when called. We could send the http request, and wait for the response from the target server, but what happens if either the network connection server is very slow, or worse yet, there is no connection at all. The request would block the thread of execution, and our REST API tocker would starve the tick-step semantics from executing for no good reason. Instead, we provide a generalized asynchronous tocker interface, that executes the http request in a separate thread. When the request has completed, then it caches the variable changes that should be applied and next time the tocker is invoked, it will simply return the cached value and start the request again.

If it is important that the tick-step must not continue until your tocker program has completed then there is no other choice than to use a blocking tocker, but in general, it is recommended to use always asynchronous tockers.

B. Parsers

As mentioned in Section III, AALTITOAD provides support for custom syntactic sugaring such as the HAWK language. At the time of writing, the project provides a single default parser that assumes the project files are written in json-format and have the keys defined by the H-UPPAAL editor [10, 11]. A user might want to use their own syntax instead, or maybe their project files are written as yaml, raw text or even binary

files. Therefore we provide the opportunity to write a custom parser plugin that, as long as the plugin will provide a valid NTTA, the engine can do all of the regular actions and analyses that it normally can.

V. EXPERIMENTAL SETUP

To evaluate the effectiveness of the maximal step semantics and Z3-based tock-space search algorithm, we will investigate how the previous release of AALTITOAD (v0.10.2) handles reachability queries on an implementation of the fischer mutual exclusion protocol compared to the new version (v1.2.0 at time of writing). An example NTTA implementation of the protocol can be seen at Figure 14. Since we are using the HAWK syntax, we can scale the test suite complexity up and down simply by instantiating or removing instances of Fischer templates in the Main template.

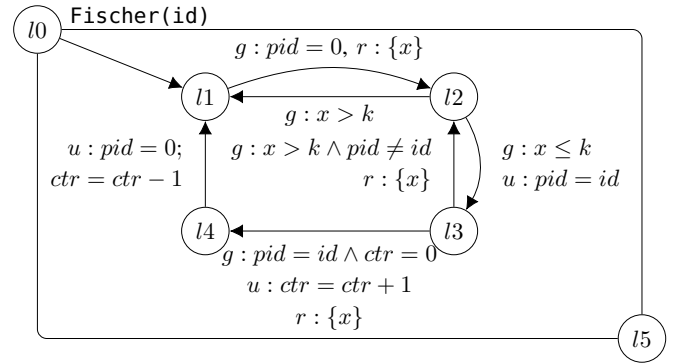


Figure 14. Fischer's mutual exclusion algorithm as a Tick Tock Automata using the HAWK sugar language

We executed a suite of reachability queries (see Figure 15) on eight variations of the protocol, on a machine equipped with an Intel i5-6300U at 3GHz and 8 gigabytes of memory. The operation was cancelled if memory consumption exceeded the 8GiB available on the system (once swap was beginning to get populated).

$$\begin{aligned}
 &\exists \diamond ctr > 1 \\
 &\exists \diamond ctr < 0 \\
 &\exists \diamond ctr = 1
 \end{aligned}$$

Figure 15. Queries used to test the fischer test suite. Only the last query is expected to be satisfied

VI. EXPERIMENTAL RESULTS

As evident by the results summary in Figure 16, the older version of AALTITOAD was unable to solve some of the fischer variations due to the drastically higher memory usage.

Interestingly however, version **v1.2.0** was generally slower but less memory intensive. Except in the fischer-3 case, this is likely to be a bug in version **v0.10.2** due to the suspicious non-exponential increase in states. When running **v1.2.0** with a profiler attached, 98% of the time is spent inside the Z3 SAT solver.

version	fischer	time	$ P $
v1.2.0	fischer-2	501ms	13
v1.2.0	fischer-3	2074ms	16
v1.2.0	fischer-4	9747ms	19
v1.2.0	fischer-5	44748ms	22
v1.2.0	fischer-6	205413ms	25
v1.2.0	fischer-7	920772ms	28
v1.2.0	fischer-8	4268392ms	31
v1.2.0	fischer-9	19152675ms	34
v0.10.2	fischer-2	81ms	127
v0.10.2	fischer-3	2634ms	3630
v0.10.2	fischer-4	2850ms	3827
v0.10.2	fischer-5	21837ms	23621
v0.10.2	fischer-6	155400ms	140367
v0.10.2	fischer-7	OOM	OOM
v0.10.2	fischer-8	OOM	OOM
v0.10.2	fischer-9	OOM	OOM

Figure 16. Test results comparing old and new versions, **OOM** = Out Of Memory. Text in **bold** means that it is the best value for the test set

When looking at a plot of the $|P|$ -column (Figure 17), we can see that the amount of reachable states increases exponentially in **v0.10.2**, and linearly in **v1.2.0**. This memory usage difference is most likely due to the way **v0.10.2** overapproximates assignments for boolean comparators in guards during interesting tock changes calculation [2], which is known to be faulty with guards containing multiple nested parentheses³ and creates more states than possibly necessary. Figure 18 depicts the amount of time, in milliseconds (logarithmic y-axis) spent finding the reachable statespace for each test suite. From this, we can deduce that **v0.10.2** is much faster at generating states due to not involving any SAT solvers, but since it generates so many more states, we end up spending a comparable amount of time searching through them compared to **v1.2.0**.

