Jonatan Wiik | Pontus Boström

Contract-Based Verification of MATLAB and Simulink Matrix-Manipulating Code

Turku Centre *for* Computer Science

TUCS Technical Report
No 1107, July 2014

Contract-Based Verification of MATLAB and Simulink Matrix-Manipulating Code

Jonatan Wiik
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5, 20520 Turku, Finland
jonatan.wiik@abo.fi
Pontus Boström
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5, 20520 Turku, Finland
pontus.bostrom@abo.fi

TUCS Technical Report
No 1107, July 2014

Abstract

MATLAB/Simulink is a popular toolset for developing embedded software. The main target of the MATLAB/Simulink toolset is numerical computing applications and the tools offer a rich language for manipulating matrices. This paper presents an approach to automatic, modular, contract-based verification of programs written in a subset of the MATLAB programming language, with focus on efficiently handling the provided matrix manipulation functions. We restrict ourselves to the subset of MATLAB suitable for code generation, which means matrix types and shapes can be determined statically. We present an approach to type and shape inference for matrices that is more strict than MATLAB, but aids verification. The type and shape information is then used in the verification. From the programs and contracts we generate verification conditions that are discharged with an of-the-shelf SMT solver. We present two approaches for verification: direct axiomatisation of built-in matrix functions and expansion of the functions. We evaluate our approaches on a number of examples and discuss challenges for automatic verification in this setting. We found that expansion of matrix functions can be very effective when the matrix sizes are relatively small, which is common in many embedded applications.

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

The MATLAB [19] environment and its toolbox Simulink, have become widely used tools for development of control systems. Simulink has even become de facto standard for model-based design in many domains. The MATLAB environment includes toolboxes for generating embedded C or C++ code for different platforms directly from Simulink models or from code written in a subset of the MATLAB language, which we refer to as Embedded MATLAB throughout this paper.

The MATLAB and Simulink programming languages are aimed at numerical computing, with matrix computations as a core feature. The languages inherently support such computations through convenient built-in functions and operators that directly operate on matrices and vectors. MATLAB is an implicitly and dynamically typed imperative language. We build on existing work on MATLAB type inference [1, 14] to obtain a type system that is suitable for static inference of types and matrix shapes. The type and shape inference closely resembles what is already done in MATLAB and Simulink when C/C++ code is generated, but it is more strict in order to aid verification.

We present a modular approach to automatic contract-based static verification of MATLAB style programs. The target of our approach is Embedded MATLAB and Simulink, but the same concept should also be applicable for reasoning about matrix code in other similar languages. We use standard assume-guarantee reasoning as found in many other verifiers, e.g. [2, 3, 7], extended with efficient handling of matrix computations. Functions are checked in isolation by assuming preconditions and asserting the postconditions. At function calls, the precondition is asserted, while the postcondition is assumed. Verification conditions are generated from the programs and contracts, which are then discharged by an automatic SMT solver. The challenges here relate to efficient handling of MATLAB's built-in matrix functions, as well as inference of information needed for efficient verification that is not given explicitly.

The work described in this paper also acts as an extension to an approach for contract-based verification of Simulink models [4, 6], by adding support for matrix computations. In that approach, Simulink models are automatically verified with respect to contracts by first generating sequential code. The target programming language used there essentially constitutes a subset of the Embedded MATLAB language.

We present and evaluate two approaches to encoding verification conditions for matrix-manipulating programs in an SMT solver. In the first approach evaluated, we view the matrix functions as a library and give them

The work described in this paper has been done in the EFFIMA program coordinated by Fimecc and the EDiHy project funded by the Academy of Finland.

pre- and postconditions, as in traditional program verification. In the second approach we use the inferred information about matrix shapes in the verification process. This information is used to automatically expand the matrix functions. As we will show, expansion can make proofs more efficient and can be very effective when the size of matrices are relatively small, which is common in many embedded applications.

The main contributions of this paper are:

- Definition of an expressive language similar to Embedded MATLAB that can be effectively encoded into verifiers.
- A type inference and two automated verification approaches for the language.
- Evaluation of the approaches on examples, as well as discussion on advantages and drawbacks of the approaches.

We have implemented the presented verification approaches in the prototype verification tool VerSÅA [5]. The tool can automatically verify Embedded MATLAB code or Simulink models involving matrix computations with respect to contracts. The contracts are written as extra annotations to function declarations in Embedded MATLAB code or in a *Description* field of subsystem blocks in Simulink models.

The paper begins with a description of the MATLAB programming language and the contract format in section 2. In section 3 we define the grammar of the language. Section 4 describes the type system and the type inference framework. In section 5 we describe the verification approach and evaluate it on some examples in section 6. Section 7 discusses related work and section 8 concludes.

2 MATLAB and contract-based verification

The programming language targeted in this paper is Embedded MATLAB, which essentially is a subset of the complete MATLAB language suitable for code generation. It is an implicitly typed imperative programming language. As opposed to the complete MATLAB language, Embedded MATLAB is statically typed, since types and matrix shapes are decided at compile-time. All data in the language is ultimately a matrix[1]. A MATLAB matrix type consists of an intrinsic type, such as **double**, **int32**, **boolean** etc., and a shape. We use $\langle m, n\rangle$ to denote a matrix shape with m rows and n columns. We use the term column vector for matrices of shape $\langle m, 1\rangle$ and row vector for matrices of shape $\langle 1, n\rangle$. We use the term vector to mean either a column

[1]MATLAB supports other types too, but we do not consider them here.

vector or a row vector. In MATLAB also scalars are considered matrices, we thus use the term scalar to mean a matrix of shape $\langle 1, 1\rangle$. In this work we restrict ourselves to two-dimensional matrix shapes, although the MATLAB language in general supports an arbitrary number of dimensions. There should, however, not be any fundamental problem in extending the approach to support more dimensions. We also require that matrices are non-empty, i.e. that the size along both dimensions is ≥ 1.

The MATLAB language has built-in functions and operators (infix functions) that directly manipulate matrices. We treat functions and operators uniformly in this paper and use the term function to refer to both of them. As an example, consider the matrices a and b of shape $\langle 2, 2\rangle$:

$$a = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad b = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

The following MATLAB code[2]:

```
c := max(a,b);
d := 2+a;
```

assigns matrices with the following values to c and d:

$$c := \begin{bmatrix} max(a_{11}, b_{11}) & max(a_{12}, b_{12}) \\ max(a_{21}, b_{21}) & max(a_{22}, b_{22}) \end{bmatrix} \quad d := \begin{bmatrix} 2 + a_{11} & 2 + a_{12} \\ 2 + a_{21} & 2 + a_{22} \end{bmatrix}$$

Both the function max and the addition operator thus *element-wise* functions on matrices. An element-wise MATLAB function or operator $f(a, b)$ is defined if a and b have the same intrinsic type and $m_1 = n_1$ and $m_2 = n_2$ or either a or b is scalar, i.e. $m_1 = m_2 = 1$ or $n_1 = n_2 = 1$. The resulting matrix will then have the same shape as a and b, or the larger one of them if either a or b is scalar. Many of the built-in MATLAB functions are element-wise, but there are some exceptions. Examples of built-in functions that are not element-wise are matrix multiplication and functions that collapse matrices. Consider, for instance, the collapsing function *sum* used in the following program code:

```
x := sum(a);
y := sum(x);
```

which assigns the following matrices to x and y:

$$x := \begin{bmatrix} a_{11} + a_{21} & a_{12} + a_{22} \end{bmatrix} \quad y := (a_{11} + a_{21}) + (a_{12} + a_{22})$$

Collapsing functions collapse (row and column) vectors to scalars and other matrices to row vectors. In the example above, for instance, the variable x is assigned a row vector containing the sum over each column of the matrix

[2]For clarity we use := for assignment in this paper, although = is used in MATLAB.

```
1  function m = max_f(a)
2  %@ typeparameters: t<:numtype, n
3  %@ types: m:t, a:matrix(t,n,1)
4  %@ ensures: all(a <= m)
5  %@ ensures: any(a == m)
6    m := a(1);
7    i := int32(2);
8    while (i <= length(a))
9    %@ invariant: 1 <= i && i <= n+1
10   %@ invariant: \forall j:int32 . (1 <= j && j < i ==> m >= a(j))
11   %@ invariant: \exists j:int32 . (1 <= j && j < i && m == a(j))
12     if (m < a(i))
13       m := a(i);
14     end
15     i := i+1;
16   end
17 end
```

Figure 1: A MATLAB function for finding the index of the minimum element in a column vector, annotated with contracts.

d. The behaviour of *sum* thus depends on the input shape. This is also the case with, for instance, the multiplication operator $*$. This operator denotes normal matrix multiplication if both arguments are matrices, but element-wise multiplication if one of the arguments is scalar. MATLAB functions are typically polymorphic, which is also the case for the built-in functions *max* and *sum* used above. The function *max*, for instance, accepts two matrices of any intrinsic type t and any shape $\langle m, n \rangle$ and returns a matrix of the same intrinsic type and shape.

A MATLAB program consists of a set of functions, of which one acts as the entry-point for the program. The aim of this paper is to enable automatic contract-based verification of such programs. We use a standard modular verification technique, checking every function with respect to its contract in isolation, using assume-guarantee reasoning. For each function body analysed, we check that the postconditions hold if the preconditions are satisfied. The contracts are written inside special comments, analogously to how it is done in e.g. JML [7] for the Java language. A small example function for computing the maximum element of a vector, is given in Fig. 1. This functionality is also implemented by the built-in MATLAB function *max* with one argument, however, the goal here is to demonstrate language features. The specification of the function, i.e. its contract, is written in comments, starting with “%@”. In addition to normal preconditions and postconditions, the function is also annotated with types for the inputs and the output.

We first consider the type annotations. Types for all inputs and the

output is given in the *types* field. The syntax matrix(t, n, m) is used to denote a matrix with intrinsic type t and shape $\langle n, m\rangle$. The short-hand form t is used to denote a scalar of intrinsic type t, i.e. matrix$(t, 1, 1)$. The *typeparameters* field declares universally quantified type parameters over which the types declared in the *types* field can be parametrised. A type parameter can parametrise over either intrinsic type or shape. For instance, the *max_f* function in Fig. 1 is parametrised to take as input a matrix of an intrinsic type t and a shape $\langle n, 1\rangle$ and outputs a scalar of the same intrinsic type t. Note that the type parameter t is bound to numeric types in this example. These type annotations are actually not needed here for type inference, as all types could be inferred if the types of the inputs of the entry-point function are known. However, the type annotations are here important from a specification point-of-view. Without the type annotations there would, for instance, be several incorrect implementations satisfying the postconditions of the *max_f* function, e.g. the implementation $m := a$. The type annotations thus provide a form of pre- and postconditions constraining the inputs and outputs on the level of types (and shapes).

