Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

Static Value Range Analysis for Matlab/Simulink-Models

Christian Dernehl[1] Norman Hansen[1] Thomas Gerlitz[1] Stefan Kowalewski[1]

Abstract: In this paper we present a static value range analysis of signals within function block diagrams developed with Matlab/Simulink. We analyse signals with respect to their data type and compute an approximation of the possible value range of each signal represented by a set of intervals. In addition the analysis reports potential problems within the model, like occurrences of NaN (Not-a-Number) and divisions by zero or infinity. We show the applicability of our analysis on a viscosity model from Matlab Central and a model from the set of automotive example models that is contained within Matlab/Simulink.

Keywords: Automotive software engineering, static analysis, Matlab/Simulink

1 Introduction

Model-based software development is used extensively within the automotive domain with one of the tools used being Matlab/Simulink. Simulink allows the creation of control models in the form of functional block diagrams, in which the data flow is expressed as a sequence of blocks connected by lines. In functional block diagrams, blocks model a function between several inputs to multiple outputs. The combination of simple functions, such as standard arithmetic, allows the modeling of complex systems as a composition of blocks from a basic block set. This design is supported by building hierarchies, to allow the reuse of existing systems and, at the same time, abstract from simple blocks. After modeling the control model, code can directly be generated from it, which is then compiled and loaded onto the target platform.

During the modeling process of a control model, errors might occur, which might eventually result in a total failure of the control system. To prevent such errors, models are tested and simulated exhaustively against a design specification. Especially in the context of safety critical systems, the behavior of a control system has to be verified to operate within specified design ranges to conform standards like the ISO26262 [IS11]. Nevertheless testing might, depending on the specification, not detect all errors and flaws within a model. Applying formal analysis in early development stages is a possible extension of the established verification process.

In this paper we present an analysis which computes an approximation of the state space of a control system modeled with Matlab/Simulink by abstracting the state space into interval sets, which are propagated along the path from inputs to outputs within the control model. Using this technique, the developer specifies sets of input intervals, as required by the ISO26262, and gets a safe approximation set of potential output intervals, which can

[1] RWTH Aachen, Informatik 11 - Embedded Software, surname@embedded.rwth-aachen.de

be used to verify the previously specified design ranges. The specification with intervals provides a simple and formal way to express the input constraints. For instance, consider an external input from an accelerometer, whose input ranges by specification are $[-4g, 4g]$, which can be read in most cases of the data sheet. Next to the resulting proposals for design ranges of each subsystem, as denoted in the ISO 26262, our analysis provides proof of the absence of values becoming NaN, division by zero/infinity and over- and underflows of signals.

Unlike abstract interpretation of the resulting C code, our analysis interprets the model. In many cases, the code generated from the model includes checks for array out of bounds and overflows, which cannot be detected by a code analysis, because the generated code handles these cases implicitly. Consider a vector, from which an element is selected using the Selector block[5], which allows to specify the index externally. In this case, code generators may include a check for the index and if the index is out of bounds, zero is returned. This mechanism proves hard to detect for static code analysis, since the array index is actually never out of bound. However, an activation of the safety procedure is hidden to the engineer.

The paper is structured as follows. In the next Section, related work in the field of value-range approximation of functional block diagrams and other domains is presented. Section 3 contains the methodology of our approach, which covers the used abstraction domain of interval sets, the set of supported functions by the analysis and how loops are approximated. The analysis is evaluated in Section 4 using a model of an open-loop and a closed-loop system. Finally, potential extensions of the approach are shown and the paper is concluded in Section 5.

2 Related Work

Thinking of the automatic verification methods for software systems, various approaches have been developed for different domains. Value set analysis, which provides a value set for each signal or variable in the program can be used to identify potential errors such as division by zero, underflows and overflows of the program variables and dead code in the program. An analysis of the resulting code from the model has been carried out to prove the absence of such errors [Si08]. Opposed to an analysis on the generated code, the Design Verifier[6] operates directly on the model by using the polyspace static analysis engine.

Depending on the configurations, the generated code of the Simulink model might be catching division by zero errors, which are still occurring in the production code, while the tester is unaware of them. Consider the case, where there is an automatic check if the divisor is zero and then the result of the function would be just 0, which is better than a runtime exception, but hides the problem. Instead our approach uses the semantics of the complete Simulink model and interprets each block individually.

