Summary

In collaboration with the tanker truck production company HMK Bilcon, we set out to continue the work on the enhancement of the model-based development (MBD) pipeline for the SafeCon III system with verification and automatic test generation initially proposed in [1, 2]. SafeCon III is a safety-critical system that monitors and controls the pump system of tanker trucks, emphasizing the need for the system to be free of serious defects. In production, the SafeCon III system is compiled and executed in a runtime environment called HAWK (Hardware Abstraction With Knowledge) developed by HMK Bilcon. Previously, to facilitate formal verification, the formalism of Tick Tock Automata (TTA) was defined to formalize the runtime environment of HAWK [2], and a model checker called AALTITOAD was created based on the described semantics [1]. Therefore, this paper focuses on automatic test generation for the SafeCon III system based on the semantics of TTA.

TTAs differ from traditional Timed Automata, as variables are globally accessible, and there are variables, called external variables, that are updated non-deterministically. In the semantics, the external variables were modelled to update non-deterministically, since the values of external variables in the SafeCon III system are updated by reading input from sensors on the trucks. Because of non-determinism, external variables greatly increase the state space, which makes the state space of the entire SafeCon III system too large to be model checked by AALTITOAD at once. As such, it is necessary to implement a way to test on a subset of the model. To this end, a formalism for translating a set of model components to a Network of Tick Tock Automata is described.

For automatic testing, the program HAALT is implemented. HAALT is an automatic test generator, which verifies whether the industrial runtime environment of HAWK is trace equivalent with the Tick Tock Automata verification engine AALTITOAD. AALTITOAD was chosen for the automatic test generation, as it follows the semantics defined for TTAs. The alternative is to translate the TTA to a common formalism such as Timed Automata and then use a different model checker. This option seems more complex, as it would therefore require a way to correctly translate concepts such as non-deterministic changes to variables and clocks in a TTA [1].

Overall, the process in HAALT is based on a method introduced in [3], which describes how to automatically generate a test suite for some coverage criterion. In HAALT, based on some coverage criterion, a TTA model is annotated, and a set of reachability queries are generated based on said annotation. This annotated model is then given as input to AALTITOAD, which produces an execution trace for each satisfiable query. Based on the trace, a set of TTA-transitions are generated such that the trace can be simulated in HAWK, which produces its own trace. Finally, by simple comparison, it is verified whether AALTITOAD and HAWK are trace equivalent for the given coverage criterion. To formalize annotation of TTA models, we introduce a formalism for converting a Tick Tock Automata (TTA) to an Annotatable Tick Tock Automata(ATTA). This conversion does not alter the original execution semantics, as the ATTA can simulate the execution semantics of the original TTA. The ATTA makes it possible to perform annotation based on logic coverage criteria, and to this end, we define and implement semantics for annotating ATTAs based on location, edge, and MC/DC coverage criteria.

To evaluate HAALT, testing was performed on a network of Fischer TTA models. It was found that the annotation methods greatly expanded the state space, limiting the practical use of HAALT. This was evident from testing on individual components of SafeCon III, as queries did not terminate for MC/DC coverage. It was, however, possible to verify test equivalence between HAWK and AALTITOAD based on location and edge coverage for simple components. When testing on

multiple components, it was immediately discovered that there are inherent differences in the way AALTITOAD and HAWK handle the notion of Network of Tick Tock Automata.

HAALT - HAWK and AALTITOAD Automatic Likeness Testing

Frederik M. W. Hyldgaard, Gustav S. Bruhns, Martin P. Hansen, and Rasmus Hebsgaard

Aalborg University, Institute of Computer Science

Abstract. In collaboration with the tanker truck production company HMK Bilcon, we set out to continue the work on enhancing the model-based development (MBD) pipeline for the SafeCon III system. As proposed by [1, 2], the pipeline can be extended to include automatic test generation and verification. This paper specifically focuses on automatic test generation. We introduce a formalism for converting a Tick Tock Automata (TTA) to an Annotatable Tick Tock Automata (ATTA), which can simulate the execution semantics of the original TTA. The ATTA makes it possible to perform annotation based on logic coverage criteria. We define semantics for annotating ATTAs based on coverage criteria (location, edge, and MC/DC coverage), which was implemented in a tool called HAALT. HAALT is an automatic test generator, which verifies whether HMK Bilcon's proprietary industrial runtime environment HAWK is trace equivalent with the Tick Tock Automata verification engine AALTITOAD. From testing, it is shown HAALT can perform verification on small components of SafeCon III. However, it was also found that the annotation methods greatly expanded the state space, limiting the practical use of HAALT.

Keywords: Automatic test generation · Tick Tock Automata · Annotatable Tick Tock Automata

1 Introduction

Model based development (MBD) is slowly gaining traction within the software industry. The development cycle involves automatically generating code based on models, which are expressed in more user-friendly formalisms than code. The simplicity of using MBD for development allows domain experts to be the ones developing the application. As described in [2], in industries with safety critical systems, errors may have serious consequences, stressing the importance of testing. To this end, model based development offers more fluid integration of software testing such as automatic testing[1], since model checking performed on the model also applies to the generated code. This is, of course, under the assumption that the code generator is semantically correct. In this paper, we will work with the use case of HMK Bilcon.

2 HMK Bilcon

HMK Bilcon A/S is a Danish tanker truck production company manufacturing custom tanker trucks. Each tank is produced as a custom order and most are equipped with the SafeCon line of systems. The system controls the piping and pump system of the trucks including loading, unloading, and transferring of liquids between tank compartments. This paper focuses on the latest version of the SafeCon system, SafeCon III. As the trucks are designed to handle oil and petrol which are highly flammable liquids, it is critical that the system always handles the liquids in a safe manner.

2.1 Current HMK Development Process

HMK Bilcon utilizes Model Based Development (MBD) to develop SafeCon III. The process begins with a specification based on customer-requested features, after which domain experts create a model in the modeling language of Tick Tock Automata (TTA). As shown on Figure 2, this model is compiled down to an executable state machine, which is run in the proprietary runtime environment HAWK (Hardware Abstraction With Knowledge) developed by HMK Bilcon. The new features are then tested manually on the truck before they are marked ready for release.

2.2 Automatic Testing

SafeCon III is a safety-critical system, emphasizing the need for the system to be free of serious defects. Because SafeCon III is developed using a MBD pipeline, it is fit to benefit from formal verification, as analyzed in [1]. In the paper, it was proposed to expand the SafeCon III MBD pipeline with formal model checking and automatic testing. For model checking, a verification engine called AALTITOAD was created based on the semantics of Tick Tock Automata (TTA) [1]. In this paper, we intend to define a method to incorporate automatic testing into the pipeline based on AALTITOAD. The new proposed pipeline is shown in Figure 1, where our proposal of *automatic testing* is marked in green.

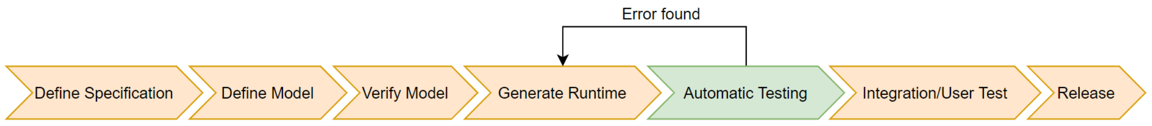

Fig. 1. Incorporating automatic testing in a new proposed pipeline.

The purpose of automatic testing is to verify whether HAWK generates a correct state machine based on the model. Therefore, if any error is found during automatic testing, it points to a fault in the runtime generation. Currently, developers from HMK Bilcon must manually test functionality by going to the truck and checking whether the functionality works as desired. Automatic testing allows for a more thorough evaluation of whether the generated state machine has similar behaviour to the model, since it allows for testing of far more unique executions of the model than manual testing.

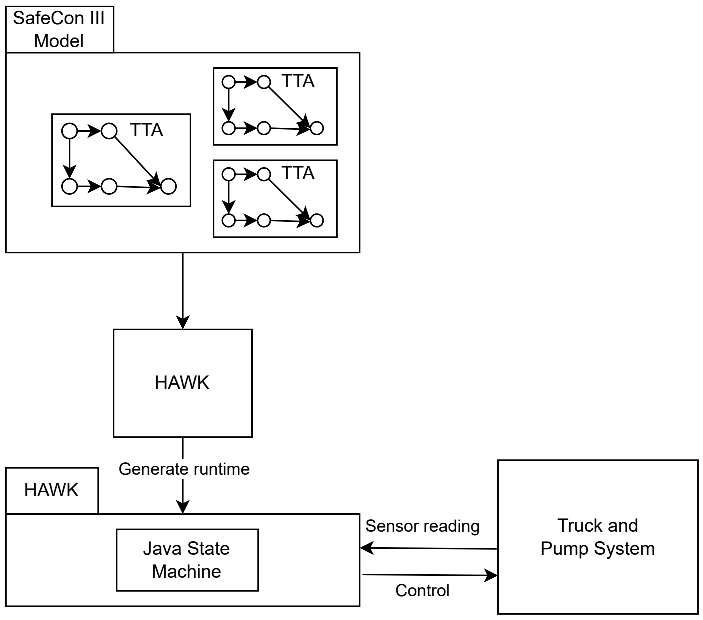

Fig. 2. Using the compiler in HAWK, a SafeCon III model composed of many TTAs is translated into a java state machine. This state machine executes in the runtime environment of HAWK on the truck and continuously interacts with the truck and pump system.

