# Combining Abstract Interpretation with Symbolic Execution for a Static Value Range Analysis of Block Diagrams

Christian Dernehl(✉), Norman Hansen, and Stefan Kowalewski

RWTH Aachen University, Lehrstuhl Informatik 11 - Embedded Software, Aachen, Germany
{dernehl,hansen,kowalewski}@embedded.rwth-aachen.de

**Abstract.** This paper presents a fully automatic verification technique for Simulink block diagrams, by combining a static value range analysis with symbolic execution. Our concept avoids a translation to other languages and, instead, extracts all necessary attributes from Simulink and interprets the model directly. With this technique, we show how user defined specifications can be validated using sound abstractions for primitives, including IEEE-754 floats, and custom data types. Moreover, we propose optimizations by exploiting the benefits of intervals and symbolic representations to apply our technique to larger models. We evaluate our solution against an industrial tool.

## 1 Introduction

With the growing use of software controlled embedded systems, the safety of programs plays an increasing role. As projects and teams become larger and more interdisciplinary, model-based design tends to improve the development process [4]. Model-based design uses graphical programming, which is easily understood by developers from different domains. Another reason for model-based design is the attempt to limit the designer to rules and, eventually, avoid certain classes of software failures. This technique has been acknowledged in safety standards for embedded software systems [9,15].

The landscape of tools supporting model-based design differs between industrial sectors. For instance, Matlab/Simulink has become a widely applied tool in the automotive domain, while some aerospace businesses prefer SCADE [3]. Both tools provide the user with a visual modeling interface for *block diagrams*, in which elements are connected via lines and the flow among the blocks defines the behavior. This paper presents solutions for block diagrams in Simulink, however, the concepts can be adapted to similar modeling tools.

In practice, code is generated from existing models and integrated, automatically at best, into custom software. As a side effect, this process enhances rapid prototyping by allowing the user to simulate and test models on desktop computers, independent of the target platform. Admitting that errors caused by invalid memory access and wrong pointer arithmetic occur rather seldom in

© Springer International Publishing Switzerland 2016
R. De Nicola and E. Kühn (Eds.): SEFM 2016, LNCS 9763, pp. 137–152, 2016.
DOI: 10.1007/978-3-319-41591-8_10

generated code, many design issues remain. Among these are unintended data type over-/underflows, irrelevant or unused model parts, invalid divisions and operations, unintended variable resets and out-of-bound access.

Since the resulting code might lead to a failure, formal methods and extensive testing is applied for validation. Nevertheless, these techniques are often used after code generation, requiring a linkage between code and model, necessitating a regeneration of the code. Instead, our aim is to provide the user during the design stage with important notifications and warnings about potential modeling flaws, so that these can be resolved immediately.

*Contribution.* In this paper, we present a detailed, sound and fully automatic verification for Simulink block diagrams using *sat modulo theory* (SMT) techniques, which are introduced in Sect. 2. Our contribution in Sect. 4 extends already existing proposals by combining a previously designed interval analysis [8] with SMT checking. We use the Microsoft Z3 SMT solver [7], which is able to represent IEEE-754 floats with bit vectors as used by Matlab/Simulink. In detail, our algorithm identifies potential design errors, including divisions by zero, under- and overflows, infinite and NaN values, out-of-bound access and boolean signals, which are constant. We classify our work with others in the field, presented in Sect. 3, and evaluate our work against an industrial tool in Sect. 5.

## 2 Background

Before presenting our method, we elaborate briefly the concept of SMT solving and block diagrams, which has already been explained in previous work [14].

### 2.1 SMT Solver

Boolean expressions combine variables with logical operators, such as and ($\wedge$), not ($\neg$) and or ($\vee$). Each variable is either true or false and thus, the boolean expression evaluates either to true or false, depending on the variables. Solving such a boolean expression is the computation of an assignment for the variables so that the expression evaluates to true. Boolean expressions can be extended, for instance, by allowing arithmetic terms which provide a broader application range. With the combination of more underlying theories such as arithmetics, bit vectors, lists or floats, the decidability, i.e. searching for a satisfying variable assignment, of the expression cannot be guaranteed [13]. SMT solvers are tools trying to find satisfying variable assignments with regard to additional theories. Because of the potential undecidability, results of the solving procedure may be either satisfiable, unsatisfiable or unknown.

For our application, we have chosen the latest Microsoft Z3 SMT solver [7] and the support of multiple theories including IEEE-754 floating point arithmetic. The interface of the solver allows users to specify either variables or constants of a given *sort*, which can be boolean, integer, real, float, bit vector or others. For real and integer expressions, a finite rational number approximation

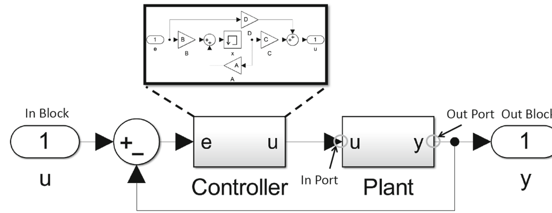


**Fig. 1.** Simulink block diagram example

is used, while floats are modeled as bit vectors, where each part, sign, fraction and exponent can be accessed individually. Since rational numbers behave differently than IEEE-754 floats, arithmetic operations are implemented individually, yielding the suited operation for a given sort.

