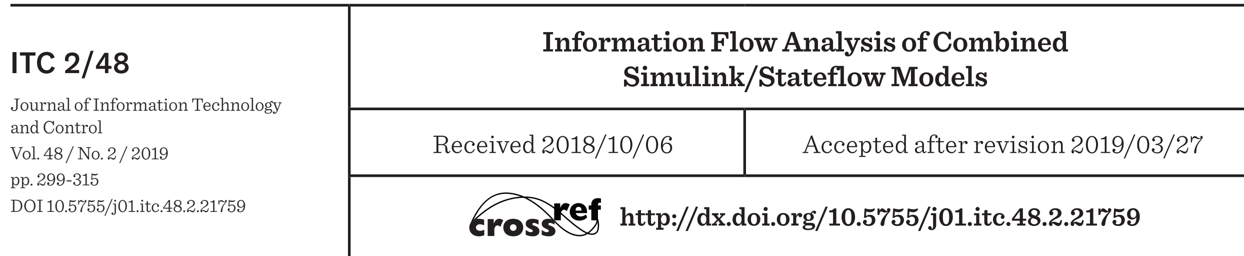

ITC 2/48
Journal of Information Technology and Control
Vol. 48 / No. 2 / 2019
pp. 299-315
DOI 10.5755/j01.itc.48.2.21759

Information Flow Analysis of Combined Simulink/Stateflow Models

Received 2018/10/06 | Accepted after revision 2019/03/27

http://dx.doi.org/10.5755/j01.itc.48.2.21759

Information Flow Analysis of Combined Simulink/Stateflow Models

Marcus Mikulcak, Thomas Göthel, Sabine Glesner

Technische Universität Berlin, Software and Embedded Systems Engineering Group Ernst-Reuter-Platz 7, 10587 Berlin, Germany, e-mail: {marcus.mikulcak, thomas.goethel, sabine.glesner}@tu-berlin.de

Paula Herber

University of Münster, AG Embedded Systems Einsteinstraße 62, 48149 Münster, Germany e-mail: paula.herber@uni-muenster.de

Corresponding author: marcus.mikulcak@tu-berlin.de

Simulink and its state machine design toolbox Stateflow are widely-used industrial tools for the development of complex embedded systems. Due to the strongly differing execution semantics of Simulink and Stateflow, the analysis of combined models poses a difficult challenge, especially when considering their timing behavior. In this paper, we present a novel approach to relate the semantics of both the dynamic Simulink components and the Stateflow controller and use it to perform an information flow analysis on combined models. The key idea of our approach is that we analyze the information flow in a given model by computing an over-approximation of the control flow through the Simulink components, and deduce whether all control flow conditions combined permit information to flow on a given path or not. The main contributions of our control flow analysis approach are: (1) we identify *timed path conditions* which capture the conditions for time-dependent information flow on paths of interest for (discrete) Simulink components, and translate them into a UPPAAL timed automata representation, (2) we translate the Stateflow components to UPPAAL timed automata, and (3) we perform model checking on the translated set of automata in order to analyze the existence of paths in the combined model. With our approach, we safely rule out the existence of information flow on specific paths through a model, which enables us to reason about non-interference between model parts and the compliance with security policies. Furthermore, our approach presents a starting point to generate feasible, efficient test cases and to perform compositional verification. We demonstrate the applicability of our approach using two versions of a complex case study from the automotive domain consisting of multiple safety-critical components communicating over a shared bus system. For this example, an approach based on timed path conditions alone is sound but highly imprecise compared to our combined approach.

KEYWORDS: Embedded Systems, Information Flow Analysis, UPPAAL, MATLAB, Simulink, Stateflow, Safety.

1. Introduction

In the area of safety-critical embedded software, such as in the automotive and aerospace domains, pro-gram- ming errors can lead to disastrous and 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 Simulink [28] 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 in conjunction with these dynamical systems, Stateflow [30], an extension to Simulink, is widely used in industrial design processes. Simulink and Stateflow are very well-suited to grasp the structure of a design on high abstraction levels and to visualize its behavior by simulation.

However, due to the complexity and the dynamic, time-dependent character of the developed models, the analysis of a given combined model is a difficult challenge, in particular if timing aspects are considered. At the same time, combining knowledge about the existence of paths, the conditions under which they are executed, and how an embedded Stateflow controller influences their behavior is a hitherto unsolved problem. This is due to the strongly heterogeneous semantics of Simulink and Stateflow.

In this paper, we present an approach for an information flow analysis (IFA) of combined discrete Simulink/Stateflow models. Our approach is threefold: First, we extract timed path conditions from discretely- timed signal-flow components developed in Simulink and prepare them for analysis by converting them into timed automata. Second, we generate UPPAAL timed automata from the embedded Stateflow controllers. Third, we use model checking to analyze whether the timed path conditions that are extracted from Simulink components are satisfiable by the timed automata representation of Stateflow components. If the conditions are satisfiable, the conditionally executed paths under analysis potentially exist in the combined model, i. e., information flow is possible. If not, they are identified as infeasible and will never be executed in the model, i. e., the absence of information flow is guaranteed. All three steps are performed fully automatically, including the generation of the verification goals for model checking.

If our analysis is applied to all possible paths of a given combined model, we can identify non-interference between model parts and, thus, reason about compliance with security policies. For example, we can verify *integrity* by checking that no information flow is possible from a non-critical to a critical component. The relevance of such integrity properties was demonstrated, for example, by the Jeep hack in [20], where the attackers gained control over the (safety-critical) speed control and braking system of a Jeep Cherokee through a vulnerability of the (non-critical) infotainment system.

We demonstrate the applicability of our approach by, among others, checking the absence of information flow from a (non-critical) odometer to a (safety-critical) braking system in multiple versions of a case study provided by our industrial partners from the automotive domain.

Note that this paper is an extended version of our work published in [17]. In this paper, we provide (1) a more exhaustive discussion of preliminaries with a special focus in information flow analysis and MATLAB Stateflow in Section 2; (2) a detailed explanation of our previously published approach to extract timed path conditions from the Simulink components of a combined Simulink/Stateflow model in Section 4; (3) an extended case study in Section 7.

The rest of this paper is structured as follows: In Section 2, we briefly introduce the necessary preliminaries. In Section 3, we present the main contribution of this paper, namely an approach for the analysis of information flow through combined Simulink/Stateflow models. In Sections 4 to 6, we give a detailed description of each step of our approach. We present our evaluation and results in Section 7. Then, we discuss related work in Section 8 and conclude in Section 9.

2. Preliminaries

In this section, we describe the basic concepts and tools employed by our approach.

2.1. Information Flow Analysis

The protection of confidentiality and integrity of information inside a software system is an 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 [5]. Such policies, based on the assignment of security levels to data elements, describe rules between which levels information flow is allowed or forbidden [26]. 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.

2.2. Path Conditions

In general, *path conditions* [12] describe sufficient conditions for information paths through code or models to be executed. In [6, 7], path conditions are used to capture all paths where information might flow from a source to a target. In contrast to static analyses, which consider all syntactically possible dependencies, path conditions take data and control flow conditions into account. Thus, they exclude, for example, information flow that depends on disjoint control flow conditions. A path condition-based analysis is therefore more precise than classical static analyses.

2.3. Simulink

Simulink [28] is an add-on to the MATLAB IDE by MathWorks that enables graphical modeling and simulation of reactive systems. In its signal-flow oriented notation [13, 23], 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. In such models, each block is interpreted as producing a piecewise-constant signal over the simulation time scale, which forms the basis for an automatic translation of the model functionality and timing behavior to code [31, 14].

