Timed Path Conditions in MATLAB/Simulink

Marcus Mikulcak, Paula Herber, Thomas Göthel, Sabine Glesner

Timed Path Conditions in MATLAB/Simulink[*]

Marcus Mikulcak, Paula Herber, Thomas Göthel, and Sabine Glesner

Technische Universität Berlin
Ernst-Reuter-Platz 7, 10587 Berlin, Germany
`marcus.mikulcak@tu-berlin.de`

Abstract MATLAB/Simulink is a widely-used industrial tool for the development of complex embedded systems. However, due to the complexity and the dynamic character of the developed models, their analysis is a difficult challenge, in particular if timing aspects are involved. In this paper, we present an approach for the construction of *timed path conditions* for MATLAB/Simulink models. Timed path conditions allow for fine-grained conclusions about the existence of possibly critical paths through a model containing time-dependent elements. With the help of timed path conditions, it is possible to identify interference and non-interference between model parts. Furthermore, they have the potential to reduce the complexity of models to improve verifiability, reason about compliance with security policies as well as generate feasible, efficient test cases. We demonstrate the applicability of our approach with a shared buffer for public as well as confidential data.

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 [11] by MathWorks, which supports the graphical design and simulation of time-continuous as well as time-discrete systems using block diagrams. Simulink is 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 character of the developed models, the analysis of a given model is a difficult challenge, in particular if timing aspects are considered. At the same time, knowledge about the existence of certain paths and the conditions under which they are executed is highly desirable. In particular, if a Simulink model becomes the main artifact in a model-based design process, the analysis of its properties becomes crucial for the correctness and reliability of the whole development process. With the help of (timed) path conditions, it is possible

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

to identify interference and non-interference between model parts and, thus, to reason about compliance with security policies. Furthermore, (timed) path conditions can be used to compute areas of low dynamic coupling for subsequent model separation. With that, they have the potential to reduce the complexity of Simulink models and thus to improve verifiability and testability. Finally, they provide a basis for generating feasible, efficient test cases for quality assurance.

In this paper, we present an approach for the construction of timed path conditions in Simulink. The main challenge we face is that all dependencies in a given design must be considered. Thereby, dependencies might be indirectly introduced via control flow, or delayed, which introduces dependencies between signals from different time slices. In our approach, we start with a static over-approximation of all potential dependencies on a path between a timed output signal and a timed input signals and collect all control flow conditions. Then, for each path, we compute a set of constraints on all input signals by performing a backward propagation of control flow conditions, which also takes timing dependencies into account. The result of our analysis is a precise description of the timed dependencies between input and output signals, represented by timed path conditions that solely depend on model-wide input variables. We demonstrate the applicability of our approach by computing timed path conditions for a case study containing a shared buffer for public as well as confidential data.

2 Preliminaries

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

2.1 Path Conditions

In general, *path conditions* [9] describe sufficient conditions for paths to be executed. In [5,6], 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. With that, they exclude, for example, information flow which is only possible if disjoint control flow conditions are satisfied. Thus, a path condition based analysis is more precise than classical static analyses.

2.2 MATLAB/Simulink

MATLAB/Simulink [11] 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.

2.3 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 [3]. Such policies, based on the assignment of security levels to data elements, describe rules between which levels information flow is allowed or forbidden [15]. 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.

3 Related Work

Path conditions [9] are heavily used in the area of symbolic execution and automatic test generation. The use of path conditions to increase granularity of information flow analysis has first been proposed in [10]. In this work, the authors describe the combination of program slicing and *Constraint Logic Programming* (CLP) to increase the precision of slicing for C programs, implemented in the *VALSOFT* Slicing System. They consider purely static slicing as too conservative and propose the extraction of conditions on the edges of the generated *Program Dependence Graphs* (PDGs). Subsequently, the concatenation of these conditions along paths of interest are analyzed by a constraint solver. However, due to the inherent differences between `C` and MATLAB/Simulink, their approach cannot directly be transferred to Simulink. For example, their work does not take timing behavior into account. They report that the precision of slicing operations can be considerably raised by the use of path conditions.