One application of SMT is to interpret program variables as SMT variables and use the solver to prove properties of the code. For instance, it might be proven that a variable value is always different from 0 or does not exceed a certain range.

### 2.2 Block Diagram

A block diagram consists of ports $P$, blocks $B$ and lines $L$, such that $P$ and $L$ form a graph. Each port is assigned to a block, whereas, subsystems, which are blocks themselves, allow a hierarchical modeling. Since subsystems can be reused, complex systems can be constructed in a bottom up approach. Figure 1 illustrates a Simulink block diagram, with subsystems *Controller* and *Plant*, root input $u$ and root output $y$, respectively. The interface of a subsystem is specified with special *In*- and *Out*-blocks. Within the Controller subsystem, the block labeled $u$ references the in port of the Controller subsystem, whereas, $y$ references the out port.

While designing systems, a user may choose from a given palette of blocks or a set of his own. For instance, the blocks shown in Fig. 1 are all part of the Simulink standard block set. Masks are user interfaces, tied to a specific block, which allows further parametrization and configuration of the block, such as setting the value for *Constant*-blocks or changing the sign within a *Sum*-block. Finally, parameters may be set in the *model workspace*, which is a container of variables, with concrete values.

A *simulation* of a block diagram consists of a certain, possibly infinite, amount of time steps, such that each step has a certain, potentially varying duration, for example 0.01 s. Additionally, an individual sample time is assigned to each block, either automatically or specifically by the user, to model systems which run at different frequencies. Blocks are active, i.e. performing a computation, if a multiple of their sample time matches the currently considered time step. If a block is inactive, due to the sample time, a zero order hold operation

is performed by default[1]. Another way to make a block inactive is by creating an enabled subsystem in which the block resides. These special systems contain an enable port with a corresponding block, making all blocks in that subsystem active or inactive.

Since loops may be constructed in block diagrams, models with circular dependencies between blocks, called algebraic loops, can be created. For example, feeding the output of a *Sum*-block back to its input yields a time synchronization problem, because the output of the block in the current time step has to be computed based on the current input, which has not yet been calculated. To avoid the occurrence of algebraic loops, a block performing a time shift or storage operation should be added within the loop. These blocks, such as delays or integrators, have therefore a *state*. On the contrary, blocks without internal states feed their input through by computing a memoryless operation.

If not otherwise specified, all states of active blocks are updated with each time step, even those, which are not needed to calculate the root outputs. Thus, the control flow of a block diagram is linear, so that at a given decision, both paths of a *Switch*-block are evaluated by default. Unnecessary computations can be avoided by using enabled subsystems.

*Signals.* In each time step, signals flow along lines from port to port through the entire model. Each block represents a function, taking the signal values of its input ports as parameters and writing the computed result to its output ports.

Whereas blocks, ports and lines specify the syntax of the block diagram, the semantics are described by signals between ports. A signal has a name and a certain value for each time step. *Concrete* signals are represented by a tensor with a data type, which is equal for all elements in the tensor, ranging over multiple dimensions. Consider a signal of type int8 containing a $2 \times 2$ matrix, then each element in the matrix has type int8. Except for special cases, such as matrix multiplication, operations on signals are defined element-wise.

Signals with different data types can be combined into one signal using *buses*. A bus can be constructed in a hierarchical fashion, so that a bus may contain concrete signals or further bus signals. Before arithmetic and other element wise operations are performed on buses, all elements of the bus are casted to a single concrete signal of the most expressive data type. However, signal routing or *Memory*-blocks, which do not change the content of a signal, do not perform type cast operations or the conversion of bus signals to concrete signals.

*Execution Order.* Finally, after the structure of block diagrams and signals has been explained, the execution order schedules the simulation of the model. Suppose a system, consisting of a *Sum*-block with feed back through a *Memory*-block, which acts like a counter, depending on the input. In the first step, the *Memory*-block, with initial output value zero, must be computed before the *Sum*-block can calculate the sum over its inputs. Therefore, a sequence, guaranteeing all inputs being available when a block is executed, is necessary. Consequently,

[1] See http://de.mathworks.com/help/simulink/slref/ratetransition.html.

source blocks, such as *Constant*- or *Inport*-blocks, must be executed before their connected blocks can be executed based on the source blocks output. This order is given by *execution contexts*, which form an ordered tree structure, in which the leafs are non-subsystem blocks. At each level of the tree, the order set of children represents a valid schedule yielding an order in a top-down structure.