[5] See `http://mathworks.com/help/simulink/slref/selector.html` .

[6] The Design Verifier is a Tool for Simulink which allows similar analysis of the model as the presented approach. `http://www.mathworks.com/products/sldesignverifier/index.html?s_tid=gn_loc_drop`

Further approaches to verify such block diagrams are performed by transforming them into an intermediate representation which is easier to analyze. For instance, converting the discrete part of Simulink to Lustre allows the utilization of existing analysis tools for synchronous languages [Tr05, Ca03]. In [CD06] a subset of the available block set within the modeling tool Simulink is transformed into an equivalent representation of the timed interval calculus, which can be used to prove formal requirements and estimate the expected value ranges of specific signals. A similar approach is presented in [CM09] where an abstract interpretation of Simulink models is presented to compute potential rounding errors when using float calculations, by computing an over approximation using Taylor series.

More exhaustive approaches include property provers in their analysis to prove the absence of bugs. In their work [RG14], Reicherdt and Glesner convert Simulink models to Boogie, which uses the Microsoft Z3 sat modulo theory (SMT) prover to prove constraints. These constraints are generated from the blocks and types in the Simulink model. Similar work has previously been done by Bauch et. al [BHB14], where models are converted to boolean formulas, using the bit vector representation.

All of the presented solutions do either an analysis of the generated code, which is very useful to check if the generated code has now flaws, or, as the latter approaches, focus on an exhaustive analysis and precise or rather numerical related computations. Therefore, different transformations into other formal representation formats are used. A general difference of our work compared to those approaches concerns the set of blocks and functionality which is supported by the described transformations and the consecutive analyses. For instance, there are limitations regarding the support of multi dimensional signals [RG14] and blocks for structural operations on such [CD06, RG14], restricting the supproted blocks to per definition *safe* blocks [CD06] or relating to the scalability of the approach for potentially very large models [BHB14].

3 Fault Detection Methods for Control Models

After having given an overview of related approaches, the focus is on our proposal. Since our analysis is based on value sets, intervals are used to represents those sets. By using sets of intervals, a more precise representation of values can be given. For each block in the model, we define a semantic with respect to finite interval sets with intervals for floating point valued signals. A finite interval set $S_I = \{I_0, \ldots, I_n\}$ contains a finite number of non-overlapping intervals $I_i, 0 \leq i \leq n$, where the interval might be either open, half-open or closed. First we define interval sets and the underlying types we use. Afterwards, the semantic for a selection of blocks, which are supported by our analysis, is presented.

3.1 Intervals and Interval Sets

Interval arithmetic is well known and widely used in different application areas [Ja12], [HJVE01], [BKS12]. However, compared to the common usage of interval arithmetic,

we check additionally faults for the IEEE 754 float. This is because many application scenarios are based on a hardware floating point unit, which eases the development process, because conversions to fixed point calculations is not required. Therefore, the symbols $-\infty$, ∞ and *nan* are also possible values in the considered interval domain. The arithmetic for intervals containing these symbols has already been implemented and defined [Wa98]. Assume the set of valid IEEE 754 floats to be $\mathbb{R}^{IEEE}$, then, the following holds $x+\infty=\infty$ (for $x\neq-\infty$), $x-\infty=-\infty$ (for $x\neq\infty$), $x*\infty=\infty$ (for $x\neq 0$), $x\circ nan=nan$ (for $\circ\in\{+,-,*,\div\}$). Note, that $[nan,nan]$ is the interval representation of the IEEE-symbol *nan*, $[\,]$ represents the empty interval and $[-\infty,\infty]$ is a valid non-empty interval.