The rest of this paper is structured as follows: Existing tools for automatic testing are explored, and a new tool capable of comparing HAWK to AALTITOAD is proposed. In sections 5 through 8, the formalism of Tick Tock Automata is extended to allow for additional annotation of the models. In sections 9 to 13, we discuss how the Annotatable Tick Tock Automata can be annotated to create a set of queries, which when given as input to AALTITOAD, becomes a set of traces covering a given coverage criteria. Lastly, the tool is evaluated and future work is discussed.

3 Related Work

The formalism of Tick Tock Automata, as defined in [2], is a relatively new research topic. In [1], Gitz-Johansen introduces the verification engine AALTITOAD for TTAs and discusses the problems that occur due to TTAs being different from Timed Automata. Pedersen et al. [4] introduce a random search model checking method for TTAs that allows for deeper searches in the state space, but does not guarantee a full exploration of the state space. Lastly, HAWK has an integrated tool which translates TTA models to *NuSMV* models [5] and utilizes a corresponding verification engine. However, it is not guaranteed that the *NuSMV* model is trace equivalent with the HAWK runtime.

Automatic and model based test generation has been explored for other formal languages. Enoiu et al. [3] propose a method, where a test suite can be generated structurally using a model checker. Function Block Diagram (FBD) programs are translated into UPPAAL[6] timed automata, and reachability properties are generated such that they fulfill a set of logic coverage criteria. Similar approaches using UPPAAL and reachability properties to generate traces that can be transformed into test cases have been proposed in [7, 8, 9]. Furthermore, tools for automatic test generation have been defined and implemented for various languages. In [10] Krenn et al. introduces a toolbox for UML models and Timed Automata that uses mutation testing to generate a test suite capable of catching specific errors. In [11] a toolbox for automatic test generation is defined for Stream X-Machines (SXM), which is an extension of finite state machines. In [12] Angeletti et al. introduces a tool, which automatically generates a test suite with full decision coverage for C programs.

In this paper, we introduce a tool, HAALT, which can verify whether a runtime can satisfy the same reachability properties as a verification engine, using the traces generated by the verification engine. To the extent of our knowledge this approach has not been used before, and certainly not for TTAs. Other forms of runtime verification have, however, been proposed for other languages such as Java [13, 14].

4 HAALT: HAWK AALTITOAD Automatic Likeness Testing

In [3], a method is outlined to automatically generate test suites for industrial systems. The method begins with choosing a set of coverage criteria to base the testing on. Based on the coverage criteria, Timed Automata models are annotated such that reachability queries can be generated, which are given to a model checker. The model checker generates a set of test traces that constitute the test suite (examples in [7, 8]). This method is highlighted, as it is relatable to the setting of HMK Bilcon, where the models would be the SafeCon III TTA models, and the model checker would be AALTITOAD. In terms of model checking, AALTITOAD can perform reachability searches on TTAs with queries of the form $\exists \diamond \varphi$ expressed as Computation Tree Logic (CTL) [15]. This CTL formula is only satisfied given there exists a path in the computation tree with a state on the path satisfying the predicate φ.

In order to apply the described method, the formalism for annotating Tick Tock Automata would need to be defined for specific coverage criteria. The alternative would be to translate Tick Tock Automata to a common formalism such as Timed Automata and use another model checker [6]. However, this option seems more complex, as it requires certainty that the translation preserves the semantics of the TTA. This would require a way to correctly translate concepts such as non-deterministic changes to variables and clocks [1].

Based on the described method, we propose a process called HAALT presented in Figure 3. In HAALT, TTAs are annotated in order to generate a test suite that fulfills a given coverage criterion. Through discussions with HMK Bilcon, we discovered that the most valuable coverage criteria for HMK Bilcon would be logic-based coverage criteria, specifically *modified condition/decision coverage* (MC/DC) [3]. They were, however, clear that they are indifferent to whether their models conform to MC/DC, and therefore, it should only be used as a method to annotate TTAs in a structured manner.

The test suite is generated by giving the annotated model and reachability queries as input to AALTITOAD, which generates an execution trace for each satisfiable query. The test suite then consists of the annotated TTA model and a test case for each trace; a trace represents the expected behavior of the model. To test for equivalence between HAWK and AALTITOAD, each AALTITOAD trace is parsed to generate a list of TTA-transitions, which can be simulated in

HAWK. By simulating these transitions, a trace is generated from the HAWK runtime, which is compared with the AALTITOAD trace. If the traces are equivalent, it is said that HAWK is consistent with the semantics of TTAs. By consistent, we mean that if HAWK performs the same transitions as AALTITOAD, it will reach the same states. If the traces deviate, it is assumed that there is an error in the implementation of HAWK. Of course, this only holds under the assumption that we have a working verification engine. As AALTITOAD may contain faults of its own, false negatives may occur, and it is ultimately up to the software engineer to determine whether the error stems from HAWK. HAALT is not able to find some errors, such as if HAWK generates outgoing edges on locations that do not have outgoing edges in the TTA. The question which HAALT answers is thus:

Can HAWK at least perform the same transitions as AALTITOAD with equivalent states between HAWK and AALTITOAD before and after each transition?

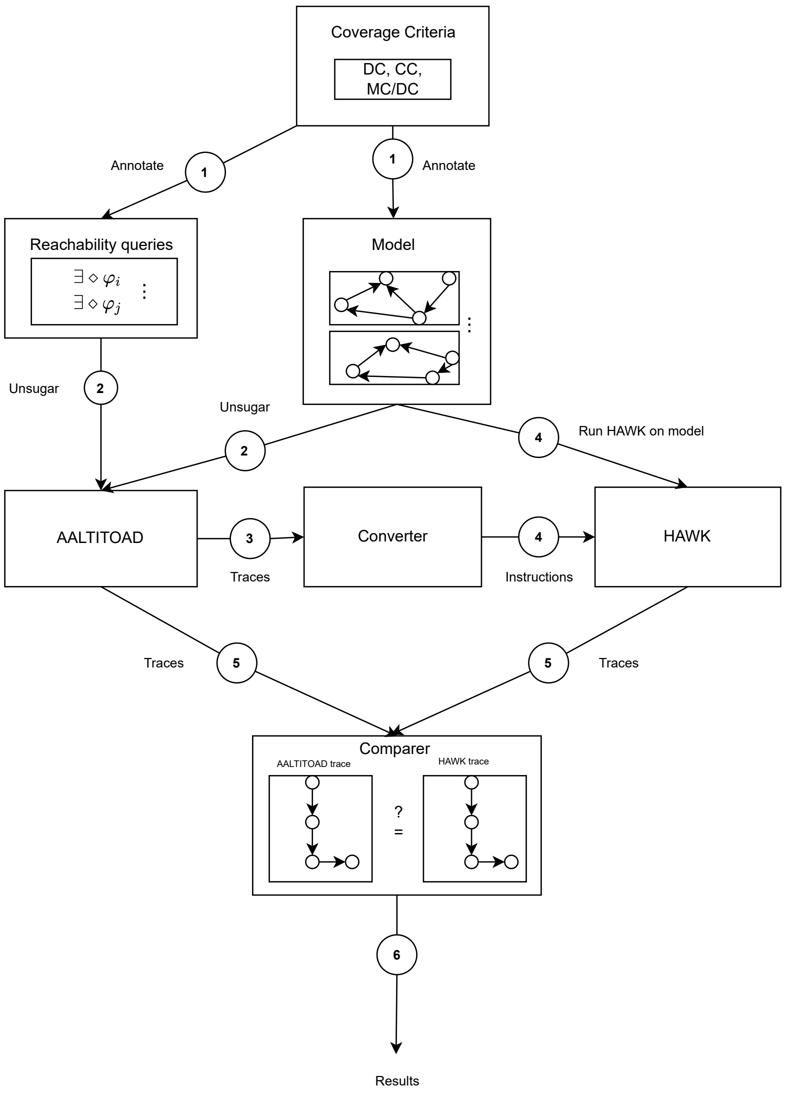

Fig. 3. Pipeline of HAALT.

5 Tick Tock Automata

To facilitate automatic testing and model verification of SafeCon III models, semantics for a new type of automata called Tick Tock Automata (TTA) were defined in [2], formalizing the state machines generated by the compiler in HAWK. The semantics are based on Timed Automata, but introduce several new notions such as tick-tock cycles and external variables. The relevant semantics for this paper will be described in the following sections; for all details of the semantics, see [1, 2].

Intuitively, the semantics describe the runtime environment of HAWK operating on a tanker truck. As the truck is operating, the system is constantly performing internal computations and regulating systems based on said computations. This is modelled as tick-transitions in TTA semantics, where a tick represents a single computation, as illustrated in Figure 4. In addition to performing computations, the truck also receives input data from sensors such as tank pressure, speed, etc. In TTA semantics, each type of input is modelled as an external variable, where the reading of input is modelled as a tock-transition. A formal tock-transition consists of non-deterministically updating the value of each external variable based on the domain of the variable and adding a random delay to all clocks, modelling the time it takes to read all sensors.

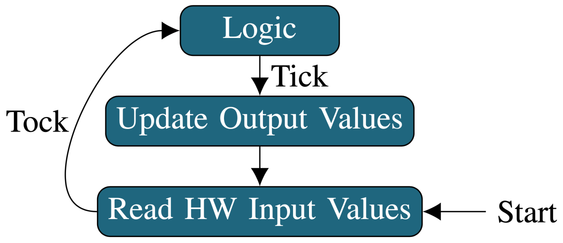

Fig. 4. Intuition behind tick-tock transitions. (Figure taken from [2])

For preliminaries, the following sets and functions are defined:

- $\mathbb{B} = \{tt, ff\}$
- $\mathbb{R} =$ reals
- $\mathbb{V} = \mathbb{R} \cup \mathbb{B}$
- A clock valuation $c : C \rightarrow \mathbb{R}_{0\leq}$
- A variable valuation $v : V \cup \Omega \rightarrow \mathbb{V}$

Then, Λ is defined as the set of variable valuations s.t. $v \in \Lambda$, and $\mathbb{R}^C$ the set of clock valuations s.t. $c \in \mathbb{R}^C$.
Formally, a TTA is defined as a $9-$tuple $(L, C, V, \Omega, E, l_0, v_0, c_0, \tau)$ where:

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

– $\tau : V \cup \Omega \rightarrow 2^{\mathbb{V}}$ is a type function defining the domain of variables.