## 3 Related Work

Analyzing and verifying Simulink diagrams is a task which has been addressed before. Reicherdt and Glesner [14] present a similar approach and translate Matlab/Simulink models into the intermediate verification Boogie language. The subsequent verification relies on the Microsoft Z3 SMT solver. Our algorithm abstracts feed through and bounded blocks, such as $\sin, \cos, \arctan$ in a similar fashion. Their algorithm supports up to 44 blocks, however, their solution has some limitations regarding buses and the soundness, since corner cases for IEEE-754 floating point types and corresponding rounding methods are not considered. Although their solution incorporates intervals specified by the user, our algorithm utilizes the fully automatically calculated intervals from a static value range analysis. Eventually, Reicherdt and Glesner prove their solution to perform in certain aspects better than the Simulink Design Verifier[2], which is a tool to verify Matlab/Simulink models, by computing reachable values and detecting design flaws. The Design Verifier uses rational numbers, as indicated by the tool, and lacks a correct abstraction of IEEE-754 floats, too. Furthermore, there are many unsupported blocks by the Design Verifier, causing large over approximations and a variety of false positive results and even undetected flaws in the model. Other techniques, such as abstract simulation by Chapoutot et al. focus on numerical errors caused by continuous models [5]. Hence, we present a static value range analysis based on abstract interpretation [6] with symbolic execution to refine derived value ranges with regard to IEEE-754 floating point arithmetic.

Apart from the verification on model level, block diagrams can be translated to intermediate representations which can subsequently be analyzed. Tripakis et al. propose in their work [16] the translation of discrete Matlab/Simulink models to Lustre. Based on the resulting Lustre representation, verification techniques can be applied. Agrawal et al. [1] convert Matlab/Simulink block diagrams to hybrid automata, which are analyzed using domain specific methods. However, the approaches based on translation of block diagrams into different representations are in general only applicable for a subset of the available Matlab/Simulink model elements and functionalities.

## 4 Concept

In this section, we give a basic introduction into our approach for abstract interpretation of Matlab/Simulink models. Consequently, the construction of SMT expressions and use of symbolic execution is presented before the combination of both approaches, is described.

[2] See http://de.mathworks.com/products/sldesignverifier/.

*Overview.* Before discussing the concrete analysis, we highlight the construction of our intermediate representation. First, a simulation of the model is launched and paused to retrieve the compiled data types of ports, model parameters and the execution order, using the Matlab API. For block parameters, which are expressions such as $3 * x + 5$, a resolution is used which first looks up the mask parameters of parent blocks in the model for matching variables and replaces variable occurrences in the expressions recursively. Second, the model workspace, where the user may set parameters, is investigated. Finally, the expression is evaluated with the Matlab API, yielding a concrete value.

The block diagram itself is represented as a graph with ports and lines, with a linking between ports and blocks. For simplicity, we enrich the graph and connect *In*- and *Out*-blocks with the matching in- and out-ports along potentially multiple hierarchical levels in the diagram. With these lines, plain subsystems without further configuration can be omitted in the analysis. For enabled, triggered or other subsystems, each affected block references its enable blocks. Signals are abstracted and are either concrete, buses or variable size signals[3]. A signal flow analysis, based on a depth-first-search, computes the hierarchical structure of each bus in the model, providing each port with a primitive or bus type.

The blocks of the model are interpreted during analysis based on previously defined abstractions and corresponding model parametrizations. As abstract domain for the interpretation, interval sets are used.

*Limitations.* Although our technique can be applied to a variety of systems, we pose some limitations on the models. First, models must be updatable and compilable, i.e. a simulation must be carried out. Note, that our method does not rely on simulation results, but rather fetches the resolved data types and signal dimensions from the model. Systems may not contain algebraic loops, since those cannot be generally resolved and no code can be generated. Furthermore, our algorithm works for discrete models with a fixed time step solver. Currently, we have implemented abstractions for over 50 blocks supporting most possible configurations and parametrizations, including basic support for custom masked blocks. Blocks without correspondent abstractions are over approximated by default.

### 4.1 Abstract Interpretation with Interval Sets

Intervals provide means to define a set of values by two boundaries, making it an efficient representation. Arithmetics and all other operations, which can be expressed by Matlab/Simulink, can be adapted to intervals [2,12]. For instance, $[1, 2] + [3, 4]$ yields $[4, 6]$, which is the set of all possible sums between values $x, y$ with $x \in [1, 2], y \in [3, 4]$. We have shown in previous work [8] how interval sets can be used for abstract interpretation of Simulink models. In our implementation, which is reused in this work, the interval analysis is a sound abstraction of IEEE-754 floats, including rounding modes after each operation and

[3] Our algorithm currently does not support all variable size operations, which are allowed by Simulink.

symbols such as $\pm\infty$ and NaN. Thus, our implemented abstract interpretation yields an interval set for each port and internal state of a block for the entire model. Although interval sets provide an efficient method for analyzing large models, interval sets lack, the capability to represent relations among multiple variables [8].