We use the standard annotations *requires* and *ensures* for function preconditions and postconditions, as well as *invariant* annotations for loops. For the *max_f* function we have the postcondition $all(a \leq m)$, stating that the output m should be greater than or equal to each element in a. We also have the postcondition $any(a = m)$, stating that there exists an element in a that is equal to m. Note that the *all* and *any* functions are built-in MATLAB functions. They correspond to universal and existential quantifiers over matrix indices and provide a compact and intuitive way to write contracts for matrix code. The invariants used for the while-loop in *max_f* are used to prove that the loop establishes the postcondition of the function. Note that the *all* and *any* functions are not used in the invariants on lines 11-12. In MATLAB these conditions could be written using *all* and *any*, e.g. $all(a(1{:}i) \leq m)$. However, we want to infer matrix shapes statically. This is not possible for this expression, as i is not constant, which consequently means that the shape of $a(1{:}i)$ is not constant and hence cannot be determined statically.

3 Language definition

In this section we define more precisely the programming language considered throughout this paper, i.e. the subset of MATLAB we intend to support. In addition to the MATLAB constructs supported, the language is also extended with some specification-oriented constructs not part of pure MATLAB. These constructs are written inside special comments to maintain compatibility with MATLAB.

In MATLAB, matrices are immutable objects presumably implemented via copy-on-write. Every function or matrix update can thus be considered to return an new matrix. A MATLAB program consists of a set of function declarations, of which one acts as the entry-point of the program. The grammar of our function declarations is the following:

$$
\begin{array}{lcl}
\mathit{FuncDecl} & ::= & \textbf{function } Id = f(Id^*) \\
 & & \mathit{TypeParams}^? \ \mathit{Types} \ \mathit{Spec}^* \ \mathit{Stmt}^? \\
 & & \textbf{end} \\
\mathit{TypeParams} & ::= & \textbf{typeparameters } (Id \ (\sqsubseteq_t t)^?)^* \\
\mathit{Types} & ::= & \textbf{types } (Id : t)^* \\
\mathit{Spec} & ::= & \textbf{requires } \mathit{Exp} \mid \textbf{ensures } \mathit{Exp}
\end{array}
\tag{1}
$$

In this grammar, x^* and $x^?$ denote zero or more and zero or one occurrences of an element x, respectively. The function declarations of MATLAB are thus extended with type annotations for the input and output parameters, as well as pre- and postcondition annotations.

The expressions supported essentially constitute a subset of the MATLAB expression language. Additionally, it also includes some logical constructs, such as universal and existential quantifiers and conditional expressions, which are not part of the MATLAB language, but often are convenient when writing specifications. The same expression language is used both in statements in function implementations and in contract expressions. The complete expression language grammar is the following:

$$
\begin{array}{ll}
\mathit{Exp} ::= & \\
\mathit{Exp_1} \ (+ \mid - \mid * \mid / \mid .* \mid ./) \ \mathit{Exp_2} \mid & \text{Arithmetic expression} \\
\mathit{Exp_1} \ (\wedge \mid \vee \mid \implies \mid \iff) \ \mathit{Exp_2} \mid & \text{Logical expression} \\
\mathit{Exp_1} \ (= \mid \neq \mid < \mid > \mid \geq \mid \leq) \ \mathit{Exp_2} \mid & \text{Relational expression} \\
(\forall \mid \exists) \ (x : t)^* \ \cdot \ \mathit{Exp} \mid & \text{Quantified expression} \\
\neg \mathit{Exp} \mid -\mathit{Exp} \mid & \text{Unary operators} \\
\mathit{Exp_1} \ ? \ \mathit{Exp_2} : \mathit{Exp_3} \mid & \text{Conditional expression} \\
\mathit{Exp_1} \ (\mathit{Exp_2} \mid :) \ (\mathit{Exp_3} \mid :)^? \mid & \text{Matrix accessor} \\
Id \ \mathit{Exp_1}, \ldots, \mathit{Exp_n} \mid & \text{Function call} \\
Id \mid & \text{Identifier} \\
\mathit{Num} \mid & \text{Numeric literal} \\
[\mathit{Exp_{11}}, \ldots, \mathit{Exp_{1n}}; \ldots; \mathit{Exp_{m1}}, \ldots, \mathit{Exp_{mn}}] \mid & \text{Matrix literal} \\
\mathit{CExp_1} {:} \mathit{CExp_2} \mid \mathit{CExp_1} {:} \mathit{CExp_2} {:} \mathit{CExp_3} \mid & \text{Range} \\
\textbf{true} \mid \textbf{false} & \text{Boolean literal}
\end{array}
\tag{2}
$$

It is worth noting that we do not separate expressions from predicates in this grammar, as this is done later in the type checking. This is consistent with how MATLAB treats predicates. It is further worth noting that matrix accessors and function calls partially share the same syntax. Again, this is consistent with MATLAB, and we handle this by requiring that variables do not have the same name as any known built-in function or function declared by the user. In MATLAB the behaviour is more complicated. Rose and

Padua developed an algorithm [14] for simulating this behaviour, which we have chosen not to repeat here.

In MATLAB there are also expressions with data-dependent shape, e.g. the Range expression in (2). The shape of a range expression $a{:}b$, where a and b are integers, is $\langle 1, b-a+1\rangle$. For these cases we define a separate language, *CExp*, for constant expressions. This language is a subset of the expressions given in (2), which can be evaluated by the type checker and hence used in expressions with data-dependent shape. Thus, the type checker does not need to support the complete expression language in order to support static shape inference of such expressions. Since the expressions in *CExp* are only used for shape information, only numeric scalars are included in the language. Support for a limited set of functions is also included. The supported functions are typecast functions for supported integer types and the *length* function. The grammar of *CExp* is hence defined as:

$$\begin{aligned} CExp \quad ::= \quad & Id \mid Num \mid int32(CExp) \mid int16(CExp) \mid int8(CExp) \mid \\ & uint32(CExp) \mid uint16(CExp) \mid uint8(CExp) \mid length(Exp) \end{aligned} \tag{3}$$

The typecast functions are needed to enable declaration of constants of different types in a way consistent with MATLAB. The *length* function returns the size of the largest dimension of its input. This function is convenient to support in the type checker since it enables the declaration of new matrices using size information from another matrix, for instance using $zeros(length(x), 1)$ to create a column vector of zeros with the same length as x. Note that the argument to *length* can be any expression in *Exp*, not only *CExp*. The reason is that only the shape of x is used and not the value of the expression. It would in principle be straight-forward to extend *CExp* to also support other language constructs and functions in the future, including matrices. For instance the *size* function, returning a vector containing the size of its input along each dimension, would be useful to support for the same reason as the *length* function.

In the MATLAB language, it is possible to assign variables (complete matrices) or elements in matrices. To reflect this, we define a separate language for the left-hand side of assignments, i.e. assignable expressions, *Asgn*:

$$Asgn \quad ::= \quad Id\ (Exp_1 \mid\ :)\ (Exp_2 \mid\ :)^? \mid Id \tag{4}$$

The special colon operator here denotes assignment to an entire row or column. Note that *Asgn* is more strict than in MATLAB, which allows assignment to any sub matrix, whereas we only allow assignment to a single element or an entire row or column.

The complete grammar of the statement language, in which function im-

plementations are written, is the following:

$$
\begin{array}{lll l}
\mathit{Stmt} ::= & \mathit{Asgn} := \mathit{Exp} \mid & \text{Assigment} & \\
& \textbf{constant}\ \mathit{Id} := \mathit{CExp} \mid & \text{Constant declaration} & \\
& \textbf{if}\ \mathit{Exp}\ \mathit{Stmt_1}\ \textbf{else}\ \mathit{Stmt_2}\ \textbf{end} \mid & \text{If-statement} & \\
& \mathit{Stmt_1} ; \mathit{Stmt_2} \mid & \text{Sequential composition} & \\
& \textbf{while}\ \mathit{Exp}\ \mathit{Inv}^*\ \mathit{Stmt}\ \textbf{end} \mid & \text{While loop} & (5) \\
& \textbf{assert}\ \mathit{Exp} \mid & \text{Assume} & \\
& \textbf{assume}\ \mathit{Exp} \mid & \text{Assert} & \\
& \mathit{Id} :\mid \mathit{Exp} \mid & \text{Non-deterministic update} & \\
& \textbf{choice}\ \mathit{Stmt_1}\ \textbf{or}\ \mathit{Stmt_2}\ \textbf{end} & \text{Non-deterministic choice} &
\end{array}
$$

While this language closely resembles the MATLAB language, there are some differences. The largest difference is the addition of the specification-oriented statements, listed in the last four lines of the grammar. These constructs are, however, primarily used internally by the verifier. The language also allows declaration of constants, which are not needed in pure MATLAB. Constants are assigned using the constant expression language *CExp* defined in (3), and they can thus be evaluated by the type checker and used as shape information in the type inference. To maintain MATLAB compatibility, the **constant** keyword of constant declarations can be written in a comment on the line above the assignment. MATLAB does not have explicit constants, but Embedded MATLAB infers that variables are constants when needed. For clarity we have, however, opted to use explicit declaration of constants here.

We recognise that the subset of supported MATLAB constructs is currently fairly small. However, it is straight-forward to extend the language and the verification approach to cover many more MATLAB constructs, such as for-loops, which would be typically used in MATLAB code targeted for code generation.

4 Type system

We want to statically determine types and shapes of data in functions written in the language described in the previous section. The type and shape information can then be used in the verification and thus the verifier does not need to quantify over types and shapes. This significantly simplifies the verification tasks, since we only need to verify functions for the instantiations of type parameters actually used instead of all valid instantiations. However, the MATLAB language is implicitly typed, and we want to avoid extra type annotations. Thus, we determine the type and shape of local variables and expressions through inference.

The approach to type and shape inference that we use is inspired by previous work of Almási, Padua and de Rose in the context of the MaJIC [1]

and FALCON [14] compilers for MATLAB code. We have, however, made several modifications to their type system, to enable efficient encoding of the programs in a verifier. The type inference we use is also stricter than MATLAB's type inference. For instance, we do not allow implicit typecasts as is commonly done in MATLAB. In MATLAB it is legal to add an integer with a double, since the double will automatically be cast to an integer. Since we do not allow typecasts like these, we also require that matrix indices are integers. Another difference compared to MATLAB is that we use a stricter separation between booleans and numeric types, in order to obtain efficient verification conditions. MATLAB does, for instance, accept numeric types as operands to logical operators and also allows booleans to be used in arithmetic expressions, which we do not allow.