In [14], the authors present an approach for slicing of 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* (IFA), i. e., does neither consider conditions nor timing along model paths, it only provides a coarse-grained dependency analysis.

In [18], the authors present an adaptation of the concept of path conditions to MATLAB/Simulink. The authors describe the translation of Simulink models into *Lustre*, a synchronous data-flow programming language [4]. On this basis, they define an IFA notation and calculate path conditions on the translated models. Their approach has been implemented in the `Gryphon` tool suite and tested using the example shown in Figure 1, which we adapted from their publication. With their approach, they are able to show non-interference between confidential and public data paths using path conditions. However, they assume that the timing

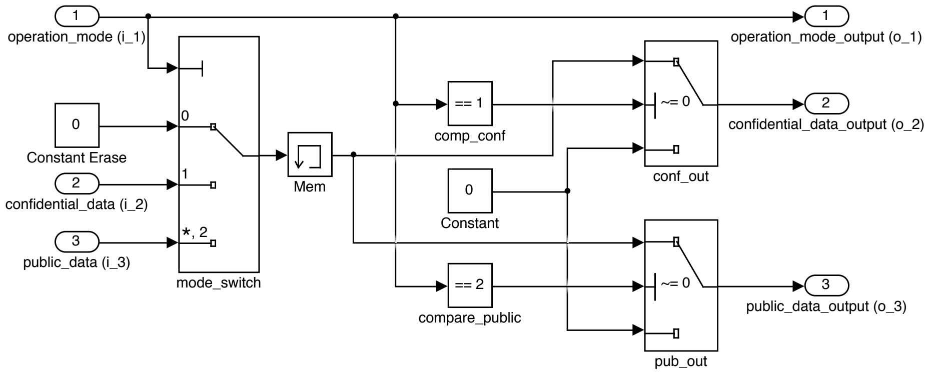

Figure 1: A shared buffer for public as well as confidential data

dependencies do not influence the information flow. Although they discuss that this assumptions is violated in their own case study, they provide no solution that takes timing dependencies into account. Possible solutions to this problem have been presented in [17], [12] and [7] via further translations of Simulink models into *Lustre*, *SIGNAL* and *UCLID*, respectively. However, as these approaches rely on a translation of models into a target language using different functional and timing semantics, properties of the original systems are lost. For example, the translation to *Lustre* maps Simulink signals onto mathematical data types, thereby losing the possibility to perform bit-precise analyses of data.

4 Timed Path Conditions in MATLAB/Simulink

In this section, we present our approach for the computation of timed path condition in MATLAB/Simulink. The main idea 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 can simply be 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, we propagate control flow dependencies backwards through the model to the input signals. With that, we can decide whether a certain path actually exists on a very fine-grained level. For both data and control dependencies, we have to take timing dependencies into account. An output might only depend on an input at certain points of time, and sophisticated routing policies might even take advantage of timing delays to make sure that two signals can never interfere. A motivating example for this case is given in the following subsection and used as a running example throughout this paper.

In order to take timing dependencies into account, we introduce the concept of *time slices*, and incorporate timing dependencies into our approach for the computation of timed path conditions by expressing all dependencies with respect to relative time slices. For the computation of timed path conditions, we use a two-step approach: (1) We (statically) identify all paths in a given Simulink model and collect all path conditions on each path. (2) For each path, we propagate all local control flow conditions backwards through the model in order to compute timed path conditions that solely depend on input variables.

In the following subsections, we first present our running example. Then, we introduce assumptions that define a Simulink subset our approach is currently able to safely analyze. In Section 4.3, we present our notations. Then, we present the computation of path conditions in Section 4.4.

4.1 Running Example

To illustrate our approach, we use a simplified version of the shared buffer presented in [18] (see Figure 1).[1] In this model, information of two different security levels (*public* and *confidential*) is fed into a shared buffer, which is implemented as a `Mem` block. According to the current operation mode, confidential (mode 1) or public (mode 2) information is saved in the buffer and passed to the corresponding output, or the contents of the buffer are erased (mode 0).