### 4.2 Symbolic Execution with SMT

After having referenced how abstract interpretation using interval set domains is carried out, we explain how SMT expressions, describing relations among signals of a model, are constructed from block diagrams.

*Types and Casting.* For the Z3 SMT expressions, there are three different sorts, *boolean*, *integer* and *float*, which we use in our algorithm. The boolean($\mathbb{B}$) and floating point ($\mathbb{F}_{32}, \mathbb{F}_{64}$) data types from Simulink can be directly mapped to the corresponding SMT sorts, while the wrapping effects or configurable saturation of integers ($\mathbb{I}_n, n = 8, 16, 32$) must be treated by adding modulo operations or respectively constraining the value. For instance, $u_0 + u_1$ becomes $(u_0 + u_1)$ mod $2^{16}$ if the types are uint16. The additional mod operation, which has to be added to ensure correct type behavior after every operation, increases complexity of the SMT expressions across the model and can be omitted if the intervals prove, that no under- and overflow occurs.

For floating point operations, a rounding mode according to IEEE-754 must be specified, such as towards zero or $\pm\infty$. By introducing a new variable $z$, the plus operation between float expressions $x$ and $y$ can be precisely specified by adding a global statement

$$\bigvee_{r \in R} z = \text{plus}(x, y, r) \tag{1}$$

where $R$ is the set of rounding modes. However, this approach leads to large SMT expressions and increases the number of variables and computation effort significantly. Therefore, we allow the user to specify a concrete rounding mode, which is used for all floating point operations. This approach seems reasonable, when assuming that neither Simulink nor external code changes the rounding mode of the floating point unit.

Type casts are handled in different fashions, depending on the input and the output type. Table 1 shows an overview of type casts, where $\varphi$ is the expression, which shall be casted. If both types are the same, the casting operation is ignored, yielding the input. Boolean casts are represented by the *if-then-else* (ite) operator, yielding one or zero depending on the boolean value. The ite is a ternary operator, so that the first argument is a boolean statement and the latter ones are the results, depending on the first argument. Casting integers or floats to boolean is modeled by setting the expression unequal to zero $\neg(\varphi = 0)$. Suppose both are integer types and the source type is larger than the destination type, then the implicitly added mod operator takes care of the overflow effect.

**Table 1.** Type casts as SMT expressions

| From | To | SMT expression | From | To | SMT expression |
|---|---|---|---|---|---|
| $\mathbb{B}$ | $\mathbb{I}$ | $\text{ite}(\varphi, 1, 0)$ | $\mathbb{I}$ | $\mathbb{B}$ | $\neg(\varphi = 0)$ |
| $\mathbb{B}$ | $\mathbb{F}$ | $\text{ite}(\varphi, 1.0, +0.0)$ | $\mathbb{I}_a$ | $\mathbb{I}_b$ | $\varphi \mod b$ if $b > a$, no op otherwise |
| $\mathbb{F}$ | $\mathbb{B}$ | $\neg(\varphi = 0)$ | $\mathbb{I}_{32}$ | $\mathbb{F}_{32}$ | bit vector to fraction if $\lvert\varphi\rvert \leq 2^{23}$ |
| $\mathbb{F}$ | $\mathbb{I}$ | via rational and modulo | $\mathbb{I}_{<32}$ | $\mathbb{F}_{32}$ | via bit vector to fraction |
| $\mathbb{F}_{64}$ | $\mathbb{F}_{32}$ | fresh variable | $\mathbb{I}$ | $\mathbb{F}_{64}$ | via bit vector to fraction |
| $\mathbb{F}_{32}$ | $\mathbb{F}_{64}$ | copy bit vector parts | | | |

Assume the Matlab/Simulink setting *saturate on integer overflow* is active, then the modulo operation is substituted by two ite operations

$$\text{ite}(\varphi > \max, \max, \text{ite}(\varphi < \min, \min, \varphi)) \tag{2}$$

which restrict $\varphi$ to the range $[\min, \max]$.

For the special case, in which a 32-bit integer is casted to a 32-bit float, several steps are taken. First, if the calculated interval proves the maximum absolute value below $2^{23}$, then the expression fits into the fraction of a 32-bit float, since the other eleven bits are used for the exponent. In this case, we can copy the bit vector representation of the integer into the fraction and set the exponent to the bias value, i.e. such that the value of the exponent is zeroed, yielding only the fraction. If the 32-bit integer exceeds $\pm 2^{23}$, then we create a fresh variable, since we cannot make any assumptions on the typecast. For 64-bit floats, no check needs to be done, since the largest integer in Simulink with 32 bits fits into the fraction of 52 bits.