In our language all data is of matrix type. Since we only consider matrices with up to two dimensions, a matrix type consists of an intrinsic type t and a shape $\langle n, m \rangle$, where n and m denotes the number of rows and columns, respectively. The intrinsic type is an element in a finite lattice L_t, formed by the elements $\mathcal{I} = \{$**boolean**, **int8**, **int16**, **int32**, **uint8**, **uint16**, **uint32**, **double**, **numtype**, **toptype**$\}$, and the comparison operator:

$$\begin{array}{rcl} L_t & = & \{\mathcal{I}, \sqsubseteq_t\} \text{ and} \\ & & \textbf{boolean} \sqsubseteq_t \textbf{toptype}, \\ & & \textbf{int8} \sqsubseteq_t \textbf{int16} \sqsubseteq_t \textbf{int32} \sqsubseteq_t \textbf{numtype}, \\ & & \textbf{uint8} \sqsubseteq_t \textbf{uint16} \sqsubseteq_t \textbf{uint32} \sqsubseteq_t \textbf{numtype}, \\ & & \textbf{double} \sqsubseteq_t \textbf{numtype}, \\ & & \textbf{numtype} \sqsubseteq_t \textbf{toptype} \end{array} \tag{6}$$

As in MATLAB, we have several different sizes of integers. We here use a different type hierarchy than that presented in [1, 14]. There integers are considered as subtypes of reals (doubles). This would not be feasible for verification, as verifiers typically uses different theories for integers and reals. Additionally, compared to [1, 14], we do not at the moment consider complex numbers. The type **toptype** is a supertype of all other intrinsic types. The type **numtype** is a super type of all numeric intrinsic types. The types **numtype** and **toptype** are in a sense abstract types. They are used as a bound for type parameters in function declarations, but they cannot be used as instantiations of type parameters.

A matrix shape consists of two dimensions $n_1, n_2 \in \mathcal{D}$, where $\mathcal{D}$ is the set of dimensions: $\mathcal{D} = \mathbb{Z}^+ \cup \{\infty\}$. Matrix shapes can then be defined as a lattice L_s, which consists of pairs of dimensions, one for the number of rows and one for the number of columns:

$$\begin{array}{rcl} L_s & = & \{\mathcal{D} \times \mathcal{D}, \sqsubseteq_s\} \text{ and} \\ & & s_1 \sqsubseteq_s s_2 \mathrel{\hat{=}} (s_1 = s_2 \vee s_2 = \langle *, \infty \rangle \vee s_2 = \langle \infty, * \rangle) \end{array} \tag{7}$$

where $*$ denotes any dimension $d \in \mathcal{D}$. Here ∞ denotes an invalid dimension,

which indicates an inference error. Programs can only be verified if every node in the abstract syntax tree has been assigned a valid (finite) shape.

In [1, 14] an unknown shape only means fallback to dynamic memory allocation. Also MATLAB has an option to generate code which uses variable-size memory. Here we require that exact types and shapes are determined statically. Because of this, we do not allow the type of a variable to change once it has been assigned for the first time. This also means that the shape of a variable cannot change once it has been assigned for the first time. However, this is something that anyway should be avoided in safety-critical embedded applications.

The type system for our language is given by the Cartesian product of the intrinsic type lattice and the shape lattice: $\mathcal{T} = L_t \times L_s$. We use $\mathrm{matrix}(t, \langle n, m\rangle)$ to denote a type $(t, \langle n, m\rangle) \in \mathcal{T}$ in the language. We can then define the language of types as follows:

$$\begin{array}{rcl} t & ::= & x \mid \textbf{boolean} \mid \textbf{int8} \mid \ldots \\ d & ::= & x \mid n \\ s & ::= & \langle d_1, d_2\rangle \\ \delta & ::= & s \mid \mathrm{max_s}(s_1, s_2) \mid \mathrm{mul_s}(s_1, s_2) \mid \mathrm{col_s}(s) \\ \tau & ::= & \mathrm{matrix}(t, \delta) \mid d \\ \alpha & ::= & \tau \mid \alpha_1 \times \alpha_2 \mid \alpha \rightarrow \tau \\ \theta & ::= & \alpha \mid \forall\ \overline{x \sqsubseteq_t t} \cdot \theta \mid \Pi\ \overline{x} \cdot \theta \mid \textbf{unit} \end{array} \tag{8}$$

Here x denotes a type parameter identifier and $\overline{x}$ denotes a list of such identifiers. We use n to denote a positive integer. Hence t and d denotes the grammar of intrinsic types and dimensions, respectively. Then τ defines the grammar of complete matrix types. We use α to define the grammar of types for functions. Finally, we use θ to define the grammar of quantified types. We have universally quantified polymorphic functions, which can be quantified over both intrinsic type and shape. We use $\forall$ to denote quantification over intrinsic type and Π to denote quantification over shape. As an example, consider the type signature for the element-wise operator addition:

$$\begin{array}{l} \forall\ t \sqsubseteq_t \textbf{numtype} \cdot \Pi\ m_1, m_2, n_1, n_2 \cdot \mathrm{matrix}(t, \langle m_1, m_2\rangle) \times \mathrm{matrix}(t, \langle n_1, n_2\rangle) \\ \quad \rightarrow \mathrm{matrix}(t, \mathrm{max_s}(\langle m_1, m_2\rangle, \langle n_1, n_2\rangle)) \end{array}$$

The quantification over intrinsic type is bounded, denoted by $t \sqsubseteq_t u$, meaning that t must be a subtype of u. Similar bounds for quantification over shape is currently not supported, and it is thus not possible to, for instance, declare functions that accepts inputs up to some specific size. This is a feature that could be added, however, we have not come across any built-in MATLAB function of interest for embedded applications, where such a declaration would be useful. The kind of polymorphism supported is similar to Let-polymorphism found in ML and related languages, i.e. type parameters

cannot be instantiated with polymorphic types. A consequence is that all valid types can be written in a form in which quantifiers only appear in the outermost position of types [21].

The shape function max_s used in the type signature above is defined as:

$$\text{max}_\text{s}(\langle m_1, m_2\rangle, \langle n_1, n_2\rangle) = \begin{cases} \langle m_1, m_2\rangle & \text{if } m_1 = n_1 \text{ and } m_2 = n_2 \\ \langle m_1, m_2\rangle & \text{if } n_1 = n_2 = 1 \\ \langle n_1, n_2\rangle & \text{if } m_1 = m_2 = 1 \\ \langle \infty, \infty\rangle & \text{otherwise} \end{cases} \quad (9)$$

The output shape for all binary element-wise functions are defined in the same way. All MATLAB functions are, however, not element-wise. The type signature for matrix multiplication, for instance, is the following:

$$\forall\ t \sqsubseteq_t \textbf{numtype} \cdot \Pi\ m_1, m_2, n_1, n_2 \cdot \text{matrix}(t, \langle m_1, m_2\rangle) \times \text{matrix}(t, \langle n_1, n_2\rangle) \\ \rightarrow \text{matrix}(t, \text{mul}_\text{s}(\langle m_1, m_2\rangle, \langle n_1, n_2\rangle))$$

where the shape function mul_s is defined in the following way:

$$\text{mul}_\text{s}(\langle m_1, m_2\rangle, \langle n_1, n_2\rangle) = \begin{cases} \langle m_1, n_2\rangle & \text{if } m_2 = n_1 \\ \langle m_1, m_2\rangle & \text{if } n_1 = n_2 = 1 \\ \langle n_1, n_2\rangle & \text{if } m_1 = m_2 = 1 \\ \langle \infty, \infty\rangle & \text{otherwise} \end{cases} \quad (10)$$

Additionally, there is also a separate case for collapsing MATLAB functions. Consider, for instance, the type signature of the *sum* function:

$$\forall\ t \sqsubseteq_t \textbf{numtype} \cdot \Pi\ n_1, n_2 \cdot \text{matrix}(t, \langle n_1, n_2\rangle) \rightarrow \text{matrix}(t, \text{col}_\text{s}(\langle n_1, n_2\rangle))$$

where col_s is given by:

$$\text{col}_\text{s}(\langle n_1, n_2\rangle) = \begin{cases} \langle 1, n_2\rangle & \text{if } n_1 > 1 \text{ and } n_2 > 1 \\ \langle 1, 1\rangle & \text{if } n_1 = 1 \text{ or } n_2 = 1 \end{cases} \quad (11)$$

In MATLAB there are also built-in functions with data-dependent output shape. Examples of such functions are $zeros(a, b)$ and $ones(a, b)$, which returns a matrix of shape $\langle a, b\rangle$ in which each element is 0 and 1, respectively. These functions are typically used to initialise matrices in Embedded MATLAB and supporting them is thus essential. However, it is obvious that, in the general case, the output shape of these functions cannot be determined at compile-time. We have opted to solve this issue by introducing constants in our language and restricting these functions to only accept constant expressions as input. Constant expressions can be evaluated during type inference and coerced to shape information, which can be used in the inference. As an example, consider the type signature for the functions *zeros* and *ones*:

$$\#a \times \#b \rightarrow \text{matrix}(\textbf{double}, \langle \#a, \#b\rangle)$$

$$\frac{\begin{array}{c} f : \forall\ \overline{x \sqsubseteq_t y} \cdot \Pi\ \overline{s} \cdot \overline{\tau_i} \rightarrow \tau_o \quad dom(\sigma_t) = \{\overline{x}\} \quad dom(\sigma_s) = \{\overline{s}\} \\ \mathcal{V}, \mathcal{C} \vdash E : \tau_i[\sigma_t, \sigma_s] \text{ forall } (E, \tau_i) \in (\overline{E}, \overline{\tau_i}) \end{array}}{\mathcal{V}, \mathcal{C} \vdash f(\overline{E}) : \tau_o[\sigma_t, \sigma_s]}\ \textbf{(fun-call)}$$

$$\frac{\mathcal{V}, \mathcal{C} \vdash E : \textbf{boolean} \quad \text{Æ} \in \{\forall, \exists\}}{\mathcal{V}, \mathcal{C} \vdash \text{Æ}\ \overline{x : \tau} \cdot E : \textbf{boolean}}\ \textbf{(quant-exp)}$$

$$\frac{\mathcal{V}, \mathcal{C} \vdash E_1 : t, \ldots, E_m : t}{\mathcal{V}, \mathcal{C} \vdash [E_1, \ldots, E_m] : \text{matrix}(t, 1, m)}\ \textbf{(mat-cons-row)}$$

$$\frac{\mathcal{V}, \mathcal{C} \vdash E_1 : \text{matrix}(t, 1, m), \ldots, E_n : \text{matrix}(t, 1, m)}{\mathcal{V}, \mathcal{C} \vdash [E_1; \ldots; E_n] : \text{matrix}(t, n, m)}\ \textbf{(mat-cons-col)}$$

$$\frac{\mathcal{V}, \mathcal{C} \vdash C : t \quad t \sqsubseteq_t \textbf{int32} \quad C > 0}{\mathcal{V}, \mathcal{C} \vdash \#C \in \mathcal{D}}\ \textbf{(shp-1)}$$

$$\frac{\mathcal{V}, \mathcal{C} \vdash C : t \quad t \sqsubseteq_t \textbf{uint32} \quad C > 0}{\mathcal{V}, \mathcal{C} \vdash \#C \in \mathcal{D}}\ \textbf{(shp-2)}$$

Figure 2: Typing rules for expressions.

where $\#a$ denotes coercion of the constant value a to shape information. The coercion is only defined for constant integer expressions, defined in (3). The arguments a and b are thus also integer scalars.

