Towards Identifying Spurious Paths in Combined Simulink/Stateflow Models[1]

Marcus Mikulcak[2], Thomas Göthel[3], Paula Herber[2] and Sabine Glesner[2]

Abstract:

MATLAB/Simulink and its state machine design toolbox Stateflow are widely-used industrial tools for the development of complex embedded systems. Due to the dynamic as well as heterogeneous nature of models that contain both Simulink and Stateflow components, their analysis poses a difficult challenge. This paper outlines an approach to relate the semantics of both Simulink and Stateflow and how to use it to perform an information flow analysis on a combined Simulink and Stateflow model. In the first step, we analyze the Stateflow automata and generate *timed output traces* for arbitrary inputs. In the second step, we use an existing *timed path condition* extraction algorithm for the Simulink components to identify conditions for information flow on paths of interest. Finally, we analyze whether the compiled sets of timed path conditions are contained in the timed output traces that we derive by using a novel trace notation for Stateflow automata. This approach makes it possible to safely rule out the existence of information flow on specific paths through a model. Further, it presents a starting point to reason about non-interference between model parts, compliance with security policies as well as the generation of feasible, efficient test cases.

Keywords: Embedded Systems, MATLAB, Simulink, Stateflow, Path Conditions, Information Flow

1 Introduction

In the area of safety-critical embedded software, such as in the automotive and aerospace domain, programming errors can lead to disastrous and, if occurring, often fatal accidents. At the same time, the complexity of such systems has increased dramatically over recent years. To cope with the steadily increasing complexity, current design processes rely more and more on models. One of the most widely-used tools for model-based design is MATLAB/Simulink [Ma15] by MathWorks, which supports the graphical design and simulation of time-continuous as well as time-discrete systems using block diagrams. To additionally support the design of state machine-based embedded controllers and model them in conjunction with these dynamical systems, Stateflow [Ma14], an extension to Simulink, is widely used in industrial contexts.

However, due to the complexity and the dynamic character of the developed models, the analysis of a given model is a difficult challenge, in particular if timing aspects are

[1] Funded by the German Federal Ministry of Education and Research as part of the research project CISMo

[2] Technische Universität Berlin, Software and Embedded Systems Engineering Group, Ernst-Reuter-Platz 7, 10587 Berlin, Germany, marcus.mikulcak@tu-berlin.de

[3] Service and Software Engineering Group, Universität Potsdam, August-Bebel-Straße 89, 14482 Potsdam, thomas.goethel@uni-potsdam.de

considered. At the same time, knowledge about the existence of certain paths, the conditions under which they are executed and how an embedded Stateflow controller influences their behavior is highly desirable.

Using knowledge about the behavior of Stateflow controllers and their influence on conditionally executed components of the model makes it possible to identify interference and non-interference between model parts and, thus, to reason about compliance with security policies. Furthermore, knowledge about the existence of paths provides a basis for generating feasible, efficient test cases for quality assurance and for calculating more precise *Worst Case Execution Time* (WCET) bounds for Simulink/Stateflow models.

Additionally, the information gathered using our approach can be used as a first step to identifying areas of low dynamic coupling in Simulink/Stateflow models and to subsequently extract abstract boundaries between these areas. This makes it possible to safely break down large models into smaller components which in turn can be analyzed using existing quality assurance methods, such as model checking, thereby avoiding the state space explosion problem. Together with our industrial partner, *Model Engineering Solutions GmbH* (MES) [ME16], we identified a number of models from the automotive sector that match the modeling style our approach is able to analyze and that therefore benefit from such a separation, such as electronic gearbox system controllers and interior and exterior lighting controllers. These models are developed using one or multiple Stateflow automata controlling data flow sections designed in Simulink and due to their size and dynamic complexity, are difficult to analyze without prior separation.

In this paper, we illustrate our three-step approach to identify spurious paths in combined Simulink/Stateflow models using a small running example. First, we extract *timed output traces* from an embedded Stateflow controller. These extracted traces form an over-approximation of the automaton behavior as we aim to support arbitrary input signals. We then use a previously published algorithm to extract timed path conditions from dynamic data-flow components developed in Simulink and prepare them for analysis. Finally, we present a comparison algorithm for timed output traces and timed path conditions to identify overlap between them. If there is an overlap, the conditionally executed paths under analysis exist. If not, they are identified as infeasible and will never be executed in the model.