In addition to potential overflows, casts from floating point to integers need special treatment for NaN and $\pm\infty$ symbols. We wrap the expression

$$\text{ite}(\text{isNaN}(\varphi), 0, \text{ite}(\text{isInf}(\varphi), \text{ite}(\text{sgn} > 0, \max, \min), \mathbb{I}(\mathbb{Q}(\varphi)))) \mod 2^n \tag{3}$$

around, where isNaN is translated to a bit vector operator by the SMT solver. The integer cast $(\mathbb{I}(\mathbb{Q}(\varphi)))$ is executed by a detour to rational number abstraction $(\mathbb{Q})$, which is then translated to an integer using the modulo operation. Similarly to the NaN check, we map the $\pm\infty$ symbols to either the minimum or maximum value of the integer, if the saturation option is activated. Thus, if $\infty$ is casted to an 8-bit integer, the result is 127. In case the saturation option of the block performing the type cast is deactivated, the first part of the isInf check is set to zero, so that a cast from $\pm\infty$ results in zero, as it does in Simulink. Consequently, the fraction and exponent are represented by rational number, which is then rounded to an integer with the configured mode. Nevertheless, a major drawback is that the bit vector constraints, which are attached to the float expression, must be propagated, causing a large overhead. Therefore, we deactivate the propagation for better efficiency, with the cost of an additional over approximation.

Finally, if both types are different floats, and the input is $\mathbb{F}_{32}$, a few checks are made. First, a fresh $\mathbb{F}_{64}$ variable is constructed and optionally constraints with respect to NaN, $\pm\infty$ are added, if the input variable cannot reach these values. This is an approximation, since no $\mathbb{F}_{32}$ except $\infty$ yields a $\mathbb{F}_{64}$ $\infty$, which holds for -$\infty$ and NaN, too. Furthermore the bit vectors from the $\mathbb{F}_{32}$ are taken into consideration, when the $\mathbb{F}_{64}$ variable is created. This procedure cannot be used the other way around, except for NaN, since certain non $\infty$ $\mathbb{F}_{64}$ values yield an $\infty$ $\mathbb{F}_{32}$ value. Furthermore, since the type and bit vectors are larger, this data cannot be copied without loss of information. Thus, a new variable with the optional NaN constraint, is created.

$$\text{ite}(\text{isNaN}(\varphi), \text{NaN}, \text{ite}(\text{isInf}(\varphi), \text{sgn}(\varphi)\infty, \varphi) \quad (4)$$

*Block Functions.* Block operations, as long as they are supported, are mapped to the according SMT operation. In addition, Simulink adds implicit type casts, when necessary. For example, if two float signals are connected to a logical operator block, an implicit cast to boolean is executed. All supported blocks perform the implicit casting operation.

Arithmetic operators are mapped to the corresponding integer or floating point, i.e. bit vector implementation. For integers, the absolute value function is mapped to the $\text{ite}(\varphi < 0, -\varphi, \varphi)$ expression, while for floating point types the sign bit is adjusted. Both, power and modulo can be expressed with integer expressions, while for floats a fresh variable is created, over approximating those functions. Finally, the square root on integers is expressed by a fresh variable $z$ and adding $\exists z\ .zz = \varphi$ to the global constraints.

Functions, which are not supported by the SMT solver, such as transcendentals, are mapped to anonymous functions, allowing the solver to choose any value of the associated sort. For example, such an over approximation is applied for trigonometric or exponential functions. In addition, constraints for bounded functions, such as sine, are added to a list of global constraints, which is included, when the solver is invoked. Concretely,

$$\bigwedge_{\substack{z\in\{\pm\infty,\\ \text{NaN}\}}} \sin(z) = \text{NaN} \wedge \forall x\ \neg(x = \pm\infty \vee x = \text{NaN}) \rightarrow \sin(x) \leq 1 \wedge \sin(x) \geq 1 \quad (5)$$

expresses the limitation of the sine function except for $\pm\infty$ and NaN as arguments. Analogous expressions can be constructed for cos and arctan. For unsupported blocks, fresh variables, which can take any value, are created. However, since a data type is assigned to each port, further global constraints for the fresh variable are added, in case of an integer type. For instance, if the output is an uint8 data type, global constraints $y_0 \geq 0 \wedge y_0 \leq 2^8 - 1$ can be added for the fresh variable $y_0$.

Unlike intervals, which loose information about the control flow and relations between values, SMT expressions abstract switches by the ite operation, keeping the condition and relation between variables available for further analysis. This memory of the formula is expressed by an abstract syntax tree (AST), which is

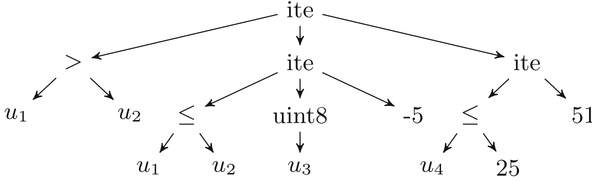


**Fig. 2.** Abstract syntax tree example

constructed internally by the Z3 solver. An illustration of an AST is given in Fig. 2, where each node represents an operator and the children the operands. There is the ite ternary operator, binary relational operators($>$, $\leq$) and an unary cast operation. Suppose the ite is created from a switch block, then during the evaluation of the root, no decision between the nodes uint8($u_3$) and $-5$ has been made. Consider further, the expression

$$\text{ite}(u_1, \text{ite}(\neg u_1, u_2, u_3), u_4) \quad (6)$$

where the result is never $u_2$, since $u_1$ cannot be true and false at the same time. This is because, if $u_1$ would be true, then the result would be $u_3$, if $u_1$ is false, then the result is $u_4$. A similar problem exists in the more complex tree in Fig. 2, between the root and the second child, because the cast to $u_3$ is never the result.