It is worth noting that there is also the opposite coercion, i.e. that shape data is coerced to an expression. Consider for instance the function $length(a)$, which returns the length of the largest dimension of the matrix a. This case is, however, trivial, since all shapes have been decided at compile-time and shapes of variables are not allowed to change. Thus it is possible to just replace the shape variable with its actual value through constant propagation. The type of shape information is considered to be the largest integer type, i.e. **int32**, at expression level.

The type signatures for all different cases of supported operators and functions are listed in Tables 1 and 2. Note that the operators $.*$ and $./$ denote element-wise multiplication and division. It is also worth noting, that for MATLAB compatibility, numeric constants can also be type parametrised. E.g. 0 can be a scalar of any type, including boolean.

Inference rules for typing expressions are listed in Fig. 2 and for statements and function declarations in Fig. 3. Here $\mathcal{V}$ maps variables to types and $\mathcal{C}$ maps constants to types. The rules describe typing of both intrinsic type and shape. As we treat operators and function calls uniformly, there is only one typing rule, **(fun-call)**, for all function calls. In the type rules, even matrix accesses are treated as function calls. The **(fun-call)** rule does, however, deserve some explanation: Function applications are typed based on the type signature of the function declaration. Let σ_t be a mapping from type

Table 1: Type signatures for binary operators and functions as well as conditional expressions

Function	**Type signature**
$a+b$ $a-b$ $a\ .{*}\ b$ $a\ ./\ b$ $max(a,b)$ $min(a,b)$ $pow(a,b)$ $mod(a,b)$	$\forall\ t \sqsubseteq_t \textbf{numtype} \cdot \Pi\ m_1, m_2, n_1, n_2\ \cdot$ $\mathrm{matrix}(t, \langle m_1, m_2\rangle) \times \mathrm{matrix}(t, \langle n_1, n_2\rangle)$ $\rightarrow \mathrm{matrix}(t, \mathrm{max_s}(\langle m_1, m_2\rangle, \langle n_1, n_2\rangle))$
$a \wedge b$ $a \vee b$ $a \implies b$ $a \iff b$	$\Pi\ m_1, m_2, n_1, n_2\cdot$ $\mathrm{matrix}(\textbf{boolean}, \langle m_1, m_2\rangle) \times \mathrm{matrix}(\textbf{boolean}, \langle n_1, n_2\rangle)$ $\rightarrow \mathrm{matrix}(\textbf{boolean}, \mathrm{max_s}(\langle m_1, m_2\rangle, \langle n_1, n_2\rangle))$
$a=b$ $a \neq b$ $a>b$ $a \geq b$ $a<b$ $a \leq b$	$\forall\ t \cdot \Pi\ m_1, m_2, n_1, n_2\ \cdot$ $\mathrm{matrix}(t, \langle m_1, m_2\rangle) \times \mathrm{matrix}(t, \langle n_1, n_2\rangle)$ $\rightarrow \mathrm{matrix}(\textbf{boolean}, \mathrm{max_s}(\langle m_1, m_2\rangle, \langle n_1, n_2\rangle))$
$a * b$	$\forall\ t \sqsubseteq_t \textbf{numtype} \cdot \Pi\ m_1, m_2, n_1, n_2\ \cdot$ $\mathrm{matrix}(t, \langle m_1, m_2\rangle) \times \mathrm{matrix}(t, \langle n_1, n_2\rangle)$ $\rightarrow \mathrm{matrix}(t, \mathrm{mul_s}(\langle m_1, m_2\rangle, \langle n_1, n_2\rangle))$
$a\ ?\ b : c$	$\forall\ t \cdot \Pi\ m_1, m_2, n_1, n_2, o_1, o_2\ \cdot$ $\mathrm{matrix}(\textbf{boolean}, \langle m_1, n_1\rangle) \times \mathrm{matrix}(t, \langle n_1, n_2\rangle)\times$ $\mathrm{matrix}(t, \langle o_1, o_2\rangle) \rightarrow \mathrm{matrix}(t, \mathrm{max_s}(\langle n_1, n_2\rangle, \langle m_1, m_2\rangle, \langle o_1, o_2\rangle))$

Table 2: Type signatures for unary functions, matrix accesses, data-dependent functions and number literals

Function	**Type signature**
$-a$ $abs(a)$ $square(a)$ $sqrt(a)$ $sin(a)$ $cos(a)$ $tan(a)$ $sgn(a)$	$\forall\ t \sqsubseteq_t \textbf{numtype} \cdot \Pi\ n_1, n_2 \cdot$ $\text{matrix}(t, \langle n_1, n_2\rangle) \rightarrow \text{matrix}(t, \langle n_1, n_2\rangle)$
$intX(a)$	$\forall\ t \cdot \Pi\ n_1, n_2 \cdot \text{matrix}(t, \langle n_1, n_2\rangle) \rightarrow \text{matrix}(\textbf{intX}, \langle n_1, n_2\rangle)$
$uintX(a)$	$\forall\ t \cdot \Pi\ n_1, n_2 \cdot \text{matrix}(t, \langle n_1, n_2\rangle) \rightarrow \text{matrix}(\textbf{uintX}, \langle n_1, n_2\rangle)$
$double(a)$	$\forall\ t \cdot \Pi\ n_1, n_2 \cdot \text{matrix}(t, \langle n_1, n_2\rangle) \rightarrow \text{matrix}(\textbf{double}, \langle n_1, n_2\rangle)$
$boolean(a)$	$\forall\ t \cdot \Pi\ n_1, n_2 \cdot \text{matrix}(t, \langle n_1, n_2\rangle) \rightarrow \text{matrix}(\textbf{boolean}, \langle n_1, n_2\rangle)$
$\neg a$	$\Pi\ n_1, n_2 \cdot \text{matrix}(\textbf{boolean}, \langle n_1, n_2\rangle) \rightarrow \text{matrix}(\textbf{boolean}, \langle n_1, n_2\rangle)$
$sum(a)$ $prod(a)$ $min(a)$ $max(a)$	$\forall\ t \sqsubseteq_t \textbf{numtype} \cdot \Pi\ n_1, n_2 \cdot$ $\text{matrix}(t, \langle n_1, n_2\rangle) \rightarrow \text{matrix}(t, \text{col}_\text{s}(\langle n_1, n_2\rangle))$
$all(a)$ $any(a)$	$\Pi\ n_1, n_2 \cdot$ $\text{matrix}(\textbf{boolean}, \langle n_1, n_2\rangle) \rightarrow \text{matrix}(\textbf{boolean}, \text{col}_\text{s}(\langle n_1, n_2\rangle))$
$transpose(a)$	$\forall\ t \cdot \Pi\ n_1, n_2 \cdot \text{matrix}(t, \langle n_1, n_2\rangle) \rightarrow \text{matrix}(t, \langle n_2, n_1\rangle)$
$length(a)$	$\forall\ t \cdot \Pi\ n_1, n_2 \cdot \text{matrix}(t, \langle n_1, n_2\rangle) \rightarrow \textbf{int32}$
$size(a)$	$\forall\ t \cdot \Pi\ n_1, n_2 \cdot \text{matrix}(t, \langle n_1, n_2\rangle) \rightarrow \text{matrix}(\textbf{int32}, \langle 1, 2\rangle)$
$a(i, j)$	$\forall\ t \cdot \Pi\ n_1, n_2 \cdot \text{matrix}(t, \langle n_1, n_2\rangle) \times \textbf{int32} \times \textbf{int32} \rightarrow t$
$a(:, j)$	$\forall\ t \cdot \Pi\ n_1, n_2 \cdot \text{matrix}(t, \langle n_1, n_2\rangle) \times \textbf{int32} \rightarrow \text{matrix}(t, \langle n_1, 1\rangle)$
$a(i, :)$	$\forall\ t \cdot \Pi\ n_1, n_2 \cdot \text{matrix}(t, \langle n_1, n_2\rangle) \times \textbf{int32} \rightarrow \text{matrix}(t, \langle 1, n_2\rangle)$
a:b	$\#a \times \#b \rightarrow \text{matrix}(\textbf{int32}, \langle 1, \#b - \#a + 1\rangle)$
a:b:c	$\#a \times \#b \times \#c \rightarrow \text{matrix}(\textbf{int32}, \langle 1, \lfloor(\#c - \#a)/\#b\rfloor + 1\rangle)$
$zeros(a, b)$ $ones(a, b)$	$\#a \times \#b \rightarrow \text{matrix}(\textbf{double}, \langle \#a, \#b\rangle)$
0 1	$\forall\ t \cdot \text{matrix}(t, \langle 1, 1\rangle)$
n	$\forall\ t \sqsubseteq_t \textbf{numtype} \cdot \text{matrix}(t, \langle 1, 1\rangle)$

$$\frac{\begin{array}{c}\mathcal{V},\mathcal{C} \vdash A : \mathrm{matrix}(t, n_1, n_2) \quad \mathcal{V},\mathcal{C} \vdash E : \mathrm{matrix}(u, m_1, m_2) \\ \mathrm{max_s}(\langle n_1, n_2\rangle, \langle m_1, m_2\rangle) = \langle n_1, n_2\rangle \quad u \sqsubseteq_t t\end{array}}{\mathcal{V},\mathcal{C} \vdash A := E : \mathbf{unit}} \ \textbf{(assign-1)}$$