The most interesting aspect of this example is that it makes use of a sophisticated routing scheme to avoid security violations. 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 `Mem` block into account. When examining the timing of the output signals we discover that if the operation mode switches from *confidential* to *public*, the outputs register a spike of the data previously stored in `Mem`: the confidential contents are sent to the public output. By computing path conditions without taking timing dependencies into account, one would falsely assume that information flow is impossible, as the control flow conditions along the path from confidential input to public output are disjoint. This shows that we can only safely use path conditions for Simulink models if we take timing behavior into account.

4.2 Assumptions

In order to apply our approach for the computation of timed path conditions, a given Simulink model has to fulfill the following assumptions:

1. It uses a *time-discrete, fixed-step* solver.
2. It does not contain algebraic loops or loop subsystems.
3. Only scalar signals are used.
4. So far, all blocks have to use the same sample time.

[1]Our simplified version does not contain the Stateflow controller used to set the operation mode present in the original.

5. For conditional execution, we support `Enabled` and routing blocks so far.
6. Control signals only pass through simple arithmetic blocks without feedback.

The first two assumptions are acceptable as we target Simulink models from the field of discrete embedded controller design, where time-continuous solvers, loop subsystems, and algebraic loops are rarely used. Assumptions 3 to 5 are imposed due to the current state of our implementation, we are confident that our approach can easily be extended to vector or matrix signals, varying sample times, and other conditional subsystems. Assumption 6 is the most serious restriction regarding typical Simulink models of interest. However, many practical Simulink models only use simple control logics. An extension of our approach to support more complex blocks and subsystems is subject to future work.

4.3 Notations

We use the following notations: B denotes the set of *blocks* and S the set of *signals* in a given model. In addition, we use I and O as the sets of incoming and outgoing ports of a model, respectively. To describe paths, we use the set $P(b_l, b_k)$ that contains all paths between blocks b_l and b_k. On a path b_l to b_k, we denote the condition for information to flow through a block b_m as $c(b_m, b_l, b_k)$. While arithmetic blocks always establish a connection between all input and output signals ($c(b_i, *, *) = \mathit{true}$), routing blocks and conditional subsystems only establish a connection under certain conditions.

In order to take timing dependencies into account, we denote the dependency of an output signal to the set of input variables at a certain point of time as (note that $t_{\max}$ designates the maximum time slice depth over all paths):

$$o_n^t = d(i_1^t, \dots i_1^{t-t_{\max}}, \dots, i_k^t, \dots i_k^{t-t_{\max}})$$

If a path starts at source block b_0 and passes through $b_1, \dots, b_{n-1}$ to the target b_n, the timed condition for the complete path $p(b_0, b_n)$ is denoted by:

$$C\big(p(b_0, b_n)\big) = \bigwedge_{i=1}^{n-1} c(b_i, b_{i-1}, b_{i+1})^{t-t_i}$$

As described above, each atomic path condition applies to the connecting signals between two neighboring blocks and not to the complete set of input and output signals. Intersecting paths through the same *routing* block therefore create different sets of conditions.

4.4 Computation of Timed Path Conditions

In this section, we describe our approach to compute timed path conditions for Simulink models. As mentioned above, we propose a two-step approach where we first identify all paths and collect all path conditions on each path, and then propagate all local control flow conditions backwards through the model in order to compute timed path conditions that solely depend on input variables. In the following, we first describe how we compute the set of all (potential) paths using a backwards depth-first search. Then, we explain how we determine timing dependencies and how we extract (local) path conditions for each path. Finally, we present our approach for the backward propagation of the local path conditions to achieve the final timed path conditions that solely depend on input variables. We illustrate each step using our running example from Section 4.1.

Finding Paths In the first step, we identify *all* potential paths between the model inports I and outports O. This is a first step to make it possible to analyze *confidentiality* of data as well as *integrity* of the model functionality, as data flowing *to* and *from* inappropriate sources can be detected (see Section 2.3). In order to find all paths $P(i_k, o_l)$, we traverse the model from o_l recursively.