2 Preliminaries

2.1 Path Conditions

In general, *path conditions* [Ki76] describe necessary conditions for paths to be executed. In [HSS08], path conditions are used to capture all paths where information might flow from a source to a target. Consider the example given in Listing 1. There, a variable `src` is assigned to an element of the array `a`. Inside the `if` statement, the variable `tgt` is assigned with an element of the array. A static analysis of the possible paths of this program would detect potential information flow from `src` to `tgt`. However, a more precise computation of path conditions takes data and control dependencies into account.

```
1 a[i + 3] = src;
2 if ((i > 10) && (j < 5))
3   tgt = a[2 * j - 42];
```

Listing 1: Path condition example

When traversing the path of information flow between lines 1 and 3, the necessary conditions are derived from the array indices and the `if`-condition and can be expressed as:

$$PC(1 \rightarrow 3) = \exists i, j\big((i > 10) \wedge (j < 5) \wedge (i+3 = 2j-42)\big)$$

The result is that there is no assignment to `i` and `j` such that the equation holds. In contrast to the result from a static analysis, this means that no information flow is possible. This simple example demonstrates that a path condition-based analysis is able to offer more precise answers about which conditions have to hold for information to flow.

2.2 Information Flow Analysis

The protection of confidentiality of information inside a software system is a long-standing and increasingly important problem in the areas of general computing as well as embedded systems. Protecting not only the data itself but also the integrity of the functionality that produces and handles data is a goal of software non-interference policies [GM82]. Such policies, based on the assignment of security levels to data elements, describe rules between which levels information flow is allowed or forbidden [SM03]. When aiming at assuring *confidentiality*, data is prohibited to flow *to* inappropriate locations, while in the context of *integrity*, data is prohibited to flow *from* inappropriate sources. As non-interference refers to the absence of information flow, it ensures both confidentiality and integrity.

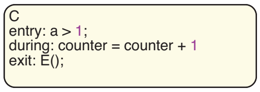
2.3 MATLAB/Simulink

MATLAB/Simulink [Ma15] is an add-on to the MATLAB IDE by MathWorks that enables graphical modeling and simulation of reactive systems. In its data-flow oriented notation, Simulink employs *blocks* which are connected using *signals*. Additionally, each block and signal is assigned a set of *parameters*.

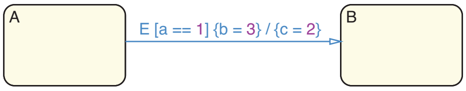
Simulation of Simulink models is performed using *solvers*, which compute the output of each block according to its semantics. *Variable step* solvers aim at automatically finding a simulation step size for each block in the model to achieve a level of precision set by the model developer. *Fixed step* solvers use a fixed simulation step size at the expense of precision while increasing performance. The former class of solvers is commonly used for hybrid or purely time-continuous systems, while the latter is used for time-discrete models.

2.4 Stateflow

Stateflow [Ma14] is a further add-on to the MATLAB IDE, especially to Simulink, and gives the designer the possibility to integrate decision logic based on state machines and flow charts into a Simulink model. Stateflow provides complex modeling styles incorporating multiple state, event and transition types as well as an execution semantic not only dependent on the structure and annotations of the model, but also on its layout. The underlying semantics resemble that of statecharts [Ha87] as they utilize hierarchical and concurrent state structures, junctions splitting transitions and actions as part of state definitions.



(a) Example of a `State` in Stateflow



(b) Example of a `Transition` in Stateflow

2.4.1 States

States form the basic building block of the decision logic implemented in Stateflow. An example of a state is shown in Figure 1a. When the execution enters a state, a set of *actions* modeled by the designer takes place. *When* these actions are performed is determined by the *action type*: `entry`, `during` and `exit`. While the sets of `entry` and `exit` actions occur only once every time the state is active, the `during` actions are performed with every simulation step and are therefore dependent on the selected solver of the Simulink and Stateflow model.

2.4.2 Transitions

To design the state change logic of an automaton, Stateflow states are connected via *transitions*. Similar to states, it is possible to add guards, trigger events and actions to transitions. Figure 1b shows an example transition containing all three mentioned semantical elements. After receiving event `E` as the `exit` action of state `C`, it is evaluated if the current transition is valid, which in turn evaluates the guard condition `a == 1` and, if true, perform the transition. As soon as this guard is evaluated to true, the *guard action* `b = 3` is executed. Before finally marking state `B` as active, however, the *transition action* `c = 2` is executed.

2.5 Timed Path Conditions in MATLAB/Simulink

We have previously presented our approach to compute *timed path conditions* in MATLAB/Simulink [Mi15], which we extend upon in this paper. The main idea of timed path conditions is to transfer the concept of path conditions from sequential programming languages like `C` to the Simulink modeling language. The main challenges are to take both data and control dependencies into account and to cope with timing dependencies. Data dependencies are simply resolved by following signal lines where each connection corresponds to a direct dependency. Control dependencies are more difficult to compute as they introduce conditional dependencies which are locally resolved. To overcome this problem, control flow dependencies are propagated backwards through the model to the input signals. With that, it is possible to decide whether a certain path actually exists on a very fine-grained level. For both data and control dependencies, timing dependencies are taken into account. An output might only depend on an input at certain points of time, and routing policies might even take advantage of timing delays to make sure that two signals can never interfere. Timing dependencies are taken into account by introducing *time slices* and timed path conditions are expressed with respect to relative time slices.

