Abstract Interpretation of MATLAB Code with Interval Sets

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

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

Abstract. In this paper we present how formal methods can be applied to MATLAB programs. We apply a static analysis based on abstract interpretation to derive reachable values and identify potential programming faults fully automatically. Our verification is built on a formalization and abstraction of matrices, structures and data types, such as integers and IEEE-754 floats. Combined with previously presented static analysis for Simulink, our tool can verify block diagrams with embedded MATLAB code. We show the feasibility of our tool and compare our solutions against a commercial tool, using real world applications.

1 Introduction

Nowadays, embedded systems are used in a variety of applications, automating many processes, which have been done manually before. Due to their strong links to the environment, embedded applications bear strong safety requirements. Software failures may cause harm to humans, the surroundings and economic success. Avoiding errors is crucial when selling cutting edge technology, which is often designed using model-based approaches. The use of mathematical models for the environment allows to test and improve the behavior of systems. Different domain specific languages, such as Mathematica, Maple, MATLAB, Modelica, SCADE, SciLab, Octave and more can be used for mathematical programming. Since MATLAB is a widely used tool in the automotive domain [8] and used to develop safety critical software, we focus on the identification of potential modeling/programming flaws of MATLAB programs.

MATLAB is both a programming language and an interpreter. In this paper, we generally mean by MATLAB the language unless stated otherwise. Rapid prototyping is supported with many built-in functions for specific domains, such as controls. Similar to other programming languages, primitive data types, custom structures and objects can be designed and reused. Control flow statements include function calls, if-else and switch branches, for and while loops and exit jumps, i.e. return statements.

In previous work [3], we have presented how to apply formal methods to block diagrams, designed with Simulink. We have used abstract interpretation with interval sets to prove the absence of certain errors, such as division by zero,

M.H. ter Beek et al. (Eds.): FMICS-AVoCS 2016, LNCS 9933, pp. 25–38, 2016.
DOI: 10.1007/978-3-319-45943-1_2

under-, overflows and dead paths. Thus, our proposal for code analysis intends to improve our block diagram verification, since Simulink allows the user to specify custom blocks with MATLAB code. Our technique has been implemented in the *artshop*[1] framework, which is a model repository and provides external tool adapters, different analyses and reporting capabilities.

1.1 Contribution

After the presentation of related work in Sect. 2, we formalize the syntax and concrete semantics of MATLAB programs in Sect. 3. Subsequently, Sect. 3 introduces the formal description of the abstraction and the interpretation rules. Section 4 evaluates the presented approach by comparison with an industrial tool before Sect. 5 concludes our work.

2 Related Work

Static Analysis and Abstract Interpretation. Static analysis describes procedures to analyze programs or models without execution and derive properties automatically. In this work, we focus on abstract interpretation [1] to compute reachable value ranges for variables at each program location. Over the last years, much research effort has been gone to proposing new abstractions, such as intervals [1], relational domains [13,14], congruences [9], digital filters [7] and more. In abstract interpretation, domains and corresponding operations are defined to create a mapping between concrete and abstract program states. With the abstractions, it may be easier to prove properties in the abstract domain, which can be transfered back to the concrete program. Suppose intervals as an abstraction and if there is no division by zero in the abstract domain, then there is none in the concrete domain. Finally, abstract interpretation is a formal methods, which has been applied to several large scale industry projects [2,17].

Analysis of MATLAB Code. To our knowledge, abstract interpretation with a value range analysis has not been applied to MATLAB code to perform a value range analysis so far, however several static analyses for MATLAB programs have been investigated. In 2003 Elphick et al. [6] proposed a static analysis of MATLAB code to derive types and dimensions of variables to improve computational speed. Doherty et al. [5] investigated later in 2011 the mapping of identifiers to functions and variables. Aiming at a formal verification of MATLAB programs, Lu and Mukhopadhyay [12] designed one year later an algorithm to transform MATLAB code into SAT modulo theory (SMT) for formal verification.

3 Abstract Interpretation of MATLAB

First we present briefly the syntax and semantics of MATLAB code. Afterwards, abstractions for variables and control flow are shown.

[1] See https://artshop.embedded.rwth-aachen.de/.

3.1 Syntax and Concrete Semantics

Similar to other programming languages, MATLAB code consists of function definitions, expression statements and control flow commands. Table 1 presents a simplified grammar, describing the syntax of MATLAB programs. Several implementation specific parts have been simplified, bitwise and postfix operators, while others have been omitted, such as sub reference expressions $expr_1(expr_2)$ for array access or function calls. Additionally, we assume for brevity that each statement is terminated with a semicolon, although statements can also be terminated with an end of line symbol. Furthermore, we have omitted that boolean, numeric and matrix expressions can be constructed by function calls. Nevertheless, all these constructs have been considered in our implementation.

Expressions can be constructed using unary ($\blacklozenge$), binary ($\lozenge$), relational ($\bowtie$) and logical (logOp) operators. Note, unlike C, MATLAB uses the () operator for both array access and function call, with array access having precedence. Additional to scalar values, MATLAB allows matrix and tensor variables. In fact, each scalar variable is treated as a 1x1 matrix. Further operators specifically for matrices, such as matrix multiplication, inversion and least square (RDIV) are available. Element wise binary operations can be specified with a dot prefix, i.e. `A.*B` multiplies matrices `A` and `B` element wise.