Our path detection starts the model traversal with a given outport block o_l and implements a *depth-first* recursive search for all paths i_k to o_l while marking already visited blocks. This makes it possible to detect cycles along paths throughout the model. After completion of the path detection, the sets $P(i,o) \bigm| \bigl(i \in I, o \in O\bigr)$ contain all paths from all input and output ports and can be analyzed further.

Running Example The results of the first step of our algorithm, the sets $P(i,o)$ of our example, are shown in Figure 2.

$$\begin{aligned}
&P(i_1,o_1) = \bigl\{\langle i_1, o_1\rangle\bigr\}\\
&P(i_2,o_1) = P(i_3,o_1) = \varnothing\\
&P(i_1,o_2) = \bigl\{\langle i_1, \texttt{comp\_conf}, \texttt{conf\_out}, o_2\rangle, \langle i_1, \texttt{mode\_switch}, \texttt{Mem}, \texttt{conf\_out}, o_2\rangle\bigr\}\\
&P(i_2,o_2) = \bigl\{\langle i_2, \texttt{mode\_switch}, \texttt{Mem}, \texttt{conf\_out}, o_2\rangle\bigr\}\\
&P(i_3,o_2) = \bigl\{\langle i_3, \texttt{conf\_out}, \texttt{Mem}, \texttt{mode\_switch}, o_2\rangle\bigr\}\\
&P(i_1,o_3) = \bigl\{\langle i_1, \texttt{compare\_public}, \texttt{pub\_out}, o_3\rangle, \langle i_1, \texttt{mode\_switch}, \texttt{Mem}, \texttt{pub\_out}, o_3\rangle\bigr\}\\
&P(i_2,o_3) = \bigl\{\langle i_2, \texttt{mode\_switch}, \texttt{Mem}, \texttt{pub\_out}, o_3\rangle\bigr\}\\
&P(i_3,o_3) = \bigl\{\langle i_3, \texttt{mode\_switch}, \texttt{Mem}, \texttt{pub\_out}, o_3\rangle\bigr\}
\end{aligned}$$

Figure 2: Detected paths through the model

Identifying Timing Dependencies With the complete set of paths between all model inports and outports, the next step in our analysis is the determination of the timing dependencies on each path $p(i_k, o_l)$. Three different cases can occur: (1) Untimed: The path neither contains time-dependent model elements nor is part of a feedback loop. (2) Fixed-Delay: The path contains time-dependent model elements but is not part of a feedback loop. (3) Feedback loop: The path is part of a feedback loop.

To compute the timing dependencies for a given set of paths, we iterate over each path and analyze it regarding time-dependent model elements and their parameters. If no timed element is found and the path is not part of a feedback loop, the untimed dependency relation $o_l^t = d(i_k^t)$ is established.

If the path is not part of a feedback loop but time-dependent model elements are encountered along the path during the iteration, a fixed-delay relation can be established and type and parameters of the blocks decide its magnitude. As explained above, we only consider discretely timed models with a fixed simulation step size so far. The behavior of a `Delay`, `UnitDelay` and a `Mem` block is therefore similar. Each time a `Mem` block is encountered, the magnitude of the fixed delay for the current path is increased by 1. When encountering a `Delay` block, after confirming the correct sampling time, its `DelayLength` parameter is read and added to the delay magnitude m of the current path, which yields $o_l^t = d(i_k^{t-m})$.

A path that is part of a feedback loop presents an *infinite* delay relation.

Running Example The result of the application of this step to our running example is shown in Table 1. As illustrated, information from the data inputs does never arrive at the outputs in the same time slice, as there are no paths between i_{2t}, i_{3t} and o_{2t}, o_{3t}. Only information from the previous time slice arrives at the outports. This also presents an indicator for the existence of a security violation between the confidential data input and the public data output. At each time t, confidential information from the previous time slice $t-1$ is still held inside the system and is released in case of a change in mode of operation in the form of a spike. Note that we use the index c to denote indirect information flow through the control signal of routing blocks [1].