2.4. Stateflow

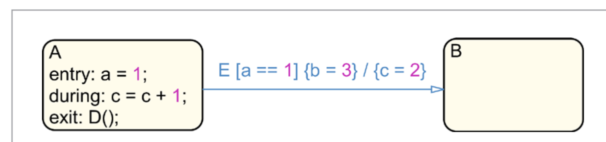
Stateflow [30] is a further add-on to the MATLAB IDE, specifically to Simulink, and gives the designer the possibility to integrate decision logic based on state machines and flow charts into a Simulink model. Stateflow makes use of complex modeling styles incorporating multiple states, event and transition types as well as an execution semantics not only dependent on the structure and annotations of the model but also on its layout. Internally, the execution of Stateflow charts is controlled via an *event queue* into which all implicit events, such as updates to input signals and transitions as well as all explicit events are ordered and evaluated in a first in, first out (FIFO) fashion, i. e., the execution semantics is purely sequential. To give an impression of the Stateflow semantics, we briefly summarize its main building blocks, i. e., states, events, and transitions. Stateflow additionally supports junctions as well as temporal logic operators to model timed conditions.

2.4.1. States

States form the basic building block of the controller logic implemented in Stateflow. An example is shown in Figure 1. If the execution enters a state, a set of *actions* modeled by the designer takes place, such as the modification of output signals of the automaton or the triggering of *events*. The *action type*, `entry`, `during` or `exit`, determines *when* these actions are performed. Depending on the type, 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 simula-

Figure 1

Stateflow example



tion step and are therefore dependent on the selected solver of the Simulink and Stateflow model. As shown in Figure 1, actions can trigger *events*. To manage the complexity of Stateflow automata, it is possible to model hierarchical states by using *superstates* and *substates*. If a superstate is triggered, its substates are either executed in an exclusive or a parallel fashion, depending on the modeling style chosen by the developer. While in an exclusive composition, only one of the mutually exclusive substates can be active at a time, multiple parallel states can be active simultaneously in a parallel composition. However, as the execution of parallel states is sequential during simulation, an ordering is imposed either explicitly via annotations made by the designer or implicitly via the relative location of the states.

2.4.2. Events

Events in Stateflow are a form of trigger mechanism for the execution of states and transitions. As seen in Figure 1, the `exit` action of state `A` triggers the *explicit* event `D()`. Whenever an event is triggered, it is broadcast to the parallel states of the current Stateflow chart.

2.4.3. Transitions

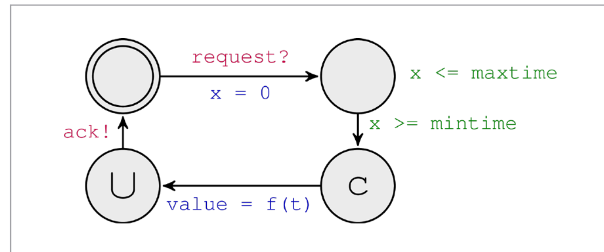
To design the state change logic of a controller, Stateflow states are connected via *transitions*. Similar to states, it is possible to add guards, trigger events, and actions to transitions. Figure 1 shows an example transition containing all three mentioned semantical elements. Event `E` triggers the evaluation of the guard condition `a == 1`. As soon as this condition evaluates to *true*, the corresponding *guard action* `b = 3` is executed. Finally, when the transition is taken, the *transition action* `c = 2` is executed and state `B` is marked active.

2.5. UPPAAL Timed Automata

Timed automata (TA) [1] are a timed extension of the classical finite state automata. A notion of time is introduced by clock variables, which are used in clock constraints to model time-dependent behavior. Systems comprising multiple concurrent processes are modeled by networks of TA, which are executed with interleaving semantics and synchronize on channels.

UPPAAL [3, 4] is a tool suite for modeling, simulation, animation, and verification of networks of timed automata. The UPPAAL modeling language extends timed automata by bounded integer variables, binary and broadcast channels, and urgent and committed locations. A small example UPPAAL timed automaton is shown in Figure 2. The initial location is denoted by ◎. The label `request?` denotes receiving on the channel `request`, while `ack!` denotes sending on channel `ack`. The clock variable `x` is first set to 0 and then used in two clock conditions: the *invariant* `x <= maxtime` denotes that the corresponding location must be left before `x` becomes greater than maxtime, and the *guard* `x >= mintime` enables the corresponding edge if `x` is greater or equal `mintime`. The symbol Ⓤ depicts an urgent location and the symbol Ⓒ a committed location. Urgent and committed locations are used to model locations where no time may pass. Leaving a committed location has priority over leaving non-committed locations.

Figure 2
UPPAAL example



The UPPAAL model checker enables fully-automatic verification of (unnested) Computation Tree Logic (CTL) formulae on a given network of timed automata.

3. Information Flow Analysis of Simulink/Stateflow Models

The heterogeneous nature of software 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 evalua-

tion rules that determine which transition fires in each step, whereby a step is made whenever one of the input signals is reevaluated, i. e., every simulation time step t_s. The main idea of our approach for the analysis of the information flow in combined, discrete Simulink/Stateflow models is to relate a Stateflow controller with its surrounding Simulink model using timed automata in order to enable model checking. For the analysis of information flow in pure Simulink components, we make use of our approach previously presented in [18]. There, we compute *timed path conditions* for a given Simulink model by performing a backwards analysis through the model. The timed path conditions extracted using our approach describe sufficient conditions for the execution of a given path, i. e., they provide a sound over-approximation of the possible information flow. For Stateflow, we utilize an approach that translates Stateflow components to a system of UPPAAL timed automata [11, 34]. With that, the semantics of a Stateflow component is precisely defined. We make use of this approach to gain a formally well-defined representation of the Stateflow components in a combined Simulink/Stateflow model, and to gain access to the UPPAAL tool suite, including the UPPAAL model checker.

To relate the timed path conditions resulting from [18] for Simulink components with the UPPAAL timed automata representation of Stateflow components, we assume that a Stateflow controller is embedded into a Simulink model and has an effect on some of its components by controlling the execution of paths through the surrounding Simulink components. Our approach to analyze the information flow in combined Simulink/Stateflow models, shown in Figure 4, is threefold:

1. We utilize our algorithm shown in [18, 19] to extract timed path conditions for all paths between a set of model elements of interest from the Simulink model (see step (1.a) in Figure 4). Along these paths, the conditions for information flow as well as their timing are gathered and expressed as sets of timed path conditions C_{TP}. To make this representation compatible with the UPPAAL timed automata representation of the Stateflow semantics presented in [19], we propose a timed automata representation of these timed path conditions and generate one automaton for each set in C_{TP} (1.b). As we extract timed path conditions for *all* paths between model elements of interest and, on these paths, extract all control flow conditions, we achieve a sound over-approximation of the possible information flow through the Simulink model.
2. We adapt the method presented in [11, 34] to translate embedded Stateflow components to a system of UPPAAL timed automata (2.a). We are confident that their translation is sound, as it provides a direct mapping of each Stateflow process into a semantically equivalent timed automata representation, and explicitly models the execution semantics of Stateflow, including the event queue. We extend the resulting system with an automaton that provides arbitrary input signals (2.b). This enables a sound and comprehensive analysis of the behavior of the Stateflow controller, as we simulate the complete environment, i. e., all possible combinations of input signals to the controller.
3. We use the information gathered in the previous steps to automatically generate reachability properties for the UPPAAL model checker (3.a) and start the checking process (3.b). In this final step, we effectively utilize model checking to analyze whether the timed path conditions derived from Simulink can be satisfied by the Stateflow controller, i. e., if one or multiple of the timed path conditions expressed in C_{TP} hold on the translated model of the Stateflow controller SF_M. If they do not hold, then we have safely shown that information flow over the paths under analysis is impossible as the path is never executed in the Simulink models and that the property of non-interference holds.