$$\frac{\mathcal{V},\mathcal{C} \vdash E : \tau \quad x \notin dom(\mathcal{V}) \quad x \notin dom(\mathcal{C})}{\mathcal{V} \cup \{x : \tau\}, \mathcal{C} \vdash x := E : \mathbf{unit}} \ \textbf{(assign-2)}$$

$$\frac{\mathcal{V},\mathcal{C} \vdash C : t \quad x \notin dom(\mathcal{V}) \quad x \notin dom(\mathcal{C})}{\mathcal{V},\mathcal{C} \cup \{x : t\} \vdash \mathbf{constant}\ x := C : \mathbf{unit}} \ \textbf{(const-assign)}$$

$$\frac{\mathcal{V},\mathcal{C} \vdash E : \mathbf{boolean} \quad \mathcal{V},\mathcal{C} \vdash S_1 : \mathbf{unit} \quad \mathcal{V},\mathcal{C} \vdash S_2 : \mathbf{unit}}{\mathcal{V},\mathcal{C} \vdash \mathbf{if}\ E\ S_1\ \mathbf{else}\ S_2\ \mathbf{end} : \mathbf{unit}} \ \textbf{(if-else)}$$

$$\frac{\mathcal{V},\mathcal{C} \vdash E : \mathbf{boolean} \quad \mathcal{V},\mathcal{C} \vdash I : \mathbf{boolean}\ \text{for all}\ I \in \overline{I} \quad \mathcal{V},\mathcal{C} \vdash S : \mathbf{unit}}{\mathcal{V},\mathcal{C} \vdash \mathbf{while}\ E\ \overline{\mathbf{invariant}\ I}\ S\ \mathbf{end} : \mathbf{unit}} \ \textbf{(while)}$$

$$\frac{\mathcal{V},\mathcal{C} \vdash E : \mathbf{boolean}}{\mathcal{V},\mathcal{C} \vdash \mathbf{assert}\ E : \mathbf{unit}} \ \textbf{(assert)} \qquad \frac{\mathcal{V},\mathcal{C} \vdash E : \mathbf{boolean}}{\mathcal{V},\mathcal{C} \vdash \mathbf{assume}\ E : \mathbf{unit}} \ \textbf{(assume)}$$

$$\frac{\mathcal{V},\mathcal{C} \vdash \overline{M : \tau} \quad \mathcal{V},\mathcal{C} \vdash E : \mathbf{boolean}}{\mathcal{V},\mathcal{C} \vdash \overline{M}\ :|\ E : \mathbf{unit}} \ \textbf{(nondet-update)}$$

$$\frac{\mathcal{V},\mathcal{C} \vdash S_1 : \mathbf{unit} \quad \mathcal{V},\mathcal{C} \vdash S_2 : \mathbf{unit}}{\mathcal{V},\mathcal{C} \vdash \mathbf{choice}\ S_1\ \mathbf{or}\ S_2\ \mathbf{end} : \mathbf{unit}} \ \textbf{(nondet-choice)}$$

$$\frac{\mathcal{V}_1, \mathcal{C}_1 \vdash S_1 : \mathbf{unit} \quad \mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \vdash S_2 : \mathbf{unit}}{\mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \vdash S_1; S_2 : \mathbf{unit}} \ \textbf{(seq-stmt)}$$

$$\frac{\mathcal{V},\mathcal{C}, \overline{x : \tau_i}, y : \tau_2 \vdash S : \mathbf{unit}}{\mathcal{V},\mathcal{C} \vdash \mathbf{function}\ y = f(x)\ S\ \mathbf{end} : \overline{\tau_i} \rightarrow \tau_o} \ \textbf{(func-decl)}$$

Figure 3: Typing rules for statements.

parameters to intrinsic types and let σ_s be a mapping from type parameters to shapes. The notation $\tau[\sigma_t, \sigma_s]$ then denotes the type τ instantiated with the mappings σ_t and σ_s. Thus, if the types of all inputs are instantiations of the inputs $\overline{\tau_i}$ of the function type signature under the type parameter mappings σ_t and σ_s, the output type of the function call will be τ_o instantiated with σ_t and σ_s. Other noteworthy rules are **(shp-1)** and **(shp-2)**, which are used for coercion of constant integer expressions to shape information in the type signatures for data-dependent expressions. This coercion is only possible for positive integers. Additionally, we also have rules for quantified expressions and matrix literals. A matrix literal consists of n rows, which in turn consists of m scalar expressions of an intrinsic type t.

Among the rules for statements in Fig. 3, it is worth noting that there are three cases for assignment statements. One for the case when a variable is assigned for the first time and one for the case when the variable has been assigned before. Separate rules are needed, since the type of a variable is not allowed to change once it is assigned for the first time. Finally, there is also a separate rule for constant declaration.

The type inference for expressions is done using a traditional unification algorithm [20, 22], where the constraints are derived directly from the typing rules. Statements, on the other hand, are handled using forward propagation of the type information. This is similar to the approach in [1, 14], where a combination of forward and backward propagation of shape and type information is used. Intrinsic type and shape are orthogonal aspects, meaning that inference can be done independently for intrinsic type and shape. Inference of intrinsic type is standard. For shape inference we use the shape functions (9), (10) and (11) to build constraints. The inference is successful if all nodes in the AST are assigned an intrinsic type and a valid (finite) shape.

5 Verification

Our verification approach is based on standard assume-guarantee reasoning. We use a modular verification technique, checking every function in isolation. The preconditions of the function are turned into assumptions and the postconditions into assertions. On function calls the preconditions are asserted and the postconditions assumed. In the type inference, we have inferred exact intrinsic type and shape (i.e. instantiation of type parameters) for all function calls. All invoked user-implemented functions are verified independently for each type instantiation occurring in the program. Thus, we do not verify that a function satisfies its contract for every valid instantiation of type parameters, but only for the instantiations actually used. Hence, the inference of types and shapes is non-modular, while the verification of func-

tions is done modularly based on the inferred type and shape information. This eliminates the need to quantify over types and shapes in the verification of functions.

The statement language, given in (5), has standard weakest precondition semantics. Loops are verified based on the classical Hoare logic in the same way as e.g. Spec# [3] and Boogie [2]. In addition to verifying conformance to contracts, we prove the absence of runtime errors in the function implementations. The runtime errors checked for are bounds on matrix accesses, integer overflow and division by zero. The verifier does, however, only check partial correctness. We do not currently check termination for neither iteration nor recursion. Termination checks could be added analogously to how it has been done in other verifiers [10], however, we have chosen to focus on verification of properties regarding matrix computations.

To verify MATLAB code that involves matrices and vectors, matrix functions need to be efficiently encoded in a verifier. We have used the SMT solver Z3 [12] by Microsoft Research as a verification backend. Z3 includes a theory for arrays [13], which we use to represent our matrices. The main constructs in this theory are the *select* and *store* expressions. The expression $select(a, i)$ returns the value stored at position i in an array, while $store(a, i, x)$ returns a new array identical to a, but with the value x on position i. We encode matrices as arrays of arrays. Each subarray is thus a matrix row. In the representation of a row vector there is only one subarray. For column vectors all subarrays are of size 1. Scalars, i.e. matrices of shape $\langle 1, 1\rangle$, are not encoded as arrays.

Matrices in this case have fixed size that can be determined statically and we can thus use this information when generating verification conditions. In this section we present two approaches to encoding the verification conditions. The approaches are then evaluated on a number of examples in section 6. In the first approach, which we call axiomatisation, we view the functions as a library and provide pre- and postconditions, as in traditional program verification. It is thus possible to axiomatise the functions directly. In the second approach we use the inferred information about matrix shapes to expand the matrix functions.

5.1 Axiomatisation

In the axiomatisation approach, matrix functions are axiomatised to have their desired meanings. Functions may have several different axioms for different input types, in which case the correct axiom is chosen based on the inferred types. Consider for instance the axioms for an element-wise function f in (12). Here f_s denotes the corresponding scalar function for f. The different axiomatisations are separated by renaming functions based on input types and shapes. We also handle polymorphism by generating

separate axioms for each type instantiation occurring in the program. Note also that all the axioms are actually quantified over the function inputs, which we have left out here for brevity. The complete axiom for a function $f(a, b)$ is thus $\forall\ a : t_1, b : t_2 \cdot A$, where A is an axiom in the format presented below.

$$\begin{array}{ll}
a : \text{matrix}(t, \langle n_1, n_2\rangle) & \forall\ i_1 : \mathbf{int32}, i_2 : \mathbf{int32} \cdot 1 \leq i_1 \leq n_1 \wedge 1 \leq i_2 \leq n_2 \\
b : \text{matrix}(t, \langle n_1, n_2\rangle) & \quad \implies f(a, b)(i_1, i_2) = f_s(a(i_1, i_2), b(i_1, i_2)) \\
 & \\
a : \text{matrix}(t, \langle n_1, n_2\rangle) & \forall\ i_1 : \mathbf{int32}, i_2 : \mathbf{int32} \cdot 1 \leq i_1 \leq n_1 \wedge 1 \leq i_2 \leq n_2 \\
b : t & \quad \implies f(a, b)(i_1, i_2) = f_s(a(i_1, i_2), b) \\
 & \\
a : t & \forall\ i_1 : \mathbf{int32}, i_2 : \mathbf{int32} \cdot 1 \leq i_1 \leq n_1 \wedge 1 \leq i_2 \leq n_2 \\
b : \text{matrix}(t, \langle n_1, n_2\rangle) & \quad \implies f(a, b)(i_1, i_2) = f_s(a, b(i_1, i_2))
\end{array} \tag{12}$$

In (13) we list axioms for some other common functions, which are not element-wise. In these formulas we have the matrices $a : \text{matrix}(t, \langle n_1, n_2\rangle)$ and $b : \text{matrix}(t, \langle n_1, n_2\rangle)$.

$$\begin{array}{lll}
a * b & : & \forall\ i_1 : \mathbf{int32}, i_2 : \mathbf{int32} \cdot 1 \leq i_1 \leq n_1 \wedge 1 \leq i_2 \leq n_2 \\
 & & \quad \implies (a * b)(i_1, i_2) = \sum_{k=1}^{k=n_2} (a(i_1, :) .* b(:, i_2))(k) \\
transpose(a) & : & \forall\ i_1 : \mathbf{int32}, i_2 : \mathbf{int32} \cdot 1 \leq i_1 \leq n_1 \wedge 1 \leq i_2 \leq n_2 \\
 & & \quad \implies transpose(a)(i_2, i_1) = a(i_1, i_2) \\
size(a) & : & size(a)(1, 1) = n_1 \wedge size(a)(1, 2) = n_2 \\
length(a) & : & length(a) = max(n_1, n_2)
\end{array} \tag{13}$$

The operator $*$ actually have several different axioms for different input types in the same way as f in (12), but we only give the matrix multiplication case here.