Table 1: Timing relations between ports of our shared buffer example

	i_1^t	i_1^{t-1}	i_2^t	i_2^{t-1}	i_3^t	i_3^{t-1}
o_1^t	$p(i_1^t, o_1^t)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
o_2^t	$p(i_1^t, o_2^t)_c$	$p(i_1^{t-1}, o_2^t)_c$	$\varnothing$	$p(i_2^{t-1}, o_2^t)$	$\varnothing$	$p(i_3^{t-1}, o_2^t)$
o_3^t	$p(i_1^t, o_3^t)_c$	$p(i_1^{t-1}, o_3^t)_c$	$\varnothing$	$p(i_2^{t-1}, o_3^t)$	$\varnothing$	$p(i_3^{t-1}, o_3^t)$

Extracting Path Conditions After the identification of timing dependencies, in the next step of our analysis, we extract the conditions necessary for information to flow along paths. These conditions are dependent on the type of the block and its semantics. While it is *true* for blocks from the functionality and timing categories, routing blocks are analyzed further to extract their behavior. Thus, we iterate over each path in $P(I, O)$ to check for the existence of routing blocks and create a set $C\big(p(i_k^{t-c}, o_l^t)\big)$ that holds the extracted conditions. If a routing block b_r in time slice $t-d$ with inputs $s_{ctrl}, s_1, \ldots, s_n$, output s_{out} and neighboring blocks b_l and b_o is found, the necessary condition for the current path is extracted and saved for later analysis by the constraint solving tool. The condition is formed depending on the type of the encountered routing block.

Running Example The goal of our running example, presented in Section 4.1, is to prove whether there is information flow between the confidential inport and the public outport. In the face of multiple routing blocks and a time-dependent model element, we will only consider one path in this example application of our approach: $p(i_2^{t-1}, o_3^t)$. After this step, the set of path conditions on this path is:

$$C\big(p(i_2^{t-1}, o_3^t)\big) = \big\{s_{ctrl}(\texttt{pub\_out})^t \neq 0, s_{ctrl}(\texttt{mode\_switch})^{t-1} == 1\big\}$$

Backward Propagation of Path Conditions As shown above, a single condition is extracted for each routing block on each path. However, as these individual conditions only contain local information about a single control signal, we propagate these control flow signals backwards to the inports of the model. This requires to take the functionalities of each block between the control signal and the inports into account. It elevates the local information about control signals in path conditions to model-wide conditions for information flow, which solely depend on input signals. To accomplish this, we analyze each control signal separately and iterate over each path from the signal to its drivers while collecting the functionality of each block. We denote the resulting dependencies as:

$$s_{ctrl}(b_r) = d(i_1^t, \dots, i_1^{t-t_{\max}}, \dots, i_j^{t-t_{\max}}, \dots, i_j^{t-t_{\max}})$$

For a single block b_l, we define its functionality as $s_o(b_l) = f_{b_l}(s_{i_1}, \dots, s_{i_n})$. When considering a complete path $p(b_1, b_n)$, the resulting function f_p is formed by the composition of each output function along the path according to its structural connections. For example, a linear chain of blocks yields:

$$f_p := f_{b_1} \circ f_{b_2} \circ \dots \circ f_{b_n}$$

Note that for each block type, a specific set of parameters is extracted and its resulting functionality is recorded. We currently support the following block types on control paths: `Bias`, `Gain`, `Abs`, `Compare`, `Add`, `Product`.

While we support untimed and fixed-delay timing relations over control paths, i. e., the existence of multiple time slices along these paths, we presently do not support feedback loops inside control flow paths, as no conclusion can be drawn under these circumstances. We plan to extend our approach with feedback loops and support for additional block types in control flow paths in future work.

