VerSÅA user manual*

Pontus Boström

Department of Information Technologies, Åbo Akademi University,
Joukahaisenkatu 3-5, 20520 Turku, Finland
pontus.bostrom@abo.fi

1 Introduction

This document gives an overview on how to use VerSAA, which is a tool for automatically verifying correctness of Simulink models with respect contract annotations. The document starts with installation instruction. Then an overview of the contract format and the capabilities of VerSAA are given. This is followed by a few examples on how to verify different aspects of a number of small systems. The last two sections give a reference of supported blocks and functions. as well as advice on handling common issues.

2 Installing and running the program

VerSAA is a program written in Java that consists of collection of jar-files and some additional scripts to conveniently run the tool. The tool is run from the command line, but it can display results from the verification graphically.

Installation instructions:

1. Ensure that that a JVM (at least Java 1.6) is installed. One can be downloaded from `http://java.com`. Note that for a 64-bit version of VerSAA, a 64-bit version of Java is needed.
2. Unpack the downloaded VerSAA package in a desired folder. This will create a folder `VerSAA-xx` where `xx` is the version with all the files. No further installation steps are needed. The folder containing the program can also be moved freely[1].
3. Note that this version of VerSAA includes the SMT-solver Z3. This means the tool cannot be used commercially unless the user already has a commercial license for Z3. See `VerSAA-xx/z3/LICENSE.txt` for details.
4. Running the tool. Enter the folder `<tool folder>/VerSAA/`
   - Windows: Run `slverifier.bat <params> <simulink model file>`
   - Unix (Linux and Mac OS X):
     Run `./slverifier.sh <params>  <simulink model file>`

* Work done in The DiHy project coordinated by Fimecc and funded by Tekes

[1] VerSAA can be run independently of Matlab/Simulink or from within Matlab

– In Matlab: The tool can also be used from Matlab by calling the function `slverifier.m`. This function takes two argument, the first is the model name and the second is a cell array of strings giving the parameters. The command `help slverifier` gives detailed information[2].

As described above the program is a command line program that takes a Simulink model as argument. There are a number of parameters that can be given to the program. An overview all the different parameters are given here in order to have a complete list in one place. In the examples shown later, more in depth presentation of what the tool actual does will be given.

– `log (all | off)`. This parameter enable control of how much information the tool will print out. When `-log all` is used a lot of debugging information will be printed. For users, most will be of little interest.
– `gui`. Shows the result of the verification in a graphical user interface. This parameter is recommended when the tool is used to verify Simulink models.
– `nowf`. Do not do well-formedness checks. When this parameter is used the tool will *not* check absence of divisions by zero, square root of negative numbers, etc., underflows or overflows for integer arithmetic operation, as well as index out of bounds matrix accesses.
– `psdf`. Print the sdf graphs that are generated in the verification generation process. They are produced as pdf files in the working directory. Note that the tool Dot, which is part of the Graphviz package, is used for generating the pictures[3]. WARNING: Any file with matching name will be overwritten. This is mostly useful for debugging of the verification process and it is not useful for users.
– `expand`. This enables expansion of all matrix operators. This is the recommended method for verifying models and programs that use matrix calculations on small matrices.

When VerSAA is used from Matlab, it will use variables in the Matlab workspace. That is, if a variable in the workspace is used in the Simulink model the value it has in the workspace is used in the verification.

VerSAA can also be used to verify Embedded Matlab code. In that case, the argument to the verifier is an .m-file. Note that only a relatively small subset of Embedded Matlab is supported. The support is aimed at providing a more simple environment than Simulink to experiment with verification of Matrix-code. Verification of code that use matrices and vectors is discussed in Section 5.2.

3 Simulink and contracts

The language used to create models in Simulink is based on hierarchical data flow diagrams [6] (see e.g. Fig. 1). A Simulink diagram consists of functional blocks

[2] In order for the built-in JVM in Matlab to find Z3, it is necessary to set *java.library.path* to include the folder *VerSAA/z3/bin*. See the Matlab documentation on how to do this.
[3] Graphviz: `http://www.graphviz.org/`

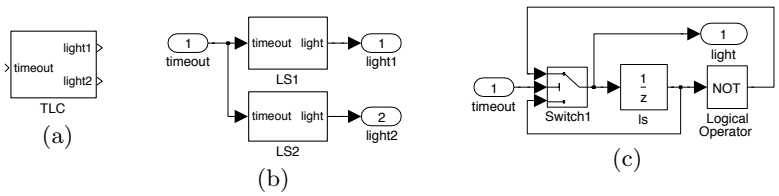

Fig. 1. (a) A subsystem that contains a simple traffic light controller, (b) its contents consisting of two individual light controllers and (c) the individual light controllers

connected by signals (wires). The blocks represent transformations of data, while the signals give the flow of data between blocks. Blocks can be parameterised with parameters that are set before model execution and remain constant during the execution. Blocks can also contain memory. Hence, their behaviour do not only depend on the current values on the inputs and the parameter values, but also on previous input values. The diagrams can be hierarchically structured using the notion of subsystem blocks, which are blocks that themselves contain diagrams. Here we use contracts [2] to give a high-level functional description of these subsystems. Simulink can model continuous, discrete and hybrid systems. The verification methods described here concern discrete periodic systems, which is one of the most common forms of control software. Note that multi-rate models are supported. However, they have to have the phase (also called offset) part of the sampling time set to zero.

To illustrate the use of Simulink, a small example that consists of a controller for a simplified traffic light system is given. The system consists of two lights that can be either *green* (true) or *red* (false). However, both lights should not be green at the same time. When a timeout signal has the value true, the lights change. The subsystem block *TLC* in Fig. 1 (a) contains the traffic light controller. A new light configuration is computed separately for each light by the subsystems *LS1* and *LS2* (Fig. 1 (b)) at each sampling instant. Both lights are switched in case *timeout* is true otherwise they retain their values (Fig. 1 (c)).