Table 1. Simplified excerpt of the MATLAB program grammar

$$
\begin{array}{ll}
\langle program \rangle & = \langle stmts \rangle \\
\langle stmts \rangle & = \textbf{\{}\ \langle stmt \rangle\ \text{';'}\textbf{\}} \\
\langle stmt \rangle & = \langle expr \rangle \mid \langle conditional \rangle \mid \langle assignment \rangle \mid \langle loop \rangle \\
\langle expr \rangle & = \langle boolExpr \rangle \mid \langle numericExpr \rangle \mid \langle matrixExpr \rangle \\
\langle boolExpr \rangle & = \langle primaryExpr \rangle \mid \langle numericExpr \rangle \bowtie \langle numericExpr \rangle \\
 & \quad \mid \langle boolExpr \rangle\ \text{logOp}\ \langle boolExpr \rangle \\
\langle numericExpr \rangle & = \langle boolExpr \rangle \mid \langle numericExpr \rangle \lozenge \langle numericExpr \rangle \mid \blacklozenge\ \langle numericExpr \rangle \\
\langle matrixExpr \rangle & = \text{'['}\ \textbf{\{}\ \textbf{\{}\ \langle expr \rangle\ \text{','}\ \textbf{\}}\ \text{';'}\ \textbf{\{}\ \langle expr \rangle\ \text{','}\ \textbf{\}}\ \textbf{\}}\ \text{']'} \mid \\
 & \quad \langle expr \rangle\ [\ \text{':'}\ \langle expr \rangle\ ]\ \text{':'}\ \langle expr \rangle \\
\langle assignment \rangle & = \langle identifier \rangle\ \text{'='}\ \langle expr \rangle, ; \\
\langle varList \rangle & = \text{'['}\textbf{\{}\ \langle variable \rangle\ \text{','}\ \textbf{\}}\ \text{']'} \\
\langle multiAssign \rangle & = \langle varList \rangle\ \text{' ='}\ \langle expr \rangle \\
\langle conditional \rangle & = \textbf{if}\ \text{' '}\ \langle boolExpr \rangle\ \langle expr \rangle\ [\ \textbf{\{}\ \textbf{elseif}\ \text{'}\langle boolExpr \rangle\ \langle expr \rangle\ \textbf{\}}\ ] \\
 & \quad [\ \textbf{else}\ \text{' '}\ \langle expr \rangle\ ]\ \text{' '}\ \textbf{end} \\
\langle loop \rangle & = \langle while \rangle \mid \langle for \rangle \\
\langle while \rangle & = \textbf{while}\ \text{' '}\ \langle boolExpr \rangle\ \textbf{\{}\ \langle stmts \rangle\ \textbf{\}}\ \text{' '}\ \textbf{end} \\
\langle for \rangle & = \textbf{for}\ \text{' '}\ \langle variable \rangle\ \text{'='}\ \langle matrixExpr \rangle\ \textbf{\{}\ \langle stmts \rangle\ \textbf{\}}\ \text{' '}\ \textbf{end} \\
\langle primaryExpr \rangle & = \langle identifier \rangle \mid \langle constant \rangle \\
\langle functiondef \rangle & = \textbf{function}\ \text{' '}\ \langle varList \rangle\ \text{' ='}\ \text{'}\langle funName \rangle\ \text{'('}\ \textbf{\{}\ \langle variable \rangle\ \text{','}\ \textbf{\}}\ \text{')'} \\
\langle \bowtie \rangle & = \text{'>'} \mid \text{'>='} \mid \text{'<'} \mid \text{'<='} \mid \text{'=='} \mid \text{'}\sim\text{='} \\
\langle \lozenge \rangle & = \text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'/'} \mid \text{'\textbackslash'} \mid \text{'\^{}'} \mid \text{':'} \mid \text{'.'}\ \langle \lozenge \rangle \\
\langle \blacklozenge \rangle & = \text{'+'} \mid \text{'-'} \\
\langle logOp \rangle & = \text{'\&'} \mid \text{'|'} \mid \text{'}\sim\text{'}\ \langle logOp \rangle
\end{array}
$$

Limitations. Our intended use of the static value range analysis for MATLAB code is the abstraction of embedded MATLAB-function blocks[2] in Simulink. Simulink constraints the use of MATLAB code within embedded MATLAB-functions to certain language constructs or restrictive use of such. These restrictions are necessary in order to be able to generate code out of and simulate the Simulink diagram. Thus, we impose limitations and assumptions about the MATLAB code to be analyzed, which are backed by the Simulink restrictions. However, we target to abstract extrinsic functions for which MATLAB code is provided and thus support some language features which Simulink does not support, too.

A first restriction of our analysis relates to the inability to analyze recursive functions or programs, which is also not supported by Simulink. Furthermore, MATLAB objects which are no structures or cell arrays are also not supported by the analysis. This includes objects such as control systems[3] and external objects, for instance from Java. As such, user defined classes with events and methods are not supported. Moreover, only integer enumerations without further functionality can be used. External code, from C/C++ and Java, either compiled or available as source code is stubbed and side effects are not considered. Error handling, using try and catch clauses is neither supported. Finally, we assume the absence of complex variables, anonymous functions and dynamic field access with the *x.(y)* operator.