Interval Set Arithmetic Consider an interval set S_I and any well-defined function $f:\mathbb{R}^{2\times n}\mapsto\mathbb{R}^2$, as in [NJC99], mapping n input intervals to an output interval. For interval sets $S_0,\ldots S_n$, we define $f_S(S_0,\ldots,S_n)$ as the union of the output interval sets, where f is applied to each combination of the input intervals.

$$f_S(S_0,\ldots,S_n)=\bigcup_{I_0\in S_0}\ldots\bigcup_{I_n\in S_n}f(I_0,\ldots,I_n) \tag{1}$$

For functions with n arguments and interval sets of cardinality m the resulting set may have m^n elements. However, the cardinality could shrink when applying functions to an interval set S, because two regions might overlap or have a common border. For instance, $\{[2;2],[4;4]\}+\{[-1;1]\}=\{[1;5]\}$. Nevertheless, our analysis limits the size of intervals of an interval set to 20, merging intervals together when the limit is exceeded, since the consecutive operations and over-/underflows might increase the number of intervals exponentially. Still, that limit is not reached in our applications. This limit can be configured to arbitrary positive integer values in case larger or smaller interval sets, leading to potentially smaller or bigger overapproximations, are desired by the user. Since the function f, serves in most cases as an operator, we call the process of applying f or f_S respectively an *operation*.

Typing We extend the concept of interval sets to typed interval sets by associating a datatype to every interval set. The currently supported datatypes are signed integer (int8, int16, int32), unsigned integer (uint8, uint16, uint32), floating point (float, double) and logical (boolean) types.

The datatype of an interval set is a property which restricts the set of values representable by an interval set. This means that an interval set of type int8 can only represent signed integer values which can be stored within 8 bits. To assure that every interval set S_i of datatype $\Delta\in\{boolean,int8,int16,int32,uint8,uint16,uint32,float,double\}$ always complies with its datatype restrictions, a type-restriction procedure ρ_Δ is applied after every operation on the interval set. The restriction operations ρ_Δ can be categorized into three categories $BOOL=\{\rho_{boolean}\}$, $FLOAT=\{\rho_{float},\rho_{double}\}$ and $INT=\{\rho_{int8},\rho_{int16},\rho_{int32},\rho_{uint8},\rho_{uint16},\rho_{uint32}\}$.

The boolean nature of an interval set is created by applying $\rho_{boolean}(S_i)$ which yields an interval set containing 0 (false) in case $0\in S_i$ and 1 (true) in case $\exists v\in S_i, v\neq 0$, while

the restriction methods of the category $FLOAT$ restrict the values of the interval set to single precision (float) or, respectively, to double precision (double) floating point values according to the IEEE 754 standard. In contrast to operations of categories $BOOL$ and $FLOAT$, all operations $\rho_{int} \in INT$ have to restrict the value range of integer values with potential under- and overflows. Due to the use of interval sets, under- and overflows can be represented without overapproximations. For instance, for interval sets yields $\rho_{int8}(\{[100;128]\}) = \{[-128;-128],[100;127]\}$, whereas $\rho_{int8}([100;128]) = [-128;127]$ yields a large overapproximation for intervals.

Both interval sets have to be of the same datatype, so that binary operations can be applied. This is no restriction compared to operations performed in Simulink or other programming languages since those perform implicit type-castings. To be able to perform these casts explicitly we provide correspondent operations for all supported datatypes which correspond to the application of the restriction method $\rho_{targettype}$. Since the interval sets are internally represented using the most expressive type, most casts can be performed using the type system of the underlying architecture. However, the cast from floating to integer types can be performed using different rounding modes. We focus on those rounding modes from [IE08] which are supported (but named differently) by Simulink, too:

- *round up*: Rounds towards $+\infty$.
- *round half to even*: Rounds to the nearest representable number. If these is ambiguous, rounds to the nearest even number.
- *round down*: Round towards $-\infty$.
- *round half up*: Rounds to the nearest representable number. If these is ambiguous, applies *round up* mode.
- *round half away from zero*: Rounds to the nearest representable number. If these is ambiguous, rounds positive values as *round up* and negative values as *round down*.
- *truncate*: Round towards 0.

Dimensions Signals in Simulink models can be of arbitrary dimension. Accordingly our domain of interval sets is extended to multidimensional matrices of interval sets where each matrix element is an interval set. Thus, operations on signals are operations on matrices of interval sets in our domain. Matrix operations in Simulink are either structural operations, e.g. concatenation along a dimension, or can be mapped to operations on the interval sets. For instance, the matrix multiplication of two matrices can be written as multiple expressions consisting of multiplications and additions. For simplicity of the explanations, we describe directly the operations on interval sets which is identical to the correspondent operation on matrices of exactly one element.