Running Example Using the previously created set of path conditions $C(i_2^{t-1}, o_3^t)$ as an input, this step of our analysis propagates the local path conditions of the routing blocks backwards and collects the functionality of each block along these paths. The model inports driving the control signals in C can be found in the first row of Table 1. There, untimed dependencies between i_1, the mode of operation and o_2^t as well as o_3^t leading through the control signals of the routing blocks can be found and the paths between the control signal and the inport i_1 are identified:

$$p(i_1^t, s_{ctrl}(\texttt{mode\_switch})) = \{i_1^{t-1}\}$$
$$p(i_1^t, s_{ctrl}(\texttt{pub\_out})) = \{\texttt{compare\_public}, i_1^t\}$$

Subsequently, we need to record the functionality of each block along these paths. The first path presents the trivial case that the control signal is directly connected to the inport of the model, we can therefore note its function as:

$$s_{ctrl}(\texttt{mode\_switch}) = f_{Inport}(i_1^{t-1}) = i_1^{t-1}$$

When collecting the functionality of the second path, we encounter a `Compare` block with a `Const` value of 2, which we translate into the following function:

$$s_{ctrl}(\texttt{pub\_out}) = f_{Compare} \circ f_{Inport}(i_1^t) = i_1^t == 2$$

With both extracted control flow relations, we have raised the scope of the path conditions from routing block-local to model-wide as path conditions are now presented as directly depending on a set of model inputs instead of local signals.

$$C\big(p(o_3^t, i_2^{t-1})\big) = \big\{(i_1^t == 2) \neq 0, i_1^{t-1} == 1\big\}$$

Translating and Solving Path Conditions We can analyze our timed path conditions, which are expressed by sets of constraints, using a constraint solver. We chose a format that resembles a set of constraints on signals. To translate this representation into a set of constraints, we first declare all encountered signals as decision variables, then to extract each condition as a constraint on the signals. Finally, the solver is instructed to find an assignment to the decision variables that does not violate any constraints. If such an assignment can be found, we can conclude that the extracted conditions along the path overlap and there is indeed the possibility for information flow. If the constraint system is unsatisfiable, the path conditions prohibit information flow.

Running Example The result from translating the path conditions extracted in the previous step into a constraint system is shown in Listing 1.1. Decision variables are declared first, then the two conditions extracted from the routing blocks along the path are shown. When solving the constraint system, we conclude that it is in fact satisfiable due to the timing annotation at the input signal i_1, allowing for information to flow from confidential to public signals.

```
var int: i_1_t;
var int: i_1_t_sub_1;
constraint (i_1_t == 2) != false;
constraint i_1_t_sub_1 == 1;
solve satisfy;
```

Listing 1.1: The translated constraint system

5 Evaluation

To evaluate our approach, we have implemented the analysis described above in `Java`. Our implementation uses an existing Simulink model parser originally developed for the *Methods of Model Quality* (MeMo) project [8]. We made our computation and implementation accessible via an `Eclipse` plug-in. While in its current state, our backward propagation algorithm only supports a small subset of simple blocks, we are confident that it still can be applied to a broad range of practical examples as this part of our approach must only be applied to the part of the design that models control signals. As a CLP language, we chose `MiniZinc` [13] for its simplicity and the possibility to be translated into multiple solver back ends. As a back end, we utilize the `Gecode` [16] constraint solver.

Table 2 shows the results of our analysis of the running example. With a complexity linear to the size of the model, our algorithm extracts the timed path conditions and passes them to the constraint solver. As can be seen in the table, both the extraction of path conditions as well as the solving of each constraint file by `Gecode` is performed in under 100 ms.

For the two paths $p(i_2^{t-1}, o_2^t)$ and $p(i_3^{t-1}, o_3^t)$ connecting the confidential and public inputs with their respective outputs, the satisfiable constraint system shows that their path conditions overlap and information flow is therefore possible. Additionally, their timing relation shows that information fed into the system at time t exits the corresponding output in the next time slice.

The constraint systems created for the two paths $p(i_2^{t-1}, o_3^t)$ and $p(i_3^{t-1}, o_2^t)$, on which confidential information crosses to the public output and vice versa, are satisfiable and therefore show that although the designer intended to use the operation mode to ensure non-interference, information flow does indeed occur whenever the operation mode is changed. The security policy of non-interference between the confidential and public data flows is therefore violated.