3.1 Verification based on contracts

Contracts describe valid behaviours of (atomic) subsystems in Simulink models, i.e., what are valid outputs in response to valid inputs. VerSAA verifies that a given Simulink subsystem annotated by a contract also satisfies it. To add a contract to a subsystem, the contract is written down in the *Description*-field[4] of that subsystem. The abstract syntax of contracts is shown in Fig 2 (a).

There c, u and y are identifiers and t is a type in the set $\{\mathsf{double}, \mathsf{int32}, \mathsf{int16}, \mathsf{int8}, \mathsf{uint32}, \mathsf{uint16}, \mathsf{uint8}, \mathsf{boolean}\}$ or matrices containing these types. E.g., a matrix of type double and dimensions $n \times m$ is declared as $\mathsf{matrix}(\mathsf{double}, n, m)$. Then

[4] Right click on the subsystem, choose block properties

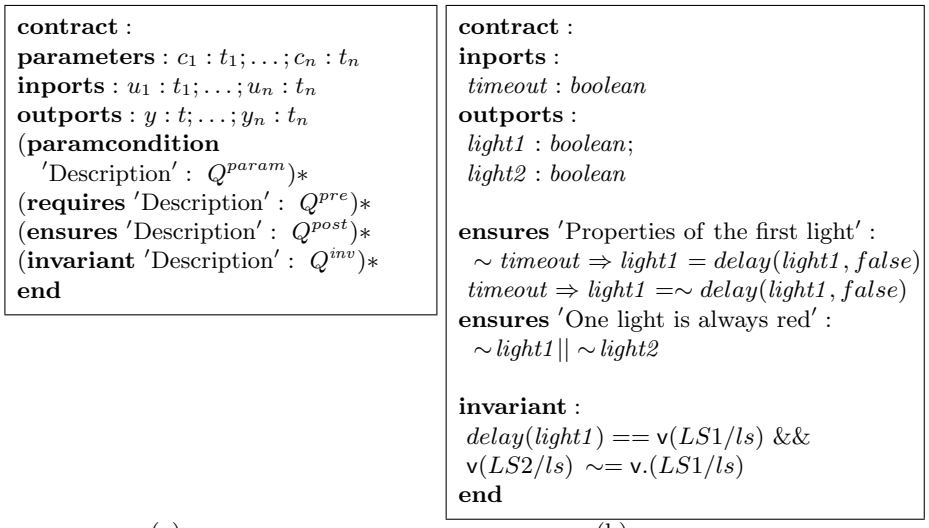

Fig. 2. (a) The abstract syntax of contracts and (b) an example contract that describes the traffic light controller subsystem

$z*$ denotes zero or more occurrences of z and Q denotes a predicate. The contract first declares the parameters, in- and out-ports of the subsystem. These are all given as lists of identifier-type pairs. The behaviour of the subsystem is described by a set of conditions. Each condition can have an optional description. Here Q^{param} describes the block parameters used in the subsystem, Q^{pre} is the precondition describing the valid inputs and Q^{post} is the postcondition describing valid outputs. In the contract conditions it is also possible to refer to old input and output values. The syntax $\mathsf{delay}(x, i)$ denotes the previous sample of port x with the initial value i. To verify the subsystem, it is then necessary to describe how these old port values are actually represented in the Simulink implementation. The condition Q^{inv} is used to describe how the delayed ports relate to block memories in the subsystem diagram. This condition describes a property that is *invariant* during system execution. This invariant property is the only thing remembered between sampling instances, so it has to carry enough information to enable the correctness proof. The contracts here have a similar structure and can describe the same type of behaviour as the ones for reactive components in [5].

To give an idea of how contracts can be used, a contract describing the functionality of the traffic light system from Fig. 1 is given in Fig. 2 (b). We want to prove that both lights are not green at the same time, that is: $\sim light1 || \sim light2$. We also want to say that the lights are only changed when the input timeout is true. The contract first declares the inputs and outputs together with their type. To give an accurate description of the system, we cannot just describe its input-output behaviour, since the output depends on the internal state given

by the *Unit Delay*-blocks. We use a delay of the first light *light1* to say that the light is unchanged if *timeout* is false and flipped if *timeout* is true. The initialisation of this light is here assumed to be red (false). The postconditions then encode the desired behaviour of the controller. The invariant in **invariant** then describes how the memories in the *Unit delay*-blocks in the subsystems *LS1* and *LS2* relate to the delay $\mathsf{delay}(\mathit{light1})$. The invariant also states that the lights always have different color. Here a function v is used to map block memories to variable identifiers. This mapping is discussed more in the examples in Section 6. The concrete syntax used in the contract conditions is inspired by the syntax of Matlab expressions [2].

4 Principles for verification

The approach to verification is described in detail in [1, 8, 2]. Here only a brief overview is given. Simulink is used to develop control systems, where the interaction of programs with their environment rather than their input-output behaviour is important. Hence, we are here interested in *reactive systems* where the execution of the behaviour of the Simulink model is observed during execution. We can observe the value of a port or block memory only at the time points when it is sampled. Note that Simulink is only guaranteed to behave in this manner for discrete models with fixed step solver. In other cases this is an abstraction. The abstraction is adequate for the discrete parts of models with even continuous time behaviour.

When verifying Simulink models we are interested in showing that the observable behaviour does not violate some desirable property P, which is here described by contracts. A Simulink model can be considered a *labelled transition system* with state s built from the block memories, as well as transitions labelled by port values p. Figure 3 illustrates the situation. Here s_i represents a value on the internal state, while p_{ij} represents a value on the ports. Consider the property P stating that the state s is either s_1 or s_3, $P \mathrel{\hat{=}} (s = s_1 \vee s = s_3)$. We can easily see that a state violating this property is not reachable from the initial state. There are two approaches to verify that property P holds.

1. We can compute all states reachable from an initial state s_i and check that we will never reach a state s where P does not hold.
2. We can prove this fact inductively. We show that any state s where P holds then P will hold again after any transition from s. Then we also need to show that the initial state satisfies P.