3.2 Operations

The following list represents a subset of function blocks, which are supported by our tool. We do not describe in this paper the standard arithmetic for intervals and the continuous

non-linear math operations, which are supported, since those have already been defined for intervals and can be mapped to interval sets as described.

- Standard arithmetic: $+, -, *, /$
- Continuous non-linear math: exp, log, trigonometry
- Relational Operators: $>, \geq, <, \leq, =, \neq$
- Logical Operators: $\neg$, $\wedge$, $\vee$
- Decisions, Switches
- Interpolation/Lookup Tables

Relational Operators First we describe, how relations are applied to interval sets. Note, that the result of a relational operator on intervals is generally not binary, but a three valued logic $\{0, 1, \perp\}$, where $\perp$ represents that the result is unknown. For instance, the outcome of the relational operation $[0,5] > [-5,3]$ is unknown. With interval sets, $\perp$ can be defined as $\{[0,0],[1,1]\}$. Let S_A and S_B be interval sets and $\succ_\perp \in \{>, \geq, <, \leq, =, \neq\}$ be a relation. Following the definition of Equation (1) $S_A \succ_\perp S_B$ is defined as $\cup_{I_A \in S_A} \cup_{I_B \in S_B} I_A \succ_\perp I_B$. In this case, the union operator might merge $[0,0]$ with $[1,1]$, representing an unknown result, although each interval relation $\succ_\perp$ yields a defined result. For example $\{[-1,-1],[1,1]\} > [0,2]$ yields for each element of the left hand side a known result, while the union of all is unknown. Nevertheless, the use of interval sets provides a more precise representation when casting boolean expressions to floating point values, e.g. as a multiplier, so that it is clear that 0.5 is not in the set.

Logical Operators For the logical operators $f_{\mathbb{B}} : \mathbb{B}^n \mapsto \mathbb{B}$, we define a new function $f_S(u_0 \neq 0, \ldots, u_n \neq 0)$ for *real* valued input interval sets $u_0, \ldots, u_n$, where 0 means $\{[0,0]\}$ in this case. Note, that $\neq$ has been previously defined on interval sets. This inequality check to zero corresponds to the standard C casting operations. The resulting output is generally calculated by applying Equation (1) on the interval sets, where the corresponding f for n inputs, and, thus, n intervals must be specified. Nevertheless, since the domain is limited to true and false, the algorithm can terminate to compute the output if two valid $x_0 \in u_0, \ldots, x_n \in u_n$ are found, making $f_{\mathbb{B}}$ become true and false.

Decisions Some blocks, like the switch block, pass a certain input through them depending on a condition c, i.e. input a is passed through if condition c is valid, otherwise input b goes through, denoted as $\text{ite}(c,a,b)$. The condition receives one parameter in form of a value computed by the model which is used to formulate the actual condition c as an relation containing the parameter value, where the result of the logical relation is an interval set as described before. If the result of the relation is $\{[0,0]\}$ or $\{[1,1]\}$, then a or respectively b is passed through the block. Additionally, a warning is issued to the user, indicating that there is a dead path in his model. However, if the result of the relation is $\{[0,0],[1,1]\}$, $a \cup b$ is the result, since the condition cannot be uniquely evaluated. Remember, due to the use of *NaN* values, the entire path of the condition has to be evaluated, and a shortcut by

evaluating for example $sin(u) > 1$ without thinking of u cannot be made, since u might be NaN.

Lookup Tables Often the behavior of specific parts in control models can be estimated with measurements, which are then fed into the model during the development process. These measurement points then realize a new function within the model. Thus, for $n \geq 2$ points $x_0, \ldots, x_n \in \mathbb{R}$ with $x_i < x_j$ for $(i < j)$ and $y_0, \ldots, y_n \in \mathbb{R}$ measurements, a function $LT(x_i) = y_i$ is defined in the model. Hence, the x_i and y_i are predefined and during execution another input x is provided, so that the output y is set to y_i if $x = x_i$. If x is in between two values x_i, x_{i+1}, then the result is either interpolated or clipped to the last y_i, which is specified by the engineer. Defining the associated interval set function for lookup tables is a two step process, in which firstly helper functions LT_C and LT_B are defined, where LT_C represents the selection of interpolation or clipping by the engineer and LT_B is a special expression handling inputs that are out of bounds, i.e. $x < x_0$ or $x > x_n$. Secondly using the ite operator, the lookup can be recursively defined. With regard to linear interpolation, the function $LT_C(I_x, y_i, y_{i+1})$ becomes