3 Path Existence in Simulink/Stateflow Models

In the following section, we describe the main contribution of this paper. First, we present an overview of our approach and an example model to illustrate the problem we aim at solving. Subsequently, we introduce the assumptions we impose on the models that our approach in its current proof-of-concept form is able to analyze. Finally, using our running example, we present our timed trace format and how to extract them from Stateflow automata, the generation of timed path conditions and the comparison algorithm for both trace sets.

3.1 General Approach

The heterogeneous nature of models containing both Simulink and Stateflow parts makes their analysis hard. The main challenge is to reconcile the inherently different semantics of Simulink and Stateflow, and in particular their timing. The semantics of Simulink is defined by the simulation semantics of the solver, where the functionality and timing depend on the simulation step size. The semantics of Stateflow is defined by evaluation rules that determine which transition fires in each step, whereby a step is made whenever one of the input signals changes. The main idea of our approach for identifying spurious paths in combined Simulink/Stateflow models is to relate a Stateflow Controller with its surrounding Simulink model using the concept of timed traces. As the identification of spurious paths in Simulink models [Mi15] is achieved by a backwards analysis through the model, the incorporation of Stateflow components requires us to analyze the effects of a Stateflow component on its (signal and variable) outputs for arbitrary inputs. Note that a

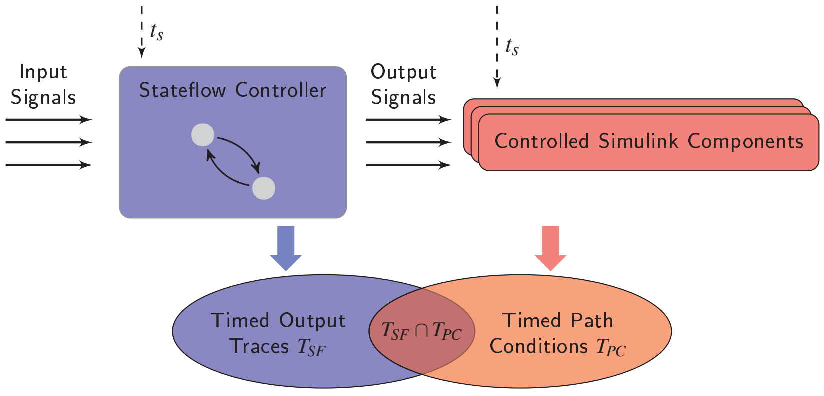

t_s
Input Signals
Stateflow Controller
Output Signals
t_s
Controlled Simulink Components
Timed Output Traces T_{SF}
$T_{SF} \cap T_{PC}$
Timed Path Conditions T_{PC}

Figure 2: Approach to identify spurious paths in combined Simulink/Stateflow models

Stateflow state machine is connected to the surrounding model via Simulink signals that form the variables used inside guards. Variables modified in state or transition actions inside Stateflow state machines form their output signals and are routed to the Simulink model. The evaluation of a Stateflow automaton is performed whenever one of the input signals to the automaton changes its value. Then, its state is reevaluated and a single possible transition is taken. We can therefore define a minimal time interval between every change in the output of a Stateflow automaton. Under the assumption of a uniform sample time throughout the model, it is equal to the simulation step size t_s. This relation between the discretely-timed solver of the Simulink model and the evaluation of the Stateflow automaton makes it possible to relate both semantics. As shown in Figure 2, we assume that a Stateflow Controller is embedded into a Simulink model and has an effect on some of its components. Our approach to identify spurious paths in combined Simulink /Stateflow is threefold: First, we analyze the behavior of an embedded Stateflow automaton under functional and timing aspects. Specifically, we extract timed traces for outputs signals influencing the control flow of the surrounding Simulink model by unrolling the automaton until a stable state is reached. The output of this step is a set of possible traces T_{SF} for an output signal of interest that forms the basis for analysis of path existence in the combined model. Second, we utilize an existing algorithm Section 2.5 to extract timed path conditions for paths of interest from the Simulink model. Along these paths that carry, e.g., safety-critical information whose flow needs to be analyzed, the conditions for information flow as well as their timing are gathered and expressed as traces T_{PC}. Third, we compare the traces derived from Simulink path conditions with the extracted timed output traces of the Stateflow automaton. In this final step, we analyze whether the sets of timed path conditions can be generated by the Stateflow automaton, i. e., if they overlap with the timed output traces $(T_{SF} \cap T_{PC})$. We explain these three steps using a running example, presented in the following section.