In VerSAA approach 2 is used. The advantages of this approach is that, given a suitable P (a suitable *invariant*), we can check the property one transition (model execution) at the time. This leads to potentially better scalability than in approach 1. The disadvantage is that a suitable P that enables the proof is needed, which might be stronger than the property of interest. This means that this approach might require more (manual) work than approach 1.

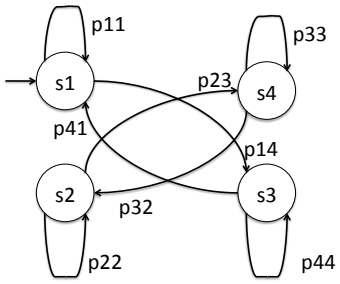

Fig. 3. A state machine used for demonstrating the principle behind verification

Consider the simple Simulink subsystem modelling a traffic light controller in Figure 1. The internal state of that model is given by the unit delay blocks *LS1/ls* and *LS2/ls*. We model the state as the tuple (*LS1/ls*, *LS2/ls*). The states in the model now correspond to the states in the transition system as follows $s1 = (true, false), s2 = (true, true), s3 = (false, true), s4 = (false, false)$. The ports labels p are given by the tuple of inport and outport values $(timeout, light1, light2)$. Here we have that $p11 = (false, true, false), p14 = (true, false, true), p44 = (false, false, true), p41 = (true, true, false), p22 = (false, true, true), p23 = (true, false, false), \ldots$. This shows how one can view the Simulink model as a labelled transition system. As can be seen from the transition system representation, a state where both lights are switched on at the same time cannot be reached from the initial state. When using approach 2 for verification, a suitable property (invariant) is $P \mathrel{\hat{=}} (s = s1 \vee s = s3)$ or in other words $P \mathrel{\hat{=}} \neg((LS1/ls) = (LS2/ls))$ as also stated in the contract in Fig. 2(b). From any such state we can then show that for any input on *timeout* the model will end up in such a state again after execution. The rest of the contract conditions are formed by combinations of states and port values. It is easy to check that they hold for each transition (model execution).

As already mentioned, VerSAA checks that each time the model is executed, it behaves according to the contract specification of the subsystems. The correctness is verified one subsystem at the time. This will guarantee that the complete model will be correct [1]. The verification is thus *compositional*. For each diagram D in subsystem with a contract the tool checks that when the internal diagram is executed in a state where the invariant on the internal state holds, then for any input satisfying the precondition the diagram will establish the postcondition and invariant again. For multi-rate models, we cannot only look at one execution of a subsystems, but we have to check the property (at least) for the shortest repeating behaviour. This means for a time period of the least

common multiple of the sampling periods, as offsets (phases) on sampling times are not allowed. For multi-rate model the invariant should then hold for all time instants when the subsystem behaviour repeats, i.e., for the period that is the least common multiple of the periods of the inputs and outputs. The technical details of the verification can be found in [2, 8, 1, 9].

The principle in VerSAA is that the verification should be completely automatic. For each subsystem, a number of verification conditions are generated. The verification conditions are such that they are only valid (hold for all variable values) if the subsystem is correct. The SMT-solver Z3 is used to automatically show validity. Z3 [3, 4] has background theories for many constructs needed in embedded systems, such as linear and non-linear arithmetic, linear integer arithmetic, arrays, bitvectors, etc. The properties that can be proved is largely dependent on this tool. Z3 is only a semi-decision procedure for some theories and thus it cannot always decide if a formula is valid or not. The result of the verification in VerSAA can be:

- *Correct* - the verified subsystem is correct.
- *Incorrect* - the verified subsystem is incorrect. Then the failed conditions are displayed. As a failed condition can be identified fairly precisely it should give a good indication of where the problem is located.
- *Unknown* - Z3 cannot decide if the subsystem is correct or incorrect. This often occurs if a subsystem is incorrect, but an undecidable theory (like non-linear real arithmetic) is used.

5 Capabilities

VerSAA can do verification of multi-rate Simulink models and a relatively small subset of Embedded Matlab. It can handle multi-dimensional data, which is typically used in control and signal processing applications. This section gives an overview of the capabilities and limitations of the tool.

5.1 General limitations

The subset of Simulink supported by VerSAA is not complete. However, there is a sufficiently large library of supported blocks to make the tool useful for verification of real systems. The limitations are currently:

- VerSAA supports verification of multi-rate subsystems where each rate has zero phase. Furthermore, only the rate transition blocks can have inputs and outputs of different rates. This is similar to Simulink Design Verifier.
- VerSAA only supports a subset of the Simulink blocks. The list of blocks supported by VerSAA can be found in Section 7.
- VerSAA supports only a subset of the built in Matlab functions. The list of built in Matlab functions supported by VerSAA can be found in Section 7.

5.2 Matrix and vector support

VerSAA has support for calculations using matrices and vectors (see [9] for details). The goal is that the operators and functions should handle multi-dimensional data in the same manner as Matlab/Simulink. Note that matrices and vectors are not allowed to be dynamically resized, but this is not recommended in embedded code anyway. The syntax is also simplified in order to allow more efficient automated verification and many of the more unusual ways to index elements in matrices are not supported. Dimensions of signals are inferred as in normal Simulink. Note that only up to 2 dimensional signals are supported. Furthermore, VerSAA has only matrices (vectors are 1xN or Nx1 matrices) and the special array type in Simulink is not used. An NxM matrix has N rows and M columns. For example, a variable or port v can be declared to be a 3x2 matrix using the notation $v : \mathsf{matrix}(\mathsf{double}, 3, 2)$. The notation $v : \mathsf{matrix}(\mathsf{double}, 3)$ states that v is a 3x1 matrix (column vector). The first index gives the number of rows and the second the number of columns. To do the dimension inference, the dimensions must be numeric constants. Hence, writing $v : \mathsf{matrix}(\mathsf{double}, N, N)$ is not allowed, unless the parameter N is defined in the Matlab Workspace (the value of N is then used in the inference) and VerSAA is used from Matlab. This also means that matrix creation functions such as *zeros*, *ones*, etc. need to have arguments that are numeric constants or the parameters can be directly found in the Matlab workspace.

