Proceedings of the 33rd European Safety and Reliability Conference (ESREL 2023)
Edited by Mário P. Brito, Terje Aven, Piero Baraldi, Marko Čepin and Enrico Zio
©2023 ESREL2023 Organizers. *Published by* Research Publishing, Singapore.
doi: 10.3850/978-981-18-8071-1_P599-cd

A Formal Verification Framework for Model Checking Safety Requirements of a Simulink Landing Gear Case Study

Tim Gonschorek, Hannes Stützer, Frank Ortmeier

Faculty of Computer Science, Otto-von-Guericke-University Magdeburg, Germany.
E-mail: fname.lname@ovgu.de

Leon Wehmeier

Department of Production and Wood Technology, OWL University of Applied Sciences and Arts, Germany.
E-mail: leon.wehmeier@th-owl.de

Michael Oppermann

Aribus Operations GmbH, Germany. E-mail: michael.oppermann@airbus.com

The request for computer-aided system verification approaches increases with rising system complexity. So, integrating formal verification approaches, e.g., model checking, into the typical engineering workflow could help keep up with this rising complexity. Therefore, however, such an analysis of a system model must cover widely applied design specification languages, e.g., Matlab Simulink or Modelica.
This paper shall provide an approach for transforming a Simulink model into the input language of a model checking verification tool. Therefore, we modeled a widely applied case study of an aircraft landing gear system to provide a formal framework that can later be translated into a Kripke structure. A widely used formalism for verification tool input languages. The set of Matlab Simulink elements for which we provide a translation is derived from the model itself. To prove that the translation preserves the model's semantics, we also define a formal representation of the modeled Simulink elements. The translation enables us to apply several model checking tools for formal verification.

Keywords: Model-based Safety Assessment, Model Checking, Integrated Safety Design.

1. Introduction

Model-based design and assessment are more and more applied within the development of dependable, software-intensive systems. Especially when the intended system behavior is specified, dynamical description formalism support the verification thereof, i.e., does the specification meet all requirements given by the authorities and corresponding norms.

Although both working with model-based approaches, designers and analysts often do not work on identical models. System designers apply languages and tools supporting the design of the specification. System analysts, however, tend to apply languages and tools that enable the use of logical reasoning and formal methods, e.g., theorem provers or model checking tools. This can lead to inconsistencies between the models and information losses, especially when the analysis results are communicated back to the designer.

What is missing, in our point of view, is a mathematically sound intermediate representation between design and verification tool languages. Such a representation would allow for automatic transformations as well as consistency checks, in both directions, between design and verification models. In contrast to already existing implementations from design languages like SysML or MatLab/SIMULINK into a single verification language, we intend to define a theoretical transformation that can be easily used for a wide set of verification languages.

To reach a wide application range, we decided to define this transformation for a subset of MatLab/SIMULINK and STATEFLOW. For defining a sound transformation, the semantics of the design model are required. In addition, we define a projection from the defined transformation into the notation of Kripke structures. Such Kripke structures can be used as a foundation for many qualitative, time-discrete state machine representation languages, often applied for model checking tools.

In this paper, we focus on time discrete, synchronous behavior description elements, which is sufficient for the majority of design models of relevant size. The SIMULINK elements chosen in this work were derived i) from a review of industry models and ii) from model relevant for an applied case study of a Landing Gear System. Further, the applicability of the translation and the applicability of the model checking approach for large scale models, are proven by analyzing the mentioned Landing Gear System case study.

In the following, we present a brief overview over related work (sec. 2). We present the formal semantics in section 3 as well as the semantics of particular SIMULINK blocks in section 4. In section 5 we validate our semantics and transformation framework on a larger case study.

2. Related Work

There already exist tools in the literature that can translate SIMULINK models into model checker input languages for formal analysis. These works, however, either focus on specific SIMULINK subsets or limit themselves to providing transformation rules into the respective formal framework's input language instead of defining explicit formal semantics for the considered SIMULINK models.

In Meenakshi et al. (2006), the authors demonstrate their tool for generating SMV code from avionic SIMULINK design models. They focus on the practical aspects of implementing a SIMULINK translation engine.

Joshi and Heimdahl (2005) briefly describe the implementation of a translation engine, but mainly focus on the generation of fault trees (resp. minimal cut sets) based on the translation of SIMULINK models. Marriott et al. (2012) describe the translation to Circus, and provide a multitude of practical translation rules for translating individual elements of SIMULINK to Circus.

Scaife et al. (2004) define a "safe" subset of SIMULINK STATEFLOW models and provide translation stubs for translating these STATEFLOW elements to Lustre. Tripakis et al. (2005) provide alternative Lustre representations for discrete-time SIMULINK models. Another such work is Zhan et al. (2017), who provide a rather comprehensive functional description and semi-formal model semantics definition. They encode SIMULINK models in the HCSP format (higher-order logic hybrid process modeling language) for the Isabelle/HOL proof environment.