TTAs transition around the state space following a tick-tock cycle. During a tick transition, for each outgoing edge $e = \langle l, g, u, r, l' \rangle$, the TTA non-deterministically performs a transition $(l, v, c) \rightarrow (l', v', c')$ for which the following holds:

$$
\begin{aligned}
&l \xrightarrow{g,u,r} l' \text{ such that}\\
&g(v,c) = tt \text{ and}\\
&u(v) = v' \text{ and}\\
&r(c) = c'
\end{aligned}
$$

where g is the guard function of an edge, u the update function, and r the clock reset function. In the case of a network of TTAs, the tick is considered to be global, and each TTA in the network performs a single tick transition simultaneously.

During a tock, all values for external variables are updated non-deterministically and some real valued time delay is added to all clocks in the system.

6 Translating from Components to TTAs

Model checking is often prone to state space explosion, and because TTAs have external variables that are non-deterministic by design state space explosion happens very quickly. While AALTITOAD applies some techniques to reduce the state space, it is still practically impossible to verify queries for the entire SafeCon III model at once [1]. Therefore, it is necessary to find a way by which smaller parts of the system can be verified. SafeCon III is based on a component structure, wherein a model is composed of a set of components that themselves are similar to TTAs. The components have some characteristics that differentiate them from TTAs. Most importantly, they are able to refer to variables and clocks that are not defined in the component itself. The variables and clocks may instead be defined in

– another component.
– specific files called *.parts* files.

.parts files are accessible from every component in the model and contain the definitions of all clocks and external variables, as well as some internal variables. In order to translate from components to testable TTAs, we define a formalism to describe the components of SafeCon III based on their locations, edges, declarations of variables and clocks, and usage of variables and clocks. There is a one-to-one correspondence between the locations, initial location, and edges of the components and the resulting TTAs. Therefore, the main task lies in identifying how variables from the components should be represented in the TTA.

Let a component be defined as $COMP = (L, E, D_v, D_c, U_v, U_c, l_0, v_0, c_0, \tau)$

– L is a set of locations.
– E is a set of edges of the form $E = L \times G \times U \times 2^C \times L$
– D_v and D_c are the sets of variables and clocks declared in the component.
– U_v and U_c are the sets of variables and clocks that are referred to in the guards and updates of the component.
– $l_0 \in L$ is the initial location.
– $v_0 \in \Lambda$ is the initial variable valuations of D_v.
– $c_0 \in \mathbb{R}^C$ is the initial clock valuations D_c.
– $\tau : U_v \rightarrow 2^{\mathbb{V}}$ where $\mathbb{V} = \mathbb{R} \cup \mathbb{B}$, is a type function that defines the domain of the variables in U_v.

The full model M can thus be seen as the set of all *COMPs* and a special

$$COMP_{parts} = (\emptyset, \emptyset, D_v^{parts}, D_c^{parts}, \emptyset, \emptyset, \epsilon, v_0^{parts}, c_0^{parts}, \tau^{parts})$$

that contains all declarations of the *.parts* files.

A TTA has two sets of variables, internal and external, which must be differentiated between during the translation. Assume that we have a set $T \subseteq M$ such that T is the set of components that should be tested. The usages of variables in a component can then be categorised into four categories as shown in Figure 5. A component can

1. read from variables declared in T
2. write to variables declared in T
3. read from variables not declared in T
4. write to variables not declared in T

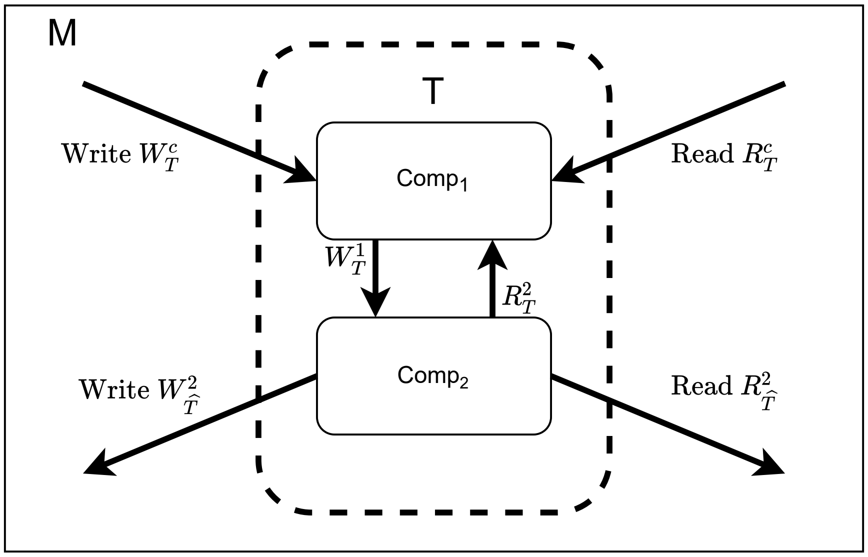

Fig. 5. An illustration of the four categories of variables. Arrows point from the usage of a variable to where it is declared.

To argue about which categories a variable lies within, U_v is decomposed into two sets such that $U_v = W_v \cup R_v$, where W_v and R_v contain all variables that are respectively written to and read from inside a given component. The categories are formally defined by

1. $R_T = \{v \in R_v \mid \exists c \in T \ s.t. \ v \in D_v^c\}$ and $W_T = \{v \in W_v \mid \exists c \in T \ s.t. \ v \in D_v^c\}$
2. $R_{\widehat{T}} = \{v \in R_v \mid \exists c \in M \setminus T \ s.t. \ v \in D_v^c\}$ and $W_{\widehat{T}} = \{v \in W_v \mid \exists c \in M \setminus T \ s.t. \ v \in D_v^c\}$

where $v \in D_v^c$ indicates that v lies in the declaration set D_v of component c.

A variable should be external in the TTA when a $COMP \notin T$ writes to a variable defined in T, or when a $COMP \in T$ reads from or writes to a variable not defined in T.

$$\Omega_T = \left(\bigcup_{c \in M \setminus T} W_T^c \right) \cup \left(\bigcup_{c \in T} R_{\widehat{T}}^c \cup W_{\widehat{T}}^c \right) \tag{1}$$

where W^c_T, $R^c_{\hat{T}}$ and $W^c_{\hat{T}}$ are the read and write sets of component c. All other variables only used in T can be internal, since it is only components in T that affect these.

$$V_T = \left(\bigcup_{c \in T} U^c_v \right) \setminus \Omega_T \tag{2}$$

A network of TTAs $\{A_1, ..., A_{|T|}\}$ can then be defined by defining the TTA A_i for all components $COMP_i = (L_i, l^i_0, E_i, D^i_v, D^i_c, v^i_0, c^i_0, U^i_v, U^i_c, \tau)$ in T such that

$$A_i = (L_i, l^i_0, C, V_T, \Omega_T, E, v_0, c_0, \tau) \tag{3}$$

Clocks and variables are shared across the network and are not dependent on i. Therefore, v_0 is the union of all v^i_0 of components in M and c_0 is the union of all c^i_0 of components in M. Furthermore C is the union of all U_c of components in T. Hence, it is shown that it is possible to translate a set of components from SafeCon III into a network of TTAs, where the internal and external variables in the network encode how the variables are used in relation to the entire model of SafeCon III.

7 Annotatable Tick Tock Automata

In HAALT, models are annotated as part of the process to specify reachability queries based on a set of coverage criteria. To describe annotation in the context of TTAs, we introduce the notion of Annotatable Tick Tock Automata (ATTA).

For ATTAs, a new set of auxiliary variables A is introduced such that the set of all variables becomes $Z = \Omega \cup V \cup A$. Then, to create a test suite for some coverage criterion, it must be possible to evaluate guards of edges and make updates to auxiliary variables based on said evaluations. However, since an update only occurs when the guard of the same edge evaluates to true, the updates to the auxiliary variables will not be performed when the guard evaluates to false. As such, updates to auxiliary variables cannot be placed on the same edge as the guard itself. Our solution is to create an ATTA, which is a copy of the original TTA, but with additional edges on which the guards of the following edges can be evaluated. The solution is illustrated in Figure 6. Each location is split into two parts; a left and right part. By splitting a location into two parts, it is now possible to place the updates to auxiliary variables before the original edges by adding new edges.

7.1 Transformation to ATTA

A TTA can be transformed into an ATTA in three steps.