Running Example To illustrate our approach, we use a model of a shared buffer originally presented in [WHW10], shown in Figure 3. There, information of the security levels *public* and *confidential* is fed into a shared buffer, implemented as a `Mem` block. According to the operation mode, confidential (mode 1) or public (mode 2), information is saved in the buffer and passed to the corresponding output. The mode of operation to access the buffer is determined by the Stateflow model in Figure 4. Write access to the buffer on the corresponding security level is requested using the `write_request` signal if the state machine is in its `waiting` state. To return to the initial state, a read request with the corresponding level is used. Data output from the automaton to the surrounding Simulink components is realized using the *entry actions* inside the states. Whenever the automaton *enters* a state, the action is performed. In our running example, the output signal `current_state` is written. While the automaton is in the `public` or `confidential` mode, the Simulink switches are set to only allow for input and output of public or confidential data, respectively.

Figures 5a to 5c illustrate that due to the timing behavior of the Simulink components in the system, information flow from the confidential input to the public output is indeed possible. Shown there, a change in operation mode from `confidential` to `public` at time $t = 5$ results in a leak of confidential information stored in the buffer to the public data output.

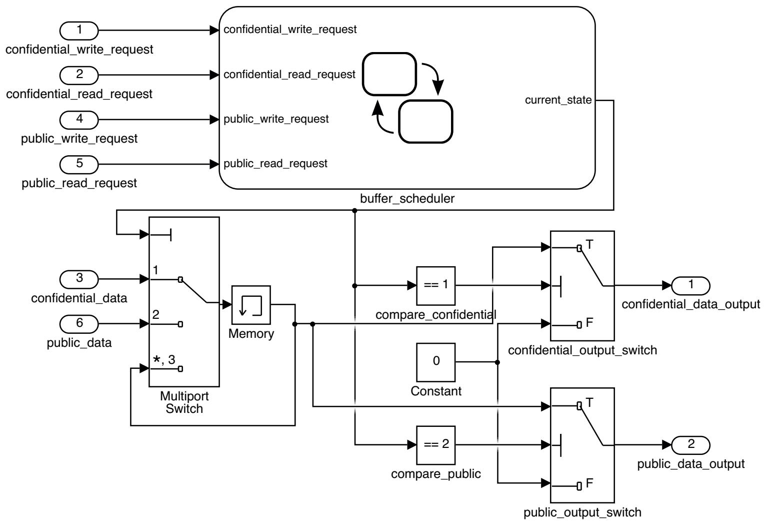


Figure 3: Model of a shared buffer in Simulink

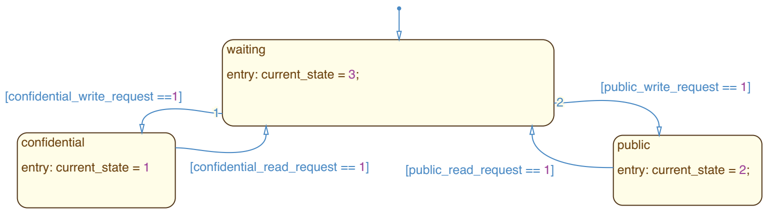

waiting
entry: current_state = 3;
[confidential_write_request ==1]
1
2
[public_write_request == 1]
confidential
entry: current_state = 1
[confidential_read_request == 1]
[public_read_request == 1]
public
entry: current_state = 2;

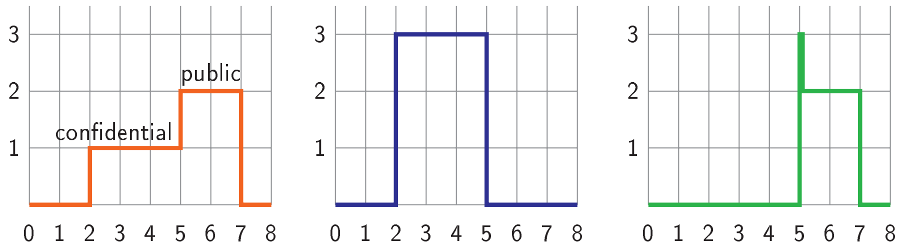
Figure 4: The Stateflow automaton of our running example

Timed Traces As a basis for comparison of the outputs of a Stateflow automaton and the sets of timed path conditions extracted using the algorithm described in Section 2.5, we define a data format to capture all required information. We call this a *timed trace*. For a given signal s_1 we capture its valuations $v_1, v_2, \ldots, v_n$ and the number of simulation steps that pass between the occurrences of these valuations. The general formats of the traces are:

$$s_1 = \{v_1 \xrightarrow{nt_s} v_2 \xrightarrow{kt_s} \ldots\}, \quad s_2 = \{v_1 \xrightarrow{\lfloor nt_s \rfloor} v_2 \xrightarrow{\lfloor kt_s \rfloor} \ldots\}$$

There, s_1 and s_2 denote signals in the Simulink model or outputs of the Simulink automaton while n and k are factors of the simulation step size t_s. Shown in s_2 is the notation of *at least* n and k simulation steps passing between the two valuations of s_2.