$$LT_C(I_x, y_i, y_{i+1}) \quad = \quad \left[\min_{x \in I_x} \frac{x}{x_{i+1} - x_i}(y_{i+1} - y_i), \max_{x \in I_x} \frac{x}{x_{i+1} - x_i}(y_{i+1} - y_i) \right] \tag{2}$$

where $I_x = [\underline{x}, \overline{x}]$ is an interval with $x_i \leq \underline{x}$ and $x_{i+1} < \overline{x}$ is a linear interpolation, which is extended to intervals. Regarding a flat behavior, the function becomes constant $LT_C(I_x, y_i, y_{i+1}) = [y_i, y_i]$. With respect to LT_B, this function checks if the lower bound is reached and uses otherwise the upper bound $LT_B(I_x) = \text{ite}(x < x_0, y_0, y_n)$. In the next step, ite is applied to check whether the input is between x_i and x_{i+1} for all valid i. If $x > x_n$, y_n is returned, yielding the following expression

$$LT_S(X) = \bigcup_{I_x \in X} \left(\bigcirc_{0 \leq i \leq n-1} \text{ite} \quad x_i \leq I_x \wedge I_x < x_{i+1} \quad LT_C(I_x, y_i, y_{i+1}) \right) LT_B(I_x) \tag{3}$$

where $\circ$ is the function application and X is the input interval set. Using currying, the bounded recursive application of the ite operator specifies a piecewise function, representing the behavior of the lookup table. For example, if $n = 3$, then the expression becomes

$$LT_S(X) = \bigcup_{I_x \in X} \text{ite}(x_0 \leq x \wedge x < x_1, y_0, \text{ite}(x_1 \leq x \wedge x < x_2, y_1, \text{ite}(x < x_0, y_0, y_n))) \tag{4}$$

where the last stacked ite becomes $LT_B(I_X) = \text{ite}(x < x_0, y_0, y_n)$. Note that LT_B only returns y_n in case $x > x_n$ since all other cases are covered by the previous ites.

3.3 Loops and Widening

In the previous section, we described how different functional blocks can be interpreted with interval sets. Considering open-loop systems whose outputs do not depend on a state

of the system, this approach is sufficient because for each block on the path the output intervals can be estimated. For closed-loop systems with hierarchical or nested loops, an approximation has to be performed, to compute the result set in finite time. Since the model is executed using a fixed step time Δt, the output interval sets of the model can be calculated for a fixed time horizon T within $T\Delta t^{-1}$ steps. However, the analysis targets not only the consideration of a given time horizon, as done by simulations, but that the system runs indefinitely. For Simulink models with loops corresponds the computation of a time step to the computation of a loop iteration for all loops in the model. For instance, the computation of all signal values for time step 0 corresponds to the computation of the first iteration of every loop in the model.

Widening is a well known abstraction procedure of values and time to analyze value ranges of variables in combination with loops. Therefore, during the analysis of the loop, a widening operator on the abstract domain overapproximates the computed reachable values [CC77]. Applying the widening operator, if chosen properly, will finally ensure that the analysis reaches a fixed-point regarding the reachable values of the loop and finally the entire system. When representing loops as a function $f(u,y)$, a fixed-point is a point y, for which $f(u,y) = y$ holds. This means, that further considered loop iterations (applications of $f(u,y)$) will not change the already computed reachable values y. We apply the widening approach as presented in [CC77] for the static value range analysis of program code to Simulink models. This has the particularity that a model might contain implicit loops introduced by Simulink-blocks of specific types. We call these loops implicit, since they are no loops which can be seen by looking on the connections between blocks in Simulink. Implicit loops are introduced by blocks which generate a non-constant outgoing signal based on no or constant input signals, e.g. Integrator-, Ramp- or Step-blocks. Figure 1 shows a simple example for a system with an implicit loop. Although, it is an open-loop system, we apply widening for the output of the integrator to ensure a correct overapproximation of the reachable values and the existence of a fixed-point for the model on the analysis domain. In case widening would not be used, the output of the Integrator-block would change for any considered timestep, a fixed-point would never be reached and the analysis would not terminate.