Note that a Stateflow state machine is connected to the surrounding model Simulink model via signals S_C that can be used as variables inside guards. Variables modified in state or transition actions inside Stateflow state machines form their output signals S and are routed to the Simulink model. There, they act as control signals that impose conditions on information flow paths from the inputs I to the outputs O. The evaluation of a Stateflow automaton is performed whenever one of the input signals to the automaton is reevaluated by the solver. Then, its state is reevaluated and a one of the possible transitions 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

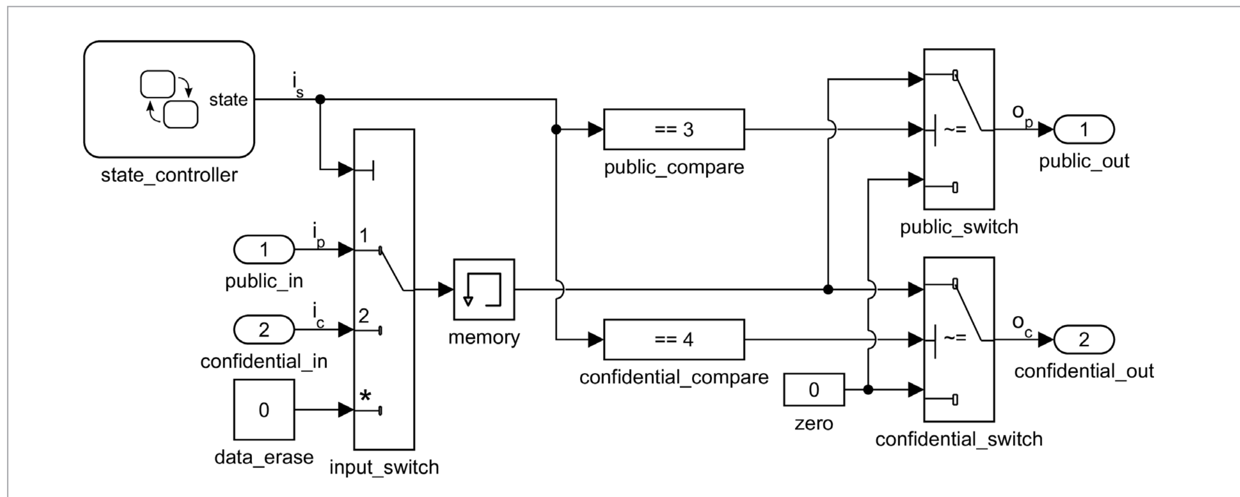
size t_s. This relation between the discretely-timed solver of the Simulink model and the evaluation of the Stateflow automaton makes it possible to relate both semantics.

In the remainder of this section, we first introduce a motivating example for our approach in Section 3.1. Then, we introduce the assumptions we impose on the models that our approach is able to analyze in Section 3.2. In Section 4, we provide an overview of our approach to extract timed path conditions from the signal-flow oriented Simulink components of combined Simulink/Stateflow models. Subsequently, we present our approach to prepare these extracted sets of timed path conditions for model checking by converting them into UPPAAL timed automata in Section 4.1. We then present the generation of networks of UPPAAL timed automata from the Stateflow components of the models as well as our generalization to support arbitrary inputs during the model checking process in Section 5. Finally, we present our automatic generation of properties using the previously extracted information and the initiation of the model checking process by our algorithm as well as the evaluation of its results in Section 6.

3.1. Motivating Example

To illustrate our approach, we use a shared communication infrastructure , such as commonly found in the design of automotive software, as a running example. Figure 3 shows the corresponding Simulink model. It uses a `Memory` block as internal buffer and switches to route the incoming and outgoing data according to their source and target, respectively. Information of two different security levels (from the public input i_p and confidential input i_c) is fed into the shared buffer. According to the current operation mode set by a controller implemented in Stateflow, confidential or public information is saved in the buffer and passed to the corresponding output. Although confidential and public data share the same memory block as buffer, the routing conditions are intended to ensure that confidential input data can never flow to the public output. To this end, the operation mode defines which input should be routed to the output. The designer did, however, not take the timing behavior of the Memory block into account. When examining the timing, we discover that if the operation mode switches from confidential to public, the buffer content that is passed to the outputs still holds the confidential data for one time unit, i. e., the confidential contents are sent to the public output. The timed path conditions for the Simulink part of this example correctly show that information flow is indeed possible from i_c to o_p, as shown in [18]. However, as the Stateflow controller is responsible for the modification of the system state i_s, an analysis of its behavior in conjunction with the timed path conditions is necessary to evaluate whether the *combined*

Figure 3
Running example



Simulink/Stateflow model suffers from the same security policy violation.

3.2. Assumptions

Our information flow analysis approach supports discrete Simulink/Stateflow models that satisfy the following assumptions:

1 Only *time-discrete, fixed-step* solvers are used [18],

2 a uniform sample time is used,

3 no algebraic loops are used,

4 control signals do not depend on any feedback loops.

All discrete Simulink/Stateflow models that satisfy these assumptions can be safely analyzed using our approach. Note that as we analyze the control flow of Simulink subsystems, we do not impose any restrictions on the data paths, i. e., complex modeling elements like integrators and transfer functions, together with arbitrary feedback loops, may be used on the data path. Note also that the existing translation mechanism for Stateflow described in [11, 34] does not impose any additional restrictions, as a full translation of all Stateflow features is provided.

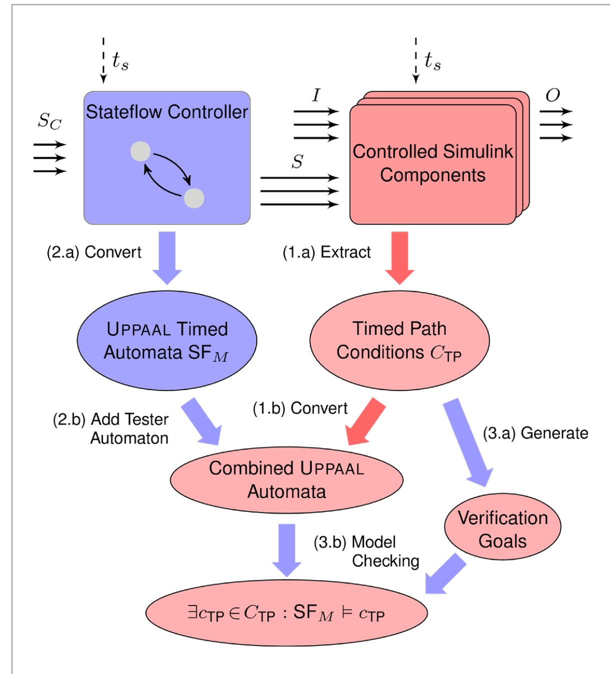
Our restriction to purely discrete models is acceptable, as we target safety-critical embedded software systems, which typically do not contain any continuous components. The restriction to a uniform sample time is mainly for simplicity of presentation and could be relaxed. Support for algebraic loops would require to incorporate a fixed-point analysis into our approach and as they are not present in the case studies provided by our industrial partners, we left this open for future work. The restriction on control signals (4) is the most serious restriction and we hope to relax it in the near future by providing a more sophisticated analysis of the control flow in a given Simulink model. However, this assumption was also met by the case studies provided by our industrial partners.

4. Extracting Timed Path Conditions from Simulink Components

In this section, we provide an overview of the first steps (see (1.a) and (1.b) of Figure 4) of our information flow analysis approach, in which we use the algorithm we have previously published in [18] to generate timed path conditions from the dynamic signal-flow (i. e., Simulink) components of the model, and translate them into a UPPAAL timed automaton representation.