1. For all locations l split it into two locations, l^l and l^r, corresponding to a left and right part.
2. Map all edges e that previously went from l_i to l_k such that they now go from l^r_i to l^l_k
3. Create a new edge e from l^l to l^r

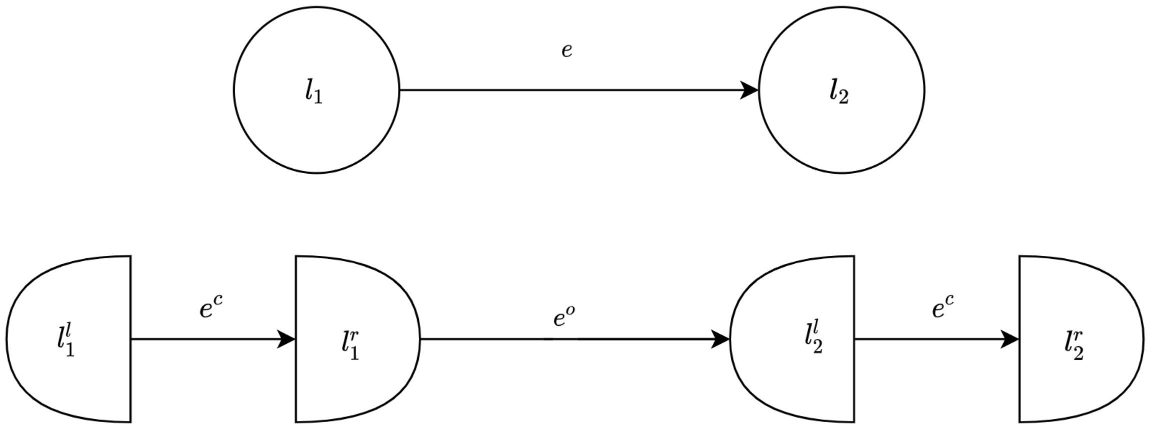

Fig. 6. A TTA and its corresponding ATTA. The TTA consists of two locations with a single edge. In the ATTA, the locations are split into two new locations, denoted l^l and l^r. The two parts of the location is then connected by an edge. The original edge going from l_1 to l_2 in the TTA, goes from l_1^r to l_2^l in the ATTA and is denoted e^o. (The locations in the ATTA have been drawn as half circles for illustration, but they act exactly as a normal location.)

Given a TTA $= (L, C, V, \Omega, E, l_0, v_0, c_0, \tau)$ define a new ATTA $= (L', C, Z, E', l_0', v_0', c_0, \tau)$. All auxiliary variables have an initial valuation of ff.

$$v_0'(x) = \begin{cases} v_0(x) & \text{if } x \in V \cup \Omega \\ ff & \text{if } x \in A \end{cases} \tag{4}$$

In Equation 5, the left and right locations are created and combined into L'.

$$L^l = \bigcup_{i=1}^{|L|} \{l_i^l\} \qquad L^r = \bigcup_{i=1}^{|L|} \{l_i^r\} \qquad L' = L^l \cup L^r \tag{5}$$

The definition of E' is split into three subsets E^o, E^{con}, and E^{aux} such that their union forms the set E'. The set E^o represents the original edges going between locations, which in the ATTA is between l_i^r and l_k^l. The set E^{con} represents the edges connecting a location l_i^l to the corresponding location l_i^r. These edges contain neither guards nor clock resets as they are included to ensure that a tick transition is always possible when in a given location l^l. This is further elaborated in section 8. E^{aux} is the only set of edges where we allow updates to auxiliary variables. Therefore, the set will be used during annotation to make updates to auxiliary variables.

$$E' = E^o \cup E^{con} \cup E^{aux} \tag{6}$$

To create E^o we redefine the original edges by moving the start and end destination of the edge to the corresponding locations after the split. This means that if an edge previously went from l_4 to l_5, it will now go from l_4^r to l_5^l.

$$E^o = \{e^o | \text{ if } e = \langle l_i, g, u, r, l_k \rangle \in E \text{ then } e^o = \langle l_i^r, g, u, r, l_k^l \rangle\} \tag{7}$$

To create E^{con}, we add an edge from the left part of a location to the right part of a location. As mentioned, this edge contains neither a guard nor clock updates.

$$E^{con} = \bigcup_{i=1}^{|L|} \{e_i\} \text{ where } e_i = \langle l_i^l, \epsilon, \epsilon, \emptyset, l_i^r \rangle \tag{8}$$

The set E^{aux} depends on the coverage criterion that is being annotated for and is therefore empty until annotation has happened.

$$E^{aux} = \emptyset \tag{9}$$

As mentioned, auxiliary variables are created as part of the annotation process. Specifically, auxiliary variables are used to track whether certain configurations of internal and external variables are possible. Due to how AALTITOAD explores the state space, external variables can only change when evaluating a guard of an edge. Therefore, to be able to evaluate whether a specific variable configuration is possible, a new auxiliary edge must be created with a guard containing the specific variable configuration as seen in Figure 9. Then, if it is possible to take the edge, it can be concluded that the specific variable configuration is possible, and the corresponding auxiliary variable can be updated.

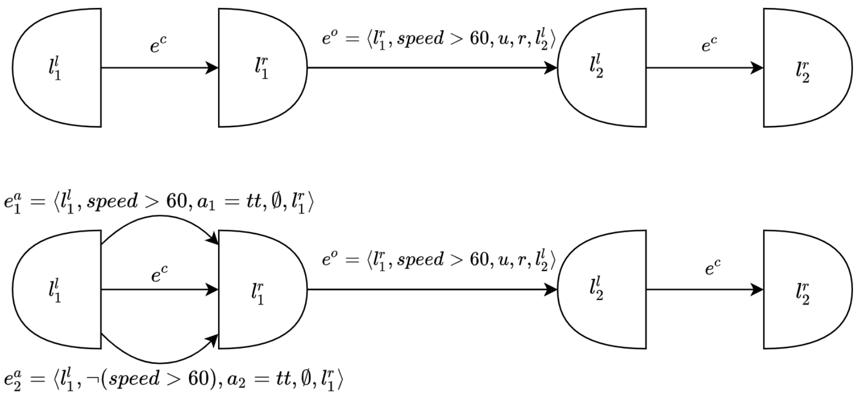
e^c
$e^o = \langle l_1^r, speed > 60, u, r, l_2^l \rangle$
l_1^l
l_1^r
l_2^l
l_2^r
$e_1^a = \langle l_1^l, speed > 60, a_1 = tt, \emptyset, l_1^r \rangle$
$e_2^a = \langle l_1^l, \neg(speed > 60), a_2 = tt, \emptyset, l_1^r \rangle$

Fig. 7. An example of how MC/DC annotation would be performed by adding edges to E^{aux} based on the guard of the original edge e^o. See Section 9.1 for details of how the guards of the auxiliary edges are formed.

Having added these locations and edges, we have created an ATTA. It can then be argued that the ATTA can simulate its corresponding TTA under a set of constraints.

7.2 Constraints for a valid ATTA

To ensure that the transformation of a TTA to an ATTA does not alter the original execution semantics, the following constraints must hold for any ATTA to be valid.
Edges in E^{aux} must only contain updates to auxiliary variables.

$$\forall e = \langle l, g, u, r, l' \rangle \in E^{aux} : \forall z \in Z : v(z) \neq v'(z) \implies z \in A \tag{10}$$

Edges in E^o must not contain updates to auxiliary variables

$$\forall e = \langle l, g, u, r, l' \rangle \in E^o : \forall z \in Z : v(z) \neq v'(z) \implies z \in Z \setminus A \tag{11}$$

$$\text{where } u(v) = v'$$

Connection edges may not have have guards nor updates.

$$\forall e = \langle l, g, u, r, l' \rangle \in E' \ : e \in E^{con} \implies g = \epsilon \wedge u = \epsilon \tag{12}$$

Auxiliary edges must only go between l^l and l^r of the same original location.

$$\forall e = \langle l_i, g, u, r, l'_j \rangle \in E' \ : e \in E^{aux} \cup E^{con} \implies l_i \in L^l \wedge l'_j \in L^r \wedge i = j \tag{13}$$

Auxiliary and connecting edges must not contain any clock resets.

$$\forall e = \langle l, g, u, r, l' \rangle \in E' \ : e \in E^{aux} \cup E^{con} \implies r = \emptyset \tag{14}$$

Given an ATTA follows these constraints, it can be argued that it can simulate a corresponding TTA, as seen in Section 8.

8 Simulation of TTAs

In this section we argue that a given ATTA can simulate its corresponding TTA maintaining the execution semantics of the original TTA. Based on the constraints put forth in Section 7.2, it follows that each tick transition in a given TTA can be simulated in the corresponding ATTA by performing two tick transitions. Intuitively, the first tick in the ATTA is necessary to take either an edge e^a or an edge e^c in the ATTA, as these edges do not appear in the original TTA. The next tick is needed to match the tick in the original TTA. This is formally defined in equation 15 for a TTA and its corresponding ATTA.

$$\text{TTA} : (l_i, v_t, c) \rightarrow_{tick} (l_j, v'_t, c') \iff \text{ATTA} : (l^l_i, v_a, c) \rightarrow_{tick} (l^r_i, v'_a, c) \rightarrow_{tick} (l^l_j, v''_a, c') \tag{15}$$

$$\text{where } \forall x \in \Omega \cup V : v_t(x) = v_a(x) \wedge v_t(x) = v'_a(x) \wedge v'_t(x) = v''_a(x)$$

By construction, the first tick in the ATTA is always possible, as at least one of the edges between locations l^l_i and l^r_i can be taken, and the constraint presented in Equation 10 will always hold. Hence, $\forall x \in \Omega \cup V : v_t(x) = v'_a(x)$. Without loss of generality, it is assumed the first tick of the ATTA is performed immediately, adding no time to the clocks. Furthermore, due to the constraint in Equation 13 the edge taken during the tick has no associated clock reset. The clock valuation is therefore constant before and after the tick.

As for the second tick, note that the ATTA is constructed such that for all $l_i \in L$ there is a corresponding pair l^l_i and l^r_i. By construction, the outgoing edges from l^r_i are the same as those going out from l_i in the original TTA. Since no variables or clocks are changed from the original TTA after the first tick in the ATTA, the set of possible edges to take are the same as those before the tick in the original TTA. Hence, after the second tick, $\forall x \in \Omega \cup V : v'_t(x) = v''_a(x)$ and the clock valuation becomes c'. Thereby, an ATTA can simulate a tick in a corresponding TTA by performing two ticks.