In Filipovikj et al. (2016), the authors describe a translation of SIMULINK models to the UPPAAL language. It focuses on the practical side of the transformation and the inclusion of statistical aspects in the SIMULINK models and analyses.

The authors of R. Reicherdt (2015) worked on translating SIMULINK models in the Boogie language. They provide a set of language-specific translation rules. They also only rely on a language-implicit model semantics definition by describing the model to Boogie transformation.

While all these works provide integrations between SIMULINK and model checking frameworks, there is, to the best of our knowledge, no mathematically rigorous semantic transformation that may be used for dynamic verification methods such as model checking.

The only further works known to us dealing with creating a mathematical semantics definition for parts of SIMULINK are Dragomir et al. (2018); Tiwari (2002). In Tiwari (2002) the authors, however, focus specifically on semantics for hybrid STATEFLOW automatons modeled in SIMULINK, providing a semantic definition specifically tailored for the STATEFLOW-component of SIMULINK. Due to their specific use case, they did not need to include semantics for regular SIMULINK blocks and the composition of blocks and models, as we did. In Dragomir et al. (2018), the authors present a formal representation. But the target is defined as a static verification in contrast to the dynamic verification approaches enabled by our presented approach.

3. A Formal Semantics for SIMULINK

Since we only consider time-discrete, synchronous[a] SIMULINK models, we can define any

[a] All standard SIMULINK blocks are time-synchronous.

block as follows.

A formal abstraction of a SIMULINK block $\mathcal{B}$ is the tuple

$$\mathcal{B} = \langle I, O, S, R, F, P_0 \rangle$$

with I being the block's input alphabet, O the output alphabet, S the set of the internal states, $R \subseteq (I \times S) \times S$ the left-total transition relation which defines the update step from the current block state $s \in S$ to the block state $s' \in S$ whenever the model progresses to the next step such that $\langle \vec{i}, \vec{s}, \vec{s'} \rangle \in R$, and $F : I, S \to O$ a function computing the output value depending on the input and block state. P_0 defines the block's initial state, i.e., its state in step 0.

3.1. *Composition of* SIMULINK *Blocks*

A SIMULINK model, in general, contains more than a single block. Blocks are connected such that the output of one block is used as input of another block with the intention to further process computational results.

Therefore, the semantics of a global SIMULINK model block is a recursive composition of all internal blocks until, on a global context, no input or output is connected to another block but rather to particular input and output blocks.

For the case that all outputs of a block are connected to inputs of a single block, the semantics of the composition of two blocks $\mathcal{B}1$ and $\mathcal{B}2$ can be defined as follows:

Definition 3.1 (I/O-Complete Composition). *If all outputs from a block $\mathcal{B}1$ are connected to the inputs of a block $\mathcal{B}2$, we call this composition $[\![\mathcal{B}1 : \mathcal{B}2]\!]$ I/O-complete. The semantics of the resulting global block $[\![\mathcal{B}1 : \mathcal{B}2]\!]$ is given by:*

$$[\![\mathcal{B}1 : \mathcal{B}2]\!] = \langle I_{\mathcal{B}1}, O_{\mathcal{B}2}, S_{\mathcal{B}1} \times S_{\mathcal{B}2}, R, F, \\ \langle P_{0_{\mathcal{B}1}}, P_{0_{\mathcal{B}2}} \rangle \rangle$$

where

$$\begin{aligned} F &: I_{\mathcal{B}1} \times S_{\mathcal{B}1} \times S_{\mathcal{B}2} \to O_{\mathcal{B}2}; \\ &\quad \langle \vec{i_{\mathcal{B}1}}, \vec{s_{\mathcal{B}1}}, \vec{s_{\mathcal{B}2}} \rangle \mapsto F_{\mathcal{B}2}(M(F_{\mathcal{B}1}(\vec{i_{\mathcal{B}1}}, s_{\mathcal{B}1})), \\ &\quad s_{\mathcal{B}2}) \\ R =& \ \{(\langle \vec{i_1}, \vec{s_1}, \vec{s_2} \rangle, \langle \vec{s'_1}, \vec{s'_2} \rangle) \\ &\quad | \langle \langle \vec{i_1}, \vec{s_1} \rangle, \vec{s'_1} \rangle \in R_{\mathcal{B}1}, \\ &\quad \langle \langle F_{\mathcal{B}1}(\vec{i_1}, \vec{s_1}, \vec{s_2} \rangle, \vec{s'_2} \rangle \in R_{\mathcal{B}2}\} \end{aligned}$$

where $i_{\mathcal{B}1} \in I_{\mathcal{B}1}, i_{\mathcal{B}2} \in I_{\mathcal{B}2}$, and $s_{\mathcal{B}1} \in S_{\mathcal{B}1}, s_{\mathcal{B}2} \in S_{\mathcal{B}2}$ and $M : O_{\mathcal{B}1} \to I_{\mathcal{B}2}$ is a mapping function to reorder the connected signals if required (e.g. if $\vec{o_1} = \langle a, b \rangle$ and $\vec{i_2} = \langle b, a \rangle$, then $M(\langle a, b \rangle) = \langle b, a \rangle$).