Types. The MATLAB language has a dynamic type system, in which variables may change their type during execution. Primitive types for scalars are signed and unsigned integers (8,16,32 bit), 32 and 64 bit floats and fixed point numbers. Floating point numbers behave according to the IEEE-754 standard[4]. Structures and cell arrays are further non primitive data types, which are considered. Let the set of primitive data types be $\mathbb{T}$, containing all integer types and floating types, i.e. `[u]intN`, $N \in \{8, 16, 32\}$ and `float32`, `float64`. Furthermore, assume $\mathbb{S}$ to be set of strings, representing all field names of all structures, then the type set is $\mathbb{D}$.

$$\mathbb{D} := \{\mathcal{T} \times \ldots \times \mathcal{T}\}_{\mathcal{T} \in \mathbb{T}} \cup \mathbb{D} \times \ldots \times \mathbb{D} \cup \mathbb{S} \to \mathbb{D} \tag{1}$$

The left most part of the union defines matrices with a primitive type, so that all elements in the matrix must have the same primitive type. Note the recursive definition allows cell arrays ($\mathbb{D} \times \ldots \times \mathbb{D}$) to contain cell arrays or structures themselves, which also holds for structures ($\mathbb{S} \to \mathbb{D}$).

A MATLAB program consists of variables $\mathcal{V}$ and operations on these variables. A program state σ assigns a concrete value to each variable of the program. During a function call, special variable environments $\mathcal{V}_{Env}$, i.e. workspaces, are created. The set of program states are all potential variable assignments for each environment.

$$\Sigma := \{\sigma | \sigma : \mathcal{V}_{Env} \times \mathcal{V} \to \mathbb{D}\} \tag{2}$$

[2] See http://www.mathworks.com/help/simulink/slref/matlabfunction.html.
[3] See http://www.mathworks.com/help/control/ref/tf.html.
[4] As long as the used machine implements the IEEE-754 standard.

With concrete values for each variable, expressions can be evaluated with the val_σ function for a given state σ.

$$\mathrm{val}_\sigma : expr \to \mathbb{D} : a \to \mathrm{val}_\sigma(a) \tag{3}$$

For scalar expressions, the valuation function is defined below for constant c and variable $v \in \mathcal{V}$.

$$\mathrm{val}_\sigma(c) := c \qquad \mathrm{val}_\sigma(\blacklozenge e) := \blacklozenge\, \mathrm{val}_\sigma(e) \tag{4}$$

$$\mathrm{val}_\sigma(v) := \sigma(v) \qquad \mathrm{val}_\sigma(e_1 \lozenge e_2) := \mathrm{val}_\sigma(e_1) \lozenge\, \mathrm{val}_\sigma(e_2) \tag{5}$$

For instance, $\mathrm{val}_\sigma(\texttt{3*x+5})$ evaluates to $(\mathrm{val}_\sigma(3) * \mathrm{val}_\sigma(x)) + \mathrm{val}_\sigma(5) = 3\sigma(x) + 5$.

Matrices. In addition to scalars, values can be grouped into matrices or tensors, leaving a single type for all values within the matrix. Compared with scalars, matrices support additional operations, such as matrix multiplication, inversion and least square optimization. For element-wise unary and binary operations, the valuation function is also applied element-wise, with $\hat{v}$ being the index vector.

$$\mathrm{val}_\sigma(\blacklozenge A) := [\mathrm{val}_\sigma\,(\blacklozenge a_{\hat{v}})]_{\hat{v}} \qquad \mathrm{val}_\sigma(A \lozenge B) := [\mathrm{val}_\sigma\,(a_{\hat{v}} \lozenge b_{\hat{v}})]_{\hat{v}} \tag{6}$$

$$\mathrm{val}_\sigma(\blacklozenge_M A) := \blacklozenge_M\, \mathrm{val}_\sigma(A) \qquad \mathrm{val}_\sigma(A \lozenge_M B) := \mathrm{val}_\sigma(A) \lozenge_M\, \mathrm{val}_\sigma(B) \tag{7}$$

MATLAB provides a variety of functions to construct standard matrices, such as *zeros*, *ones* and *eye*, which yield matrices filled with zeros or ones and the identity matrix for given dimensions.

Structures. Since matrices cannot group values of different types, MATLAB provides structures, similar to structs in the C programming language. Each structure has several fields representing an object. A field is accessed using the dot operator(.), where the structure variable is on the left and field name is on the right hand side, i.e. *struct.fieldname*. Consider a structure s with fields x, y and z. $s.z$ would access the field z within s. Element-wise operations are not allowed on structures. Since a structure is mapped to a variable, the valuation function results also in a structure.

$$\mathrm{val}_\sigma(s.x) := \mathrm{val}_\sigma\,(\mathrm{val}_\sigma(s).x) \tag{8}$$

The inner valuation function resolves the structure variable to a concrete structure, while the outer valuation extracts the object x within s.

Cell Arrays. Matrices compose scalars of the same type with a given dimension. Cell arrays extend matrices by allowing, similar to structures, different types and dimensions in each part. Thus, only structural operations, such as reshaping the matrix structure can be applied. In the following listing, the variable a is a cell array in which the element at position (1,1) is of type `uint8`, while the other matrices are of type `double`.

```
a = {uint8(eye(5)), ones(3,7); zeros(3,2), eye(2)};
```

A cell within a cell array can be accessed by the $\{,\}$ operators, hence `a1,1` accesses the element at position (1,1). The concrete value of a cell array are all elements within the array.

$$\mathrm{val}_\sigma\,(\{c_{\hat{v}}\}) = \{\mathrm{val}_\sigma\,(c_{\hat{v}})\} \tag{9}$$

Although MATLAB code in Simulink and code generation do not support cell arrays, we define the correspondent abstractions. These can still be of use when extrinsic function calls from Simulink to externally defined MATLAB functions are contained within the model.