We further have the following axioms for collapsing functions. Here we have a matrix $a : \text{matrix}(t, \langle n_1, n_2\rangle)$ and a row vector $b : \text{matrix}(t, \langle 1, n_2\rangle)$, the transpose b^T is then the corresponding column vector:

$$\begin{array}{lll}
sum(a) & : & \forall\ i_2 : \mathbf{int32} \cdot 1 \leq i_2 \leq n_2 \\
 & & \quad \implies sum(a)(1, i_2) = \sum_{k=1}^{k=n_1} a(k, i_2) \\
sum(b) & : & sum(b) = \sum_{k=1}^{k=n_2} b(1, k) \\
sum(b^T) & : & sum(b^T) = sum(b) \\
all(a) & : & \forall\ i_2 : \mathbf{int32} \cdot 1 \leq i_2 \leq n_2 \\
 & & \quad \implies all(a)(1, i_2) = \forall\ k : \mathbf{int32} \cdot 1 \leq k \leq n_1 \cdot a(k, i_2) \\
all(b) & : & all(b) = \forall\ k : \mathbf{int32} \cdot 1 \leq k \leq n_2 \implies b(1, k) \\
\ldots & &
\end{array} \tag{14}$$

Collapsing functions, such as Σ here, must be defined recursively in SMT solvers. The functions *all* and *any* are, however, exceptions, as these functions can be directly encoded using universal and existential quantifiers.

The axioms for element-wise functions, such as f in (12), can be directly and efficiently encoded in SMT solvers. Efficient encoding of collapsing functions and other recursively defined functions, on the other hand, is hard [18].

The reason is that proofs of properties regarding these functions are typically done by induction, which typically cannot be done automatically by an SMT solver. The needed specifications for induction proofs for these functions would thus have to be provided manually, which is not feasible in practice.

5.2 Expansion

Instead of using axioms, we can utilise the inferred shapes of matrices to expand the definitions of matrix functions. We will see later that this approach is very efficient for matrices of relatively small size.

We have the matrices $a : \mathrm{matrix}(t, \langle n_1, n_2\rangle)$, $b : \mathrm{matrix}(t, \langle n_1, n_2\rangle)$ and $c : \mathrm{matrix}(t, \langle 1, n_2\rangle)$. Then $[\![a]\!]$ denotes the syntactically expanded matrix:

$$[\![a]\!] = \begin{bmatrix} [\![a(1,1)]\!] & \cdots & [\![a(1,n_2)]\!] \\ \vdots & \ddots & \vdots \\ [\![a(n_1,1)]\!] & \cdots & [\![a(n_1,n_2)]\!] \end{bmatrix} \tag{15}$$

where also matrix accesses are expanded. The matrix accessor $a(i_1, i_2)$ is defined as:

$$[\![a(i_1, i_2)]\!] = [\![a]\!](i_1, i_2) \tag{16}$$

If a is an expanded matrix and its indices are constant, the correct element can be directly chosen. However, we still need to use arrays in the SMT encoding as indices in matrix accesses are not always constant. If a is an identifier then the expansion $[\![a]\!]$ does nothing. This is actually important in order to handle many cases efficiently. This means that we can use quantified expressions to write down expressions that are effectively scalar and hence not expanded. This is particularly useful for verification of loops, such as in Fig. 1, where no matrix functions other than matrix accessors are used in the invariants.

The expanded definition of an element-wise function f applied to two expanded matrices $[\![a]\!]$ and $[\![b]\!]$ is given as follows:

$$[\![f(a,b)]\!] = \begin{bmatrix} f_s([\![a(1,1)]\!], [\![b(1,1)]\!]) & \cdots & f_s([\![a(1,n_2)]\!], [\![b(1,n_2)]\!]) \\ \vdots & \ddots & \vdots \\ f_s([\![a(n_1,1)]\!], [\![b(n_1,1)]\!]) & \cdots & f_s([\![a(n_1,n_2)]\!], [\![b(n_1,n_2)]\!]) \end{bmatrix} \tag{17}$$

Again, the function f_s here denotes the corresponding scalar version of the function f. The expansions of all other element-wise functions follow the same pattern.

Collapsing functions, such as *sum* and *all*, are also expanded. The expansions for *sum* are defined in the following way:

$$[\![sum(c)]\!] = [\![c(1)]\!] + \ldots + [\![c(n_1)]\!] \quad [\![sum(c^T)]\!] = [\![c(1)]\!] + \ldots + [\![c(n_1)]\!]$$

$$[\![sum(a)]\!] = \left[[\![a(1,1)]\!] + \ldots + [\![a(n_1,1)]\!] \quad \ldots \quad [\![a(1,n_2)]\!] + \ldots + [\![a(n_1,n_2)]\!]\right] \tag{18}$$

Finally, there is also a special case for expansion of the multiplication operator:

$$[\![a * b]\!] = \begin{bmatrix} [\![sum(a(1,:) .* b(:,1))]\!] & \cdots & [\![sum(a(1,:) .* b(:,n_3))]\!] \\ \vdots & \ddots & \vdots \\ [\![sum(a(n_1,:) .* b(:,1))]\!] & \cdots & [\![sum(a(n_1,:) .* b(:,n_3))]\!] \end{bmatrix} \quad (19)$$

All other functions are encoded in a similar way as the functions presented above.

As an example on how the expansion works, consider the expansion of an expression $all(sum(x + ones(3,3)) > 0)$, where the variable x has the type matrix(**double**, $\langle 3, 3\rangle$):

$$\begin{aligned} &(x(1,1)+1)+(x(2,1)+1)+(x(3,1)+1)>0 \\ \wedge\ &(x(1,2)+1)+(x(2,2)+1)+(x(3,2)+1)>0 \\ \wedge\ &(x(1,3)+1)+(x(2,3)+1)+(x(3,3)+1)>0 \end{aligned}$$

Note that $>$ is an element-wise operator. Not also that *ones* is expanded to a matrix literal of the given size, in which each element is 1. The expansion of the matrix accessor can then directly pick the correct element.

6 Benchmarks

In this section we evaluate the two approaches to encoding verification conditions, described in the previous section, on a number of small examples. We have used Z3 version 4.3.0 on a modern laptop in the evaluation. We also compare the performance of our approaches to Simulink Design Verifier[3] (SLDV). SLDV is a MATLAB toolbox for verification Simulink models, which also handles a large subset of Embedded MATLAB. We used MATLAB 2014a and SLDV 2.6 in the evaluation.

We start with proving associativity of matrix addition and associativity of matrix multiplication using our encoding in an SMT solver. The execution times for different sizes of matrices are listed in Table 3. In the table, we use *Ax* to denote axiomatisation and *Exp* to denote expansion. For the element-wise function matrix addition, the axiom encoding is very efficient and the execution time is invariant with respect to the size of the matrix. For the matrix multiplication function, which involves recursively defined functions, axiomatisation is not a feasible approach, since inductive proofs regarding the properties of the function would be needed. The SMT solver is only able to unfold the definition for matrices up to the size $\langle 3, 3\rangle$. Expansion, on the other hand, is an efficient approach as long as the matrices are kept fairly small. SLDV can prove associativity of matrix addition, but the scalability is far lower than for both of our approaches. Proving associativity of matrix

[3] http://www.mathworks.com/products/sldesignverifier/

Table 3: Execution times for proving associativity of matrix addition and matrix multiplication for different matrix sizes

Addition			
$\langle n,n\rangle$	Ax (s)	Exp (s)	SLDV (s)
25	1.1	2.0	8.0
50	1.1	3.7	202.0
100	1.1	10.2	$>$ 300.0
200	1.1	35.3	$>$ 300.0
400	1.1	151.0	$>$ 300.0

Multiplication			
$\langle n,n\rangle$	Ax (s)	Exp (s)	SLDV (s)
3	1.1	1.2	n/a
4	n/a	1.3	n/a
5	n/a	1.6	n/a
10	n/a	6.2	n/a
20	n/a	81.0	n/a

multiplication fails, because SLDV cannot handle nonlinear arithmetic on real numbers.

We would like to point out that the most interesting part of these benchmarks is the growth rate of the execution times rather than the actual execution time. This is because of the current implementation of the expansion in the tool, which generates a new AST from the unexpanded AST. Copying of subtrees in this step currently amounts to the vast majority of the execution time. This step could easily be optimised by doing the expansion and encoding into the SMT format in one step.

Fig. 4 lists a function that creates a square matrix of the size given by the input parameter n, where each element in row i contains the value i. This example demonstrates how the MATLAB built-in functions can be used to write compact and readable specifications. It also demonstrates the use of the special colon operator for assigning or accessing entire rows or columns, which can also be useful for writing specifications. Note also that the function input parameter n is used as a type parameter. This implicitly requires that n is a scalar of intrinsic type **int32** and is declared as a constant in the calling function. It also means that the input parameter n cannot be assigned in the function, which is normally possible. Note that the functionality provided by the function could be implemented in one line, $m := transpose(double(1{:}n)) * ones(1, n)$, but the goal here is to demonstrate language features.

Also recursive functions are supported. Fig. 5 lists a recursive function calculating the factorial $x!$ of a positive integer x. We do not, however, check termination for recursion, and thus only partial correctness is verified. Note that user-defined functions cannot be expanded, as is done for built-in functions, since the verifier cannot automatically deduce how a function should be expanded. Thus, axiomatisation is always used for these functions.