Assumptions As we present a proof-of-concept implementation for the evaluation of path existence in combined Simulink/Stateflow models, both the Stateflow as well as Simulink components have to fulfill certain requirements. For the latter, the requirements are listed in [Mi15]. For the former, we as of now assume a basic form of Stateflow automaton consisting of only states and transitions without any hierarchical grouping or the use of



(a) Operation mode (b) Confidential data output (c) Public data output

Figure 5: Timing of the shared buffer signals

junctions. However, we are confident that the presented approach can be extended to support a wide range of Stateflow modeling features and styles which we describe in Section 6.

3.2 Generating Timed Traces from Timed Path Conditions

In the first step of our approach, we use the algorithm summarized in Section 2.5 to generate timed path conditions from the dynamic data-flow components. The solved constraints in our example model result in the following path conditions:

$$C\big(p(o_2^t, i_3^{t-1})\big) = \big\{(s_{\text{control}}^t == 2), s_{\text{control}}^{t-1} == 1\big\}$$

This set shows the conditions for information to flow through the path starting at data inport i_3 for confidential data and leading to the data output port i_2 for public data. As can be seen there, two conditions are imposed on the signal denoting the current state of the system (either `confidential` or `public`). The conditions denote that information flows from the public inport to the confidential output whenever s_{control} is first set to 1 (at an arbitrary time $t-1$) and then set to 2 in the next time step (at time t). Note that $t-1$ in the algorithm denotes a time that is a single simulation step earlier than t, i. e., time $t-t_s$. We can therefore rearrange the condition set into a timed trace $\tau \in T_{\text{PC}}$ as follows:

$$\tau(s_{\text{control}}) = \{1 \xrightarrow{t_s} 2\}$$

In our example, the trace $\tau(s_{\text{control}})$ defines the conditions for a security policy violation to occur in the Simulink components, i. e., confidential data is led to the public data outport. Therefore, if we are able to identify $\tau(s_{\text{control}})$ in the output traces of the Stateflow automaton, the security violation occurs in the combined system and vice versa.

3.3 Establishing a Control Relation

To elevate the timed path conditions and to generate a global statement about the existence of traces, a connection between the signals controlling the execution of the path and the output signals A of the Stateflow automaton needs to be identified. The timed path condition extraction algorithm aims to unify the signals controlling all control flow elements such as `Switch` and `If-Action-Subsystem`. Using this information, our approach follows these signals backwards through the Simulink model until it encounters a Stateflow automaton.

Running Example In our example, this relation is established by following the signal path such that:

$$s_{\text{control}} = o_{\text{current\_state}}, \quad \text{with } o_{\text{current\_state}} \in O$$

If a connection is found, we extract output traces from the Stateflow automaton, specifically for the control signals of the paths. This process is illustrated in the following section.

3.4 Extracting Timed Stateflow Automata Traces

In this section, we present our approach to extract timed output traces from Stateflow automata. As explained above, a prerequisite to this analysis is an established connection between the control signals of the path through the Simulink components, i. e., the path under analysis, and the Stateflow controller setting these signals. The inputs to this step are the traces generated from the timed path conditions and their connection to the automaton.

In this step, we especially consider the possibilities of modeling states and transitions in Stateflow. When a state is entered, a set of *actions* takes place, such as the modification of output signals of the automaton or the triggering of *events*. *When* these actions are performed is determined by the *action type*: `entry`, `during` and `exit`. Depending on this choice by the programmer, the timing behavior and the frequency of the modifications and triggers changes. While the sets of `entry` and `exit` actions occur only once every time the state is active, the `during` actions are performed with every simulation step and are therefore dependent on the selected solver of the Simulink and Stateflow model. To design the state change logic of an automaton, Stateflow states are connected via *transitions*. Similar to states, it is possible to add guards, trigger events and actions to transitions.