Sometimes SMT formulas can be theoretically constructed for blocks, but in practice this is not feasible. Consider a large lookup table, which can be expressed by a nested combination of ite operators. For each point in the lookup table, a new ite expression is constructed, yielding a large overhead. In this case, our implementation omits the computation of the SMT expression and yields a fresh variable with additional constraints. These constraints are taken from the interval analysis to limit the created variable.

Finally, the selector block allows the user to select a subset from an input vector or matrix, such as the first two entries of a vector of length 20. Additionally, the selection can be specified by a signal, which potentially changes with each time step. Furthermore, if the selection signal is above 20, an unintended data access occurs. Our verification solution adds constraints to verify, that the selection signal stays within bounds.

*Model Sources.* Inputs to the entire system are expressed by *In*-blocks at root level, which receive a special handling, since other *In*-blocks are virtually connected to the corresponding *Out*-blocks during the enrichment of the model. For each scalar in each root *In*-block, a new variable is created $u_0, \dots u_n$, such that the SMT sort is chosen based on the data type. Constants with an associated data type, such as block parameters, are expressed by the corresponding SMT constants. Since the inputs may change over time, new variables for the model sources have to be created for each time step.

*Path Encoding.* With the generation of symbolic expressions for single blocks given, this technique can be adapted to entire block diagrams. The execution

order from Simulink is used as a schedule for the symbolic execution, such that for each block a symbolic output expression can be computed based on the symbolic input expression. For blocks with internal states, such as integrators, delays or others, symbolic representation is computed, too.

### 4.3 Combining Abstract Interpretation with Symbolic Execution

Our technique, based on abstract interpretation, terminates if a fix point for the model is found, i.e. any further considered time step does not change the computed reachable values for any signal [6,8]. Assuming an infinite run time, approximations need to be made for blocks with states and models with loops to enforce the existence of a fix point, since our analysis would potentially not terminate otherwise. After a specified time horizon, states, for which no fix point was reached yet, are widened to the biggest interval for the states type. Since each loop contains at least a block with a state, a fix point for every loop is eventually reached.

By comparison, widening for SMT formulas is hard, if no fresh variable without any constraints is used [11]. To avoid the construction of symbolic expressions over multiple considered model time steps, we introduce new variables where constructed expressions would depend on expressions of a previous time step, which is the case for blocks with states. However, our technique exploits the computed intervals and uses the interval constraints for the further analysis. For example, if an interval is only positive ($[0, \infty)$), the constraint $v_0 \geq 0$ is added for the newly created variable $v_0$.

Nevertheless, the calculation and evaluation of SMT operations takes the largest share of the computational time. Therefore, we add further optimizations, so that information among the two domains is shared. First, before a boolean signal is evaluated by the SMT solver, the interval is checked, whether it actually contains both boolean values and may thus be shrinked by evaluating the SMT expression. This improvement provides a performance benefit, especially for variant models, in which constant booleans are used to configure a variant. A similar enhancement is applied during the evaluation of divisions. First, the interval is evaluated if it contains zero and then the SMT solver is invoked to find a solution, yielding a division by zero.

Vice versa, optimizations from the SMT solver are feed back to intervals. First, if the SMT solver proves or disproves a boolean formula, the interval is adjusted and the according constant is removed. Second, if a divisor is proven to be non-zero, the zero is removed from the interval, before the operation is carried out, which reduces the number of potential division by zero warnings.

Since formulas grow along paths, we evaluate each boolean expression along the path to potentially decrease the size of the formula. In this way expressions like $x \wedge \neg x$ are reduced to zero, which is the signal value being passed to the next ports.

## 5 Evaluation

After having presented, how we combine abstract interpretation with interval sets and symbolic execution with SMT to perform value range analysis on block diagrams, we demonstrate the performance and discuss benefits and drawbacks of our approach. We compare results of our analysis with results of the MathWorks Simulink Design Verifier (SLDV)[4], which performs a value range analysis on Simulink models, too. We have applied our methods to several block diagrams to highlight the main aspects of our technique. Table 2 provides an overview of all models which have been evaluated. The second and third column in Table 2 lists blocks and lines, while the fourth column presents the number of virtual blocks[5], i.e. blocks performing no operation and serving only as visual aids. Structural properties of the systems are given in the last three columns. The fifth column lists the number of subsystems, including conditional and atomic ones, while column six denotes the hierarchy, i.e. the maximum number of nested subsystems. Finally, the last column determines, whether the system is a feedback systems and contains a loop.