Due to the introduction of auxiliary variables that depend on external variables, we have introduced a need for tocks to occur during the double tick of the ATTA, if we want to take any auxiliary edge. An ATTA may therefore perform the actions $tick \rightarrow tock \rightarrow tick \rightarrow tock$. The ATTA is, however, still able to simulate any TTA with any Tick-Tock cycle, since all left and right parts of locations are connected by the empty edges in E^{con}. Hence, it is always possible to perform the same amount of double ticks in the ATTA, as the amount of ticks in the Tick-Tock cycle of the original TTA. If the original TTA has a Tick-Tock cycle with multiple ticks for every tock, the addition of additional tocks is still not a problem. This is due to the fact that there will always be a possible branch that allows for the same execution as the original TTA, since the external variables are chosen without constraints.

9 Test coverage criteria

In the software industry, coverage criteria are used to assess the thoroughness of test cases used in any testing of code. Coverage criteria are also used in the setting of automatic unit testing of models. Various types of test coverage exist, such as model-based coverage criteria [16] and logic-based coverage criteria. As mentioned in Section 4, the logic based criteria MC/DC coverage is the most valuable coverage criteria for HMK Bilcon.

9.1 MC/DC for TTAs

To understand Modified condition/decision coverage in the context of a TTA, we must first define what is meant by *decision* and *condition*. Decisions in TTAs are the points, where it is determined whether an edge is taken or not. Thus, if there are multiple possible edges to take from a location in a given TTA, then there are multiple decisions. Whether it is possible to take an edge depends on the guard of the edge. Each guard consists of one or more conditions combined through boolean operators (and, or). Therefore, a single decision is covered if it is possible to have the guard of the given edge evaluated to true in one trace and false in another. Let $\{d_i\}$ be the set of decisions and $\{c_{ij}\}$ be the set of conditions in decision d_i.

To fulfill decision coverage, each decision must be able to evaluate to both true and false.

$$\varphi_1 : \exists \diamond d_i \quad and \quad \varphi_2 : \exists \diamond \neg d_i \tag{16}$$

To fulfill condition coverage, each condition c_{ij} in decision d_i must be able to evaluate to both true and false

$$\varphi_1 : \exists \diamond c_{ij} \quad and \quad \varphi_2 : \exists \diamond \neg c_{ij} \tag{17}$$

For a test suite to fulfill MC/DC, it must be checked that all conditions c_{ij} in d_i determine the evaluation of d_i. c_{ij} determines d_i if there exists a configuration of values of the variables in the conditions of d_i except c_{ij} such that the evaluation of d_i is different for the two different evaluations of c_{ij}. Furthermore, it must be verified that it is possible for c_{ij} to evaluate to both true and false. Whether c_{ij} determines d_i can be checked using the following expression, where $\oplus$ is the binary operation XOR.

$$d_i[c_{ij} \mapsto tt] \oplus d_i[c_{ij} \mapsto ff] \tag{18}$$

Thus the check of MC/DC requires two tests for every condition. These two queries together constitute an MC/DC pair.

$$\varphi_1 : \exists \diamond c_{ij} \wedge (d_i[c_{ij} \mapsto tt] \oplus d_i[c_{ij} \mapsto ff]) \tag{19}$$

$$\varphi_2 : \exists \diamond \neg c_{ij} \wedge (d_i[c_{ij} \mapsto tt] \oplus d_i[c_{ij} \mapsto ff]) \tag{20}$$

9.2 Model-based coverage criteria

Other than logic-based coverage criteria, annotation for the model-based coverage criteria location coverage and edge coverage are implemented as well.
As defined in [8], a test suite satisfies the edge coverage criterion, when it is possible to take each edge in the model. The location coverage criterion is satisfied by a test suite when each location in a model is reached during execution of the tests [8]. Including these criteria allows us to compare the runtime and test suite size between the different criteria.

9.3 Test suite size

Since a test suite consists of a set of query traces, the size of the test suite is at most the number of queries generated for the model being tested. In many instances, however, the actual size will be smaller, since AALTITOAD only generates traces for satisfiable queries.

For location coverage, one query is generated for each location in the model, and for edge coverage, one query for each edge. Lastly, the amount of queries for MC/DC coverage depends on the number of MC/DC pairs in the model and, thereby, the number of edges with guards, the size of the guards, and how the guards are constructed. For example, for each variable in the guard of the form $a \vee b \vee c \vee d$, there is an MC/DC pair, so it is linear in the number of queries it generates. Considering a guard of the form $(a \wedge b) \vee (c \wedge d)$, the number of MC/DC pairs more than triples for every two variables added. As a result, the test suite for MC/DC is often bigger than both location and edge coverage with the potential of having exponentially many queries based on the number of conditions in the guard.

10 Location Annotation

To generate queries for the location coverage criterion, it is necessary to create a set of auxiliary variables that keep track of whether a location has been visited. In an ATTA, this can be done by annotating the auxiliary edges.

$$A = \bigcup_{i=1}^{|E^{con}|} \{a_i\} \tag{21}$$

Each auxiliary edge is then responsible for updating its own auxiliary variable such that

$$E^{aux} = \left\{ e_i^a = \langle l^l, \epsilon, u, \emptyset, l^r \rangle \mid e_i^c = \langle l^l, \epsilon, \epsilon, \emptyset, l^r \rangle \in E^{con} \right\} \tag{22}$$

$$\text{where } u(v) = v\Big[a_i \mapsto tt\Big]$$

It is then possible to verify whether a location l_i is reachable by checking if the auxiliary variable a_i is eventually true:

$$\exists \diamond\, a_i == True$$

11 Edge Annotation

Given an ATTA, an auxiliary variable must be created and maintained for each edge in the original TTA. Let $\lambda(e)$ be a function $\lambda : E \rightarrow 2^E$, such that

$$\lambda(e_\lambda = \langle l_\lambda, g_\lambda, u_\lambda, r_\lambda, l'_\lambda \rangle) = \{e = \langle l, g, u, r, l' \rangle \in E \mid l'_\lambda = l\} \tag{23}$$

The set given by $\lambda(e_\lambda)$ is thus the set of all edges that start in the location where edge e_λ ends. For each edge in the original set of edges, add one auxiliary variable to the set A.

$$A = \bigcup_{i=1}^{|e_i^c \in E^{con}|} \bigcup_{j=1}^{|\lambda(e_i^c)|} \{a_{ij}\} \tag{24}$$

Then, the auxiliary variables are updated on the newly created edges.

$$E^{aux} = \bigcup_{i=1}^{|e_i^c \in E^{con}|} \bigcup_{j=1}^{|e_j^o \in \lambda(e_i^c)|} \{e^a = \langle l_i^l, g, u, \emptyset, l_i^r \rangle | g = g_j^o \text{ and } u(v) = v[a_{ij} = tt]\} \tag{25}$$

Thus, a set of tests that cover all edges is a set of queries verifying whether for all edges in the original TTA, the annotation variable a_{ij} is eventually true:

$$\exists \diamond \, a_{ij} == True$$

12 MC/DC Annotation

Given an ATTA, two auxiliary variables must be created and maintained for every MC/DC pair on every edge in the original TTA.
Let σ be a function such that $\sigma(e \in E')$ maps to the n-tuple K of conditions that constitute the guard of edge e. Each condition $k \in K$ is then a mapping $k : \Lambda \times \mathbb{R}^C \to \mathbb{B}$.
Let $\alpha(g, k)$ be a function that for a given guard g and condition k in $\sigma(e)$ returns the set of MC/DC pairs of k, where an MC/DC pair is a configuration of conditions in the set $\sigma(e) \setminus k$ such that k exclusively determines the value of g. For example, given $g = ((a \wedge b) \vee c)$ then $\alpha(g, c) = \{(tt, ff), (ff, tt), (ff, ff)\}$. Intuitively, each $q \in \alpha(g, c)$ represents an MC/DC pair for condition c, where the configuration for a and b is given by q and the value of c is respectively tt and ff, forming a pair. This is shown in Table 1.

a	b	c	$(a \wedge b) \vee c$
tt	ff	tt	tt
tt	ff	ff	ff
ff	tt	tt	tt
ff	tt	ff	ff
ff	ff	tt	tt
ff	ff	ff	ff

Table 1. The three MC/DC pairs for condition c.

For each edge in the original set of edges, iterate through all of the conditions k and add two auxiliary variables to A for each MC/DC pair.

$$A = \bigcup_{i=1}^{|e_i^c \in E^{con}|} \bigcup_{j=1}^{|e_j^o \in \lambda(e_i^c)|} \bigcup_{m=1}^{|k \in \sigma(e_j^o)|} \bigcup_{n=1}^{|\alpha(g_j^o, k)|} \{a_{ijmn}, a'_{ijmn}\} \tag{26}$$

For all MC/DC pairs add two auxiliary edges, one for each auxiliary variable of the pair. The first edge represents the condition k, and the second represents the negation of k.

$$\begin{aligned} E^{aux} = & \bigcup_{i=1}^{|e_i^c \in E^{con}|} \bigcup_{j=1}^{|e_j^o \in \lambda(e_i^c)|} \bigcup_{m=1}^{|k \in \sigma(e_j^o)|} \bigcup_{n=1}^{|q \in \alpha(g_j^o, k)|} \\ & \{e_{ijmn} = \langle l_i^l, g, u, \emptyset, l_i^r \rangle | g = k \wedge \Big(\bigwedge_{w=1}^{(|\Sigma = (\sigma(e_j^o) \setminus k)|)} \Sigma_w = q_w \Big) \text{ and } u(v) = v[a_{ijmn} \mapsto tt]\} \cup \\ & \{e_{ijmn} = \langle l_i^l, g, u, \emptyset, l_i^r \rangle | g = \neg k \wedge \Big(\bigwedge_{w=1}^{(|\Sigma = (\sigma(e_j^o) \setminus k)|)} \Sigma_w = q_w \Big) \text{ and } u(v) = v[a'_{ijmn} \mapsto tt]\} \end{aligned} \tag{27}$$

For each condition in a given guard, MC/DC is achieved if for each condition k, it holds that the queries for at least one pair are satisfied.

$$\exists \diamond a_{ijmn} == True$$

$$\exists \diamond a'_{ijmn} == True$$

13 Non-determinism

One of the key requirements of HAALT is that it is possible to recreate an exact trace found in AALTITOAD in HAWK. It is therefore a problem when a non-deterministic choice happens in AALTITOAD, since HAWK might not make the same decision. HAWK handles non-deterministic decisions naively by always choosing the *first* edge. *First* is defined by the order in which the edges appear in the *.json* files that constitute a model. In contrast, AALTITOAD handles non-determinism by choosing a random valid edge. This discrepancy between strategies is an obvious error in either the implementation of HAWK or AALTITOAD that would be caught by HAALT. However, since it is a known error, it is not useful to generate a test suite to show that HAWK cannot not do the same as AALTITOAD in case of non-determinism. Therefore, the functionality to remove non-determinism from models has been implemented.