Fig. 6 lists a function returning a column vector containing the n first Fibonacci numbers. This example also illustrates the use of description strings for contract conditions. These strings are used both as a description in the source code for the developer and also by the verification tool to help the

```
1  function m = matrix_create(n)
2  %@ typeparameters: n
3  %@ types: n:int32, m:matrix(double,n,n)
4  %@ ensures: all(all(m == transpose(double(1:n))*ones(1,n)))
5    i := 1;
6    m := zeros(n,n);
7    while (i <= n)
8    %@ invariant: 1<=i && i<=n+1
9    %@ invariant: \forall j:int32 .
10   %@   (1<=j && j<=i-1 ==> all(m(j,:)==double(j)))
11     m(i,:) := double(i);
12     i := i+1;
13   end
14 end
```

Figure 4: A MATLAB function creating a matrix of shape $\langle n, n \rangle$, where each element in row i has the value i.

```
1  function y = fac(x)
2  %@ types: x:int32, y:int32
3  %@ requires: x >= 0
4  %@ ensures: x > 0 ==> y == x*fac(x-1)
5  %@ ensures: x == 0 ==> y == 1
6    if(x == 0)
7      y := 1;
8    else
9      y := x*fac(x-1);
10   end
11 end
```

Figure 5: A recursive MATLAB function.

```
 1  function fibs = fibonacci(n)
 2  %@ typeparameters: n
 3  %@ types: n:int32, fibs: matrix(double,n,1)
 4  %@ requires 'Inputs larger than 1 supported' : n > 1
 5  %@ ensures 'Fibonacci numbers 3 to n':
 6  %@   \forall j:int32 .
 7  %@   ((3<=j && j<=n) ==> (fibs(j) == fibs(j-1)+fibs(j-2)))
 8  %@ ensures 'The first fibonacci number is 0' : fibs(1) == 0
 9  %@ ensures 'The second fibonacci number is 1' : fibs(2) == 1
10    i := 2;
11    fibs := zeros(n,1);
12    fibs(1) := 0;
13    fibs(2) := 1;
14    while (i < n)
15    %@ invariant 'Loop index between 2 and n' : 2<=i && i<=n
16    %@ invariant 'Initial fibonacci numbers' : fibs(1)==0 && fibs(2)==1
17    %@ invariant 'Elements up to i calculated' :
18    %@   \forall j:int32 .
19    %@   ((3<=j && j<=i) ==> (fibs(j) == fibs(j-1)+fibs(j-2)))
20      i := i+1;
21      fibs(i) := fibs(i-1)+fibs(i-2);
22    end
23  end
```

Figure 6: A MATLAB function for calculating the n first Fibonacci numbers.

developer locate the error in case the program does not verify.

The next example, listed in Fig. 7, is significantly more complex than the previous examples. It is an implementation of the Gaussian elimination method for solving systems of linear equations. The function provides the solution to the equation system $A * x = f$, where A is a matrix, f is a column vector and x is a column vector of unknown values. Here we use the function $answer(A, f)$ to specify the desired solution. The postcondition of the $answer$ function states that the result x satisfies the condition $A * x = f$. However, we do not provide an implementation for the $answer$ function, and hence the function is not verifiable. We use the function $old(x)$ in the invariants, denoting the initial value of an input parameter x. The **error** statement used on line 14 is used to model exceptions. It is a short-hand for **assume false**, which here intuitively can be understood as an infinite loop. This works, as we are only interested in the case when the program terminates normally, i.e., partial correctness. It is also worth noting that ~ denotes negation in MATLAB and ~= hence denotes the relational operator not equal to.

The function consists of two loops. The first loop transforms A into an upper triangular matrix, while the second one transforms A into a diagonal matrix with only ones in the diagonal. All transformations preserve

the property $all(A * answer(old(A), b) = f)$. This means that at the end $answer(old(A), b) = f$, i.e. f contains the solution to the equation system.

The first loop traverses all columns and transforms all the rows below the diagonal in such a manner that the elements below the diagonal become zero. If an element in the diagonal would be zero, the matrix is singular and there are no solutions for the equation system. In this case we have an error. The nested loop performs the transformations, one row at the time. Each row j below and including the diagonal at column k is transformed as $A(j,:) := A(j,:) - A(k,:) .* A(j,k) ./ A(k,k)$. This has the effect of setting all elements $A(j,k)$ below the diagonal element $A(k,k)$ to zero, while also not changing elements to the left of column k in row j that are already zero. The right-hand side f is transformed in the same way. The loop invariants capture the progress up to index k and j, respectively.

The second loop performs back substitution of the results. Starting from the lower right corner of A, the value of $x(k)$ is stored in $f(k)$ and the corresponding diagonal element of A is set to one. The inner loop traverses all rows j above the diagonal for a column k and subtracts the term $f(k) .* A(j,k)$ from $f(j)$ while $A(j,k)$ is set to zero. This essentially moves the term $x(k) .* A(j,k)$ to the right-hand-side of the equation.

The advantage of expanding function definitions is here that we can use matrix multiplication directly in specifications, without any extra lemmas regarding the recursively defined *sum* function. This decreases annotation overhead and improves ease of use, when the user can analyse the problem on a higher level of abstraction. The major problem with the specification of the *gauss* function, from a usability point-of-view, is still the large amount of invariant conditions needed to track state information. In particular, here we often need duplicate information in order to state that the inner loops maintain the invariant properties of the outer loop. This could potentially be partially remedied by techniques for inference of invariants [15, 16].

Verification benchmarks for the example programs described above, as well as the *max_f* function listed in Fig. 1, are given in Table 4. The benchmarks lists execution times for verification of the programs with different input sizes. Again, expansion works well for matrices of relatively small size. The axiomatisation approach is only effective for the *fibonacci* example, where no recursively defined functions are used. In the *max_f* case, the axiomatisation approach works to prove the postcondition on line 4, which uses the *all* collapsing function. The SMT solver is, however, not able to prove the postcondition on line 5, which uses the *any* function. The problem seems to be the combination of universal and existential quantifiers used in the axiom for the *any* function: $\forall\ a : \text{matrix}(\mathbf{boolean}, \langle n, 1\rangle) \cdot (any(a) = \exists\ j : \mathbf{int32} \cdot 1 \leq j \leq n \wedge a(j))$. It seems that the SMT solver is not able to instantiate these quantifiers successfully. In general, we noted that the SMT solver seems to quickly run into problems with the axioms for *all* and *any*.

```
1  function f = gauss(A,b)
2  %@ typeparameters: n
3  %@ types: A:matrix(double,n,n), b:matrix(double,n,1), f:matrix(double,n,1)
4  %@ ensures: all(f == answer(A,b))
5    k := 1;
6    f := b;
7    while(k <= length(b))
8    %@ invariant: 1<=k && k<=length(b)+1
9    %@ invariant: all(A*answer(old(A),b)==f)
10   %@ invariant:
11   %@   \forall u:int32, v:int32 . (1<=u && u<k && u<v && v<=n ==> A(v,u)==0)
12   %@ invariant: \forall v:int32 . (1<=v && v<k ==> A(v,v)~=0)
13     if (A(k,k) == 0)
14       error 'Matrix is singular'
15     end
16     j := k+1;
17     while (j <= length(b))
18     %@ invariant: 1<=k && k<=length(b)
19     %@ invariant: k<j && j<=length(b)+1
20     %@ invariant: all(A*answer(old(A),b)==f)
21     %@ invariant:
22     %@   \forall u:int32, v:int32 . (1<=u && u<k && u<v && v<=n ==> A(v,u)==0)
23     %@ invariant: \forall v:int32 . (1<=v && v<=k ==> A(v,v)~=0)
24     %@ invariant: \forall v:int32 . (k<v && v<j ==> A(v,k)==0)
25       f(j) := f(j)-f(k).*A(j,k)/A(k,k);
26       A(j,:) := A(j,:)-A(k,:).*A(j,k)./A(k,k);
27       j := j+1;
28     end
29     k := k+1;
30   end
31   k := n;
32   while (1 <= k)
33   %@ invariant: 0<=k && k<=n
34   %@ invariant: all(A*answer(old(A),b)==f)
35   %@ invariant: \forall v:int32 . (1<=v && v<=n ==> A(v,v)~=0)
36   %@ invariant: \forall v:int32 . (k<v && v<=n ==> A(v,v)==1)
37   %@ invariant:
38   %@   \forall u:int32, v:int32 . (1<=u && u<=n && u<v && v<=n ==> A(v,u)==0)
39   %@ invariant:
40   %@   \forall u:int32, v:int32 . (k<u && u<=n && 1<=v && v<u ==> A(v,u)==0)
41     f(k) := f(k)/A(k,k);
42     A(k,k) := 1;
43     j := k-1;
44     while (1 <= j)
45     %@ invariant: 1<=k && k<=n
46     %@ invariant: 0<=j && j<k
47     %@ invariant: all(A*answer(old(A),b)==f)
48     %@ invariant: \forall v:int32 . (k<=v && v<=n ==> A(v,v)==1)
49     %@ invariant: \forall v:int32 . (1<=v && v<=n ==> A(v,v)~=0)
50     %@ invariant:
51     %@   \forall u:int32, v:int32 . (1<=u && u<=n && u<v && v<=n ==> A(v,u)==0)
52     %@ invariant:
53     %@   \forall u:int32, v:int32 . (k<u && u<=n && 1<=v && v<u ==> A(v,u)==0)
54     %@ invariant: \forall v:int32 . (j<v && v<k ==> A(v,k)==0)
55       f(j) := f(j)-f(k).*A(j,k);
56       A(j,k) := 0;
57       j := j-1;
58     end
59     k := k-1;
60   end
61 end
```

Figure 7: Gaussian elimination example program

```
1 function y = testfunc(x)
2 %@ types: y:matrix(double,1,2), x:matrix(double,1,2)
3 %@ requires: all(x == 0)
4 %@ ensures: all(y >= 0)
5    y := x;
6 end
```

Figure 8: The axiomatisation of the *all* and *any* functions seems to cause problems for the SMT solver.

Table 4: Benchmarks for the example programs

max_f			
n	Ax (s)	Exp (s)	SLDV (s)
10	n/a	1.2	11.0
20	n/a	1.2	38.0
30	n/a	1.2	200.0
1000	n/a	4.5	> 300.0
2000	n/a	16.5	> 300.0
3000	n/a	61.3	> 300.0

matrix_create			
n	Ax (s)	Exp (s)	SLDV (s)
10	n/a	1.4	< 1.0
25	n/a	2.2	2.0
50	n/a	6.0	33.0
75	n/a	13.8	178.0
100	n/a	30.2	> 300.0
150	n/a	111.8	> 300.0

fibonacci			
n	Ax (s)	Exp (s)	SLDV (s)
250	1.3	4.5	n/a
500	1.3	13.7	n/a
1000	1.3	50.5	n/a

gauss			
n	Ax (s)	Exp (s)	SLDV (s)
2	n/a	3.3	n/a
3	n/a	10.3	n/a
4	n/a	n/a	n/a

This is also demonstrated in the example listed in Fig. 8, which the verifier is unable to prove using the axiomatisation approach. The verifier is not able to utilise the information expressed using the axiomatised version of *all* in the precondition to establish the postcondition.

For the *gauss* example, the verifier is only successful on inputs up to size 3. The problem is proving invariant preservation of $all(A * answer(old(A), b) = f)$ after the row update in the first inner loop. The SMT solver returns unknown, possibly due to the complex expression with many array updates in the SMT encoding, due to the combination of matrix multiplication and row update.