Figure 4

Overall approach



For the computation of timed path conditions from the Simulink components of a combined system, we use a two-step approach: (1) Statically identify all paths in a given Simulink model and collect all path conditions on each path. (2) For each path, propagate all local control flow conditions backwards through the model in order to compute timed path conditions that solely depend on input variables.

Using this approach, it is possible to express conditions on paths through Simulink models containing time- dependent elements. In general, for a single path, these conditions take the following form shown in Equation (1):

$$c_{TP}(i \rightarrow o) = \bigwedge_{l=0}^{T} \bigwedge_{j=1}^{n} p(s_j)^{t-l \cdot t_s} \tag{1}$$

with $\{i \in I, o \in O, s_j \in S\}$.

There, T denotes the depth of the path under analysis in *time slices* [18], $s_1,...,s_n$ denote all control signals,and t_s denotes the simulation step size. Each path condition $p(s_j)$ corresponds to a conjunction of the atomic control flow conditions that are collected for each control signal in each time slice $t-l\cdot t_s$ on the given path from i to o during the backward analysis. The timed path condition $c_{\mathrm{TP}}(i\to o)$ defines a sufficient condition for information to flow through the path starting at data input i and leading to the data output o.

From our running example shown in fig:running_example, we are able to extract the following conditions for information to flow from the confidential data input i_c to the public data output o_p:

$$c_{\mathrm{TP}}(i_c\to o_p)=(i_s==4)^{t}\wedge(i_s==1)^{t-t_s}. \quad (2)$$

4.1. From Timed Path Conditions to Timed Automata

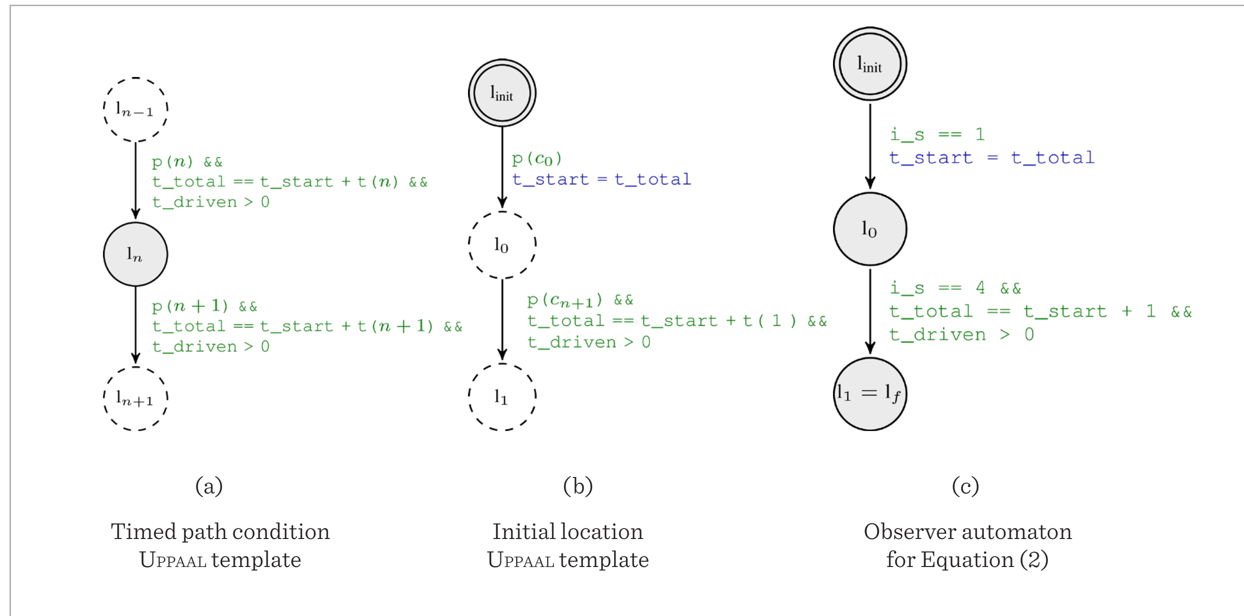
In this step, we present our approach to create compatibility between the extracted sets of timed path conditions and the Stateflow controllers translated into networks of UPPAAL timed automata. Specifically, our main contribution in this step is the creation of *observer automata* [22] for the information flow paths under analysis.

The main purpose of our translation from timed path conditions to timed automata is to make our representation of the timed paths in a Simulink model compatible with the network of UPPAAL timed automata generated from the Stateflow components. The key idea of our translation is to encode the timed path condition into an observer automaton, which observes a timed automata model of a Stateflow component and checks whether the timed path conditions derived from a Simulink model can be satisfied by the Stateflow component. Note that the timed path condition is satisfiable if the final location of this observer automaton is reachable, which can conveniently be checked using the UPPAAL model checker.

The translation of Stateflow automata to UPPAAL presented in [11, 34] does not utilize clocks. We are therefore forced to rely on a different mechanism to synchronize the automaton to be created with the main system of automata created from the Stateflow model. As shown in the following, we make use of two variables introduced by the authors of [11, 34] that emulate the simulation time found in Simulink/Stateflow, both globally (`t_total`) and inside states (`t_driven`).

The input to our generation algorithm is a list of timed path conditions, i. e. , the extracted set of timed path conditions c_{TP} for a path under analysis sorted by time slices, as shown in Equation (1). The maximum length of this list is the time slice depth of the current path T. We perform multiple iterations over the list of timed path conditions we receive as an input. First, we add an initial state and one location for every entry in the list. Second, we create all forward transitions between the created locations. Every transition is guarded by the correct valuation of the next entry in the list. Additionally, whenever the first guard is evaluated to *true*, an internal variable t_start is set to the current value of the global time variable t_total and is used in all forward transitions to evaluate whether the timing requirement between input list conditions is met. We then add backward transitions from each state back to the initial state. These transitions are taken whenever the timing or control signal requirement to proceed to the next entry in the set is not met. Further, we add backward transitions from every state back to the state corresponding to the first entry in the list. These transitions are taken whenever the timing and data requirements are not fulfilled to enter the next state in the list but match its first entry. In the next step, we add all required self-loops to the automaton. These self-loops are taken whenever the simulation time has not progressed far enough for the next forward transition to be taken or whenever the backward transition guard cannot be evaluated to *true*. Finally, the automaton will be added to the translated Stateflow system. We have illustrated our generation algorithm using UPPAAL timed automata templates shown in Figures 5a and 5b. There, dashed states denote placeholders for subsequent iterations of the algorithm during generation and the functions $p(n)$ and $t(n)$ extract the n th atomic condition and the n th timing requirement from the input list. Revisiting our running example, the automaton created from the timed path condition $c_{\mathrm{TP}}(i_c\to o_p)$, shown in Figure 5c. There, we initially wait until $i_s==1$. Whenever this condition is satisfied, we check whether $i_s==4$ in the next time slice. If this happens, we reach the final location l_f of our observer automaton, which means that the timed path condition can actually be

Figure 5
Observer automata



(a)
Timed path condition
UPPAAL template

(b)
Initial location
UPPAAL template

(c)
Observer automaton
for Equation (2)

satisfied. If $i_s == 1$ is satisfied but $i_s == 4$ is not satisfied in the next step, we reset the observer automaton.

By checking initially whether the timed path condition that refers to the earliest time slice is satisfied, and then subsequently checking whether all timed path conditions referring to the subsequent time steps are satisfied, we ensure that we detect all sequences of outputs of the Stateflow controller which might satisfy the overall timed path condition, i. e., our approach is sound.