One significant difference from Simulink is that VerSAA does not do implicit type conversions. In particular, the indices in matrices and vectors are integers (int32). This means that all expressions used as matrix indices must be of type integer and not double. Explicit type conversion blocks or functions must be used otherwise.

Matrices are translated to (two dimensional) arrays in Z3. There are two ways to verify programs that use matrices. The default approach is to use axioms defining the properties of the supported functions. This works well in some cases. However, if functions that have recursive definitions are used then performance will be poor. This is fairly common, e.g., matrix multiplication, functions *sum*, *prod*, *min*, etc. with one argument are all examples of this type of function. The second approach is enabled with the parameter `-expand` to VerSAA. The matrix operators are expanded in such a manner that Z3 only deals with scalar operators and functions. This is possible, since the size of all data is static and known. This also keeps the used subset of the array theory decidable. This will work well for relatively small matrices and vectors. However, this does not scale to very large matrices and vectors, since the expanded formulas become too large.

VerSAA can handle the most common element-wise operators and functions. It can also do matrix multiplication and knows the functions e.g. *zeros*, *ones*, *size*, *length* and *transpose*. A more detailed list of supported functions and blocks can be found at the end of the document.

5.3 Runtime errors

Aside from checking that subsystems fullfill their contracts, VerSAA will by default also check that all operations are well-defined. VerSAA will check that there are no over- or underflows in integer calculations (the standard signed and unsigned integer types are supported). VerSAA will also check that all functions are used correctly. That is, VerSAA will check that, e.g., there is no division by zero, square root is not taken of a negative number and that exponentiation returns a real number[5]. Integers are used to index matrices. VerSAA will also check that all matrix accesses are within matrix bounds.

The software in control systems is often arithmetic intensive. Conceptually, the calculations are usually then carried out using real arithmetic. However, in Simulink these calculations are then often implemented using floating-point arithmetic according to IEEE 754 standard. Verification involving floating-point computation is hard [7]. For example, Simulink Design Verifier approximates floating point arithmetic with rational number arithmetic [6]. We also approximate floating-point numbers with real numbers. This approach helps to show that the principles of the system are correct. However, many defects relating to when the behaviour of floating-point and real arithmetic differ will go undetected. This problem is not just theoretical, as rounding errors commonly cause significant problems with accuracy of the results of calculations.

6 Examples

To get a better idea of how VerSAA can be used to verify different types of properties about Simulink models, a few small examples are given.

6.1 A multi-rate Simulink model

The following multi-rate model computes decimal value of the four latest binary inputs delayed by one. The output is updated on a rate four times slower than the input. Hence, the sampling period of the input is one, while the sampling period of the output is four time units. The diagram implementing the functionality is shown in Figure 4. The contract is written in the *Description* field of the *Block properties* of the subsystem. In order to remember the four last inputs, delay expressions are used. Delay expressions can only be used to delay inports or outports or delays of them. This restriction is to simplify handling of sampling times. The contract is shown below:

```
contract:
  inports:
    Bool:boolean
  outports:
    Decimal:double
  ensures 'Main condition' :
```

[5] This is handled in Simulink by returning mathematically incorrect (but useful) results.

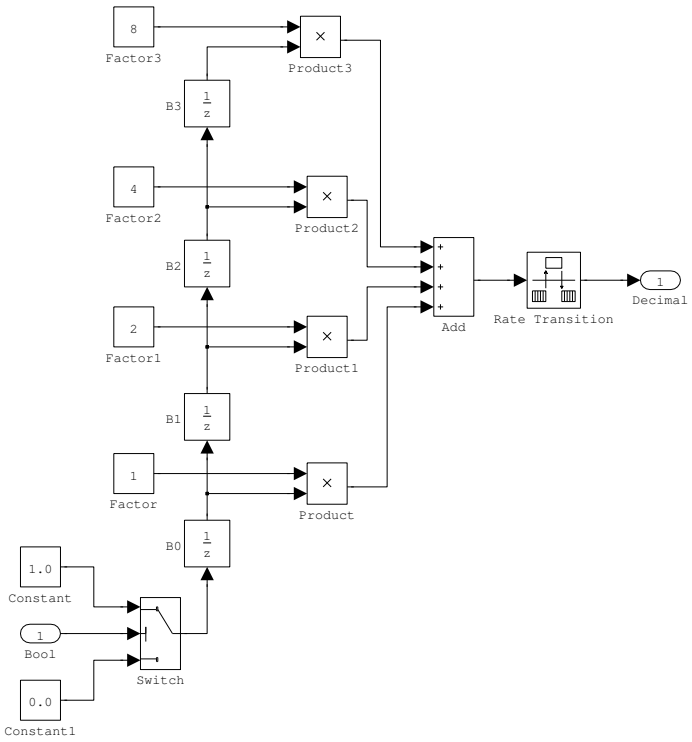

Fig. 4. A Simulink diagram implementation of the binary to decimal converter

```
    Decimal==
    bool2real(delay(Bool,0))*1+
    bool2real(delay(delay(Bool,0),0))*2+
    bool2real(delay(delay(delay(Bool,0),0),0))*4+
    bool2real(delay(delay(delay(delay(Bool,0),0),0),0))*8

  ensures 'Output should be within limits'  :
    Decimal<=4;
    Decimal>=2

  invariant    'First part':
    bool2real(delay(Bool,0))==B0$X &&
    bool2real(delay(delay(Bool,0),0))==B1$X

  invariant     'Second part':
    bool2real(delay(delay(delay(Bool,0),0),0))==B2$X &&
    bool2real(delay(delay(delay(delay(Bool,0),0),0),0))==B3$X

end
```

A delay expression has the same sampling as the port that it delays. Using delay expressions is recommended over explicitly declared memories[6], since it usually results in specifications that are shorter and easier to read. Note also