³See issue #14 on the source repository: github.com/sillydan1/aaltitoad

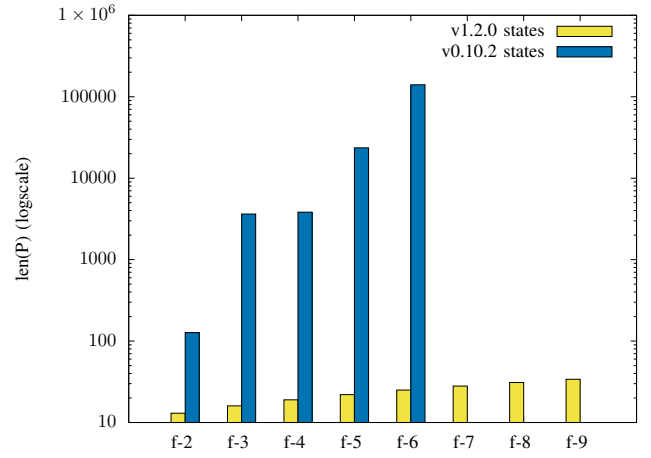


Figure 17. Unique states generated ($|P|$) per fischer test set. Note logarithmic y-axis

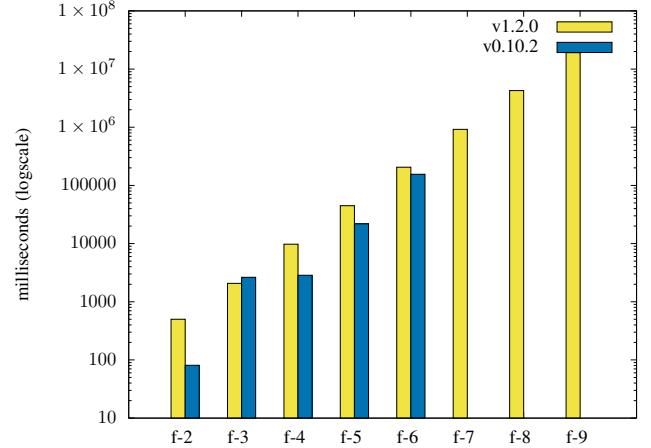


Figure 18. Amount of time (milliseconds) used to find the reachable state space. Note logarithmic y-axis

In Figure 19 and Figure 20 we see the amount of states present in the waiting set W for each iteration of the forward reachability search algorithm described in Algorithm 4 for **v1.2.0** and **v0.10.2** respectively. As expected the figures depict a hump where the peak is somewhere in the middle of the search, followed by a decline with occasional minor bumps as the search algorithm starts to revisit states from P again. **v1.2.0** is obviously much more memory conservative and doesn't even break into the double digits when searching through the fischer-9 test variant.

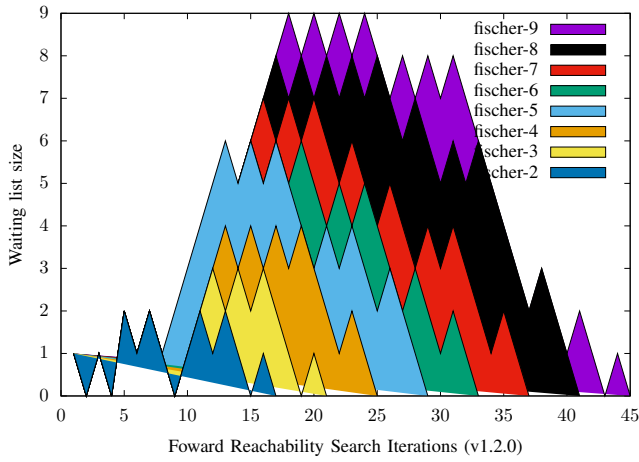


Figure 19. Waiting set size ($|W|$) during forward reachability search in AALTITOAD version v1.2.0