Our analysis, seen in Algorithm 1, computes an over-approximation of all traces that can be produced by a given Stateflow model by performing a depth-first search on its state space. It starts with the initial node of a system and a set of empty traces $T(o)$ for all output signals of the Stateflow automaton. When processing a node, the function `nodeAnalysis` first analyzes the set of actions assigned to the current state. If any of the actions contains an assignment to o_{control}, the value of this assignment is appended to $\tau(o_{\text{control}})$. Additionally, our algorithm internally saves the origin of the assignment, i. e., the state and the action type containing the assignment. If the trace was not previously empty, the time step calculation, which we present below, is initiated. After the time interval to the previously appended value is calculated and added to the timed trace, or if the trace was previously empty, the algorithm continues by adding all outgoing transitions to a work list. For every transition on the work list, a similar analysis is performed by the function `transitionAnalysis`. As we assume arbitrary inputs to the system, we do not need to include guards into our analysis and consider them to be $true$ in all cases. Similarly, we assume that transition trigger events are always active. When taking a transition from the worklist, its condition and transition

```
Function extractTraces
    Input : output signal of interest $o_{\text{control}}$
            worklist of transitions $t_{\text{work}}$
            visited transitions $t_{\text{visited}}$
            state $v$
    // Does current state contain assignment to control signal?
    // If so, write them into $\tau(o_{\text{control}})$
    nodeAnalysis $(v, o_{\text{control}}, \tau(o_{\text{control}}))$;
    // If state does not have outgoing transitions, end recursion
    if $|t_{out}(v)| == 0$ then return;
    for $t : t_{out}(v)$ do
        // return when encountering previously visited transitions
        if $t \in t_{visited}$ then return;
        else $t_{\text{work}}$.add($t$);
    end
    $t_{\text{next}} = \text{pop}(t_{\text{work}})$;
    // Search next transition for occurrence of control signal assignment
    transitionAnalysis $(t_{\text{next}}, o_{\text{control}}, \tau(o_{\text{control}}))$;
    $v_{\text{next}} = \text{dst}(t_{\text{next}})$;
    // work on next state
    extractTraces $(o_{\text{control}}, t_{\text{work}}, t_{\text{visited}}, v_{\text{next}})$;
```

Algorithm 1: Extracting traces from Stateflow automata

actions are searched for assignments to o_{control}. If none is found, the algorithm continues by visiting the next state and performing the analyses explained above. For every outgoing transition on the initial node, the algorithm is run recursively while maintaining the worklist of unvisited transitions, possibly creating an over-approximation for states included in loops. However, this behavior is deliberate to facilitate a trace comparison across loops. Whenever a previously visited transition is encountered or when a state does not have any outgoing transitions, the base case of the recursion is reached and a set of traces for outputs of interest is returned. Internally, an ordered list of visited states is maintained to facilitate the time step calculation. This way, starting with the initial state, all states are visited and all transitions are taken while traces for all variables of interest are constructed.

Time Step Calculation As explained above, we aim at relating the timing of changes to the outputs of a Stateflow automaton to each other. We are interested in the *minimal* time interval between such changes, as these define the influence of the automaton on the surrounding model when analyzing information flow. This calculation is triggered whenever an assignment to the control signals under analysis is recognized in the actions of a state or a transition. In this case, the current state and the list of visited states, i. e., the path of the current recursion are used to perform a backwards analysis of the minimal time interval t_{diff} between the control signal assignments. The calculation is performed by iterating backwards over the path of visited states and transitions while observing the transition guards. If a guard does not use temporal operators, t_{diff} is increased by $1\,t_s$ as the reevaluation of Stateflow automata is triggered every simulation step of the surrounding Simulink model with an interval of t_s. If it does use temporal condition operators, further analyses are necessary. These operators take one of two forms: *event-based* and *absolute-time*. Event-based operators, such as `after`, `at`, `every` and `before` appear in the form of, e. g., `after(5, E)`. In this example, the operator evaluates to true after at least 5 occurrences of event `E` have taken place since activation of the Stateflow chart. Absolute-time operators work in a similar fashion. However, they do not count the number of occurrences of an event but evaluate the simulation time of the model. For our implementation, we currently only support the extraction of the event-based notation `after(n, E)` from which the number of event occurrences n is extracted and added to t_{diff}. As we are extracting a minimal time interval, we do not need to evaluate the precise timing of event occurrences. Therefore, due to the Stateflow automaton evaluation interval, n occurrences of an event take at least $n t_s$.

Running Example In the case of our motivating example, the (over-approximated) set of traces T_{SF} extracted by our algorithm is the following:

$$T(o_{\text{current\_state}}) = \left\{ \{3 \xrightarrow{\lfloor t_s \rfloor} 1 \xrightarrow{\lfloor t_s \rfloor} 3 \xrightarrow{\lfloor t_s \rfloor} 2 \xrightarrow{\lfloor t_s \rfloor} 3\}, \{3 \xrightarrow{\lfloor t_s \rfloor} 2 \xrightarrow{\lfloor t_s \rfloor} 3 \xrightarrow{\lfloor t_s \rfloor} 1 \xrightarrow{\lfloor t_s \rfloor} 3\} \right\}$$