**Table 2.** Model metrics

| Model | Blocks | Lines | Virtual blocks | Subsystems | Hierarchy | Closed loop |
|---|---|---|---|---|---|---|
| ABS Brake | 48 | 50 | 16 | 5 | 2 | yes |
| Quarter Car | 57 | 70 | 11 | 3 | 2 | yes |
| Suspension | 46 | 55 | 13 | 3 | 1 | yes |
| DAS | 970 | 915 | 562 | 189 | 13 | yes |
| 8 Bit Counter | 190 | 213 | 126 | 20 | 4 | no |

The first three systems represent applications from the automotive domain and contain continuous blocks, which are not supported by the SLDV. Therefore, we have discretized the systems with MathWorks model discretizer, using Tustins method [10] (trapezoidal integration). Further adaptions, which are described below, were necessary for some models. The ABS Brake represents an anti-lock braking system and is taken from the Simulink examples[6]. In addition to the discretization, we replaced the stop simulation block with an *Out*-block yielding a feasible model for the SLDV. Given our current implementation, we had to exchange the user defined function block, computing the relative slip[7] by $1 - u_1/u_2$ with a subsystem, an addition and division block, respectively. The Quarter Car (we analyzed not the entire system) and Suspension systems, both

[4] See http://www.mathworks.com/products/sldesignverifier/.

[5] See http://mathworks.com/help/simulink/ug/nonvirtual-and-virtual-blocks.html.

[6] See http://de.mathworks.com/help/simulink/examples/modeling-an-anti-lock-braking-system.html.

[7] In case $u_2 = 0$, $\varepsilon$ is used, which is considered in our verification.

model vehicle suspension in the automotive domain and are taken from the Simulink examples and Matlab Central[8]. To analyze both models, no further modification, except the discretization, was necessary. As a pure discrete system, the 8 Bit Counter, taken from Matlab Central and extended by an additional *Out*-block, is being evaluated. Due to the nature of the hardware related model, the entire system consists mainly of *Memory*- and *Truth Table*-blocks. The DAS model is an industrial example of an assistance system from the automotive domain.

Our evaluation platform is a computer with an Intel i5 2.67 GHz CPU, eight gigabytes memory, with a 64-bit Windows 7 operating system and Matlab 2015b. The logged analysis times of the SLDV exclude the duration of model compilation and translation to the internal intermediate representation. Thus, we exclude for our algorithm the time for starting Matlab, loading the model and translating it to our intermediate abstract block diagram representation. Additionally to time elapse for analysis, we compare the number and type of issued warnings.

**Table 3.** Analysis results

| Model | Time (s) | | | Warnings | | |
|---|---|---|---|---|---|---|
| | SMTR | SMTF | SLDV | SMTR | SMTF | SLDV |
| ABS Brake | 4.838 | 81.777 | 102 | 30 (1) | 30 (1) | 4 |
| Quarter Car | 1.315 | 1.255 | 11 | 35 (0) | 35 (0) | 2 |
| Suspension | 28.483 | 30.388 | 12 | 83 (0) | 83 (0) | 1 |
| DAS | 37.658 | 1317.478 | 75 | 225 (5) | 225 (5) | 31 |
| 8 Bit Counter | 44.273 | 32.165 | 44 | 97 (0) | 97 (0) | 12 |

In Table 3, an overview of the comparison is given. On the left part of the table, the time elapse is denoted, while the right part contains the number of warnings issued by each algorithm, including symbolic execution with reals (SMTR), floats (SMTF) and the Simulink Design Verifier (SLDV). Since the SLDV uses rational number approximations and no IEEE-754 floats, we extended the evaluation by adding the symbolic execution with reals, to highlight the computational cost for sound floating point abstractions. Hence, our solution issues warning types, which are not considered by the SLDV, such as potential NaN values or implicit rate transitions. Consequently, our algorithm issues more warnings on all chosen models. Therefore, we indicated the number of warnings, excluding warning types not being issued by the SLDV, in parenthesis.

As expected, because of the more complicated theory used for SMTF, the time elapse using SMTR is for most models lower. Moreover, SMTF scales worse

[8] See http://de.mathworks.com/matlabcentral/.

than SMTR to larger models. Comparing SMTR to SLDV, it can be noticed, that SLDV is faster analyzing the Suspension model. Regarding the issued warnings, SMTR and SMTF differ for no evaluated model, which we find to be plausible. This is due to the fact, that no model was constructed, using IEEE-754 specific blocks, such as isNaN or isInf. In addition, no model was chosen which exploits differences between SMTR and SMTF. The SMTR yields also NaN warnings, because IEEE-754 operations are also covered by the interval sets. However, differences between the number of warnings of our approach and the SLDV will further be discussed for the evaluated models.