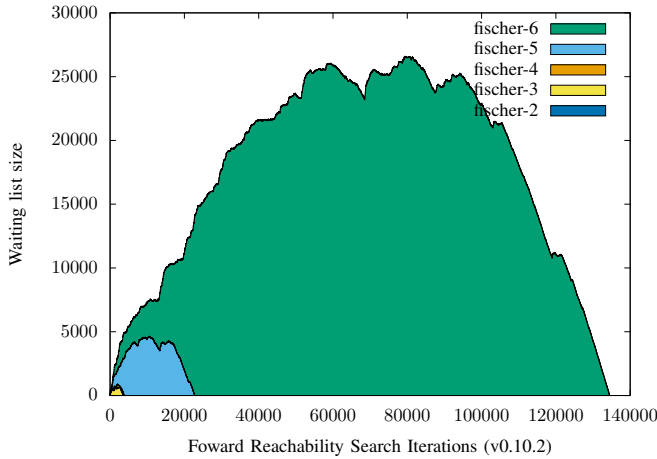


Figure 20. Waiting set size ($|W|$) during forward reachability search in AALTITOAD version v0.10.2

VII. CONCLUSION

The new release of AALTITOAD provides a more versatile experience for developers wanting to embrace a Model Based Development pipeline. It is much more extendable and can act as a drop-in replacement for the old version with only some tweaks to the commandline flags. It provides a suite of front-ends, and new front-ends are easy to put together such as additional static-analysis tools for the MBD pipeline. The new maximal-step tick-semantics provide a heuristic to the runtime when encountering non-determinism and a more thorough, accurate and memory efficient verification experience. The Z3 SAT solver tocker integration enables us to answer more queries using much less memory and with only a minor time-performance hit.

VIII. FUTURE WORK

A. Extended Query Language Support

Currently, the verifier front-end only supports reachability queries using Computation Tree Logic (CTL) as a query language. It would be straight forward to implement support for safety CTL queries, since it is just a negated version of reachability. However, CTL is able to describe more than just reachability and safety queries and it would be nice if the verifier supported the entire query language. In addition to CTL, it would also be beneficial to the practicality of the tool to support queries written with Linear Temporal Logic (LTL). This would require a whole different approach to the search algorithm, but by using other verification engine implementations as inspiration, this should be achievable even with the small development team currently available.

B. Better MBD Tool Suite Integration

To provide a smooth and as helpful as possible MBD experience, AALTITOAD should provide a specification and reference implementation of a GDB-style debugging experience. This protocol specification could be implemented by graphical model editors such as H-Uppaal, so a developer can simulate locally or "attach" to a remote server. From here they should be able to add and remove breakpoints, inspect symbols, location state, edit the current state and perform general debugging on their model. A good inspiration reference would be to see how the VSCode Debug Adapter Protocol [12] is architected. In order to help the developer during the modelling phase, AALTITOAD should also provide a specification and reference implementation of a linter-like system, that provides modelling hints to the developer in real-time, like many Integrated Development Environments have today. Both of these specifications would need some careful consideration of what exactly a model-developer needs and wants to know during development and how to provide it most intuitively.

C. Additional HAWK Sugar Extensions

Currently, the HAWK syntax only allows instantiation of TTA templates in a parallel composition manner, but it would make it easier to segment models into specialized pieces of code if the language also allowed for sequential composition where the parent TTA "enters" a child TTA.

D. Compiling and Loading Binary Encodings of NTTA

One may consider the AALTITOAD HAWK parser module as a type of compiler. It translates a JSON formatted list of files into a binary datastructure in internal memory that can then be executed. This parsing is fairly fast today, but it could be improved even further if the datastructure could be dumped and loaded directly from/to the disk. This would cause some issues if the binary version was loaded on a different CPU architecture, but with a quick checksumming scheme, this is not an unsolvable problem.

REFERENCES

- [1] Asger Gitz-Johansen. Tick tick automata: A modelling formalism for real world industrial systems. 2020.
- [2] Asger Gitz-Johansen. Aaltitoad a tick tick automata verification engine. 2021.
- [3] Peter B. Greve Peter H. Taankvist Alexander Bilgram, Emil Ernsten and Thomas Pedersen. Hmkaal: Formal verification and detection of non-determinism and data races in industrial systems. 2020.
- [4] Martin P. Hansen Frederik M. W. Hyldgaard, Gustav S. Bruhns and Rasmus Hebsgaard. Haalt - hawk and aaltitoad automatic likeness testing. 2022.
- [5] Joost-Pieter Katoen Christel Baier. *Principles of Model Checking*. The MIT Press, Cambridge, Massachusetts, London, England, 2008.
- [6] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.
- [7] The z3 theorem prover. URL <https://github.com/Z3Prover/z3>.
- [8] Z3 prover development team publications list. URL <https://github.com/Z3Prover/z3/wiki/Publications>.
- [9] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi: 10.1137/0201010. URL <https://doi.org/10.1137/0201010>.
- [10] Rasmus Holm Jensen Niklas Kirk Mouritzen and Ulrik Nyman. H-uppaal. 2017.
- [11] H-uppaal github repository. URL <https://github.com/DEIS-Tools/H-Uppaal>.
- [12] Debug adapter protocol. URL <https://microsoft.github.io/debug-adapter-protocol>.