3.5 Comparing Trace Sets

In the last step, we aim to establish a relation between the two sets of traces, i. e., the set of path conditions expressed as timed traces and the set of possible output traces of the Stateflow automaton. In order to show that information flow on the path under analysis can never occur, the relation $T_{\text{SF}} \cap T_{\text{PC}} = \emptyset$ has to hold. If it does not, then at least one of the path condition traces is contained in the set of possible output traces and information flow can therefore occur. Note that the *contains* relation does not only look for matching values in traces but also analyzes timing similarities. The comparison analyzes each pair of the cross product of both trace sets. In the first step, the values of τ_{PC} are compared in order with the values of its corresponding pair element τ_{SF}. Note that the values of τ_{PC} only need to appear in τ_{SF} in their correct order. The latter can contain additional assignments. If all values in the correct order are found, the second step identifies the timing relation of both traces. Starting with the initial value in τ_{PC} ($1^{\diamond}$) and its corresponding value in τ_{SF} (1^{*}) , all time steps in τ_{SF} are added until the next value of τ_{PC} ($2^{\mp}$) is encountered ($2^{\star}$) If this number is smaller or equal than the time step in τ_{PC}, the timing of the value update matches and the analysis continues until each time step is matched. If the values in their correct order cannot be found or their respective timing relations do not match, τ_{SF} does not contain τ_{PC}.

Running Example For our running example, our approach identifies the following sets:

$$\begin{aligned} T_{\text{SF}} &= \{\tau(o_{\text{current\_state}})_1, \tau(o_{\text{current\_state}})_2\} \\ &= \left\{ \{3 \xrightarrow{\lfloor t_s \rfloor} 1^{*} \xrightarrow{\lfloor t_s \rfloor} 3 \xrightarrow{\lfloor t_s \rfloor} 2^{\star} \xrightarrow{\lfloor t_s \rfloor} 3\}, \{3 \xrightarrow{\lfloor t_s \rfloor} 2 \xrightarrow{\lfloor t_s \rfloor} 3 \xrightarrow{\lfloor t_s \rfloor} 1 \xrightarrow{\lfloor t_s \rfloor} 3\} \right\} \\ T_{\text{PC}} = \tau_{\text{PC}} &= \left\{ \{1^{\diamond} \xrightarrow{t_s} 2^{\mp}\} \right\} \end{aligned}$$

When analyzing the value and timing relation of these sets $T_{SF} \cap T_{PC}$ using our algorithm, the result is that their intersection is empty. This is due to the fact that although the values of τ_{PC} appear in order in $\tau(o_{current\_state})_1$, their timing does not match. In the controller, a change of its output `current_state` from 1 to 2 takes place only with $t_{diff} = 2$. The violation of the security policy that is present in the timing behavior of the Simulink components of the model can therefore never occur in the combined Simulink/Stateflow model.

4 Evaluation

To evaluate our approach, we have implemented it in `Java`. It uses a modified version of the path condition extraction architecture described in [Mi15]. We use the `Java`-based constraint solving system `JaCoP` [KS13] instead of the external `Gecode` solver to increase performance and to facilitate its integration into our tool. We have extended it with a parser for Stateflow automata and the analysis algorithms described above. Our implementation is accessible via an `Eclipse` plug-in. Table 1 shows the results of our analysis of the running

Automaton Size	Time		
	Extraction T_{PC}	Extraction T_{SF}	Comparison
3	58 ms	14 ms	4 ms
10		39 ms	12 ms
100		258 ms	113 ms
1,000		2,183 ms	946 ms

Table 1: Evaluation results

example as well as of generated Stateflow automata. Note that we only adapted the size of the controller by randomly adding states and unguarded transitions while maintaining the size of the Simulink components of the system as well as the size of T_{PC}. In the first row, the results of with an automaton size of 3 states can be seen. The following rows show the performance of our algorithm for automaton sizes of 10, 100 and 1,000 as well as the algorithm run times necessary for a complete analysis. The extraction of T_{PC} includes both the extraction of sets of timed path conditions as well as the constraint solving process by `JaCoP`. As can be seen in the table, the extraction of both sets as well as the comparison in the case of our motivating example is performed in under 100 ms. For the generated Stateflow automata, the extraction and the comparison scale linearly.

5 Related Work

Extensive work has been done in the area of translating subsets of Simulink/Stateflow models into formals language with well-defined semantics, especially Lustre and the graphical modeling suite SCADE, in order to perform model checking on the translated systems [Sc04, Mi05, WHW10]. However, as these approaches rely on a translation of models into a target language using functional and timing semantics different to those of Simulink and Stateflow, properties of the original systems are lost and the timing of models cannot be analyzed precisely. Further, the translation process for industrial-sized models poses strong restrictions on their design and is therefore often not applicable [WL15].