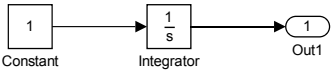

Fig. 1: Open loop system with implicit Simulink loop

A second particularity regarding widening and the associated detection of a fixed-point for Simulink models is the "'delay along paths"' (DAP). Assume a system containing a loop with a Delay-block. Such a block delays its incoming signal for some timesteps. The model shown in Figure 2, for instance, matches this description. The delay block delays the incoming signal for three timesteps, which means that its outgoing signal is by default zero for those timesteps. If widening is directly applied for the second loop iteration (second timestep), no overapproximation would be made and a fixpoint detected, because a further loop iteration does not change any computed reachable values. However, considering further iteration, the delay changes its outgoing values. Such fixpoint detection problems are avoided by calculating the maximal DAP, which is the maximal number of

timesteps (respectively loop iterations) a signal value needs to influence the value at the end of the path in the model, and apply widening on the abstract domain only in case enough timesteps have been considered. Because of the manifold ways to create dependencies between loops in Simulink models, we compute the maximal delay of the entire model as overapproximation of the DAP for single paths and use this value instead. Moreover, we take the maximal delay of the model into account for fixed-point detection to ensure, that no further iteration of any loop may change the computed reachable values. In case of the model from Figure 2, the maximal delay of the model is three and equal to the maximal delay of the path from the Constant- to the Outport-block.

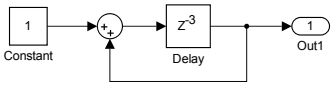

Fig. 2: Explicit loop with DAP from Constant to Out1 of three

4 Evaluation

The aim of the evaluation of our proposal is to show the feasibility on the one hand, while discussing drawbacks on the other hand. For the feasibility, the analysis has been executed on a computer with a 2.6 GHz 64 bit Intel i5 4-core CPU with 8GB memory running Windows 7 and 64 bit Java 8. With respect to the data, the evaluation is carried out on two models, both configured to a fixed step size of 0.01. The first model is a viscosity model and illustrated in Figure 3.

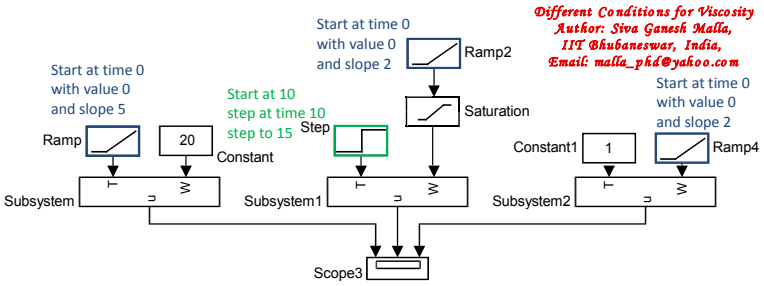

Fig. 3: Viscosity Model (taken from [Ma12]) with annotations

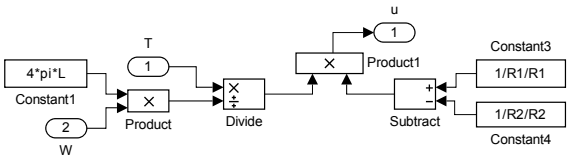

Fig. 4: Viscosity Model Subsystem (taken from [Ma12])

The internals of the models subsystems are identical and shown on Figure 4. However, the values for L, R1 and R2 are different from subsystem to subsystem, since they are configured using masks. Table 1 shows the values for each subsystem (rounded for better readability). The specific parameters for the Ramp-, Step- and Saturation- blocks are annotated in Figure 3.

	Subsystem	Subsystem1	Subsystem2
L	14e-3	14	14
R1	10.5e-3	10.5	10.5
R2	12e-3	12	12

Tab. 1: Mask parameters of the subsystems (Viscosity model)

The analysis of the viscosity model takes about 13.782 seconds and determines the reachable output values for the subsystems as given in Table 2. Potential problems, e.g. the occurrence of *NaN* as a value, of the model are indicated through warning messages. For instance, the analysis indicates a potential division by zero at the Divide-block in Subsystem1 and Subsystem2 and a potential division by infinity at the Divide-block in Subsystem2. This is accurate considering that the Ramp-blocks produce signals starting at zero and that the system runs for an infinite amount of time.