In contrast to the I/O-complete compositions, SIMULINK blocks can be parallel, i.e., they do not have any connection via input or output.

Definition 3.2 (Parallel Composition). *The semantics for a parallel composition $[\![\mathcal{B}1 \| \mathcal{B}2]\!]$ of blocks $\mathcal{B}1$ and $\mathcal{B}2$ is defined as*

$$[\![\mathcal{B}1 \| \mathcal{B}2]\!] = \langle I_{\mathcal{B}1} \times I_{\mathcal{B}2}, O_{\mathcal{B}1} \times O_{\mathcal{B}2}, S_{\mathcal{B}1} \times S_{\mathcal{B}2}, R, \\ F, \langle P_{0_{\mathcal{B}1}}, P_{0_{\mathcal{B}2}} \rangle \rangle$$

where

$$\begin{aligned} R =& \ \{\langle (\vec{i_{\mathcal{B}1}} {}^\frown \vec{i_{\mathcal{B}2}}), (\vec{s_{\mathcal{B}1}} {}^\frown \vec{s_{\mathcal{B}2}}), (\vec{s'_{\mathcal{B}1}} {}^\frown \vec{s'_{\mathcal{B}_2}}) \rangle | \\ &\quad \langle \langle \vec{i_{\mathcal{B}1}}, \vec{s_{\mathcal{B}1}}, \vec{s'_{\mathcal{B}1}} \rangle, \langle \vec{i_{\mathcal{B}2}}, \vec{s_{\mathcal{B}2}}, \vec{s'_{\mathcal{B}2}} \rangle \rangle \in R_{\mathcal{B}1} \times \\ &\quad R_{\mathcal{B}2}\} \\ F &: (I_{\mathcal{B}1} \times I_{\mathcal{B}2}) \times (S_{\mathcal{B}1} \times S_{\mathcal{B}2}) \to O_{\mathcal{B}1} \times O_{\mathcal{B}2}; \\ &\quad \langle \vec{i_1}, \overline{\vec{i_2}}, \vec{s_1}, \vec{s_2} \rangle \mapsto F_{\mathcal{B}1}(\vec{i_1}, \vec{s_1}) {}^\frown F_{\mathcal{B}2}(\overline{\vec{i_2}}, \vec{s_2}) \end{aligned}$$

b

At the moment, we covered the compositions for a complete connection of the I/O relation of two blocks (*cf.* Def. 3.1) and no connection between two blocks (*cf.* Def. 3.2). In most cases, however, it is the case that some outputs of $\mathcal{B}1$ are connected to inputs of $\mathcal{B}2$ and the other open inputs and output are connected to other blocks or the outputs act as global SIMULINK block outputs. For this partial sequential composition the resulting block's $\mathcal{B}$ is given in Def. 3.3.

Definition 3.3 (Partial Sequential Composition). *The semantics of a partial sequential composition $[\![\mathcal{B}1; \mathcal{B}2]\!]$, where only a subset of the outputs of $\mathcal{B}1$ map to inputs of $\mathcal{B}2$ is given by:*

$$[\![\mathcal{B}1; \mathcal{B}2]\!] = \langle I, O, S_{\mathcal{B}1} \times S_{\mathcal{B}2}, R, F, \langle P_{0_{\mathcal{B}2}}, P_{0_{\mathcal{B}2}} \rangle \rangle$$

[b]The notation $(\vec{a} {}^\frown \vec{b})$ is used to express the concatenation of the vectors $\vec{a}$ and $\vec{b}$, i.e. $\vec{a} {}^\frown \vec{b} = \begin{pmatrix} a \\ b \end{pmatrix}$.

where:

$$
\begin{aligned}
I = {} & I_{\mathcal{B}1} \times \{\langle i_{\mathcal{B}2_j}\rangle | i_{\mathcal{B}2} \in I_{\mathcal{B}2}, i_{\mathcal{B}2_j} \textit{ not connected} \\
& \quad \textit{to any output of } \mathcal{B}1\} \\
O = {} & \{\langle o_{\mathcal{B}1_j}\rangle | o_{\mathcal{B}1} \in O_{\mathcal{B}1}, \textit{output port number j} \\
& \quad \textit{is not connected to } \mathcal{B}2\} \times O_{\mathcal{B}2} \\
F\ : {} & I \times S_{\mathcal{B}1} \times S_{\mathcal{B}2} \to O; \\
& \langle \vec{i_1}, \vec{i_2}, \vec{s_1}, \vec{s_2}\rangle \mapsto \\
& F_{\mathcal{B}1_{unconnected}}(\vec{i_1}, \vec{s_1})^\frown \\
& F_{\mathcal{B}2}(M(F_{\mathcal{B}1}(\vec{i_1}, \vec{s_1}), \vec{i_2}), \vec{s_2})
\end{aligned}
$$