Only few authors have addressed the problem of formalizing the behavior of Stateflow automata. Although we also do not provide a complete formal semantics, timed traces partially formalize some of the Stateflow semantics. In [HR04] and [Ha05], operational and denotational semantics for a subset of Stateflow are presented. While they represent a wide range of specific functionality of the controller semantics, they do not consider the timing of automata and the connection with surrounding Simulink models. Due to the similarities between Stateflow and Statecharts, we analyze previous formalization efforts for such models, most notably [Ha87]. However, these similarities are merely superficial, as the underlying solver for Stateflow automata works in a purely sequential fashion, and their semantic differences make an elevation of the this approach infeasible.

6 Conclusion

In this paper, we have presented an approach to extract *timed output traces* from Stateflow automata and compare them to *timed path conditions* extracted from the Simulink parts of a combined Simulink/Stateflow model. The resulting conditions can be used to, e.g., reason about the existence of paths. We have argued that for embedded software models only a combined analysis of both the behavior of the controller (in Stateflow), and the dynamic data-flow components (in Simulink) can cope with the complex timing behavior. Using the example of a shared buffer for confidential and public data using an embedded controller, we have demonstrated the usability of our approach in an *Information Flow Analysis* (IFA). Thereby, we have shown that although timed path conditions alone detect a security policy violation, it is recognized as spurious when also analyzing the controller component.

To increase the precision of our approach, we aim to extend the support for further Stateflow design constructs, such as hierarchical states, events, junctions, further state action types and temporal condition operators. We plan to incorporate these elements into a Stateflow formalization to generate timed traces for single Stateflow output signals and timing relations between multiple output signals to support non-unified control signals as explained in Section 3.3. Furthermore, we aim at supporting additional design patterns for the integration of Stateflow controllers into Simulink models, such as scheduling of individual Simulink components using an automaton or the integration of Simulink functions into controllers. Finally, an interesting artifact in the quality assurance process of embedded software is the trace of input data to the Stateflow automaton that leads to the output of a certain trace, which in turn executes a path in the Simulink components of the model. We plan to extend our algorithm to support the generation of such traces.

References

[GM82] Goguen, Joseph A; Meseguer, José: Security Policies and Security Models. In: IEEE Symposium on Security and Privacy. 1982.

[Ha87] Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: Science of Computer Programming. volume 8. Elsevier, 1987.

[Ha05] Hamon, Grégoire: A Denotational Semantics for Stateflow. In: ACM International Conference on Embedded Software. ACM, 2005.

[HR04] Hamon, Grégoire; Rushby, John: An Operational Semantics for Stateflow. In: Fundamental Approaches to Software Engineering. Springer, 2004.

[HSS08] Hammer, Christian; Schaade, Rüdiger; Snelting, Gregor: Static Path Conditions for Java. In: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. ACM, 2008.

[Ki76] King, James C: Symbolic Execution and Program Testing. Communications of the ACM, 1976.

[KS13] Kuchcinski, Krzysztof; Szymanek, Radoslaw: Jacop - Java Constraint Programming Solver. In: International Conference on Principles and Practice of Constraint Programming. unpublished, 2013.

[Ma14] MathWorks Stateflow. www.mathworks.com/products/stateflow/, Accessed: 06/2016.

[Ma15] MathWorks MATLAB/Simulink. www.mathworks.com/products/simulink, Accessed: 06/2016.

[ME16] Model Engineering Solutions GmbH. www.model-engineers.com, Accessed: 06/2016.

[Mi05] Miller, Steven; Anderson, Elise; Wagner, Lucas; Whalen, Michael; Heimdahl, Matts: Formal verification of flight critical software. In: Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit. 2005.

[Mi15] Mikulcak, Marcus; Herber, Paula; Göthel, Thomas; Glesner, Sabine: Timed Path Conditions in MATLAB/Simulink. In: International Embedded Systems Symposium. Springer, 2015.

[Sc04] Scaife, Norman; Sofronis, Christos; Caspi, Paul; Tripakis, Stavros; Maraninchi, Florence: Defining and Translating a Safe Subset of Simulink/Stateflow into Lustre. In: International Conference on Embedded Software. ACM, 2004.

[SM03] Sabelfeld, Andrei; Myers, Andrew C: Language-based Information-Flow Security. IEEE Journal on Selected Areas in Communications, 2003.

[WHW10] Whalen, Michael W.; Hardin, David; Wagner, Lucas G.: Model Checking Information Flow. In: Design and Verification of Microprocessor Systems for High-Assurance Applications. Springer US, 2010.

[WL15] Walde, Georg; Luckner, Robert: Automatic Translation of Complex Flight Control Systems from Simulink/Stateflow to SCADE - An Experience Report. Technical report, Deutsches Zentrum für Luft- und Raumfahrt, 2015.