5. Stateflow Controllers as Timed Automata

In this section, we present our adaption of an existing mechanism to translate Stateflow controllers into networks of UPPAAL timed automata. The translation makes it possible to elevate the analysis of the controller behavior into a formally well-defined representation and enables the usage of the UPPAAL model checker to verify properties on the controller behavior. Specifically, we aim at combining the generated observer automata presented in the previous section with the networks of automata created from the controller in order to verify whether the extracted timed path conditions are satisfiable. Our main contribution in this section is the generalization of the execution behavior to allow model checking of the controller for arbitrary inputs. We utilize a technique developed in [34] that converts Stateflow automata to UPPAAL timed automata. In the following, we briefly introduce the concepts behind their work and present our adaptations and extensions.

5.1. State Transformation

Every state of the Stateflow automaton is converted into four separate automata: (1) a Controller Automaton, handling activation and deactivation of child states and activation of time-dependent events, such as `after(n, sec)`, (2) a Controller Action Automaton, responsible for performing `Entry`, `During`, `Exit` actions for the state, (3) a Condition Automaton, evaluating conditions and performing condition actions of outgoing transitions, and finally (4) a Common Automaton that reacts on conditions generated by the Condition Automaton, performs the transition actions on outgoing transitions and activates the next state.

5.2. Transition Transformation

For each state, all outgoing transitions are saved in an array sorted by the implicit and explicit transition priority. From this array, the Condition Automaton selects the next transition to be evaluated and taken.

5.3. Time

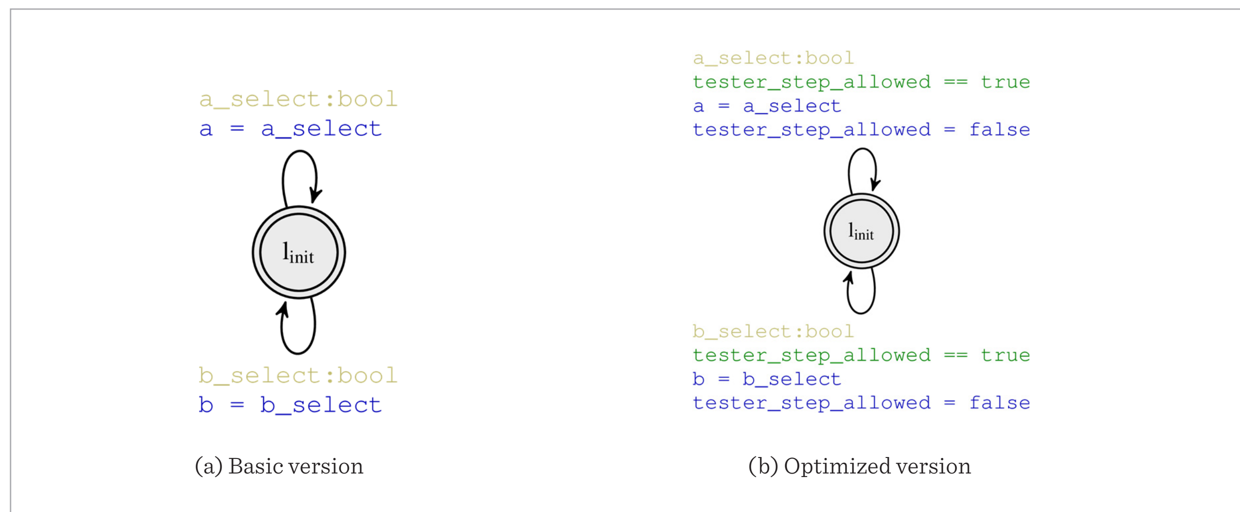
In Stateflow, time can be modeled absolute or event-based. As an absolute value, time is used by, e.g. `after`, `before`, `at` and `every` in conjunction with a time unit, e. g., `after(2, sec)`. This absolute form is based on the simulation time of the surrounding Simulink model and can be used to synchronize the timing behavior of the Stateflow controller with that of its environment. The second form utilizes the same keywords but makes use of *events* inside the Stateflow controller, e. g., `every(3, e)`, which executes the associated action with every third occurrence of the event `e`. To emulate this behavior in UPPAAL, the authors have implemented a *virtual event stack* as a structured array in UPPAAL. To model the simulation time of the Stateflow controller, the authors do not rely on *clocks*, but implemented two distinct integer variables: `t_total` and `t_driven`. The former implements the simulation time of the overall system, while the latter implements simulation time passed inside the current state.

5.4. Generalization to Arbitrary Inputs

The translation from Stateflow to UPPAAL does not include the possibility to simulate the environment of the Stateflow controller. This means that if, for example, transitions are guarded by signals driven by the Simulink model the controller is embedded into, their value will be held as *uninitialized* and will not be part of the verification of the controller. This makes it impossible to utilize the translation in its original form, as the signals driving the Stateflow controller are an integral part of its functionality. We therefore extended the translation by a *generic tester automaton* [25].

A *generic tester automaton*, in its most basic form, can be seen in Figure 6a. There, a timed automaton with a single state and two edges is shown. On each edge, a `Boolean` signal, `a` or `b`, is set non-deterministically to `true` or `false` by using the `select` syntax built into UPPAAL. This makes it possible to simulate every combination of both signals in every step of the model checking process as the tester automaton runs concurrently to all other automata. Our extension to the original translation analyzes all input signals to the Stateflow controller as well as their data types. Using this information, a tester automaton for the controller under analysis is constructed to simulate arbitrary inputs to the controller, thereby acting as the non-deterministic environment of the controller.

Figure 6
Generic tester automata



(a) Basic version (b) Optimized version

Additionally, we have optimized the behavior of the tester automata in combination with the networks of UPPAAL timed automata. In the basic implementation, the tester automaton selects a new value at every point in the model checking process. However, due to the complex design of the translation from Stateflow to UPPAAL, only a small number of transitions directly relate to transitions taken in the Stateflow controller. We therefore have added a global `Boolean` variable `tester_step_allowed`, seen in Figure 6b. It is added to the guards of the edges and consumed as soon as the edge is taken. This flag is raised only when input values to the controller are allowed to change, i. e., with every (emulated) simulationstep.

6. Information Flow Analysis Using UPPAAL

In the final step of our algorithm, we combine the timed automata generated from the sets of timed path conditions in the first step with the translated networks of timed automata in the previous step. Using information from the first steps, we generate verification goals for the UPPAAL model checker in order to verify whether the sets of timed path conditions are satisfiable by the translated Stateflowcontroller.

6.1. Generation of Verification Goals

In order for the UPPAAL verifier to check whether the timed path conditions can be satisfied by the generated system of timed automata, we create appropriate queries. Due to the design of the timed path condition observer automata, we only need to create a single CTL query for each automaton to check reachability of its final location. As seen in Section 4.1, we save information about the final location, therefore, the necessary queries take the form of *exists eventually* statements for the existence of a path to the final location l_f for every automaton. Hence, the queries have the form EF (l_f) and are generated automatically.

6.2. Model Checking Using UPPAAL

In the final step of our analysis process, the generated queries are verified on the translated UPPAAL timed automata model which has been extended with one automaton for each set of timed path conditions and a tester automaton to simulate arbitrary inputs. For each entry in the query file, UPPAAL verifies the stated property on the complete system, i. e., we check whether the timed path conditions can be satisfied by the Stateflow automaton. If so, information is potentially able to flow along the path under analysis. If not, the path does not exist and information flow is therefore shown to be impossible.