Workspaces. During the execution of programs, variables are kept in a *workspace*. Hence, the MATLAB developer interface itself has a root workspace and for each function a new empty workspace is created. For function calls, the arguments, which are passed as call by value, are stored in the workspace.

Furthermore, there are global variables, which are visible among several functions and thus are decoupled from any local workspace. Persistent variables do not loose their value after exiting a function. Hence persistent variables can be understood as static variables of a certain function.

Functions. When resolving the parenthesis operator, the value on the left hand side can either be a matrix or a function identifier. Functions can be specified in several ways. Built-in functions are shipped with the MATLAB interpreter and are resolved internally. User defined functions can either be specified by a MATLAB file (.m), containing a function definition and code. External functions may be given in other languages, such as C/C++ or Java. Generally, C/C++ functions can be compiled from MATLAB with a compiler to a mex[5] file, for which the source code must not be provided during run-time. Similarly, for functions programmed in Java, no source code is necessary for execution.

Whether functions are executed or variables are addressed, depends on an internal MATLAB algorithm. However, the following example makes it plausible, that variables have priority.

```
>>zeros = ones(5);
>>zeros(3)
ans = 1
```

In the example, a variable *zeros* is created by calling the built-in function *ones*, yielding a matrix filled with ones. When calling *zeros* in the second line, the variable *zeros* is resolved, prior to the built-in function.

Type and Dimension Inference. So far, we have assumed that constants and declared variables have a given type. However, MATLAB uses a type inference system, which has previously been investigated [6]. Regarding our implementation, consider the following example from the MATLAB console.

```
>>class(double(1.3) * uint8(3.5) * double(2.2))
ans = uint8
>> class(single(1.3) * uint8(3.5) * double(2.2))
Error using *
Integers can only be combined with integers of the same class, or scalar
    doubles.
>> uint8(5) * double(0.2)
ans = 1
```

From this behavior, we have derived the following facts. First, expressions with double are automatically casted up to double, which is also the default type for constants. This simplifies building expressions of constants with other types, including matrices. Second, after type inference, the result is casted down automatically. However, this behavior is limited to double constants. For 32 bit single floats and other primitive types, no automatic casting is performed. Thus, in our implementation, we infer double

[5] See http://www.mathworks.com/help/matlab/ref/mex.html.

constants by the closest defined expression. For assignments of fresh variables, we derive the type of the right hand side expression.

Deriving dimensions of matrix expressions is solved by defining additional matrix operations, such as addition or multiplication with scalars. In these cases, the scalar value is applied element-wise. Corner cases, in which a variable has multiple dimensions, but only contains a single element, such as $[1, 1, 1]$ are also considered, i.e. `ones(1,1,1)*ones(5)`.

3.2 Abstract Semantics

For our abstraction, we use an interval sets [3] domain, i.e. an ordered set of IEEE-754 intervals [10] without overlapping intervals for each concrete scalar. With this approach, we tend to increase precision, since unconnected intervals can be expressed. For instance, the interval from $[-1, 1]$ without zero can be expressed by interval sets. When applying binary operations, all resulting combinations of both sets are merged. To avoid uncontrolled growth of intervals in the set, we limit the intervals per set to a configurable number. If this number is exceeded, the intervals with the smallest distance between their bounds are merged to a single interval, as overlapping intervals are in general. Depending on the primitive data type, several interval implementations are used to capture the entire behavior of each type. Therefore, interval sets for unsigned and signed 8,16 and 32 bit integers and 32 and 64 bit floats are implemented. For floats, the user may specify an explicit IEEE-754 rounding mode to be applied when performing IEEE-754 operations. Otherwise, the result of all rounding modes is merged after each operation introducing an additional over approximation. Furthermore, special handling for operations producing and using symbols, such as NaN and $\pm\infty$ have been considered.

State Abstraction. For each abstract state ρ, there is an abstract valuation function, yielding an abstract value for expressions.

$$\mathrm{val}^{\#}_{\rho} : Expr \to \mathbb{I}_{\mathbb{D}} : a \to \mathrm{val}^{\#}_{\rho}(a) \tag{10}$$

In addition, abstract unary $\blacklozenge^{\#}$ and binary $\lozenge^{\#}$ operations can be carried out on the abstract domain.

$$\mathrm{val}^{\#}_{\rho}(c) := \{[c, c]\} \qquad \mathrm{val}^{\#}_{\rho}(\blacklozenge e) := \blacklozenge^{\#}\, \mathrm{val}^{\#}_{\rho}(e) \tag{11}$$

$$\mathrm{val}^{\#}_{\rho}(v) := \rho(v) \qquad \mathrm{val}^{\#}_{\rho}(e_1 \lozenge e_2) := \mathrm{val}^{\#}_{\rho}(e_1) \lozenge^{\#} \mathrm{val}^{\#}_{\rho}(e_2) \tag{12}$$

The $^{\#}$ denotes the corresponding abstraction of the operation on the abstract interval set domain. For binary operations on interval sets IS_1 and IS_2, we compute all combinations of all intervals, i.e. $IS_1 \lozenge^{\#} IS_2 := \cup_{I_1 \in IS_1, I_2 \in IS_2} I_1 \lozenge_I I_2$, where $\lozenge_I$ is the operation for intervals, as presented in [10].