$$
\begin{aligned}
R = \{ & \langle\langle \vec{i}, \vec{s_1}, \vec{s_1'}\rangle, \langle \vec{j}, \vec{s_2}, \vec{s_2'}\rangle\rangle | \\
& \langle \vec{i}, \vec{s_1}, \vec{s_1'}\rangle \in R_{\mathcal{B}1}, \langle \vec{j}, \vec{s_2}, \vec{s_2'}\rangle \in R_{\mathcal{B}2}, \\
& \vec{j} = M(F_{\mathcal{B}1}(\vec{i}, \vec{s_1}), I_{\mathcal{B}2})\}
\end{aligned}
$$

Applying these three definitions we are able to derive the overall semantics of a global SIMULINK block (or model) from its individual elements by computing the definitions bottom up. The connections of the block are, therefore, encoded in the derived transition relation R by applying the output F of a block as input of the composed block.

3.2. *Block Types*

We introduce the definitions of stateful and stateless blocks to differentiate between blocks only requiring the computation of a formula on the input values (i.e., are basically time-invariant and independent of previous model states) from those where the block's output depends not only on the current inputs but also the block's state (which in turn depends on previous states and inputs).

A **stateless** block can be defined as a block $\langle I_{\mathcal{B}}, O_{\mathcal{B}}, S_{\mathcal{B}}, R_{\mathcal{B}}, F_{\mathcal{B}}, P_{0_{\mathcal{B}}}\rangle$ where the state is irrelevant, i.e., $\forall i \in I.\forall q \in S.\forall s \in S.F(i,s) = F(i,q)$ and immediate inputs relevant for computing the output value via F, i.e., $\neg(\forall i \in I.\forall j \in I.\forall s \in S.F(i,s) = F(j,s))$

A **stateful** block $\langle I_{\mathcal{B}}, O_{\mathcal{B}}, S_{\mathcal{B}}, R_{\mathcal{B}}, F_{\mathcal{B}}, P_{0_{\mathcal{B}}}\rangle$ can be defined as a block where $\neg(\forall i \in I.\forall j \in I.\forall s \in S.F(i,s) = F(j,s))$ and $\neg(\forall i \in I.\forall q \in S.\forall s \in S.F(i,s) = F(i,q))$, i.e., the current input and the block state is required to compute the current outputs. We can further differentiate between stateful blocks by dividing them into **stateful immediate** and **stateful non-immediate** blocks.

Stateful non-immediate blocks are blocks which's outputs only depend on the current block state and not the current input, like for example a memory block[c]: $\forall i \in I.\forall j \in I.\forall s \in S.F(i,s) = F(j,s) \wedge \neg(\forall i \in I.\forall q \in S.\forall s \in S.F(i,s) = F(i,q))$.

Stateful immediate blocks are blocks which's outputs depend on the current block state and current input values, i.e., inputs immediately affect output, and state affects output: $\neg(\forall i \in I.\forall j \in I.\forall s \in S.F(i,s) = F(j,s)) \wedge \neg(\forall i \in I.\forall q \in S.\forall s \in S.F(i,s) = F(i,q))$

3.3. *Execution Semantics/Traces and Paths*

The semantics $[\![\langle I_{\mathcal{B}}, O_{\mathcal{B}}, S_{\mathcal{B}}, R_{\mathcal{B}}, F_{\mathcal{B}}, P_{0_{\mathcal{B}}}\rangle]\!]$ of a tuple $\langle I_{\mathcal{B}}, O_{\mathcal{B}}, S_{\mathcal{B}}, R_{\mathcal{B}}, F_{\mathcal{B}}, P_{0_{\mathcal{B}}}\rangle$ can be defined as the set of all (infinite) sequences Π of tuples $\Pi_k = \langle \vec{i_k}, \vec{o_k}\rangle$ such that all $\vec{i_k}$ are valid input vectors for the system at step k, and valid initial values for $k = 0$ or possible successors to a previous input vector $\vec{i_{k-1}}$, and such that all $\vec{o_k}$ are valid initial output vectors, i.e., $\vec{o_k} = F_{\mathcal{B}}(\vec{i_k}, \vec{s_k})$ where $\vec{s_k} \in S_{\mathcal{B}}$ is the current system state computed by repeatedly applying the transition relation to the system state and input values $\vec{i_j}$ for $0 \leq j \leq k$. Formally, $\begin{cases} \langle \vec{i_{k-1}}, \vec{s_{k-1}}, \vec{s_k}\rangle \in R_{\mathcal{B}} & k > 0 \\ \vec{s_k} \in S_0 & k = 0 \end{cases}$. Thus,