The main difference concerns the *Result could be NaN warning* which is non-existent for SLDV. However, SLDV computes for many signals the same reachable values as SMTF/SMTR, which are often $(-\infty; \infty)$ due to overapproximations and thus could lead to a NaN result in case the sum of two signals with reachable values $(-\infty; \infty)$ is computed, as in many of the benchmark models. The further discussion will focus on warnings which are supported by both approaches. SLDV detects for the ABS Brake model, two *potential divisions by zero* (DbZ) and two potential *data type overflows* (DTO). Using SMTF/SMTR, we were able to avoid the detection of three false positives to only one potential DbZ warning. Regarding the Quarter Car model, SLDV issues two DbZ warnings. However, these result from constant values which were different from zero at the time of analysis and were thus not detected by SMTF/SMTR. The Suspension model causes SLDV to detect a false positive DTO warning. The SMTF/SMTR warnings, however, are limited to potential NaN and *implicit rate transition* warnings.

Comparing the warnings and computed reachable values for the DAS model, we detected that SLDV is in general less overapproximative regarding data stores (which are not yet supported by our tool) and triggered subsystems. However, SLDV issues 19 DbZ and 12 DTO warnings which are caused by lookup tables and divisions by constants or constant signals and data types as float64 which may not overflow. Besides warnings for unsupported features, NaN occurrences and implicit rate transitions, we were able to detect two paths in the model which do not contribute to any model result. Furthermore a violation of the specified design ranges has been detected, which might be a false positive warning.

For the 8 Bit Counter model, SLDV issues 12 overflow warnings. However, these relate to signals which are either of integer types or of boolean type as a result of a lookup. Inspecting the correspondent blocks and paths of the model, these warnings can be identified as false positives and are, furthermore, not detected using SMTR/SMTF.

## 6 Conclusion

This paper presented the combination of abstract interpretation with symbolic execution based on SMT with IEEE-754 floating point arithmetic for static value range analysis of block diagrams. The evaluation of the presented approach against an industrial state of the art tool showed, that the industrial tool scales

better to large models regarding the time elapse for analysis. However, it is not able to detect IEEE-754 related modeling flaws, such as potential occurrences of NaN or correct handling of infinity values due to the used rational number approximation. Moreover, we were able to show that our presented approach is able to reduce the number of false positives regarding warnings for potential overflows and divisions by zero, compared to the industrial tool.

Future work will focus on extending the support of Simulink features, e.g. Stateflow which is only partly supported yet, and to reduce over approximations for special system classes.

## References

1. Agrawal, A., Simon, G., Karsai, G.: Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. Electron. Notes Theor. Comput. Sci. **109**, 43–56 (2004)
2. Alefeld, G., Mayer, G.: Interval analysis: theory and applications. J. Comput. Appl. Math. **121**(12), 421–464 (2000)
3. Bochot, T., Virelizier, P., Waeselynck, H., Wiels, V.: Model checking flight control systems: the airbus experience. In: ICSE Companion (2009), pp. 18–27 (2009)
4. Broy, M., Kirstan, S., Krcmar, H., Schätz, B., Zimmermann, J.: What is the benefit of a model-based design of embedded software systems in the car industry? In: Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications, p. 310 (2013)
5. Chapoutot, A., Martel, M.: Abstract simulation: a static analysis of simulink models. In: International Conference on Embedded Software and Systems, 2009. ICESS 2009, pp. 83–92, May 2009
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252 (1977)
7. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Dernehl, C., Hansen, N., Kowalewski, S.: Static value range analysis for Matlab/Simulink-models. In: 13. Workshop Automotive Software, INFORMATIK 2015, pp. 1649–1660 (2015)
9. ISO: ISO 26262–6 - Road vehicles - functional safety - Part 6 product development software level. Technical report, Geneva, Switzerland (2011)
10. Korlinchak, C., Comanescu, M.: Discrete time integration of observers with continuous feedback based on Tustin's method with variable prewarping. In: 6th IET International Conference on Power Electronics, Machines and Drives (PEMD 2012), pp. 1–6. IET (2012)
11. Leino, K.R.M., Logozzo, F.: Using widenings to infer loop invariants inside an SMT solver, or: a theorem prover as abstract domain. In: Workshop on Invariant Generation, pp. 70–84 (2007)
12. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. Society for Industrial and Applied Mathematics, Philadelphia (2009)

13. de Moura, L., Bjørner, N.: Satisfiability modulo theories: an appetizer. In: Oliveira, M.V.M., Woodcock, J. (eds.) SBMF 2009. LNCS, vol. 5902, pp. 23–36. Springer, Heidelberg (2009)
14. Reicherdt, R., Glesner, S.: Formal verification of discrete-time MATLAB/Simulink models using Boogie. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 190–204. Springer, Heidelberg (2014)
15. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5), 19–25 (2003)
16. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time Simulink to Lustre. ACM Trans. Embed. Comput. Syst. (TECS) **4**(4), 779–818 (2005)