[6] See [2, 1] for this way to specify memory of subsystems

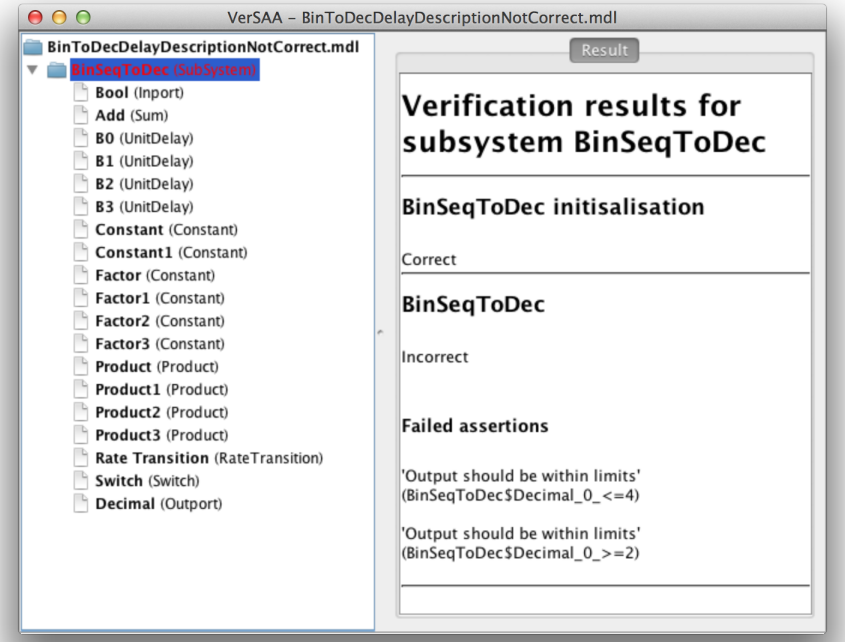

Fig. 5. Result of verifying the BinToDec model, where some postconditions do not hold

that initialisation (the zeros) can be omitted, as long as the delay is initialised at least once. In the invariant we need to say how block memories relate to the delays. We refer to the memory of e.g. the delay block $B0$ as $B0\$X$. To refer to a block B in a subsystem S, the syntax $S\$B$ is used.

The result from verifying the model with VerSAA can be seen in Figure 5. The command used was here:

```
slverifier  -gui <BinToDec model>
```

The entire model is verified when running VerSAA on a model. The user interface is divided into two parts: a tree view to the left and a text box to the right. The tree view shows an overview of the verification results of the Simulink model, while the text box can show more detailed information for the chosen subsystem. The subsystems with contracts are marked in different colors in the tree view depending on the verification result:

– Green: The subsystem is correct.

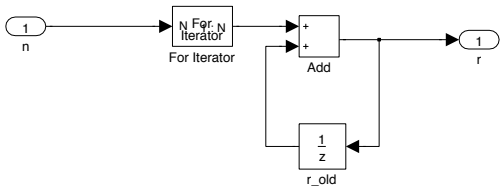

Fig. 6. A Simulink diagram from a for-iterator subsystem to compute the sum of all natural numbers up to n

- Red: The subsystem is incorrect. The properties that were not verified are shown to the right. By writing properties separately with good descriptions, it is easier to locate the problem precisely to some simple condition.
- Orange: The status of the subsystem is unknown. Also here the properties that were not verified are shown to the right.
- Gray: The subsystem could not be verified. This is typically because the subsystem contained some block (e.g. Stateflow) the tool did not understand.

By clicking on a desired subsystem in the tree view more information is provided to the right. In Figure 5 the subsystem *Bin2Dec* has two failed assertion. The postcondition *'Output should be within limits'* does not hold. The real lower limit is 0 and the real upper limit is 15. The failed assertions are shown to the right. Note that since VerSÅA handles multi-rate models the buffer index for the port the problem occurred is also given. In this case it is the first sample directly, which is denoted by the suffix _0_ after the port name.

6.2 A for-iteration subsystem

This example involves a for-iterator subsystem that computes the sum of all natural numbers up to a limit given as parameters. We have first defined the function sum_up. This function is defined recursively. We have the following axioms:

$$\forall x : int32.(x > 0 \Rightarrow \mathsf{sum\_up}(x) = x + \mathsf{sum\_up}(x-1)) \quad (1)$$

$$\mathsf{sum\_up}(0) = 0 \quad (2)$$

The verification of the subsystem will also proves the well-known formula $\mathsf{sum\_up}(n) = n(n+1)/2$. Contents of the subsystem is shown in Figure 6. The inport n gives the number to sum up to and the outport gives the result. The delay block stores the intermediate values. The contract for this subsystem is given as:

```
contract:
inports:
   n:int32
outports:
   r:int32
requires:
   n>0
```

```
ensures:
   r==sum_up(n) &&
   r==(n*(n-1))/to_int32(2)
invariant:
   r_old$X==sum_up(k-1)&&
   r_old$X==(k*(k-1))/to_int32(2)
end
```

The contracts for for-iterator subsystems are a bit different than for normal subsystems. The idea here is that the for-iterator subsystem computes a function where the inports are arguments and the outports provides the results (there is no state remembered between invocations here). The contract cannot contain delay expressions (or specification variables (memories) [1]). The memories of the blocks in the diagram are used to store intermediate results. To verify the for-loop we need a loop invariant that describes the intermediate results. The loop invariant captures the situation between loop iterations. We construct the loop invariant in such a manner that when the invariant holds and the loop has finished (the loop guard does not hold anymore) then that implies the desired postcondition. The idea in the verification is that if we assume that the loop invariant holds before an arbitrary iteration and then check that it holds after, then it will always hold between iterations. Then in the final step we prove that, given that the invariant holds, when the loop has terminated the desired post-condition has been reached. Note that a loop index variable k is automatically defined and can be used in the invariant. Here we prove inductively that the block memory $r\_old\$X$ contains the intermediate result during the execution of the loop. Note also that the invariant in this case can refer to the inports. The postcondition then only relates inputs and outputs. Based on the invariant we can then prove the postcondition when we know we just exited the loop. We have the following restrictions for for-iterator subsystems:

- Only one-based indexing is supported
- The iteration limit has to be given as an inport
- The increment is one and not given as a inport.
- Memories are reset at the start of every execution of the block.