Removing non-determinism is achieved by numbering every edge in the annotated ATTA using an external variable p before it is given to AALTITOAD. Since each value of p directly corresponds to an edge taken by AALTITOAD, it is possible to tell HAWK exactly which edge it should take, depending on the assigned value of p. The set of edges in the $ATTA = (L, l_0, C, Z, E, v_0, c_0, \tau)$ is thus

$$E = \{e'_i = \langle l, g', u, r, l' \rangle \mid e_i = \langle l, g, u, r, l' \rangle \in E \; and \; g' : ((g) \wedge p = i)\} \tag{28}$$

and $\Omega = \Omega \cup \{p\}$.

It must be noted that this solution of adding the p-value to the original edges directly changes the model and allows for more possible states to be explored. The advantage is that it makes it possible to ignore errors caused by non-determinism, and instead, find other types of errors. A disadvantage is that the TTA no longer has multiple legal edges to take in a single state, which parts of the SafeCon III system are modelled after. This entails that AALTITOAD may find a trace that is not possible if HAWK is run independently of the trace from AALTITOAD. However, the main goal of HAALT is to test whether HAWK can follow the same trace as AALTITOAD. Thus, whether it is actually possible in the production environment on the trucks is not the purpose of the tests.

14 Determining trace similarity

Based on a set of reachability queries, the goal of HAALT is to run AALTITOAD on a network of ATTAs (NATTA) to determine a set of instructions to give HAWK. Both of these processes generate traces in the form

$$s_1,\ s_2,\ \dots,\ s_n$$

$$s_i = (\bar{l_i}, v_i, c_i) \in S = 2^L \times \Lambda \times \mathbb{R}^C$$

The states can then be compared one by one, and if any of the states do not match in the two traces, an error is found. The algorithm is shown in 1, where a single query is given. It is, however, trivial to extend the algorithm to allow for a set of queries.

Algorithm 1 Pseudo code for testing trace similarity.

```
 1: procedure TestSimilarity($NATTA$, $\varphi$)
 2:     $T_A = \text{AALTITOAD}(NATTA, \varphi)$
 3:     $I = \text{GenerateInstructions}(T_A)$
 4:     $T_H = \text{SimulateHAWK}(NATTA, I)$
 5:     for each $s_i = (\bar{l_i}, v_i, c_i) \in T_A$ do
 6:         if $s_i \neq T_{H,i}$ then
 7:             Error($s_i$)
 8:         end if
 9:     end for
10: end procedure
```

Determining the set of instructions to give HAWK is trivial assuming the usage of ATTAs, since it is impossible in ATTAs to have an edge going to and from the same location. This is because any loops on location l in the original TTA will go from l^r to l^l in the corresponding ATTA. Therefore, if the location changes, a tick has occurred, otherwise it was a tock.

Algorithm 2 Pseudo code for generating instructions

```
 1: function GenerateInstructions($T = s_1, s_2, .., s_n$)
 2:     Let $I$ be a new list
 3:     for each $s_i, s_{i+1} = (\bar{l_i}, v_i, c_i), (\bar{l_{i+1}}, v_{i+1}, c_{i+1}) \in T$ do
 4:         if $\bar{l_i} \neq \bar{l_{i+1}}$ then
 5:             Append $tick$ to $I$
 6:         else
 7:             Append tock($s_i, s_{i+1}$) to $I$
 8:         end if
 9:     end for
10:     return $I$
11: end function
```

A tock(s_i, s_{i+1}) contains the set of external variables that have been changed between s_i and s_{i+1}.

15 Results and Discussion

To gauge the practical use of HAALT, it was tested on SafeCon III components. Additionally, due to state space explosion, the ability of AALTITOAD to solve the generated queries is the main computational bottleneck of HAALT, so it is important that the annotation process does not increase the state space more than necessary. Therefore, we verify how many additional states we introduce for each annotation method to further reason about the practical use of HAALT.

15.1 Testing on SafeCon III

Throughout the development process of HAALT, SafeCon III has been the target model for testing. Individual parts of the model have been tested based on the formalism in Section 6. From testing single components at a time, HAALT has verified trace equivalence between HAWK and AALTITOAD based on location and edge coverage criteria. However, it is evident that when testing for the MC/DC coverage criterion, most components are simply too large, as AALTITOAD does not terminate for some queries. Additionally, when testing multiple components at a time, it is evident that there are inherent differences between AALTITOAD and HAWK, as HAALT immediately finds differences in the traces. This will be further elaborated in Section 16.2.

15.2 Experimental Setup

To evaluate the efficiency of queries running on ATTA models in comparison to TTA models, we utilize the setup used in [1]. The setup consists of a network of TTAs that are based on the Fischer's mutual exclusion algorithm shown in Figure 8. Two experiments are trialled: in experiment 1, the query $\varphi_1 : \exists \diamond ctr > 1$ is tested and for experiment 2, $\varphi_2 : \exists \diamond ctr = n$, where n is the number of Fischer components being tested. The queries were originally designed as safety queries, but suffice as an indicator of the size of the state space. For φ_2, the $\wedge\ ctr = 0$ part is omitted from the guard between $l3$ and $l4$, as the purpose is to see if every model can reach $l4$. For each experiment, AALTITOAD is run on the Fischer TTA model as well as the corresponding ATTA model for location, edge, and MC/DC coverage. Additionally, the pick-first state-picking strategy is used for consistency. The experiments are run on a PC with 8-core Intel i7-4770 3.40 GHZ CPU and 16GB of RAM on a Ubuntu 21.10 operating system.

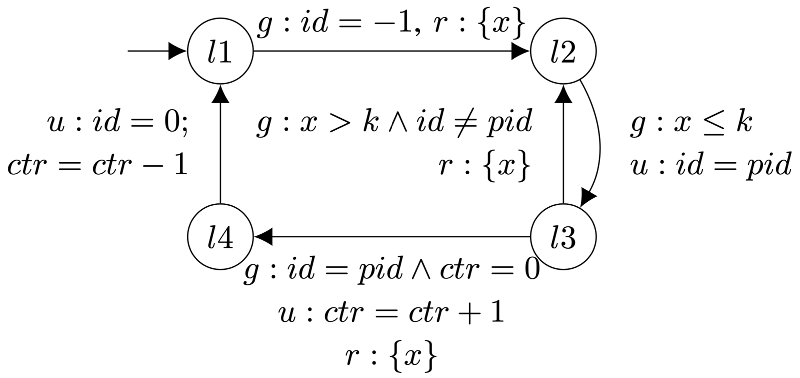

Fig. 8. Fischer model as a TTA. x is a timer, pid is the process identifier, id is a control variable, ctr is a counter for the number of parallel TTAs positioned in $l4$, and k is a constant $k = 2$. Taken from [1].

15.3 Data

The data is illustrated in Table 2 and Table 3 respectively, where data is collected for 1 to 5 parallel Fischer TTA models. Fi* denotes that each Fischer model is modified so that the guard on the edge between $l3$ and $l4$ now is only $id = pid$. In this context, N/A denotes that AALTITOAD terminated without results due to running out of memory. It is evident that the annotated models introduce a non-linear amount of states to the explored state space. Furthermore, the queries do not terminate for Fischer-4 in experiment 1 nor Fischer-5 for experiment 2. Annotation for MC/DC coverage is especially limited, as φ_1 only terminates for Fisher-1 and Fischer-3 for φ_2.

	Not annotated	Location	Edge	MC/DC
F1	9	21	34	173
F2	99	1367	4234	N/A
F3	739	44952	259043	N/A
F4	4421	N/A	N/A	N/A

Table 2. Number of unique explored states for φ_1.

	Not annotated	Location	Edge	MC/DC
F1*	6	22	18	48
F2*	52	144	500	587
F3*	379	9938	6110	11111
F4*	1421	17362	70753	N/A
F5*	19061	N/A	N/A	N/A

Table 3. Number of unique explored states for φ_2.

15.4 Evaluation of Results

Based on the results, it is clear that performing annotation greatly increases the amount of states to be explored. As seen in Table 2, when running AALTITOAD on an annotated model consisting of four Fischer components, it runs out of memory as a result of the increased state space for all annotation methods. For MC/DC, AALTITOAD could only handle one Fischer component. Considering the small size of a Fischer component, this means that queries for annotated models with even a few small components can be practically impossible for AALTITOAD to satisfy. In addition, unlike in SafeCon III, no external variables are used in the Fischer models, which would increase the set of possible values for variables and states. This is worsened when testing on a subset of the SafeCon III model, as some internal variables are made external. Hence, AALTITOAD has similar problems running reachability searches on SafeCon III models with few annotated components.