For instance, $\mathrm{val}^{\#}_{\rho}$(`3*x+5`) evaluates to $\mathrm{val}^{\#}_{\rho}(3) * \mathrm{val}^{\#}_{\rho}(x)) +^{\#} \mathrm{val}^{\#}_{\rho}(5) = \{[3, 3]\} *^{\#} \rho(x) +^{\#} \{[5, 5]\}$. Similarly, relational $\bowtie$ and logical operators *logOp* are abstracted.

$$\mathrm{val}^{\#}_{\rho}(e_1 \bowtie e_2) := \mathrm{val}^{\#}_{\rho}(e_1) \bowtie^{\#} \mathrm{val}^{\#}_{\rho}(e_2) \tag{13}$$

$$\mathrm{val}^{\#}_{\rho}(e_1\ logOp\ e_2) := \mathrm{val}^{\#}_{\rho}(e_1)\ logOp^{\#}\ \mathrm{val}^{\#}_{\rho}(e_2) \tag{14}$$

Matrix Abstraction. Similar to scalars, fixed size matrices can be abstracted.

$$\mathrm{val}^{\#}_{\rho}(\blacklozenge A) := \left[\mathrm{val}^{\#}_{\rho}(\blacklozenge a_{\hat{v}})\right]_{\hat{v}} \qquad \mathrm{val}^{\#}_{\rho}(A \lozenge B) := \left[\mathrm{val}^{\#}_{\rho}\,(a_{\hat{v}} \lozenge b_{\hat{v}})\right]_{\hat{v}} \tag{15}$$

$$\mathrm{val}^{\#}_{\rho}(\blacklozenge_M A) := \blacklozenge^{\#}_M \mathrm{val}^{\#}_{\rho}(A) \qquad \mathrm{val}^{\#}_{\rho}(A \lozenge_M B) := \mathrm{val}^{\#}_{\rho}(A) \lozenge^{\#}_M \mathrm{val}^{\#}_{\rho}(B) \tag{16}$$

Although abstract matrices have in many cases a fixed length, the dimension of matrices may depend on the run of the program. Consider Listing 1.1, where the length of x depends on a random number.

Listing 1.1. Variable Size Variables

```
if( rand() > 0.5 )
    x = [1 2 3];
else
    x = [1 2];
end
```

In this case, the length of x is either two or three. Since the random number function `rand()` is abstracted by an interval $[0, 1]$, both paths have to be taken in the abstract execution. For such cases, we use a matrix with the maximum dimension size, in this case with three elements. Additionally, the potential sizes are stored in an interval set. In case a statement yields potential access to an element of the matrix, a warning is issued. However, the abstract interpretation is continued and correct access is assumed. If our analysis can prove that an invalid access occurs, the algorithm is aborted and a message is issued.

Structures. Structures are represented by abstract structures, mapping strings to abstract values. Thus, the abstract valuation of a structure variable yields an abstract structure. In order to extract a field, we extend the abstract valuation function.

$$\mathrm{val}^{\#}_{\rho}(x.a) := \mathrm{val}^{\#}_{\rho}\left(\mathrm{val}^{\#}_{\rho}(x).a\right) \tag{17}$$

Note, that the outer valuation function might yields an abstract object, which itself might be an abstract cell array, structure or matrix.

Cell Arrays. As with matrices, cell arrays may also have a variable size during execution, for example, `cell(uint8(rand() * 10))` yields a cell array with zero up to 10 elements. Therefore, we also store bounds on the size of a cell array. As with matrices, our implementation issues warnings for potential out of bound accesses and continues if possible, assuming a valid cell access. Our abstraction is a set of cell elements, which are abstract objects.

$$\mathrm{val}^{\#}_{\rho}\left(\{c_{\hat{v}}\}\right) := \left\{\mathrm{val}^{\#}_{\rho}\,(c_{\hat{v}})\right\} \tag{18}$$

Reshaping Operations. Matrices and cell arrays provide both structural operations, which change the dimensions, while keeping elements untouched. Functions, such as `squeeze` and `reshape` perform a rearrangement of elements. Assuming variable size abstractions, we issue a warning if the reshape operation might be out of bounds and assume a correct access and continue with the analysis.

3.3 Abstract Interpretation

With data types and structures abstracted, interpretation rules for the program execution must be defined. These rules describe a transition relation, switching between two states. Formalizations for these rules have been defined, for instance for the C programming language [1,2]. Therefore, we focus in this part on differences to MATLAB code.

Built-in Functions. Since built-in functions cannot be analyzed, due to lacking MATLAB code, we have defined abstractions for a certain set of functions. These individual abstractions increase the precision of the analysis. However, we have derived the behavior from the official documentation and experiments. Hence, there is no guarantee, that our abstractions are correct. We have defined abstractions for matrix constructors, such as *zeros*, *ones*, *rand*, *eye*, *diag* and math functions, such as *sum*, *norm*[6], *min*, *max*, *cell* and *class*[7].

Function Stubbing. External functions, for which no MATLAB source code is available, are stubbed, i.e. over approximated by the extreme values of the corresponding type. By default, the type and dimensions are computed by type inference, however, if this yields no result, a scalar function with type `double` is assumed. Nevertheless, in practice the user could specify type and dimensions for unsupported functions.