	Subsystem	Subsystem1	Subsystem2
Reachable values at outport	$\{[0;\infty]\}$	$\{[6.0478E-6;0.0060418],[\infty;\infty]\}$	$\{[0;6.0417E-4],[\infty;\infty]\}$

Tab. 2: Results of the value range analysis (Viscosity model)

The second model we chose for evaluation purpose is the "'Automotive Suspension"' [TM] model from the Simulink example models. This model contains 44 basic Simulink blocks of 10 different types (Sum, Gain, Step, Constant, Integrator, Mux, Demux, Subsystem, Outport, Inport). In contrary to the viscosity model, it contains feedback-loops that depend on each other. We replaced the continuous integrators of the model with discrete integrators, which use the forward euler method. That is because the support for different continuous integration solvers is not yet completely finished. Furthermore, we switched the "'acceleration due to gravity"' block, which was originally a Constant-block, to an Inport-block. This allows to vary the gravity depending on the considered location. Since our static analysis of the model operates on interval sets, we are able to analyze the model for arbitrary user specified value ranges of the gravity. We chose as input for the gravity the interval sets $A = [-9.81;-9.81]$, $B = [-9.80665;-3.71]$ and $C = [-9.82;-9.80]$. Those sets represent the commonly used value for gravity in our region (A), the range of gravity values above the zero level on earth [TT08] and mars[7] (B) and an arbitrary, but similar to A chosen range (C). Since we know that the model converges to a fixpoint for the constant -9.81, we perform the analysis without use of a widening operator, which yields a simulation of the model on the interval set domain terminating in case a fixpoint is reached. Regarding the results of the simulation with interval set input A we could determine that our results match exactly (using 64 bit double precision) the values computed by Simulink for every signal of the model at any considered time. This gives us a hint on the numerical compliance of our implementation with Simulink.

In general, we use widening to ensure the existence of a fixpoint for systems with loops [CC77]. Table 3 shows our results[8] using a simple widening procedure for the discussed

[7] `http://nssdc.gsfc.nasa.gov/planetary/factsheet/marsfact.html`

[8] We show the results as intervals instead of interval sets and rounded for better readability

Input Interval Set	Widening	THETA (Integrator out)	Z (Integrator out)	Time elapse(s)
A	OFF	$[-0.01098; 0.00862]$	$[-0.15372; 0.0]$	39.779
A	ON	$[-\infty; \infty]$	$[-\infty; \infty]$	31.772
B	ON	$[-\infty; \infty]$	$[-\infty; \infty]$	17.192
C	ON	$[-\infty; \infty]$	$[-\infty; \infty]$	17.570

Tab. 3: Results of the value range analysis (Automotive Suspension model)

inputs. The results indicate, that widening (if enabled) is applied before a fixpoint could be determined and thus a huge overapproximation is achieved. This overapproximation leads to warnings with false positive character indicating potential problems where none can occur. For instance, the possible occurrence of the value *NaN* as a result of the operation $[-\infty; \infty] + [-\infty; \infty]$ is indicated due to the overapproximation. Our future work focuses on approaches to reduce this amount of overapproximation for loops using alternative widening approaches specific to the static analysis of Simulink models, as other additional analysis domains.

5 Conclusion

This paper presents a formal approach to perform static value range analysis of functional block diagrams, in particular Matlab/Simulink, before generating code. The approach allows the direct detection of modeling flaws during the modeling phase, early in the development process. Based on user-provided value ranges for inputs, as demanded by ISO26262 [IS11], a safe value range of outputs is computed. Therefore, the abstract domain of interval sets, a more precise representation than the widely applied intervals, is used. We show, that the application of a type concept to this domain can cope very precisely with numeric issues, e.g. rounding modes, and type related over- and underflows. The evaluation of the approach shows that it is suited to detect potential errors as divisions by zero, infinity or occurrences of NaN. In the future, we will evaluate the value range analysis on more realistic examples. However, the use of interval sets can lead to similar large overapproximations as intervals in combination with widening for loops of the model. Thus, the approach is not sufficient to describe the behavior of a dynamic system, but indicates potential modeling flaws. The next step is to represent the relations between signals of the model, for example $u_0(t) > u_1(t)$, and the rates of the signals. With this information more precise conclusions, e.g. reduction of potential overapproximations, about the system can be drawn.