Since the test suite for MC/DC coverage is the most thorough, it is expected that it generates the most states to explore. While this requires more memory, it is not inherently bad, since the goal of performing MC/DC is to test a wide range of possible program executions. Similarly, edge coverage will generally generate more states to explore than location coverage.

While the verification of the queries for MC/DC coverage requires many states to be explored, our annotation method introduces many new states that are only considered new by AALTITOAD due to the auxiliary variables. Specifically, when auxiliary variables are updated as a result of taking an auxiliary edge, AALTITOAD will regard it as a new state. Considering that there can be multiple auxiliary edges between two locations, many states can be added because of this. An example of this is illustrated in Figure 9. For the given example, the amount of new states for AALTITOAD to search grows exponentially. Additionally, to take an auxiliary edge, external variables may have to be changed to satisfy the guard of the auxiliary edge, expanding the state space further.

However, these changes to auxiliary variables and external variables have no impact on possible future states, as auxiliary variables never appear in guards, and external variables can at all times be given new values non-deterministically. As such, to make it feasible for AALTITOAD to solve more queries, states added due to the annotation of the model could be ignored. Specifically, if a tick is performed, and the new state is only different from a previously explored state due to external or auxiliary variables, the state should not be considered unique nor be further explored.

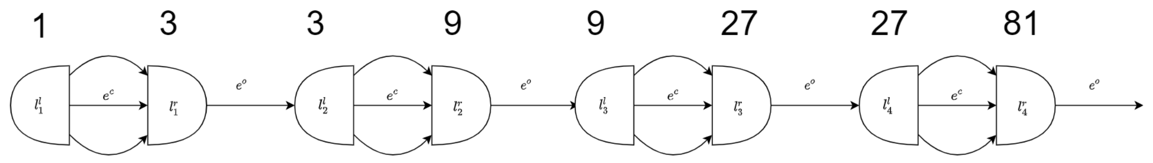

Fig. 9. If the annotation method adds two additional edges for each location, the amount of additional states for AALTITOAD to search is equal to 3^n, where n is the number of locations in the original TTA.

16 State of Aaltitoad

An underlying assumption of HAALT is that AALTITOAD is a working verification engine for Tick Tock Automata. To ensure that this is a reasonable assumption, the state of AALTITOAD has continuously been surveyed for bugs, and improvements have been made throughout the process of developing HAALT in collaboration with HMK Bilcon. Many errors have been found in AALTITOAD and almost as many have been fixed. In the following sections, the current state of AALTITOAD will be accounted for including its current limitations.

16.1 State space explosion

As discussed in [1], AALTITOAD is limited in the size of TTAs it can model check, due to state space explosion. HAALT depends on AALTITOAD to generate traces, so these limitations also apply to HAALT, which makes it practically impossible to use HAALT on non-trivial components of SafeCon III. In [1], Gitz-Johansen discusses some method to improve the state space exploration algorithm in AALTITOAD, which may allow for query verification on larger TTAs.

16.2 Network of tick tock automata

In its current state, AALTITOAD does not adhere to the semantics described in [2] concerning the behavior of a network of Tick Tock Automata (NTTA). The intended behavior is that a tick-transition is global, where all TTAs in the NTTA must change location if a tick-transition is valid. This is not the case in AALTITOAD, where it is possible for TTAs to change states one at a time despite other TTAs having valid tick-transitions. Consequently, the only result you get when testing multiple components is that HAWK and AALTITOAD generate different traces.

16.3 Parsing guards

AALTITOAD incorrectly parses guards containing external variables and parentheses. The extend of this error is not completely understood, but if a guard contains an external variable and parentheses, AALTITOAD may evaluate the entire guard to false. Since the translation from a set of components to a network of TTAs often changes variables to external, this error limits the amount of queries that can be satisfied by AALTITOAD.

17 Conclusion

In this paper, we introduced and formalized the notion of Annotatable Tick Tock Automata models based on different test coverage criteria. This includes semantics for annotation methods based on location coverage, edge coverage, and MC/DC coverage. Following the semantics, the different annotation methods were implemented in a program called HAALT, which is used for automatic test generation. Using the verification engine, AALTITOAD, HAALT is able to verify whether the industrial runtime environment HAWK is trace equivalent with the semantics of Tick Tock Automata for some coverage criterion. However, it was found that HAALT currently has limited practical use, as the annotation methods introduce too many new states for AALTITOAD to handle with the current state of AALTITOAD. Testing on the industrial system SafeCon III revealed that HAALT is able to perform automatic verification on small components. When testing multiple components at a time, an error is immediately found, indicating an inherent difference in the handling of Network of Tick Tock Automata between HAWK and AALTITOAD.

HAALT is currently available for HMK Bilcon to use and further develop.

18 Future work

As HAALT is dependent on AALTITOAD for verification, resolving the limitations outlined in Section 16 would also benefit HAALT. This is a plausible solution, as AALTITOAD is open source. Improvements can also be made to the annotation process in HAALT, especially for MC/DC. The main improvement would be to reduce the number of duplicate states, which are caused by auxiliary edges and auxiliary variables. For a tick transition, the resulting state should only be considered a unique state, if there is a difference in internal variables, and not auxiliary or external variables.

References

[1] Asger Gitz-Johansen. *AALTITOAD: A Tick Tock Automata Verification Engine*. visited on 05-05-2022. 2020. URL: `http://ulrik.blog.aau.dk/hmk/`.

[2] Asger Gitz-Johansen. *Tick Tock Automata: A Modelling Formalism for Real World Industrial Systems*. visited on 05-05-2022. 2020. URL: `http://ulrik.blog.aau.dk/hmk/`.

[3] Eduard P Enoiu et al. "Automated test generation using model checking: an industrial evaluation". In: *International Journal on Software Tools for Technology Transfer* 18.3 (2016), pp. 335–353.

[4] Thomas Pedersen et al. *HMKAAL: Formal Verification and Detection of Non-determinism and Data Races in Industrial Systems*. visited on 05-05-2022. 2020. URL: `http://ulrik.blog.aau.dk/hmk/`.

[5] Alessandro Cimatti et al. "NUSMV: a new symbolic model checker". In: *International journal on software tools for technology transfer.* 2.4 (2000). ISSN: 1433-2779.

[6] Johan Bengtsson et al. "UPPAAL in 1995". In: *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 1996, pp. 431–434.

[7] Anders Hessel et al. "Testing real-time systems using UPPAAL". In: *Formal methods and testing*. Springer, 2008, pp. 77–117.

[8] Anders Hessel et al. "Time-optimal real-time test case generation using UPPAAL". In: *International Workshop on Formal Approaches to Software Testing*. Springer. 2003, pp. 114–130.

[9] Anders Hessel and Paul Pettersson. "A Global Algorithm for Model-Based Test Suite Generation". In: *Electronic Notes in Theoretical Computer Science* 190.2 (2007). Proceedings of the Third Workshop on Model Based Testing (MBT 2007), pp. 47–59. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/j.entcs.2007.08.005`. URL: `https://www.sciencedirect.com/science/article/pii/S1571066107005403`.

[10] Willibald Krenn et al. "Momut:: UML model-based mutation testing for UML". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–8.

[11] Dimitris Dranidis, Konstantinos Bratanis, and Florentin Ipate. "JSXM: A tool for automated test generation". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2012, pp. 352–366.

[12] Damiano Angeletti et al. "Using bounded model checking for coverage analysis of safety-critical software in an industrial setting". In: *Journal of Automated Reasoning* 45.4 (2010), pp. 397–414.

[13] Martin Leucker and Christian Schallhart. "A brief account of runtime verification". In: *The Journal of Logic and Algebraic Programming* 78.5 (2009). The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07), pp. 293–303. ISSN: 1567-8326. DOI: `https://doi.org/10.1016/j.jlap.2008.08.004`. URL: `https://www.sciencedirect.com/science/article/pii/S1567832608000775`.

[14] Cyrille Artho et al. "Combining test case generation and runtime verification". In: *Theor. Comput. Sci.* 336 (2005), pp. 209–234.

[15] Edmund Clarke, E. Emerson, and Joseph Sifakis. "Model checking". In: *Communications of the ACM* 52 (Nov. 2009). DOI: `10.1145/1592761.1592781`.

[16] Sonali Pradhan, Mitrabinda Ray, and Srikanta Patnaik. "Coverage criteria for state-based testing: a systematic review". In: *International Journal of Information Technology Project Management (IJITPM)* 10.1 (2019), pp. 1–20.

Appendices

Appendix A

Design of HAALT

In this appendix, we document the design considerations and implementation of HAALT.

1 Annotators

The design of the *annotator* classes is shown in Figure 10. The annotators are designed based on the template pattern inspired by the semantics. A super class *annotator* contains two abstract public methods *annotate* and *CreateQueries*, which are implemented by the respective sub classes, where a sub class is defined for each type of coverage criteria. Additionally, the annotator class contains methods to split each location into two and adding a connection edge according to the semantics described in Section 7.