7. Evaluation

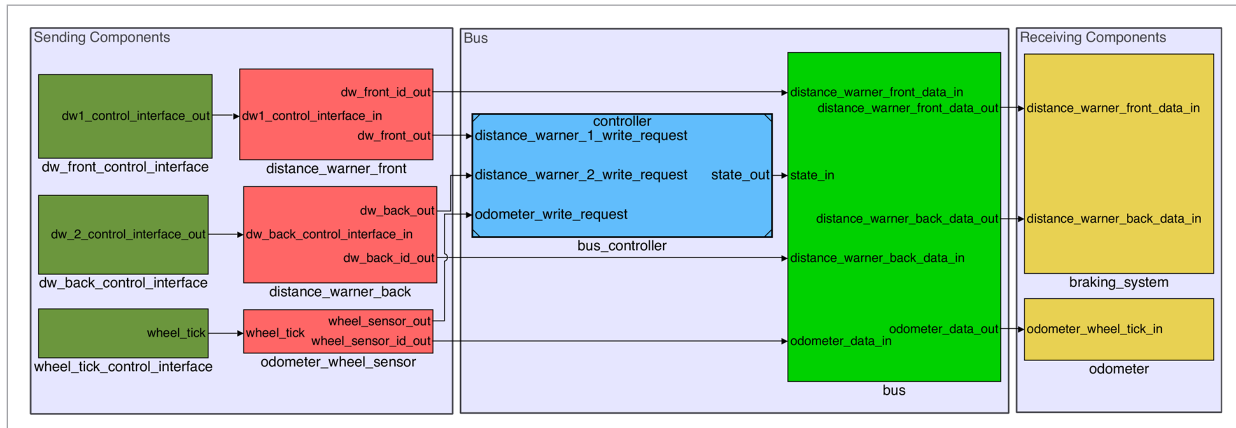
To evaluate our approach, we have implemented it as an `Eclipse` plug-in in `Java`. It utilizes our path condition extraction algorithm for Simulink [18] and a Stateflow to UPPAAL conversion [11, 34].

7.1. Case Study

To show the practical applicability of our approach, we have used an industrial case study from the automotive domain. Its core is a communication infrastructure over which two distance warners, supplied by our industrial partner Assystem GmbH [2], and a non-critical component, an odometer, supplied by Model Engineering Solutions GmbH [15], send and receive data. Figure 7 shows the structure of our case study in Simulink/Stateflow. The distance warners, situated at the front and at the back of the car, send their analysis results, i. e., proximity alerts, to the receiving component, an automated braking system. The odometer receives data from sensors on the axes of the car. The distance warners together with the automated braking system perform inherently safety-critical functions, especially under timing aspects, as dropped or delayed warning signals either to the driver or an automated braking system while traveling at high speeds could cause serious accidents. The most important property of the design from a security perspective is that the design has to guarantee that the braking system only receives messages from the distance warners, i. e., that information flow from the non-critical odometer to the critical braking system is prohibited and, consequently, *integrity* is ensured. An additional interesting property is that no information flows from the distance warners to the odometer, as this may indicate that they are not properly received by the braking system. The overall model consists of 905 blocks and

Figure 7

Shared communication infrastructure in a car



multiple layers of subsystems, making it comparable, in size as well as complexity, to models with similar functionality used by our industrial partners in the automotive domain. Our running example, shown in Section 3.1, shows a simplified version of the routing mechanism and controller utilized in our case study. The main challenge for the analysis of this case study is that the correct routing inherently depends on the timing of the control flow.

Note that we utilize two versions of the bus and the arbitration controller: In the original version, the shared infrastructure can only be used by a single sending component at a time, while in the extended second version, which we have implemented for demonstration reasons, multiple senders can utilize the shared bus at the same time. In both versions, all three sending components seen on the left in Figure 7 utilize the bus to send their unique `id` to the receiving components on the right. Inside the channel, a system of switches reacts to the state currently set by the controller and routes the data to and from the communication channel accordingly. While in the first version of the case study, the switches that control inputs and outputs to the bus are controlled by the same control signal, they are controlled by two control signals in the second version.

In the following, we present the analysis results for both versions of our case study, i. e., the timed path conditions extracted from the Simulink components of the bus and the controller translated to UPPAAL as well as analysis results and computation times.

7.2. Results

Using our approach, a designer is able to analyze arbitrary paths through the model. For this section, we chose to analyze two paths on which information flow can lead to critical errors in both versions of our case study. To this end, we verify the *integrity* of the automated braking system by analyzing whether information from the odometer can ever reach either input of automated braking system (which should not happen). Additionally, we check whether data sent from the front and back distance warners through the bus is able to reach the odometer display (which also should not happen). Note that the former property is crucial to verify that there is no information flow from the non-critical odometer to the safety-critical braking system. The latter property is important as information flow between the distance warner to the odometer may lead to a dropped proximity warning, which in turn may lead to a failure to brake by the automated braking system. In the following, we present the results of the individual steps taken by our algorithm in order to prove non-interference on the selected paths.

7.2.1. Paths Under Analysis

As explained above, we aim at analyzing potentially critical information flow from the odometer sensors to the braking system as well as from both distance warners to the odometer display. We denote the paths as follows, where *I* refers to the first version of our case study, and *II* to the second version

$$P_1^I = P_1^{II} = P(\text{odometer_sensor_out} \rightarrow \text{bs_dw_front_data_in})$$
$$P_2^I = P_2^{II} = P(\text{odometer_sensor_out} \rightarrow \text{bs_dw_back_data_in})$$
$$P_3^I = P_3^{II} = P(\text{dw_front_out} \rightarrow \text{odometer_wheel_tick_in})$$
$$P_4^I = P_4^{II} = P(\text{dw_back_out} \rightarrow \text{odometer_wheel_tick_in}).$$

In the next step of our algorithm, the paths are analyzed and sets of timed path conditions are extracted.

7.2.2. Extracted Timed Path Conditions

The sets of timed path conditions extracted for each path are shown in the following:

$$c_{\text{TP}}(P_1^I) = \{(s_{\text{state}} == 1)^t, (s_{\text{state}} == 3)^{t-d_1 \cdot t_s}\}$$
$$c_{\text{TP}}(P_2^I) = \{(s_{\text{state}} == 2)^t, (s_{\text{state}} == 3)^{t-d_1 \cdot t_s}\}$$
$$c_{\text{TP}}(P_3^I) = \{(s_{\text{state}} == 3)^t, (s_{\text{state}} == 1)^{t-d_1 \cdot t_s}\}$$
$$c_{\text{TP}}(P_4^I) = \{(s_{\text{state}} == 3)^t, (s_{\text{state}} == 2)^{t-d_1 \cdot t_s}\}$$
$$c_{\text{TP}}(P_1^{II}) = \{(s_{\text{input_state}} == 1)^t, (s_{\text{output_state}} == 3)^{t-d_2 \cdot t_s}\}$$
$$c_{\text{TP}}(P_2^{II}) = \{(s_{\text{input_state}} == 2)^t, (s_{\text{output_state}} == 3)^{t-d_2 \cdot t_s}\}$$
$$c_{\text{TP}}(P_3^{II}) = \{(s_{\text{input_state}} == 3)^t, (s_{\text{output_state}} == 1)^{t-d_2 \cdot t_s}\}$$
$$c_{\text{TP}}(P_4^{II}) = \{(s_{\text{input_state}} == 3)^t, (s_{\text{output_state}} == 2)^{t-d_2 \cdot t_s}\}.$$

In these sets, d_1 and d_2 denote the timing depths on the communication channels in *time slices* [18]. For the first version of our case study, it is calculated as 3. For the second version, it is calculated as 5. At this point in the analysis, due to their timing behavior, it is impossible to rule out the existence of information flow on these paths, as $s^{t-n \cdot ts}$ has to be considered a distinct signal for each time slice. It is therefore necessary to continue the analysis, i. e., to generate timed automata from each set of timed path conditions and verify whether these sets of conditions are satisfiable with the observer automata combined with the translated Stateflow controller.

7.2.3. Generated Observer Automata

From each of these sets of path conditions, our approach generates a single UPPAAL timed automaton. As explained in Section 4.1, each automaton consists of an initial state as well as one state per entry in the condition set, i. e., three states.