$$
\begin{aligned}
[\![\langle I_{\mathcal{B}}, & O_{\mathcal{B}}, S_{\mathcal{B}}, R_{\mathcal{B}}, F_{\mathcal{B}}, P_{0_{\mathcal{B}}}\rangle]\!] = \{\Pi | \Pi = \langle \Pi_0, ..., \\
& \Pi_n, ...\rangle, \Pi_k = \langle \vec{i_k}, \vec{o_k}\rangle, \langle \vec{i_0}, \vec{s_0}, \vec{s_1}\rangle \in R_{\mathcal{B}}, \\
& \vec{o_k} = F_{\mathcal{B}}(\vec{i_k}, \vec{s_k}), s_0 \in S_0, P_{0_{\mathcal{B}}} \models S_0\}
\end{aligned}
$$

3.4. *Transforming a* SIMULINK *Block to a Kripke Structure*

Most input languages of qualitative model checking tools are based on Kripke structures as input semantics. For the sake of making the defined SIMULINK block semantics as applicable as possible with other model checking tools, we define a mapping to a general Kripke structure.

[c]The current inputs, of course, affect the block's next state via R and therefore cannot be discarded.

Definition 3.4 (Kripke Structure). *As defined in Clarke et al. (2018), a Kripke structure K over a set of atomic propositions AP is represented by the tuple*

$$K = (S, S_0, T, L)$$

with S as a finite set of states, $S_0 \subseteq S$ representing the non-empty set of all initial states, $T = S \times S$ as a left-total transition relation, such that for every state $s \in S$ there exists a state $s' \in S$ with $(s, s') \in T$, and $L : S \rightarrow 2^{AP}$ as a function labeling each state with the atomic propositions evaluating to true in the state.

The transformation from $\mathcal{B} = \langle I_\mathcal{B}, O_\mathcal{B}, S_\mathcal{B}, R_\mathcal{B}, F_\mathcal{B}, P_{0_\mathcal{B}}\rangle$ to $K = (S, S_0, T, L)$ is straight forward. For the set of states S of the resulting Kripke structure, we must take into account $S_\mathcal{B}$ but also the input values $I_\mathcal{B}$. Therefore, we first generate an internal Kripke structure K^* from $S_\mathcal{B}, R_\mathcal{B}, P_{0_\mathcal{B}}$ with $S = S_\mathcal{B}$, $T = R_\mathcal{B}$ and S_0 as the set of all possible initial states that can result from $I_\mathcal{B}$ and $P_{0_\mathcal{B}}$. The set of atomic propositions AP can be generated by using the names of the internal block variables and a unique id for each SIMULINK block. The labeling function L now maps this AP to the corresponding state from S where the actual value of the SIMULINK variable fits to the proposition. As all (considered) Simulink data types are strictly speaking finite and discrete, they can be encoded in AP - for example as bit vectors. For the behavior of the global block, the overall system output O and corresponding output function F are not of interest. Further, the output is enclosed within the state space and, therefore, can be stored as additional formula F for later use. For input blocks we can also generate another Kripke structure K^I, such that the size of S equals the number of possible input values. AP and L are defined accordingly such that each input value bijectively maps to a state of the Kripke structure. And, as the input values are chosen non-deterministically, $S_0 = S$ and $T = S \times S$, i.e., each value can occur at each time. With K^* and K^I the overall semantics of K is the parallel composition of the two Kripke structures

$$[\![K]\!] = [\![K^* \| K^I]\!]$$

To create a bisimilar Kripke structure K for any given (Simulink) model $\langle I_\mathcal{B}, O_\mathcal{B}, S_\mathcal{B}, R_\mathcal{B}, F_\mathcal{B}, P_{0_\mathcal{B}}\rangle$, using the semantics notation as defined earlier, written as

$$K \sim \langle I_\mathcal{B}, O_\mathcal{B}, S_\mathcal{B}, R_\mathcal{B}, F_\mathcal{B}, P_{0_\mathcal{B}}\rangle$$

and therefore

$$\begin{aligned}&\exists F_{map}.\forall \Pi \in [\![\langle I_\mathcal{B}, O_\mathcal{B}, S_\mathcal{B}, R_\mathcal{B}, F_\mathcal{B}, P_{0_\mathcal{B}}\rangle]\!].\\ &\quad \forall \pi \in [\![K]\!].\\ &\quad F_{map}(\Pi) = \pi \wedge F_{map}^{-1}(\pi) = \Pi\end{aligned}$$

the following mappings can be used to generate K^I and K^*:

For generating K^*, an (arbitrary) bijective mapping of the model's states $S_\mathcal{B}$ (being a state vector containing the current values for each state variable) to a state set S_{K^*} as well as a labeling function L_{K^*} are required, i.e.,

$$\begin{aligned}\exists F_{SL} :& S_\mathcal{B} \rightarrow \{L_{K^*}(s_K) | s_K \in S_{K^*}\}.\\ & F_{SL}^{-1}(F_{SL}(s)) = s\end{aligned}$$

with AP and S_{K^*} chosen appropriately. The output values o of the model and blocks forming the model can be computed by applying $F_\mathcal{B}$ to the inverse mapping F_{SL}^{-1} of the Kripke structure's states. For K^I, we choose S_{K^I}, AP, L_{K^I} such that there is one state in K^I with appropriate labeling for each possible input in the original model $\mathcal{B}$, i.e., $\exists F_{map^I} : I_\mathcal{B} \rightarrow \{L_{K^I}(s) | s \in S_{K^I}\}$. The corresponding transition relation $T_{K^I} \in S_{K^I} \times S_{K^I}$ is derived from the possible model inputs defined by $R_\mathcal{B}$ for the (non-deterministic) inputs $I_\mathcal{B}$:

$$\begin{aligned}T = \{\langle s, s'\rangle | & i = F_{map^I}^{-1}(s), i' = F_{map^I}^{-1}(s'),\\ & s_\mathcal{B} \in S_\mathcal{B}, s'_\mathcal{B} \in S_\mathcal{B}, s''_\mathcal{B} \in S_\mathcal{B},\\ & \langle i, s_\mathcal{B}, s'_\mathcal{B}\rangle \in R_\mathcal{B}, \langle i', s'_\mathcal{B}, s''_\mathcal{B}\rangle \in R_\mathcal{B}\}\end{aligned}$$

This makes creating the transition relation T_{K^*} possible by deriving it from $R_\mathcal{B}$ and K^I as follows: Inputs i are modeled by the parallel structure K^I (see above). Combined with F_{SL} from earlier, which allows us to map between $S_\mathcal{B}$ and S_{K^*}, we have mapped and modeled the inputs in their entirety and defined a mapping for the model's states. This enables us to define a transition $T_{K^*} \subseteq S_{K*} \times S_{K*}$ modeling the Simulink model's transition $R_\mathcal{B} \subseteq I_\mathcal{B} \times S_\mathcal{B} \times S_\mathcal{B}$:

$$\begin{aligned}T_{K^*} = \{&\langle s_{K_j}, s'_{K_j}\rangle |\\ & s_{K_j} = F_{SL}(s_\mathcal{B}), s'_{K_j} = F_{SL}(s'_\mathcal{B}),\\ & i_\mathcal{B} = F_{map^I}^{-1}(L_{K^I}(s_{K^I})), \langle i_\mathcal{B}, s_\mathcal{B}, s'_\mathcal{B}\rangle \in R_\mathcal{B},\\ & s_{K^I} = \pi_j^{K^I}, \pi^{K^I} \in [\![K^I]\!], s_{K_j} = \pi_j^{K^*},\\ & \pi^{K^*} \in [\![K^*]\!]\}\end{aligned}$$

Finally, we combine the parallel Kripke structures K^I and K^* to $K = K^I \| K^*$. The resulting structure K is, by construction, a bisimulation of the original model $\mathcal{B}$.

4. Semantics of Various Simulink Blocks

The considered SIMULINK blocks are either purely computational *stateless* blocks (sums, logical operators, etc.) or *stateful* blocks, i.e., blocks that also depend on the current (and past) system states[d]. The following subsections provide a brief excerpt of the semantics of some of the most commonly used blocks in our analyzed models.

4.1. *Stateless Operators*

Logical Operators For the stateless, binary logical operators *and*, *or*, *xor*, their respective negations, and the unary operator *not*, their respective semantics are given by $[\![logop]\!] = \langle I, O, \emptyset, R, F, \emptyset\rangle$ where $I = \{\top, \bot\}^2$, $R = \{\langle i, \emptyset, \emptyset\rangle \,||i \in I\}$ and a block specific output function F.

Stateless Relational and Math Operators Similarly, for the relational operators *less than*, *less than or equals*, *greater than*, *greater than or equals*, *equals*, and *not equals*, their respective semantics can be defined as $[\![relop]\!] = \langle I, O, \emptyset, R, F, \emptyset\rangle$ where $I = \{DATATYPE^2\}$, R and O as before, and $F : I \times S \to O; \langle\langle i_1, i_2\rangle, \emptyset\rangle \mapsto i_1 \bullet i_2 = \top$ with $\bullet$ representing the block's selected operator - e.g. $\leq$. Mathematical operation blocks, e.g. sum, product, can be defined analogously.