We also evaluated the performance of SLDV on the example programs. SLDV unfolds loops for verification, which means static loop bounds are needed. We modified the programs to use for-loops instead instead of while-loops, since this seemed to significantly more efficient in SLDV. The tool was able to verify *max_f* and *matrix_create*, but the scalability is significantly lower than for our expansion approach. Presumably because of the loop unfolding. Verification of the *gauss* program failed because it involves non-

linear arithmetic. We were not able to encode the specification of *fibonacci* in SLDV in a good way, as there is no support for quantifiers.

The results indicate that expansion can be a robust and efficient approach, while the performance of axiomatisation heavily depends on how well the verifier does quantifier instantiation in a given situation. We have here only evaluated the axiomatisation and expansion approaches separately. The results, however, suggests that a hybrid approach, where only functions that are problematic to axiomatise effectively are expanded, could be efficient. It is worth noting that we have used Z3 with the default settings when obtaining our benchmarks. Z3 uses model-based quantifier instantiation (MBQI) [**?**] by default. Compared to using patterns for quantifier instantiation, MBQI seems to significantly improve the performance in our case. It may, however, be possible to tune the settings for even better performance.

7 Related work

Contract-based static verification has been implemented for many different programming languages e.g., Java [7, 9], C# [3], .NET [15] and C [11]. Arrays are supported in all these verifiers. It would be possible to implement the matrix functions as a library in any of these frameworks. However, there are three challenges: 1) The languages are statically and explicitly typed. 2) We would need to manually provide contract annotations to recursively defined functions such as e.g. the sum in matrix multiplication to inductively prove the needed properties of them. Explicitly providing all lemmas that can be possibly needed in practise for e.g. matrix multiplication is not practical. 3) Arrays are mutable objects, which complicates reasoning.

The main challenge here is automated verification of recursive functions, such as the functions *sum* and *prod* with one argument. In [18] they discuss axiomatisations of comprehension functions, which are similar to our recursive functions, suitable for use in SMT solvers. There the bounds on the comprehensions are not static. Their focus is on verification of loops that computes results involving these comprehensions, not programs that use functions specified by them. This means that the recursive definitions typically have to be unfolded only a few times. Their work focuses on bounding the unfolding. Here the problem is that unless we manually provide the desired properties of recursively defined functions, the function definitions have to be unfolded all the way. This becomes extremely inefficient in an SMT solver. However, all our recursive functions can only be applied to matrices with known static bounds.

Simulink Design Verifier (SLDV) can handle a large subset of the built-in functions in Embedded MATLAB. How matrix calculations are handled cannot be found in the documentation. The performance is not always good,

e.g. proving associativity of matrix multiplication fails (in MATLAB 2014a, SLDV 2.6), since nonlinear arithmetic is not supported for rational numbers. Loops are unfolded, which means static bounds on loops are needed. SLDV is perhaps more targeted towards verification of Simulink models where a lot of the functionality is described as state machines using Stateflow.

Our verification approach relies on inference of intrinsic types and shapes of matrices. Inference of this type of information for MATLAB programs have been studied before in the context of program optimisation [14, 1, 17]. The goal there is to calculate shapes and types of matrices in order to pre-allocate matrices and avoid bounds checks at runtime. They do not use any type annotations to guide the inference. In their case, an unknown type or shape only means fallback to dynamic inference. In this paper concrete values for the matrix shapes need to be given, but the axiomatic approach to verification should work with symbolic bounds also. One important difference is that we require that matrices do not change their shape, which is handled in their frameworks. This is not a necessary restriction in our case either. However, we can only handle size of matrices that depend on constants, i.e., data-dependent sizes in loops and recursion is not allowed. In [14, 1] they use a combination of forward propagation and backward propagation of type information. This is very similar to our constraint-based type inference. In [17] they use algebraic properties regarding shapes of the matrix operators in MATLAB to perform shape analysis. They allow matrices with arbitrary many dimensions. They can also infer other relationships between data than concrete values obtained by forward and backward propagation in [14, 1]. Using the constraint systems obtained in our inference approach to obtain dynamic constraint will probably be far less efficient due to the large numbers of constraints generated. MATLAB also performs static type and shape analysis for data in Embedded MATLAB. It appears that forward propagation of matrix shapes is performed. Variable-sized data can be used, but it has to be explicitly enabled. As function parameters can be declared to be constants, we can use these parameters in matrix shapes, which cannot be done for fixed-sized data in Embedded MATLAB. However, currently they can use more complex expressions in matrix creation expressions. These limitations can be remedied in our tool also, by increasing the subset of the language handled by the type checker.

Languages that are designed to be aware of the shape of the data exist. The language FiSH [8] allows type annotations involving matrix and vector shapes. Also an inference algorithm is discussed. However, they do not allow converting values to matrix shapes, as we do e.g. for the function *zeros*. A dependent type system in ML [23, 24] has also been studied. One goal with this type system is analysis of array bounds. The approach is based on user annotations of types together with a local inference algorithm. The index language used for array shapes can be arbitrarily complex only limited by

the choice of constraint solver. This approach is potentially more general than ours, but it is aimed at a functional language.

Many functions are partial, e.g. division and matrix access. We do special well-formedness checks to ensure that functions are only applied in their domain. This is similar to many other verifiers [9, 3, 15, 11].

To handle the problem with recursive functions, abstract interpretation techniques [15, 16] could be used to automatically infer properties about the results. In [16] they infer loop invariants that are then used to prove that the resulting matrix is e.g. upper-triangular, diagonal, etc. However, this approach seems to require that the result has similar shape as the argument(s).

8 Conclusions

In this paper we have described an approach to automatically verify that programs that manipulate matrices satisfy specifications given as contracts. Furthermore, the verifier checks well-formedness constraints such as absence of errors of integer over- and underflow, division by zero and matrix accesses out of bounds. The target for our approach is Embedded MATLAB and Simulink, which are programming languages with a strong focus on numerical computing. The most important goal is efficient handling of the built-in operations for matrix manipulation. Our approach handles the most common matrix manipulation functions in Embedded MATLAB, while-loops and recursion. The key feature is type inference to statically infer intrinsic types and shapes of matrices. Hence, the size of all matrices are static and determined at compile-time, which is the standard behaviour for Emedded MATLAB and Simulink. This allows efficient handling of the different MATLAB functions in a SMT solver. We evaluated two approaches: direct axiomatisation of matrix functions and expansion of matrix functions. We found that expansion works very well for relatively small matrices commonly found in embedded control and signal processing applications. This allows complete automation with relatively small annotation overhead. In the axiomatisation approach we need to either manually provide the needed specifications for recursively defined functions or have the verifier unfold the function definition, neither of which are desirable. We demonstrated the usefulness of our approach on a number of examples. The approach has also been used successfully to check a Simulink model that contained several hundred blocks. This model involved scalar control logic mixed with matrix calculations. Most of the matrix calculations were carried out element-wise on small vectors (1x3 and 1x4 element row vectors). However, matrix multiplication involving up to 6x4 element matrices was also included. Even if the verified properties focused on control logic, the example demonstrates that the approach scales to non-trivial models.

There are many directions for future work. Matrix accesses are now limited to one element or a complete row or column. Embedded MATLAB allows more flexibility and allows choosing any submatrix. This should not present any fundamental problem for our approach. Complete support for the control flow constructs in Embedded MATLAB should also be provided. Currently, only if-statements and while-loops are supported. The verifier also only checks partial correctness. The plan is to implement checks for termination of both iteration and recursion. Techniques based on abstract interpretation, e.g. [16], could perhaps also be used to infer the needed properties of recursively defined functions, which would allow for more automation when axiomatisation of functions are used. However, we believe that our approach is a good start towards fully automated efficient verification of Embedded MATLAB programs.

References

[1] G. Almási and D. Padua. MaJIC: Compiling MATLAB for speed and responsiveness. *SIGPLAN Not.*, 37(5):294–303, 2002.

[2] M. Barnett, B.-Y. E. Chang, R. Deline, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer and et. al., editors, *FMCO'05*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[3] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *Communications of the ACM*, 54(6), 2011.

[4] P. Boström. Contract-based verification of Simulink models. In *ICFEM2011*, volume 6991 of *LNCS*. Springer, 2011.

[5] P. Boström, R. Grönblom, T. Huotari, and J. Wiik. An approach to contract-based verification of Simulink models. Technical Report 985, TUCS, 2010. Tool: `http://users.abo.fi/pbostrom/slverificationtool/`.

[6] P. Boström, L. Morel, and M. Waldén. Stepwise development of Simulink models using the refinement calculus framework. In C. B. Jones, Z. Liu, and J. Woodcock, editors, *ICTAC'07*, volume 4711 of *LNCS*, pages 79–93. Springer, 2007.

[7] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[8] C. C. B. Jay and P. Steckler. The functional imperative: Shape! In *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP '98, pages 139–153. Springer-Verlag, 1998.

[9] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO2006*, volume 4111 of *LNCS*. Springer, 2006.

[10] B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Commun. ACM*, 54(5):88–98, 2011.

[11] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *SEFM'12*, volume 7504 of *LNCS*, 2012.

[12] L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[13] L. de Moura and N. Bjorner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 45–52, Nov 2009.

[14] L. de Rose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21:286–323, 1999.

[15] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS'10*, volume 6528 of *LNCS*. Springer, 2011.

[16] T. A. Henzinger, T. Hottelier, L. Kovács, and A. Voronkov. Invariant and type inference for matrices. In *VMCAI2010*, volume 5944 of *LNCS*, 2010.

[17] P. G. Joisha and P. Banerjee. An algebraic array shape inference system for MATLAB®. *ACM Trans. Program. Lang. Syst.*, 28(5):848–907, September 2006.

[18] K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT-solvers. In *SAC'09*. ACM, 2009.

[19] Mathworks Inc. Simulink. `http://www.mathworks.com`, 2014.

[20] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[21] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[22] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, January 1965.

[23] H. Xi. Dependent types in practical programming. In *POPL'99*. ACM, 1999.

[24] H. Xi. Dependent ML: An approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2), 2007.

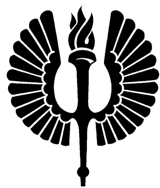
TURKU

CENTRE *for*

COMPUTER

SCIENCE

Joukahaisenkatu 3-5 A, 20520 TURKU, Finland | www.tucs.fi



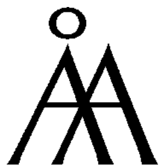
University of Turku

Faculty of Mathematics and Natural Sciences

- Department of Information Technology
- Department of Mathematics

Turku School of Economics

- Institute of Information Systems Sciences



Åbo Akademi University

- Department of Computer Science
- Institute for Advanced Management Systems Research

ISBN 978-952-12-3053-0
ISSN 1239-1891