6.3 Finding the minimum element in a vector

This example illustrates the use of multi-dimensional data. The example consists of a for-iterator subsystem that returns the minimum value (v) and the index (i) of the minimum value from a vector a. The diagram for the subsystem is shown in Fig. 7. The contract is given as:

```
contract:

inports:
   a:matrix(double,1,N)

outports:
   i:int32;
   v:double

ensures 'Index limits':
```

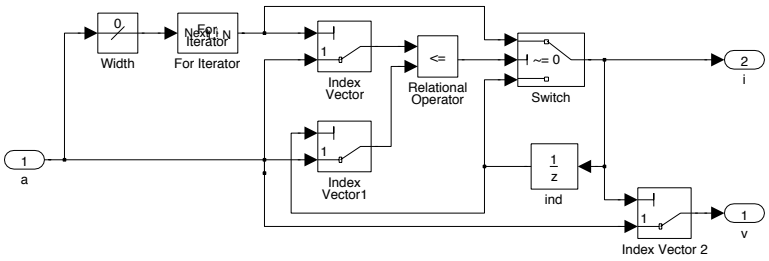

Fig. 7. A Simulink diagram for a for-iterator subsystem to find the minimum element of a vector a

```
    1<=i && i<=length(a)

ensures 'Properties of min index':
    all(a(i)<=a)

ensures 'Properties of min value':
    a(i)==v

invariant:
   1<=k &&k<=length(a);
   1<=ind$X && ind$X<=length(a);
   \forall j:int32 . ((1<=j && j< k) ==> a(ind$X)<=a(j))

end
```

The contract first gives the types of the inputs and outputs. Here N must be given a value in the base workspace of Matlab, in order for the type inference to infer dimensions. The type of indices in matrices is $int32$. The first postcondition states that the index i is within matrix bounds, the second states that all the elements in the vector a are greater or equal than the element at position i and the third one states that v is equal to the smallest element. The $<=$ works elementwise on the vector a and the functions *all* and *any* are generalised AND and OR respectively. Using the builtin Matlab functions it is possible to write very compact specifications. The tool can also expand these definitions, and the verification can therefore be very efficient. The invariant is then needed to prove the postcondition. Recall that k is the automatically generated loop index. We need to state explicitly that k is within matrix bounds. Then we also need to state that the memory $ind\$X$ in the delay block *ind* in the diagram, which is used to stored the minimum found so far is within matrix bounds. Finally, we need to state that $ind\$X$ contains the minimum found in vector a up to k.

6.4 Examples involving embedded Matlab

VerSAA can also be used to verify programs written in a subset of Embedded Matlab. This features is mostly aimed at giving a simple environment to try out verification of matrix manipulations, but it can be used for real verification tasks as well. An example of an Embedded Matlab program is given in Fig. 8 and more examples can be found in [9]. The function of interest is max_f that finds the

maximum element in the array a. Note that in order to infer numeric dimensions, the program need to have dimensions on all parameters given. Hence, the small function *test_max* that calls the function *max_f* is needed. Here *max_f* is tested for a 10×1 vector. One can think of it as exhaustive testing of the function for a vector of a specific size.

Specifications are given after %@, since then the specifications will be comments in Matlab and the functions are valid Matlab code. The first specification regards type parameters, as the functions are polymorphic. The function works for any numeric type t (double, int32, int16,...) and for any dimension n. The parameter a is then declared to be of type $\mathsf{matrix}(t, n, 1)$ meaning a matrix with element of type t and dimension $n \times 1$. The return value is then a scalar (a 1×1 matrix) of type t. There is no precondition. The postcondition states that all elements of a are smaller or equal to m and that there exists an element in a equal to m.

The implementation is essentially a while loop where the minimum element is searched for. Note that we initialise the loop index i to be of type $int32$. This is necessary, since integers need to be used as indices into matrices. Here it is easy to find the loop invariant. We say in the invariant that m is the minimum element in vector a up to index i. We also need to state the bounds on i. When the loop guard does not hold anymore, then $i = n + 1$ and then according to the loop invariants m is the minimum of the whole vector a. Finding the loop invariants can be very challenging for more complicated programs. However, in more complex algorithms the same information is usually captured in diagrams describing the situations between loops. Then it is just a matter of expressing this information formally.

Note that VerSAA does not check that the loops actually terminate. Hence, VerSAA checks *partial correctness*, which means that it only guarantees that if all the functions terminate the result produced is correct. Termination checks could be added to VerSAA, but at the moment they are not supported. In the case of the for-iterator subsystem, termination is guaranteed by behaviour of the subsystem under the restrictions used in VerSAA.

6.5 Comparision with Simulink Design Verifier

From a user point of view the biggest difference between the tools is probably that verification conditions and assumptions are given as special blocks in Simulink Design Verifier (SLDV). The verification approach is quite similar in VerSAA and Simulink Design Verifier. SLDV uses an approach based on k-induction where it tries to infer some invariant over a number of steps and then verify that the model satisfies the invariant. Bounded model checking can also used to search for violations over only a finite time. In VerSAA, an inductive invariant has to be given explicitly for the internal state (given in the *invariants* clause of the contract). Given this invariant and inputs satisfying the preconditions, then the postcondition and the invariant is established. Furthermore, VerSAA supports refinement-based compositional verification for building correctness arguments for large systems.

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

Fig. 8. A MATLAB function for finding the index of the minimum element in a column vector, annotated with contracts.