References

[BHB14] Bauch, Petr; Havel, Vojtech; Barnat, Jirí: Accelerating temporal verification of Simulink diagrams using satisfiability modulo theories. Software Quality Journal, pp. 1–27, 2014.

[BKS12] Biallas, Sebastian; Kowalewski, Stefan; Schlich, Bastian: Range and Value-Set Analysis for Programmable Logic Controllers. In: Proceedings of the 11th International Workshop on Discrete Event Systems. IFAC, Guadalajara, Mexico, pp. 378–383, 2012.

[Ca03] Caspi, Paul; Curic, Adrian; Maignan, Aude; Sofronis, Christos; Tripakis, Stavros; Niebert, Peter: From Simulink to SCADE/Lustre to TTA: A Layered Approach for Distributed Embedded Applications. In: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems. LCTES '03, ACM, New York, NY, USA, pp. 153–162, 2003.

[CC77] Cousot, P.; Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM Press, New York, NY, Los Angeles, California, pp. 238–252, 1977.

[CD06] Chen, Chunqing; Dong, JinSong: Applying Timed Interval Calculus to Simulink Diagrams. In (Liu, Zhiming; He, Jifeng, eds): Formal Methods and Software Engineering, volume 4260 of Lecture Notes in Computer Science, pp. 74–93. Springer Berlin Heidelberg, 2006.

[CM09] Chapoutot, A.; Martel, M.: Abstract Simulation: A Static Analysis of Simulink Models. In: Embedded Software and Systems, 2009. ICESS '09. International Conference on. pp. 83–92, May 2009.

[HJVE01] Hickey, T.; Ju, Q.; Van Emden, M. H.: Interval Arithmetic: From Principles to Implementation. J. ACM, 48(5):1038–1068, September 2001.

[IE08] IEEE: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008, pp. 1–70, Aug 2008.

[IS11] ISO: ISO 26262-6 - Road vehicles - Functional safety - Part 6 Product development software level. Technical report, Geneva, Switzerland, 2011.

[Ja12] Jaulin, L.; Kieffer, M.; Didrit, O.; Walter, E.: Applied Interval Analysis: With Examples in Parameter and State Estimation, Robust Control and Robotics. Springer London, 2012.

[Ma12] Malla, Siva: Calculation of Viscocity. 2012. `http://de.mathworks.com/matlabcentral/fileexchange/39413-calculation-of-viscocity`.

[NJC99] Nedialkov, N.S.; Jackson1, K.R.; Corliss, G.F.: Validated solutions of initial value problems for ordinary differential equations. Applied Mathematics and Computation, 105(1):21 – 68, 1999.

[RG14] Reicherdt, Robert; Glesner, Sabine: Formal Verification of Discrete-Time MATLAB/Simulink Models Using Boogie. In (Giannakopoulou, Dimitra; Salaün, Gwen, eds): Software Engineering and Formal Methods, volume 8702 of Lecture Notes in Computer Science, pp. 190–204. Springer International Publishing, 2014.

[Si08] Simon, A.: Value-Range Analysis of C Programs. Springer, August 2008.

[TM] The MathWorks, Inc.: Automotive Suspension. `http://de.mathworks.com/help/simulink/examples/automotive-suspension.html?prodcode=SL`.

[Tr05] Tripakis, Stavros; Sofronis, Christos; Caspi, Paul; Curic, Adrian: Translating Discrete-time Simulink to Lustre. ACM Trans. Embed. Comput. Syst., 4(4):779–818, November 2005.

[TT08] Thompson, Ambler; Taylor, Barry N.: Guide for the Use of the International System of Units (SI). National Institute of Standards and Technology Special Publication 811 (2008 Edition). National Institute of Standards and Technology / U.S. Department of Commerce, 2008.

[Wa98] Walster, G. William: The Extended Real Interval System. Technical report, 1998.