7.2.4. Generated UPPAAL Timed Automata

In the next step, we translate the controllers of both case studies into networks of UPPAAL timed automata. The Stateflow controller of the first version of our case study consists of three states and five transitions managing the current state of the shared bus. The controller for the second version consists of two states and five transitions, implementing the FIFO-like behavior of the shared bus. The corresponding translated networks of UPPAAL automata consist of eleven automata, ranging in size between one and four states with a large number of self loops. Ten automata emulate the functionality and semantics of the Stateflow controller and one, the added generic tester automaton, acts as the non-deterministic environment.

7.2.5. Verification Results

In the final step, our approach combines the translated Stateflow controller with the generated observer automata by adding them to the UPPAAL system declaration and generates a single verification goal for each c_{TP}. The results of the verification process are shown in Table 1. As can be seen there, the first step of our approach, the extraction of constraints from the combined models and the generation of corresponding UPPAAL automata takes between approximately 300 and 400 ms[1]. The translation of the Stateflow controller for the first and second version of our case study takes 830 ms and 761 ms, respectively, and only has to be performed once per model as we store the translation result for each model revision. Finally, for cases in which the observer automaton does not reach its final location l_f, namely on P_1^I and P_2^I, the verification of the combined controllers takes approximately 10 s while in all other cases, the property is verified after 5 s. The respective similarities in verification times are due to the complex structure of the Stateflow controller behavior emulation in comparison to the observer automata.

Unfortunately, the composition of large models does not generally scale well, as model checking has exponential complexity. However, by only using the timed path conditions as an over-approximation of the control flow within the Uppaal model (and not the com-

1 Tested on a 2.6 GHz Intel Core i7 with 16 GB main memory.

Table 1
Evaluation results

Path	Time			$EF(l_p)$
	Extract c_{TP}	Build SF_M	Verification	
P_1^I	379 ms	830 ms	10.247 s	×
P_2^I	327 ms		10.559 s	×
P_3^I	354 ms		5.971 s	✓
P_4^I	302 ms		5.788 s	✓
P_1^{II}	390 ms	761 ms	5.398 s	✓
P_2^{II}	387 ms		5.279 s	✓
P_3^{II}	349 ms		5.262 s	✓
P_4^{II}	356 ms		5.737 s	✓

plete Simulink model), our approach scales comparatively well for the practical examples we have seen at our partners from the automotive industry, where the Stateflow controllers are typically comparatively small. Note that using our optimization described in Section 5.4, we were able to decrease the necessary verification times from multiple hours to the significantly lower values seen in Table 1.

As our analysis shows, our approach successfully verified the *absence* of information flow over the critical paths $P_3^I, P_4^I, P_1^{II}, P_2^{II}, P_3^{II}, P_4^{II}$. For the first version, our approach showed that there is information flow possible on the first two paths under analysis, i. e., the non-critical odometer may have access to the braking system. This is a severe violation of the property of *integrity*, which potentially leads to disastrous consequences. To overcome this, we have corrected and successfully verified the controller implementation as presented in the following.

7.2.6. Correcting the Controller Implementations

As Table 1 shows, we were able to identify information flow on P_1^I and P_2^I. At this point, it is up to the designer to identify the source of the error. In our case, the original implementation of the controller contained faulty timed transition guards, which did not correspond to the time slice depth of the shared channel. After correcting these guards, the analysis correctly shows the absence of information flow on both paths with verification times of 5.381 s and 5.293 s and, with that, the validation of all requirements under analysis.

8. Related Work

Extensive work has been done in the area of translating subsets of combined Simulink/Stateflow models into formal languages with well-defined semantics, especially Lustre and the graphical modeling suite SCADE, in order to perform model checking on the translated systems [27, 21, 33]. 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 [32].

Only few authors have addressed the problem of formalizing the complete behavior of Stateflow automata. In [9, 8], the authors have presented operational and denotational semantics for a subset of Stateflow. While they succeed in representing a wide range of the Stateflow functionality, they do not consider the timing and the connection with surrounding Simulink models. In contrast, the approach for an automatic translation from Stateflow to UPPAAL presented in [11, 34] has the advantage that it captures both the functionality and the precise timing of Stateflow and enables automatic verification via model checking. We utilize this in our approach.

In [24], the authors present an approach for slicing Simulink models. Their algorithm identifies model parts that influence the computation of a given block. However, as their approach does not have the characteristics of an information flow analysis, i. e., does neither consider conditions nor timing along model paths, it only provides a coarse-grained dependency analysis.

A number of model analysis techniques are integrated into the *Simulink Design Verifier* [29]. It offers the designer the possibility to generate test cases and detect design errors, such as integer overflows or division by zero by utilizing static analysis methods. Its scalability, however, is strictly limited as it utilizes a model checking-based approach to analyze the model as a whole. Further, it is not possible to formulate information flow properties using the provided verification blocks. Additionally, the Design Verifier supports the generation of slices to identify data dependencies through

the model. However, it provides only a very strong over-approximation as it takes neither control nor timing dependencies into account.

Due to the similarities between Stateflow and Statecharts, it is reasonable to analyze previous formalization efforts for such models, most notably [10]. 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 approach presented in this work infeasible.

9. Conclusion

In this paper, we have presented a novel information flow analysis for combined Simulink/Stateflow models. By being applicable to combined Simulink/Stateflow models, our approach can be used to analyze information flow in embedded software models that consist of one or more embedded controllers (modeled in Stateflow), and a number of dynamic signal-flow components (modeled in Simulink). The main idea of our approach is to translate *timed path conditions* of the dynamic signal-flow, i. e., of the Simulink components of a given model, into UPPAAL timed automata, and to combine them with a timed automata representation of the embedded controllers modeled in Stateflow. For the combined model, we automatically generate verification goals which in turn enable us to automatically check the absence of information flow between arbitrary input and output ports using the UPPAAL model checker. The result is a fine-grained analysis of the information flow along paths of interest through combined Simulink/Stateflow models under both data as well as timing aspects. Note that for the computation of timed path conditions of dynamic signal-flow components, we utilize our approach presented in [18], and for the conversion of Stateflow controllers into UPPAAL, we have adapted and extended the translation from Stateflow to UPPAAL presented in [34, 11]. Our extensions include the addition of generic tester automata and optimizations of their interaction with the surrounding timed automata.

The verification times necessary to rule out the existence of safety-threatening paths using our approach are largely dependent on the size of the Stateflow controller. As these are usually relatively small compared to the surrounding Simulink components, our approach scales well for complex Simulink/Stateflow models.

We have demonstrated the practical applicability of our approach using two versions of a complex industrial case study that implements a shared communication infrastructure for a safety-critical automotive system. There, we have shown that although timed path conditions alone detect a safety requirement violation, our approach efficiently recognizes it as spurious in approximately 6 s.

In the future, we aim at relaxing our assumptions on the control flow paths, and at supporting additional design patterns for the integration of Stateflow controllers into Simulink models, such as scheduling of individual Simulink components using Stateflow or the integration of Simulink functions into controllers. Furthermore, as UPPAAL offers the possibility to display counterexamples, i. e., paths in the model checking process that led to a violation of the formula under analysis, we are confident that we are able to use this information to provide the developer with more feedback about the precise sources of possible information flow through a model by, e. g., highlighting them directly in the source model.

Acknowledgments

This work was funded by the *German Federal Ministry of Education and Research* as part of the *ECoSMo* project [16].