VerSAA (via Z3) supports non-linear arithmetic better than SLDV. However, currently VerSAA does not support common Simulink features such as Stateflow and embedded MATLAB[7], which SLDV supports. The support for matrix data seems fairly weak in SLDV, where scalability is often poor [9]. Furthermore, SLDV unrolls loops when checking iteration and fixed loop bounds need to be provided. This is more restrictive than VerSAA, but it is still useful for embedded software where the number of iterations should have a strict upper bound. Also no loop invariants need to be provided when unrolling loops. Note that VerSAA does not currently prove termination of loops (it proves *partial correctness*)

7 Supported blocks and functions

Supported blocks. A relatively large number of blocks are supported. The supported structuring mechanisms are: *Atomic subsystems*, *virtual (normal) subsystems*, *enabled subsystems* and *for-iterator subsystems*. A list of the supported builtin blocks is given in Table 1. Additionally, almost all sink blocks are also supported. However, their behaviour is not modelled. The block definitions are

[7] Embedded Matlab in Simulink models is actually a Stateflow chart with a Embedded Matlab function embedded, hence it is not supported in verification of Simulink models

Table 1. List of the supported blocks

Block	Comment
Sum	Supports arbitrary number of inputs, as well as giving operators for ports
Product	Supports arbitrary many ports as well as giving operators for ports (see limitations for $*$ and / in Table 2)
Gain	Supports both elementwise and matrix multiplication
Logic	Supports AND, OR and NOT. Arbitrary many inputs for AND and OR supported
Math	See Table 2 for supported functions
MinMax	Supports both one and two input version of *min* and *max*
Trigonometry	See below for which functions are supported
Relational Operator	All alternatives are supported, but only = and $\sim=$ works for booleans
Signum	
Abs	
Unary Minus	
Switch	Comparing with $\sim= 0$ also works for booleans
Constant	If a type of the output is given in the block it is respected
Unit Delay	
Memory	
Dead Zone	
Saturate	
Bias	
Datatype conversion	Support conversion between most types
Multiport Switch	Supports arbitrary many inputs
Index Vector	
Matrix Selector	
Mux and Demux	
Width	
Rate transition	This is the only block that tolerates different rates on the ports
Backlash	
Enabled subsystems	The enable port block is then also supported
Merge	
For iterator subsystems	The limitations discussed earlier applies
InPort & OutPort	

stored in an XML- file

`<tool-folder>/config/functions.xml`

The format of the XML-file is described in [2]. This file contains the list of all blocks that are currently supported, as well as the behaviour they are assumed to have. However, it is currently not possible to just add new block definitions to this file. The reason is that the blocks also need to be known by the typechecker. This is something that will be fixed in the future.

There is an option for handling blocks that have no definitions in VerSAA. The verification of a subsystem relies on contract descriptions of subsystems lower in the subsystem hierarchy. If an unknown block is encapsulated in a subsystem with a contract describing its behaviour, VerSAA can use the contract information to verify the rest of the model. However, as the content of that subsystem (the unknown block) cannot be analysed, VerSAA cannot check that the subsystem correctly satisfies its contract. The correctness has to be verified e.g. by testing or with Simulink Design Verifier. As the correctness has to be verified externally, the contract must be written carefully to not introduce false assumptions. More generally VerSAA will not verify subsystems with contracts that contain unknown blocks. However, it will use the contract descriptions when verifying the rest of the model. This enables verification of large models where some parts might contain unsupported constructs.

Supported functions. A number of functions from Matlab are also supported. There are two classes of supported functions. The first class is completely supported functions, where VerSAA knows the function definition. The second class is functions, where VerSAA will treat the function as an uninterpreted function symbol. However, VerSAA might contain assumptions about the function. The currently supported functions, as well as the assumptions known about them are listed in Table 2. The table gives the function definitions for elementwise functions when real scalar numbers are used. The definitions are straightforward to extend to matrices.

All supported functions with corresponding assumptions known about them is given in the file:

`<tool-folder>/config/functions.def`

The user can add functions definitions here. Note also the pattern declaration $: pat\{\ldots\}$ for each condition in the file. This is used by Z3 as hints for how to instantiate the quantifiers [4].

8 Common issues

VerSAA complains about syntax errors. The following problems are common: The arguments for the range operator : have to be constants, which is checked already at the syntactic level. Also syntax such as $a(2:)$ for indexing

Table 2. List of the supported functions

Function	Assumption
$+, -, .*, *, ./, /$	Arithmetic opertaors. / can only be applied to scalars.
$<, <=, ==,$ $\sim=, >=, >$	Relational operators. Only == and $\sim=$ can be applied to booleans.
$\&\&, \|\|, \sim$	Logical operators. The arguments have to be booleans.
$a : b : c$	The range operator : works as in Matlab, but the arguments a, b and c must be numeric constants.
$a(:, c)$	The range indexing operator : works as in Matlab, but it can only be used to index a whole row or column. Here c must be a scalar.
min	$\forall x : t, y : t.(\mathsf{min}(x, y) = \textbf{if } x \leq y \textbf{ then } x \textbf{ else } y \textbf{ end})$
max	$\forall x : t, y : t.(\mathsf{max}(x, y) = \textbf{if } x \geq y \textbf{ then } x \textbf{ else } y \textbf{ end})$
abs	$\forall x : t.(\mathsf{abs}(x) = \textbf{if } 0 < x \textbf{ then } -x \textbf{ else } x \textbf{ end})$
sgn	$\forall x : t.(\mathsf{sgn}(x) = \textbf{if } 0 = x \textbf{ then } 0 \textbf{ else if } x < 0 \textbf{ then } -1 \textbf{ else } 1 \textbf{ end end})$
square	$\forall x : t.(\mathsf{square}(x) = x. * x)$
sqrt	$\forall x : t.(x >= 0 \Rightarrow \mathsf{sqrt}(x). * \mathsf{sqrt}(x) = x)$
sin	$\forall x : t.(-1 \leq \mathsf{sin}(x) \wedge \mathsf{sin}(x) \leq 1)$
cos	$\forall x : t.(-1 \leq \mathsf{cos}(x) \wedge \mathsf{cos}(x) \leq 1)$
tan	Uninterpreted function
pow	Supported by Z3
saturate	$\forall x : t, y : t, z : t.(\mathsf{saturate}(x, y, z) = \textbf{if } x < y \textbf{ then } y \textbf{ else if } z < x \textbf{ then} z \textbf{ else } x \textbf{ end end})$
to_real	Built into Z3 to convert integers to reals
to_int	Built into Z3 to convert reals to integers. There are also functions to_int32, etc for converting to particular integer types.
intXX	Converts a double to the given integer type XX.
uintXX	Converts a double to the given unsigned integer type XX.
isinteger	$\forall x : double.(\mathsf{isinteger}(x) \Leftrightarrow (x = \mathsf{to\_real}(\mathsf{to\_int}(x))))$
bool2real	$\mathsf{bool2real}(True) = 1.0 \wedge \mathsf{bool2real}(False) == 0.0$
real2bool	$\forall x : double.(\mathsf{real2bool}(x) = (x \neq 0.0))$
size	Returns the size of the argument as a two element vector
length	Returns the length of the argument
zeros	Returns a matrix filled with zeros. The arguments must be constants.
ones	Returns a matrix filled with ones. The arguments must be constants.
all	Generalised AND that collapses a matrix
any	Generalised OR that collapses a matrix
sum	Sum of one argument that collapses a matrix
prod	Product of one argument that collapses a matrix
min, max	Min and Max of one argument that collapses a matrix
transpose	Transpose of a matrix
factorial	$\forall x : t.(x > 0 \Rightarrow (\mathsf{factorial}(x) = x * \mathsf{factorial}(x - 1)))$ $\mathsf{factorial}(0) = 1$
sum_up	$\forall x : t.(x > 0 \Rightarrow (\mathsf{sum\_up}(x) = x + \mathsf{sum\_up}(x - 1)))$ $\mathsf{sum\_up}(0) = 0$