Conditional Statements. For conditional statements, such as `if(u > f(x))`, we improve the quality of the analysis by narrowing the intervals, based on the constraint. If the expression might be true and has the form $\bigwedge\bigvee v \bowtie expr$ where $v \in \mathcal{V}$, we can improve the analysis by narrowing the interval set. With previously computed intervals for all variables, a solution can be computed using interval arithmetic with additional checks for special IEEE-754 behavior. Solutions for variables in disjunctions are unionized, while conjunctions are cut. Although, we are limited in our current implementation to these simple constraints, more advanced techniques, such as relational domains [13], interval linear programming [15] or symbolic methods [4,16] can provide solutions for complex constraints. A similar and generally better approach has been applied for different programming languages in other work [2].

In case the expression cannot be narrowed, our algorithm continues with the previously computed interval sets. We repeat the procedure for `elseif` parts, while for the `else` branch the complement of the interval set is calculated[8]. Similar to other value range analyses, we interpret both branches if necessary and merge the results.

Abstract interpretation for `switch` statements is processed in the same way, except that MATLAB has a slightly different syntax here. Instead of having multiple `case` statements without a break, MATLAB uses cell array notation. Furthermore, MATLAB does not require a `break` statement during a `switch`, so that `break` commands always refer to loop statements. For example, the following code assigns to y the value 1 if x is 1,2 or 3, so there is no fall through after the first `case` statement. In fact, a `break` statement within a `switch` would be associated to the outer `while` or `for` loop.

```
switch(x)
    case {1,2,3} y = 1;
    case 4 y = 2;
end
```

[6] Only p vector norms and the 1,∞ norms for matrices.
[7] Only for primitive types.
[8] Which might include NaN values for floating point types.

Loops. Not only `switch` statements have a special syntax in MATLAB, but also `for` loops are more restricted, i.e. MATLAB requires loops with the `for` keyword to be of finite length. In detail, a matrix has to be supplied with a finite number of elements, since MATLAB only allows matrices to a user-defined maximum length[9]. Changes to the iterating variable within the loop are executed, but do not affect the termination condition of the `for` loop, i.e. in each iteration the next element from the vector is taken. Consequently, our abstract interpretation checks whether the supplied vector size is below a specified, but low, threshold and executes eventually all iterations.

In contrary, the condition of `while` loops is checked before each iteration. Hence there is no guarantee of termination. In both cases, i.e. large `for` and `while` loops, widening, as presented in [1] and adapted to interval sets in [3], is applied. Thus the set of reachable values is widened to $[-\infty; \infty]$ after a configured number of loop iterations, being a valid fix point for IEEE-754 values.

As for conditional statements, we narrow the interval set based on the same procedure. Especially for `for` loops the union of all elements with the given matrix can be used to shrink the interval set.

4 Evaluation

After having presented our approach to derive reachable value ranges for MATLAB code, we evaluate it against an industrial tool.

We selected three MATLAB files from external sources to be used for evaluation purpose. Those are a Kalman filter (KALMAN) implementation[10], an implementation of ℓ_1 trend filtering (L1TF)[11][11] and the MATLAB implementation of the matrix exponential function *expm()* of MATLAB R2014b (EXPM).

We intend to refine and extend our existing analysis of Simulink models (AV) by abstract interpretation of MATLAB-function blocks (MATLAB code). Since the Simulink Design Verifier (SLDV), which has been used for evaluation purpose in previous work, is able to analyze MATLAB-function blocks too, the SLDV is used for evaluation purpose. Thus, we build Simulink models containing MATLAB-function blocks implementing the examples mentioned above[12].

All constructed Simulink models are build of a MATLAB-function block containing one of the three identified examples, an outport block and either constant blocks (indicated by $modelname_{CI}$) to provide input values to the MATLAB-function block or inport blocks to provide free input (indicated by $modelname_{FI}$). Since none of the examples is intended to be used as code in embedded MATLAB-function blocks, slight modifications due to the restrictive support of the MATLAB language inside MATLAB-function blocks are necessary. These modifications are the conversion of MATLAB scripts into functions with input and output parameters, the removal of

[9] See http://www.mathworks.com/help/matlab/matlab_env/set-workspace-and-variable-preferences.html.

[10] http://www.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/37782/versions/1/previews/kalman.m/index.html.

[11] http://stanford.edu/~boyd/l1_tf/.

[12] Note that we use Simulink only to provide input to the MATLAB-function and do not model relevant functionality for the analysis to work or causing additional over approximation due to widening for loops in the Simulink model. Hence, the quality of the results is not affected by the MATLAB-function being integrated into Simulink.

code parts which are responsible for visual and textual output, the elimination of cell array use and the removal of variables with increasing or variable sizes by pre-allocation. We succeeded to adapt the examples accordingly, resulting in executable Simulink models. However, the modifications restrict the original functionality of the examples, e.g. with regard to the applicability to arbitrary large matrices for EXPM, which has been restricted to two dimensions. Since the KALMAN script contains two method calls `randn()` to generate noise, which will cause analyses to assume return values in the range $[-\infty; \infty]$, we construct an additional variation where `randn()` is replaced by `zeros()`. The model variants with this modification are KALMAN$_{CI}Z$ and KALMAN$_{FI}Z$ respectively.