References

1. Alur, R., Dill, D. L. A Theory of Timed Automata. In: Theoretical Computer Science, 1994, 126(2), 183-235. https://doi.org/10.1016/0304-3975(94)90010-8
2. Assystem. Assystem Germany GmbH. Jan. 2019. URL: https://www.assystem-germany.com/.
3. Behrmann, G., D. A., Larsen, K. G. A Tutorial on UPPAAL. In: Formal Methods for the Design of Real-Time Systems. Springer, 2004, 200-236. https://doi.org/10.1007/978-3-540-30080-9_7
4. Bengtsson, J., Yi, W. Timed Automata: Semantics,

Algorithms and Tools. In: Lectures on Concurrency and Petri Nets. Springer, 2004, 87-124. https://doi.org/10.1007/978-3-540-27755-2_3

5. Goguen, J. A., Meseguer, J. Security Policies and Security Models. In: IEEE Symposium on Security and Privacy, 1982, 11-20. https://doi.org/10.1109/SP.1982.10014
6. Hammer, C., Krinke, J., Snelting, G. Information Flow Control for Java Based on Path Conditions in Dependence Graphs. In: IEEE International Symposium on Secure Software Engineering, 2006, 87-96.
7. Hammer, C., Schaade, R., Snelting, G. Static Path Conditions for Java. In: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, ACM, 2008, 57-66. https://doi.org/10.1145/1375696.1375704
8. Hamon, G. A Denotational Semantics for Stateflow. In: ACM International Conference on Embedded Software, ACM, 2005, 164-172. https://doi.org/10.1145/1086228.1086260
9. Hamon, G., Rushby, J. An Operational Semantics for Stateflow. In: Fundamental Approaches to Software Engineering. Springer, 2004, 229-243. https://doi.org/10.1007/978-3-540-24721-0_17
10. Harel, D. Statecharts: A Visual Formalism for Complex Systems. In: Science of Computer Programming, 8(3), Elsevier, 1987, 231-274. https://doi.org/10.1016/0167-6423(87)90035-9
11. Jiang, Y., Yang, Y., Liu, H., Kong, H., Gu, M., Sun, J., Sha, L. From Stateflow Simulation to Verified Implementation: A Verification Approach and a Real-Time Train Controller Design. In: 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016, 1-11. https://doi.org/10.1109/RTAS.2016.7461337
12. King, J. C. Symbolic Execution and Program Testing. In: Communications of the ACM, 1976, 19(7), 385-394. https://doi.org/10.1145/360248.360252
13. Kuo, B., Golnaraghi, F. Automatic Control Systems, 9th ed. Hoboken, N. J: Wiley, 2009. ISBN: 978-0470048962.
14. Lee, E. A., Neuendorffer, S. Concurrent Models of Computation for Embedded Software. In: IEE Proceedings - Computers and Digital Techniques, 2005, 152(2), 239-250. ISSN: 1350-2387. https://doi.org/10.1049/ip-cdt:20045065
15. MES. Model Engineering Solutions GmbH. June 2016. URL: model-engineers.com.
16. [16] Mikulcak, M., Glesner, S., and Herber, P. The ECoS-Mo Project, 2019.
17. Mikulcak, M., Herber, P., Göthel, T., Glesner, S. Information Flow Analysis of Combined Simulink/Stateflow Models. In: 2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2018, 223-228. https://doi.org/10.1109/WETICE.2018.00050
18. Mikulcak, M., Herber, P., Göthel, T., Glesner, S. Timed Path Conditions in MATLAB/Simulink. In: Götz, M., Schirner, G., Wehrmeister, M. A., Al Faruque, M. A., Rettberg, A. (Eds.), System Level Design from HW/SW to Memory for Embedded Systems. Cham: Springer International Publishing, 2017, 64-76. ISBN: 978-3-319-90023-0. https://doi.org/10.1007/978-3-319-90023-0_6
19. Mikulcak, M., Herber, P., Göthel, T., Glesner, S. Towards Identifying Spurious Paths in Combined Simulink/Stateflow Models. In: Heinrich, C. M., Martin, P. (Ed.), INFORMATIK 2016, P-259, Lecture Notes in Informatics (LNI). Gesellschaft für Informatik, GI Bonn, 2016, 1495-1508. URL: https://dl.gi.de/20.500.12116/1037.
20. Miller, C., Valasek, C. Remote Exploitation of an Unaltered Passenger Vehicle. In: Black Hat USA, 2015.
21. Miller, S., Anderson, E., Wagner, L., Whalen, M., Heimdahl, M. Formal Verification of Flight Critical Software. In: Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit, 2005, 15-18. https://doi.org/10.2514/6.2005-6431
22. Mokadem, H. B., Berard, B., Gourcuff, V., De Smet, O., Roussel, J.-M. Verification of a Timed Multitask System with UPPAAL. In: IEEE Transactions on Automation Science and Engineering, 2010, 7(4), 921-932. https://doi.org/10.1109/ETFA.2005.1612699
23. Parhi, K. K., Chen, Y. Signal Flow Graphs and Data Flow Graphs. In: Bhattacharyya, S. S., Deprettere, E. F., Leupers, R., Takala, J. (Eds.), Handbook of Signal Processing Systems. MA: Springer US, 2010, 791-816. ISBN: 978-1-4419-6345-1. https://doi.org/10.1007/978-1-4419-6345-1_28
24. Reicherdt, R., Glesner, S. Slicing MATLAB/Simulink Models. In: 34th International Conference on Software Engineering (ICSE), IEEE, 2012, 551-561. https://doi.org/10.1109/ICSE.2012.6227161
25. Robinson-Mallett, C., Hierons, R. M., Liggesmeyer, P. Achieving Communication Coverage in Testing. In: ACM SIGSOFT Software Engineering Notes, 2006, 31(6), 1-10. https://doi.org/10.1145/1218776.1218786
26. Sabelfeld, A., Myers, A. Language-Based Information-Flow Security. In: IEEE Journal on Selected Areas in Communications, 2003, 21(1), 5-19. https://doi.org/10.1109/JSAC.2002.806121

27. Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F. Defining and Translating a Safe Subset of Simulink/Stateflow into Lustre. In: International Conference on Embedded Software, ACM, 2004, 259-268. https://doi.org/10.1145/1017753.1017795
28. The MathWorks. MATLAB Simulink. r2017b. 1 Apple Hill Drive, Natick, MA, Sept. 2017. URL: www. mathworks.com/products/simulink.
29. The MathWorks. Simulink Design Verifier. r2017b. 1 Apple Hill Drive, Natick, MA, Sept. 2017. URL:www.mathworks.com/products/sldesignverifier.
30. The MathWorks. Stateflow. r2017b. 1 Apple Hill Drive, Natick, MA, Sept. 2017. URL: www.mathworks. com/products/stateflow/.
31. Tripakis, S., Sofronis, C., Caspi, P., Curic, A. Translating Discrete-Time Simulink to Lustre. In: ACM Transactions on Embedded Computing Systems (TECS), 2005, 4(4), 779-818. https://doi.org/10.1145/1113830.1113834
32. Walde, G., Luckner, R. Automatic Translation of Complex Flight Control Systems from Simu- link/Stateflow to SCADE - An Experience Report. Tech. rep. Deutsches Zentrum für Luft- und Raumfahrt, 2015.
33. Whalen, M. W., Hardin, D., Wagner, L. G. Model Checking Information Flow. In: Design and Verification of Microprocessor Systems for High-Assurance Applications. Springer, 2010, 381-428. ISBN: 978-1-4419-1538-2. https://doi.org/10.1007/978-1-4419-1539-9_13
34. Yang, Y., Jiang, Y., Gu, M., Sun, J. Verifying Simulink Stateflow Model: Timed Automata Approach. In: 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, Singapore: ACM, 2016, 852-857. ISBN: 978-1-4503-3845-5. http://doi.acm.org/10.1145/2970276.2970293.