Table 2: Evaluation results

Path	Constraints	Sat	Time	
			Extraction	Solver
$p(i_1^t, o_1^t)$	$\varnothing$	-		-
$p(i_2^{t-1}, o_2^t)$	$\{i_1^{t-1} == 1, (i_1^t == 1) \neq 0\}$	✓		38 ms
$p(i_2^{t-1}, o_3^t)$	$\{i_1^{t-1} == 1, (i_1^t == 2) \neq 0\}$	✓		29 ms
$p(i_3^{t-1}, o_2^t)$	$\{i_1^{t-1} == 2, (i_1^t == 1) \neq 0\}$	✓	73 ms	27 ms
$p(i_3^{t-1}, o_3^t)$	$\{i_1^{t-1} == 2, (i_1^t == 2) \neq 0\}$	✓		33 ms

6 Conclusion

In this paper, we have presented an approach to extract *timed path conditions* from Simulink models. These conditions can be used to reduce model complexity and as an IFA tool to argue about the existence of paths between arbitrary blocks in a model. We have shown how we find paths between inputs and outputs to the model and how we determine timing dependencies of signals along these paths. Further, we have demonstrated how we extract conditions from routing blocks on paths and how we identify control flow relations between blocks to be able to draw conclusions about the existence of paths using a constraint solving tool. Using the example of a shared buffer for confidential as well as public data, we have demonstrated the usability of our approach in the context of an IFA. Thereby, we have shown how timed path conditions can be used to both detect as well as rule out security policy violations.

To increase the precision of our approach, we are planning to extend its functionality to include more parts of the Simulink design library, such as `IndexVector` and `Selector` blocks to support non-scalar signals. Furthermore, we see high potential to increase the width of our approach by supporting Stateflow [11], an extension to Simulink with functionality and semantics similar to state machines. Stateflow is widely-used to model control logic within Simulink, i.e., to drive the control signals of routing blocks within the model. Finally, we aim at extending our approach to support more Simulink-specific features used in industrial applications, such as bit-precise variable modifications and the TargetLink block set [2] used in the development of implementation-level Simulink models.

References

1. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM (1977)
2. dSpace: TargetLink Embedded Code Generator. https://www.dspace.com (2015)
3. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 2012 IEEE Symposium on Security and Privacy. IEEE Computer Society (1982)
4. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language lustre. Proceedings of the IEEE (1991)
5. 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)
6. Hammer, C., Schaade, R., Snelting, G.: Static path conditions for Java. In: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security. ACM (2008)
7. Herber, P., Reicherdt, R., Bittner, P.: Bit-precise formal verification of discrete-time MATLAB/Simulink Models using SMT solving. In: Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on (2013)
8. Hu, W., Wegener, J., Stürmer, I., Reicherdt, R., Salecker, E., Glesner, S.: MeMo - Methods of Model Quality. In: MBEES (2011)
9. King, J.C.: Symbolic execution and program testing. Communications of the ACM (1976)
10. Krinke, J., Snelting, G.: Validation of measurement software as an application of slicing and constraint solving. Information and Software Technology (1998)
11. MathWorks: MATLAB/Simulink. http://www.mathworks.com/products/simulink/ (2015)
12. Messaoud, S.: Translating Discrete Time Simulink to SIGNAL. Ph.D. thesis, Virginia Tech (2014)
13. NICTA: The MiniZinc Constraint Programming Language. http://www.minizinc.org/ (2014)
14. Reicherdt, R., Glesner, S.: Slicing MATLAB/Simulink Models. In: Software Engineering (ICSE), 2012 34th International Conference on. IEEE (2012)
15. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. Selected Areas in Communications, IEEE Journal on (2003)
16. Schulte, C., Lagerkvist, M., Tack, G.: Gecode: Generic constraint development environment. In: INFORMS Annual Meeting (2009)
17. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time Simulink to Lustre. ACM Transactions on Embedded Computing Systems (TECS) (2005)
18. Whalen, M.W., Hardin, D., Wagner, L.G.: Model Checking Information Flow. Springer US (2010)