Other Stateless Operators Another commonly used block is the *Switch* block which passes through one of its two input signals i_1 and i_3, depending on whether a specified condition is met by the boolean input i_2. The semantics using our framework can be defined as $[\![switch]\!] = \langle I, O, \emptyset, R, F, \emptyset\rangle$ where the input and output domains follow their intuitive definitions, R as previously and

$$\begin{cases} F : I \times S \to O; \langle\langle i_1, i_2, i_3\rangle, \emptyset\rangle \mapsto \langle i_1\rangle & i_2 = \top \\ F : I \times S \to O; \langle\langle i_1, i_2, i_3\rangle, \emptyset\rangle \mapsto \langle i_3\rangle & i_2 = \bot \end{cases}$$

[d]Thus, the stateless blocks only affect the transition relation of the underlying product of the stateful components' automatons.

4.2. *Stateful Simulink Blocks*

Now we take a demonstrative look at model components requiring *state*, i.e., access to information from previous steps.

Memory A SIMULINK *memory* block and the variants thereof delay the input data by a number of steps before committing them to the output signal. Formally, $[\![mem]\!] = \langle I, O, S, R, F, P_0\rangle$ with $I,\ O,\ S$ equivalent to the domains of the input signal (plus the configured initial output value, if that is outside the input signal's domain), $R = \{\langle\vec{i}, \vec{s}, \vec{s'}\rangle | s' = \vec{i}\}$, $F : I \times S \to O; \langle\langle i_0\rangle, \langle s_0\rangle\rangle \mapsto \langle s_0\rangle$, and P_0 as configured initial output value of the block parameters. The semantics of memory elements with a depth of more than 1 can be defined as chaining multiple 1-step memory blocks using the composition theorems defined earlier.

4.3. *Stateflow*

In our review of relevant design models, we found that for more complex designs STATEFLOW state charts were often used to express the desired behavior. This section provides a brief conceptual overview of STATEFLOW, and specifies commonly used STATEFLOW automaton types and their respective semantics.

4.3.1. *Stateflow Automatons*

In the interest of brevity, we shall keep the formalization of the semantics of STATEFLOW's basic mechanism to a minimum. Basically, the variable assignments occurring while a STATEFLOW automaton is in a specific state (*during* assignment) as well as transition activation conditions and variable assignments linked to transitions can be expressed as a hybrid of Moore and Mealy automatons, which is easy to map to the presented framework: The transition relation R is transparently mapped, the input alphabet is given by I, the outputs are given by applying the output relation on the current state and inputs, and the required states encoded in S.

4.3.2. *Stateflow Entry and Exit Conditions*

Besides the *during* assignments of variables in a state, STATEFLOW also supports *entry* and *exit* assignments. For exit assignments, the relevant state variable(s) are assigned the specified value in the step in which the state machine's state changes from the state with the exit assignment to any different state. *Entry* assignments are assigned in the step in which a transition leading into the relevant state activates. This behavior is simple to encode.

5. Prototyping and Early Observations

Using the semantics as described previously, we created a SIMULINK translation tool to validate our proposed semantics. This transforms information from the SIMULINK model into SAML Güdemann and Ortmeier (2010), which is a relatively straight-forward process using the mapping of the formal semantics to Kripke structures (as Kripke structures can easily be mapped to SAML). The tool extracts the blocks, block parameters, and signals from a given SIMULINK model. From this information, the tool creates an abstract syntax tree (AST), where every block is represented by equivalent SAML elements. Signals are implicitly translated by assigning a unique name to the AST elements that map the blocks and referencing them to exchange data. Once the SAML generation is complete, we use the VECS analysis framework to create a compatible input for the nuXmv model checker and parse the nuXmv results.

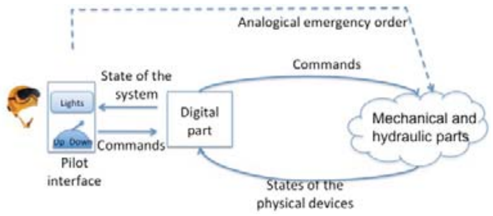
Analogical emergency order
Commands
State of the system
Lights
Up Down
Commands
Digital part
Mechanical and hydraulic parts
Pilot interface
States of the physical devices

Figure 1. Global architecture of the landing gear system Boniol and Wiels (2014).

As an example, the landing gear system introduced by Boniol and Wiels (2014) Boniol et al. (2017) was translated into a Simulink 2019b model. The global architecture of the system is depicted in Figure 1.

For the safety analysis, three Failure Conditions (FCs) were inserted in a Failure Condition Observer Simulink block (see Table 1). A FC describes a hazardous system state, which can lead to an unwanted or dangerous system behavior. The requirements R_{42} and R_{11} introduced in the case study Boniol and Wiels (2014) were modeled assuming the system is in normal mode, as well as an additional one from an aircraft level view on the system.

Table 1. FCs represented in the Simulink model.