vectors is not supported. The transpose operator $'$ cannot be used, since it is used for other purposes. The function *transpose* has to be used instead. Note also that that the equals operator is == and assignment operator is =.

VerSAA complains about unknown blocks. VerSAA can only handle a subset of the Simulink blocks. See Table 1 for a list of supported blocks. See also the previous section on how to deal with the situation. Note that even the built in Simulink blocks are sometimes library blocks (included in models as references). VerSAA cannot handle library blocks, so the best option is to inline all reference blocks before attempting verification.

VerSAA complains about unknown identifiers. Check that all variables are spelled correctly in the contract. Also check that all references to block memories in the *invariant* section are done correctly. VerSAA only obtains variable values from the Matlab workspace when run from Matlab. Therefore all block parameters (which are not numeric values) used in a diagram need to be declared in the contract of a containing subsystem otherwise.

VerSAA complains that it cannot derive sampling rates. The inference of sampling times in VerSAA is more strict than in Simulink. It also essentially only works forward in the model. This means that typically all constant blocks and inports on the highest level in the subsystem hierarchy need to have sampling times given. Rate transition blocks also have to be used to connect parts of the model with different sampling times.

VerSAA complains about type errors even though the Simulink approves of the typing. The type inference algorithm in VerSAA might not produce exactly the same typing as the one in Simulink. The typing in VerSAA is more strict in order to generate efficient verification conditions for the verifier. There are a number of things to consider:

- Doubles and integers cannot be mixed in computation. Thus it is impossible to e.g. add an integer with a real number. The reason is that real and integer arithmetic are separate theories in Z3.
- A number written with a decimal point, e.g., 2.0 will always be treated as a double. If a number is written as an integer, e.g., 2 its type can be either an integer or double depending on where it is used. Additionally 1 and 0 can also be booleans. If VerSAA complains about typing problems it might be because it cannot determine the correct type of the numbers involved. This can be a problem especially in functional blocks if parameters are given as numeric values. One suggestion is to use the the type casting function e.g. $int32$ to cast an expression to the desired type. However, this might affect negatively on the performance of Z3 (but it seems to work well).

– Prefer to use parameters declared in contracts (or in the Matlab workspace) as parameters to blocks instead of numeric values. Then the type and dimension is given explicitly, which makes the inference more accurate. Also, if parameters are set in the Matlab workspace, they have to be given the correct type (not the default double if that is not the desired type, of course).
– Check that all ports in the contracts have correct names (corresponds to the ones in the subsystems). Non-matching names can give strange type errors.

VerSAA complains about matrix dimensions even though the Simulink approves of them. Also here VerSAA is more strict. There is no array type in VerSAA and all vectors are either $n \times 1$ or $1 \times n$ vectors, so this has to be handled carefully.

VerSAA returns the result *Unknown* even though the subsystem seems to be correct. Z3 is only a semi-decision procedure in many cases. Thus sometimes it cannot verify that some property holds even though it does. One useful trick is to assign concrete values to parameters to eliminate non-linear arithmetic (or reduce the amount of it). In case the tool is called from Matlab, then parameter definitions from the Matlab root workspace will be used, which should help. One can also give concrete values to the inports in the precondition in order to verify the system for some specific input values. Note that for incorrect models where matrices or non-linear arithmetic is used, *unknown* will often be reported. Z3 will often also get stuck in an infinite loop then.

References

1. P. Boström. Contract-based verification of Simulink models. In *ICFEM2011*, volume 6991 of *LNCS*. Springer, 2011.
2. P. Boström, R. Grönblom, T. Huotari, and J. Wiik. An approach to contract-based verification of Simulink models. Technical Report 985, Turku Centre for Computer Science (TUCS), 2010.
3. L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS2008)*, volume 4963 of *LNCS*, pages 337–340, Budapest, Hungary, 2008. Springer.
4. L de Moura and N. Bjørner. Z3 - a tutorial. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.
5. F. Maraninchi and L. Morel. Logical-time contracts for reactive embedded components. In *30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04*, Rennes, France, August 2004.
6. Mathworks Inc. Simulink. `http://www.mathworks.com/products/simulink`, 2010.
7. D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3), 2008.
8. J. Wiik. Contract-based verification of multi-rate Simulink models. Master's thesis, Åbo Akademi University, 2012.
9. J. Wiik and P. Boström. Contract-based verification of MATLAB and Simulink matrix-manipulating code. Technical Report 1107, TUCS, 2014.