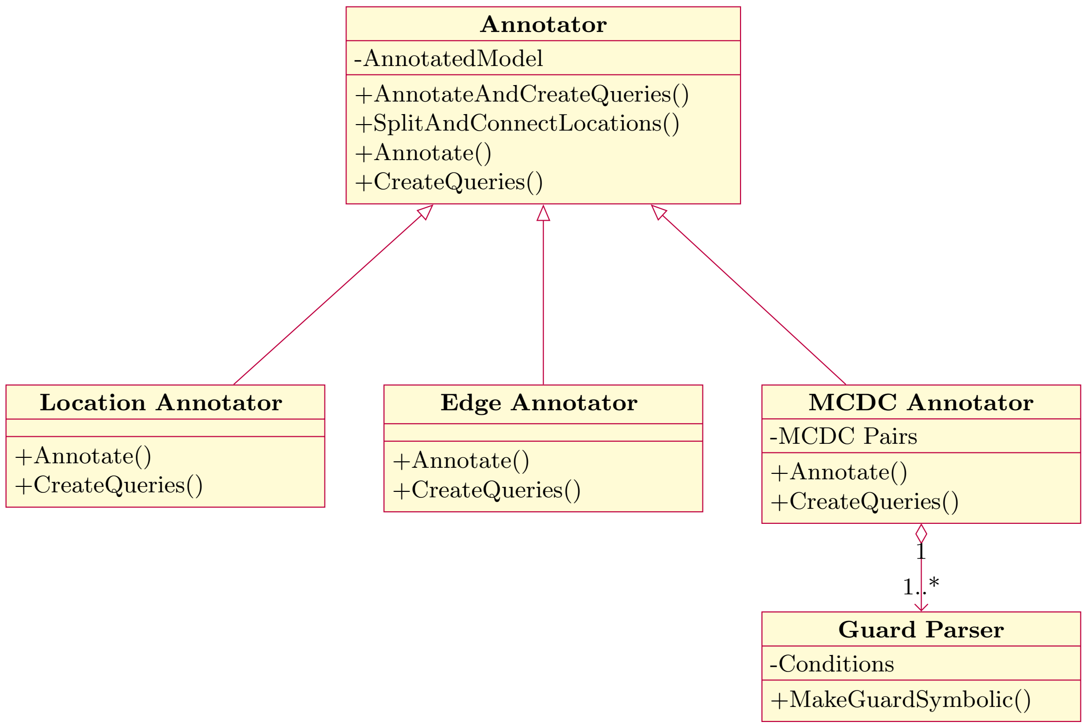

Fig. 10. Design of annotators for each coverage criteria.

2 MC/DC Annotator

The MC/DC pairs are found by first splitting a guard into all of its condition. For this task, a guard parser was implemented, which given a guard, returns the guard split into its conditions, where the conditions have template variable names. MC/DC pairs are found for each condition in the guard, by testing each possible configuration, and testing if the condition evaluates the decision. If it does, an MC/DC pair is found. For each pair, two queries and two edges are added to the ATTA.

3 Guard-parser

In order to generate MC/DC pairs, a *guard parser* module for guards is implemented. The main feature of the module is extracting the individual conditions of a guard as a dictionary and creating a symbolic representation of the guard. The module is implemented as a top-down recursive descent parser based on the ll(1) context free grammar in Appendix 3. Using this method, a parse tree is created after which a series of tree walks reduce the tree and find conditions seperated by disjunctions and conjunctions. Reductions include reducing logical expressions such as $A == B$ to a single auxiliary variable c_i that symbolically represents the condition. Note that the grammar does not enforce precedence between neither logical separators nor arithmetic operators. This is because the symbolic representation of the guard preserves the ordering of the conditions as well as placement of parentheses, and as such, the Python Eval function is simply used to enforce precedence.

```
Guard  ::=  A

A      ::=  B A'
A'     ::=  || A  |  && A  |  ϵ

B      ::=  C  B'
B'     ::=  == B   |  != B   |  ϵ

C      ::=  D C'
C'     ::=  > C   |  < C  |  <= C   |  >= C   |  ϵ

D      ::=  E D'

D'     ::=  + D  |  - D   |  * D  |  / D  |  % D  |  ϵ

E      ::=  G  |  - G  |  ! G

F      ::=  id  |  number  |  false  |  true  |  ( A )
```

4 Dependency graph

The *dependency graph* module is used to find all dependencies between components, as described in Section 6. The module is instantiated by finding all dependencies between all components and *.parts* files in the SafeCon III model. Dependencies are represented as a dependency graph, where the nodes are components and edges go from usage to defintion. The module takes as input a set of components to be tested and returns a set of variables that need to be made external based on the set definitions in Section 6. An example of a dependency graph is shown in Figure 11, where dependencies are shown between the component, the context (other components not part of the test bundle), the *main* component, and the *.parts* files.

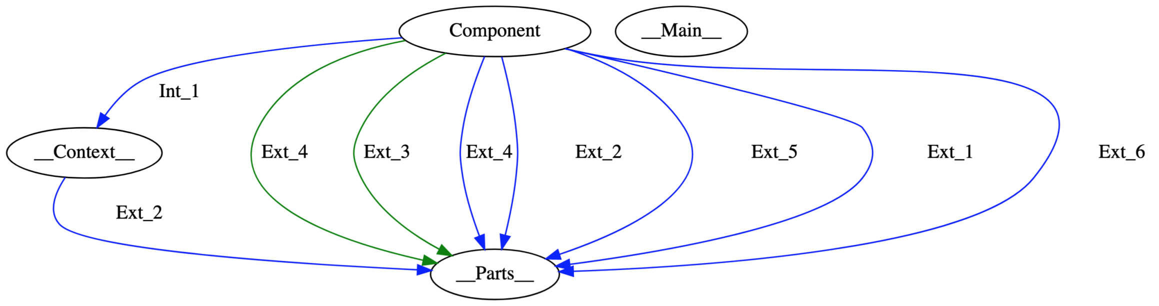

Fig. 11. Example of dependency subgraph. Dependencies are shown between __Parts__, __Main__, __Context__, and the component. Blue edges are reads, green edges are writes.

5 Modelcreator

The *modelcreator* module is responsible for taking the user specified components, and creating a model consisting of only those components, which can then be annotated and run by HAWK and AALTITOAD. This corresponds the conversion of a set of components T to a network of TTAs described in Section 6. As shown in Figure 12, the modelcreator uses the dependency graph module to find which variables to keep internal and which to make external. Additionally, because HAWK allows for generic components, which can be instantiated with input parameters, the modelcreator has to create a component for each instantiated component. This encompasses declaring variables to the value of the input parameter of the component. To accomplish this, the modelcreator goes through all of the specified components and creates a new component file that includes the relevant declarations for internal variables, and external variables are placed in the parts file. Lastly, all of the components are specified as subcomponents inside a newly generated main file.

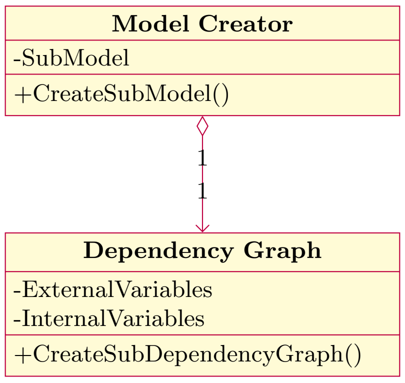

Fig. 12. Design of model creator.

6 Comparer

The *comparer* is responsible for comparing the AALTITOAD trace with the HAWK trace, and generating appropriate error/warning/success messages. The outputs are not immediately comparable, due to differences in formatting standards and it therefor needs to be parsed. Afterwards, each state in the traces are compared, which is the implementation of line 6 in Algorithm 1. Since we want to give precise error and warning messages, we have multiple checks for comparability. The different errors that can be found are:

- Different locations
- Different amount of states
- Different variables
- Different variable values

If an error is encountered, a long error message containing information about the found differences, query, annotator, and both before and after states from HAWK and AALTITOAD are written to a log file. If a warning is found, a similar message explaining the warnings is created and written to a warning file. Lastly, a short message is written to the console summarizing the errors.

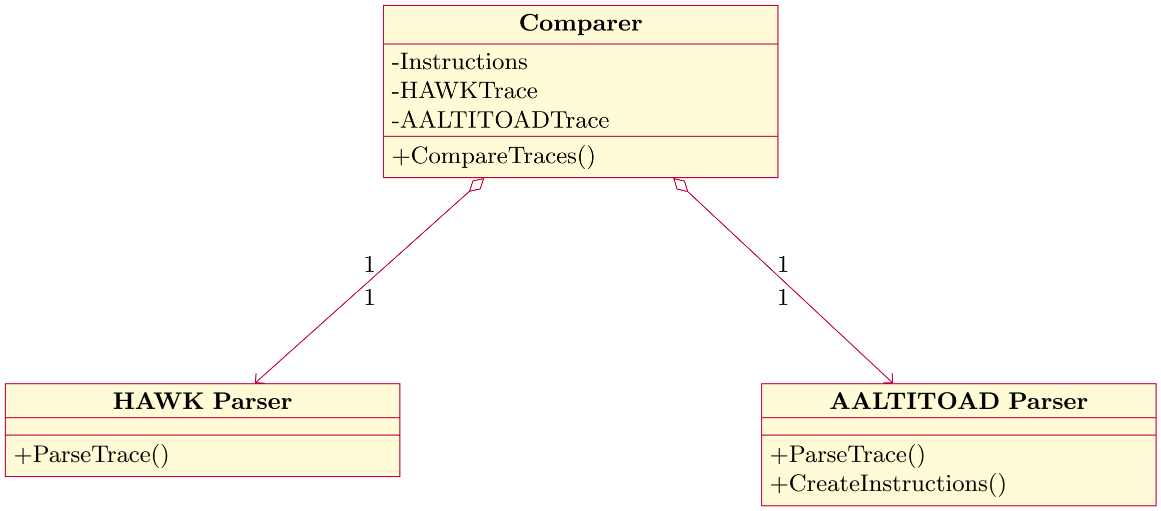

Fig. 13. Design of trace comparer.