FC	Description
FC1	Both the electro-valves controlling gear extension and retraction are open.
FC2	At least one gear is not locked extended or at least one door is not closed 15 seconds after the command handle has been moved to the extension position and remained in that position.
FC3	The green light of the pilot interface is on and all others are off indicating that the gears are locked extended, even though at least one of them is not.

In the nuXmv model, the FCs are translated into safety properties, i.e., invariants. For all three FCs, their absence could be proven using the nuXmv model checker. The runtimes were 112 seconds for FC1, 2.51 seconds for FC2 and 28.2 seconds for FC3. For validating our results, we, on the one hand, negated the FCs and generated a suitable counterexample and, on the other hand, generated a set of representative sequences the model must contain. By encoding those sequences into the corresponding acceptor automaton, we prove that these sequences were contained within the formal model encoding. This generation of execution traces, however, is in our point of view a benefit of the model-based analysis applying model checking in contrast to static analysis. The mentioned SIMULINK, SAML and nuXmv models are available online Stützer et al. (2023).

6. Conclusion

We presented a concept for a comprehensive, extensible formalism to define model semantics for discrete time SIMULINK models. Based on these formal semantics, we created an automated translation to generate SAML code from arbitrary SIMULINK models, which can be used to apply state-of-the-art formal analysis techniques to SIMULINK models. By explicitly defining the formal semantics underlying our transformation, we enable easy extensibility of our tool and lay the foundations for integrations with other analysis and transformation tools by providing a shared, unambiguous understanding of the analyzed SIMULINK models. We showed the applicability of our transformation and corresponding verification on a realistic case study taken from the literature.

Acknowledgement

This work was partially funded by the german federal state Saxony Anhalt and the European Regional Development Fund (ERDF) under grant number ZS/2020/12/153625 FuE 2104/00021 (Autosafe).

References

Boniol, F. and V. Wiels (2014). The landing gear system case study. *Communications in Computer and Information Science 433*.

Boniol, F., V. Wiels, Y. Aït-Ameur, and K.-D. Schewe (2017, Apr). The landing gear case study: challenges and experiments. *International Journal on Software Tools for Technology Transfer 19*(2), 133–140.

Clarke, E. M., T. A. Henzinger, H. Veith, and R. Bloem (2018). *Handbook of Model Checking*. Cham: Springer International Publishing.

Dragomir, I., V. Preoteasa, and S. Tripakis (2018). The refinement calculus of reactive systems toolset. In D. Beyer and M. Huisman (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Cham, pp. 201–208. Springer International Publishing.

Filipovikj, P., N. Mahmud, R. Marinescu, C. Seceleanu, O. Ljungkrantz, and H. Lönn (2016). Simulink to uppaal statistical model checker: Analyzing automotive industrial systems. In *FM 2016: Formal Methods*, Volume 9995 of *LNCS*, pp. 748–756. Cham: Springer International Publishing.

Güdemann, M. and F. Ortmeier (Eds.) (2010). *A Framework for Qualitative and Quantitative Model-Based Safety Analysis*.

Joshi, A. and M. P. E. Heimdahl (2005). Model-based safety analysis of simulink models using scade design verifier. In *Computer Safety, Reliability, and Security*, Volume 3688 of *LNCS*, pp. 122–135. Berlin, Heidelberg: Springer Berlin Heidelberg.

Marriott, C., F. Zeyda, and A. Cavalcanti (2012). A tool chain for the automatic generation of circus specifications of simulink diagrams. In *Abstract State Machines, Alloy, B, VDM, and Z*, Volume 7316 of *LNCS*, pp. 294–307. Berlin, Heidelberg: Springer Berlin Heidelberg.

Meenakshi, B., A. Bhatnagar, and S. Roy (2006). Tool for translating simulink models into input language of a model checker. Volume 4260 of *LNCS*, pp. 606–620. Berlin, Heidelberg: Springer Berlin Heidelberg.

R. Reicherdt (2015). *A Framework for the Automatic Verification of Discrete-Time MATLAB Simulink Models using Boogie*. Phd thesis, TU Berlin, Berlin.

Scaife, N., C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi (2004). Defining and translating a safe subset of simulink/stateflow into lustre. In G. Buttazzo (Ed.), *Proceedings of the fourth ACM international conference on Embedded software - EMSOFT '04*, New York, New York, USA, pp. 259. ACM Press.

Stützer, H., W. Outzen, L. Bedau, and T. Gonschorek (2023). Landing gear system models.

Tiwari, A. (2002). Formal semantics and analysis methods for simulink stateflow models.

Tripakis, S., C. Sofronis, P. Caspi, and A. Curic (2005). Translating discrete-time simulink to lustre. *ACM Transactions on Embedded Computing Systems 4*(4), 779–818.

Zhan, N., S. Wang, and H. Zhao (2017). *Formal Verification of Simulink/Stateflow Diagrams*. Cham: Springer International Publishing.