We perform the evaluation on a 64-bit Windows 7 operating system running on an Intel i5 2.67 GHz CPU with eight gigabytes memory. The model files where constructed with MATLAB R2014b and the imports of the models for AV and the SLDV analyses performed with MATLAB 2015b. Since AV is implemented in Java, the 64 bit version of Java 8 update 77 is used.

SLDV and AV allow the user to configure the analysis in various ways. We consider two configurations ($SLDV_V$ and $SLDV_{DL}$) for SLDV analyses. $SLDV_V$ is the default configuration with additionally enabled option *Out of bound array access* for design error detection. $SLDV_{DL}$ is the same configuration as $SLDV_V$ but with enabled dead logic detection. Since the activation of dead logic detection disables the detection of other modeling flaws, such as potential divisions by zero, we analyze every evaluation model using both configurations consecutively. As for evaluations in our previous work, the default configuration of AV is used.

Besides the issued warnings, we compare the duration of the analyses AV and SLDV. Since the SLDV excludes the duration of model opening, compilation and translation to internal intermediate representation from its time measurement, we exclude the time required to start MATLAB, load the model and translate it to our intermediate abstract block diagram representation, too.

Table 2 shows the resulting time elapses and issued warnings for the computation of the different analyses and their correspondent configuration for the evaluation models and their variants. The X entries indicate that the correspondent analysis was unable to analyze the model using the given configuration due to incompatibility although the model is, due to the described adaptions, executable in Simulink. The SLDV states,

Table 2. Time elapse for analysis computation in seconds

Model	Logical lines of	Time elapse (s)			Warnings		
	MATLAB code	$SLDV_V$	$SLDV_{DL}$	AV	$SLDV_V$	$SLDV_{DL}$	AV
EXPM$_{CI}$	80	X	X	2.4	X	X	1
EXPM$_{FI}$	80	X	X	1.4	X	X	50
L1TF$_{CI}$	82	X	X	0.8	X	X	10
L1TF$_{FI}$	82	X	X	0.4	X	X	20
KALMAN$_{CI}$	44	37	37	0.4	57	1	30
KALMAN$_{FI}$	44	116	44	0.7	55	1	62
KALMAN$_{CI}Z$	44	44	42	0.9	52	0	0
KALMAN$_{FI}Z$	44	111	39	0.9	51	0	57

that the reason for the incompatibility regarding the EXPM model is its inability to determine the size of the expressions

```
floor((((j-1)*n\_m)+1)/2):floor((((j-1)*n\_m)+n\_m)/2)
floor(((j*n\_m)+1)/2):floor((((j*n\_m)+n\_m)/2))
```

which were introduced as modified expressions to use matrices instead of cell arrays. The L1TF model cannot be analyzed using the SLDV due to the use of logical indexing with index variables `negIdx1` and `negIndx2`, which is not supported by the SLDV inside MATLAB-function blocks.

For EXPM_{CI}, the AV issues a single warning regarding dead code in the MATLAB program. In particular, the `else` branch of the `if` statement is detected to be unreachable, which is true since the constant input causes the condition to be always satisfied. Allowing arbitrary two-dimensional input values (EXPM_{FI}), AV detects two potential divisions by zero, two potential divisions by $\pm\infty$ and 46 potential occurrences of NaN (Not-a-Number). Because EXPM_{FI} considers the input to the MATLAB-function to be within the interval $[-\infty;\infty]$ and values are computed based on the input, both division operators cause each a warning about division by zero, $\pm\infty$ and the potential occurence of NaN as a result of the operation. Since NaN values are propagated along operations, e.g. $x+\text{NaN} = \text{NaN}$, further warnings about potential occurring NaN values are issued[13].

Regarding the models L1TF_{CI} and L1TF_{FI}, AV detects two false-positive warnings, a potential overflow and a potential underflow for the statement `status = 'solved';`, which is a false positive. For both models, all other warnings are NaN warnings. For the L1TF_{CI} model with constant input, the NaN warnings are introduced due to the over approximation of the contained `for` loop.

Before comparing the analysis results of AV with the results of SLDV, we take a look at the warnings issued using $SLDV_{DL}$ for KALMAN_{CI} and KALMAN_{FI}. The SLDV output report states for both model variants, that the MATLAB-function block of the model is unsupported and thus, only partial results could be computed. Since the $SLDV_{DL}$ does not produce any other results, we will further focus only on $SLDV_V$.

Table 3 gives a detailed overview of the amount and types of warnings being produced by $SLDV_V$. It can be seen, that although there is no division by zero due to constant inputs for KALMAN_{CI} and $\text{KALMAN}_{CI}Z$, both division operators in the MATLAB program cause corresponding warnings. Considering free input, one division by zero is detected and by * is indicated, that a further division by zero was undecidable. Furthermore, the mentioning of an unsupported MATLAB-function blocks originates from the over approximation of the `randn()` method call in KALMAN_{CI} and KALMAN_{FI}. There are several warnings about array bounds for all models. However, these are false-positive warnings in all cases since the array sizes and accesses are all constant. Similar, the overflow warnings are of false-positive nature since all programs work only with the default type double. The use of `randn()` causes additional overflow warnings, too.

Table 4 presents the results using the AV analysis for the Kalman filter models. Comparing the results obtained with AV and $SLDV_V$, it can be noticed that there are no overflow warnings and considerably less array bound warnings. Because of the IEEE-754 double type, data type overflows are impossible and thus not issued. Nevertheless,

[13] The consecutive issuing of NaN warnings caused by NaN propagation can be disabled in order to identify only the cause of potential NaN occurrences. However, this option is disabled in the used default configuration.

Table 3. Detailed overview of warnings being issued analyzing the Kalman filter examples with $SLDV_V$

	KALMAN_{CI}	KALMAN_{FI}	$\text{KALMAN}_{CI}Z$	$\text{KALMAN}_{FI}Z$
Division by zero	2	1*	2	1*
Division by $\pm\infty$	0	0	0	0
Overflow/underflow	31	31	27	27
Array bounds	23	23	23	23
Unsupported block	1	1	0	0

Table 4. Detailed overview of warnings being issued analyzing the Kalman filter examples with AV

	KALMAN_{CI}	KALMAN_{FI}	$\text{KALMAN}_{CI}Z$	$\text{KALMAN}_{FI}Z$
Division by zero	0	1	0	1
Division by $\pm\infty$	0	2	0	2
Overflow/underflow	0	0	0	0
Array bounds	2	2	0	0
Unsupported block	0	0	0	0
Result could be NaN	28	57	0	54

the detected potential array bound violations are false-positive results. Comparing the division related warnings of both analyses, AV does not assume every division to be a division by zero. However, for arbitrary inputs ($\text{KALMAN}_{FI}Z$ and $\text{KALMAN}_{FI}Z$) potential divisions by zero and $\pm\infty$ are recognized.

As to expect, the amount of NaN related warnings increases for the models with free input and the `randn()` method due to the assumed input or return values $[-\infty; \infty]$ and correspondent operations on $\pm\infty$. Consequently, the model $\text{KALMAN}_{CI}Z$ could be proven to be safe with regard to divisions by IEEE-754 values NaN, $\pm\infty$, array access and data type overflows.

5 Conclusion

This paper provides a formal approach to derive reachable value ranges for MATLAB programs based on abstract interpretation. Combined with an existing static value range analysis for Simulink models, it enables the detection of potential modeling flaws during model-based development of embedded systems within Simulink MATLAB-function blocks. Evaluating our approach with three MATLAB programs, we were able to show the support of Simulink functionality which is not supported by the Simulink Design Verifier. Moreover, analysis time and false-positive warnings regarding divisions, overflows and array accesses are reduced. Due to the sound abstraction of IEEE-754 floating point arithmetic, IEEE-754 related warnings such as potential occurrences of NaN or divisions by $\pm\infty$ are detectable. Future work will focus on the reduction of over approximations and the extension of supported MATLAB functionality.

Acknowledgements. This project was funded within the Priority Programme "Cooperatively Interacting Automobiles" of the German Science Foundation DFG (SPP 1835). The authors acknowledge the fruitful collaboration with the project partners.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model forstatic analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1977, New York, NY, USA (1977)
2. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Why does Astrée scale up? Formal Methods Syst. Des. **35**(3), 229–264 (2009)
3. Dernehl, C., Hansen, N., Gerlitz, T., Kowalewski, S.: Static value range analysis for MATLAB/simulink-models. In: INFORMATIK 2015 (2015)
4. Dernehl, C., Hansen, N., Kowalewski, S.: Combining abstract interpretation with symbolic execution for a static value range analysis of block diagrams. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 137–152. Springer, Heidelberg (2016). doi:10.1007/978-3-319-41591-8_10
5. Doherty, J., Hendren, L., Radpour, S.: Kind analysis for MATLAB. ACM SIGPLAN Not. **46**(10), 99–118 (2011)
6. Elphick, D., Leuschel, M., Cox, S.: Partial evaluation of MATLAB. In: Pfenning, F., Macko, M. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 344–363. Springer, Heidelberg (2003)
7. Feret, J.: Static analysis of digital filters. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 33–48. Springer, Heidelberg (2004)
8. Gerlitz, T., Minh Tran, Q., Dziobek, C.: Detection and handling of model smells for MATLAB/simulink models. In: Proceedings of the International Workshop on Modelling in Automotive Software Engineering. CEUR (2015)
9. Granger, P.: Static analysis of arithmetical congruences. Int. J. Comput. Math. **30**, 165–190 (1989)
10. Hickey, T., Ju, Q., Van Emden, M.H.: Interval arithmetic: from principles to implementation. J. ACM (JACM) **48**, 1038–1068 (2001)
11. Kim, S.J., Koh, K., Boyd, S., Gorinevsky, D.: l1 trend filtering. SIAM Rev. **51**, 339–360 (2009)
12. Lu, Z., Mukhopadhyay, S.: Model-based static code analysis for MATLAB models. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 474–487. Springer, Heidelberg (2012)
13. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 3–17. Springer, Heidelberg (2004)
14. Miné, A.: The octagon abstract domain. High.-Order Symb. Comput. **19**, 31–100 (2006)
15. Rohn, J.: Interval linear programming. In: Rohn, J. (ed.) Linear Optimization Problems with Inexact Data, pp. 79–100. Springer US, Boston (2006)
16. Rümmer, P., Wahl, T.: An SMT-LIB theory of binary floating-point arithmetic. In: International Workshop on Satisfiability Modulo Theories (SMT) (2010)
17. Stattelmann, S., Biallas, S., Schlich, B., Kowalewski, S.: Applying static code analysis on industrial controller code. In: 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). IEEE (2014)