Symbolic Simulation of SystemC AMS Without Yet Another Compiler

Carna Zivkovic, Christoph Grimm
TU Kaiserslautern, Chair of Design of Cyber-Physical Systems
Gottlieb-Daimler-Strasse 49, 67663 Kaiserslautern, Germany
Email: zivkovic|grimm@cs.uni-kl.de

***Abstract*—Modeling languages first of all support simulation that is considered as reference by designers. Formal verification and synthesis share the same languages. However, they usually require a separate, dedicated compiler. As modeling languages such as SystemC have reached a high complexity, it is very hard to support reasonably large language subsets, and to guarantee consistency with simulation semantics.**

This paper shows a way to use the simulator itself to generate a formal model of the complete dynamic behavior. Compared with using yet another compiler, this permits to improve consistency with simulation semantics, and to combine simulation with other use-cases. We concretely focus on symbolic simulation of SystemC AMS ([1], [2], IEEE Std 1666.1-2016), for which we generate AADD and BDD for symbolic simulation.

***Index Terms*—Symbolic simulation, formal verification, SystemC AMS, Compiler**

I. Introduction

Electronic design automation is based on modeling languages that allow designers to describe models of hardware or software. Examples are VHDL, Verilog, and SystemC (based on C++), just to name a few. The main purpose of such modeling languages is simulation.

However, other use cases such as synthesis and formal verification require formal models like automata or binary decision diagrams (BDD) that represent the complete behavior of a model. The straightforward way to get such models is to write yet another compiler. Besides the huge effort for writing a compiler for languages such as C++ or VHDL, this is likely to introduce limitations to subsets, and maybe inconsistencies to the simulation results. The most significant disadvantage of two separate compilers or tools is that close integration of simulation with other use-cases can provide them with useful information, e.g. for concolic (mixed concrete/symbolic) verification or synthesis based on simulation-in-the-loop.

The main contribution of this paper is

- to show a way to use the existing proof-of-concept implementation of SystemC (AMS) for symbolic simulation, based on polymorphism and operator overloading.
- in particular we present the first approach to handle control flow statements (selection statement, iteration statement) in that way, just by code-instrumentation and not using an additional compiler.

The concrete use-case is symbolic simulation of SystemC AMS. However, the approach can also be used with other modeling languages and for other use-cases like high-level synthesis. In the following, we first review related work to generate formal models from modeling or programming languages with a focus on SystemC. In Section II, we give an overview and introduce AADD. Section III explains the symbolic execution of C++. In Section IV, we introduce the methods for block condition tracking and for generating AADD. In Section V, we discuss the integration of the approach in different models of computation. In Section VI, we give two SystemC AMS examples. In Section VII, we close with a summary and conclusion.

A. State of the art and related work

The straightforward way to translate a modeling language into a formal model is to write a compiler. For example, SystemC is translated by a compiler into intermediate representations like IVL [3], UPPAAL's timed automata [4], or simple sequential C [5]. A comprehensive overview including the particular challenges and limitations is given in [6].

The need of a separate compiler can be overcome by frameworks such as Java or the Clang C++ compiler that offer well-specified, open intermediate representations: the Java virtual machine, or the LLVM. This approach is taken by SystemC-clang [7] and PINAVM [8]. However, still a number of limitations remain. This includes e.g. the use of C++ pointers and loops (see [8], [6] for details).

In particular for testing and verification it is very useful to combine concrete and symbolic techniques (concolic testing [9]) in a single framework. Object-oriented features of modern languages permit e.g. [10], [11], [12] the symbolic execution within a regular compiler or simulator. In PyExZ3 [10], Python programs are translated to the input language of the automated theorem prover Z3. For this purpose, Python's integer objects, operators and functions are replaced by a symbolic type that produces the Z3 input language. In [11], [12], SystemC operators and functions are overloaded to permit symbolic simulation.

The approach taken in PyExZ3 [10] supports control flow by overloading functions that implement conditional statements. Compared with [10], we introduce a method that also handles the non-functional control-flow statements of procedural languages: block condition tracking. This allows us to generate AADD in a simple yet efficient way.

II. Symbolic simulation of SystemC and AADD

A. Overview of tool flow

This work describes the language-related part of a toolkit for verification of mixed-signal systems. The toolkit targets the concolic (mixed concrete/symbolic) verification based on SystemC. To increase verification coverage, simulation of critical parts is done in a symbolic way. To improve interoperability with existing SystemC models and verification infrastructure, and to allow designers to deal with scalability issues, we permit the concrete simulation of the other parts. The tool flow is shown in Fig. 1.

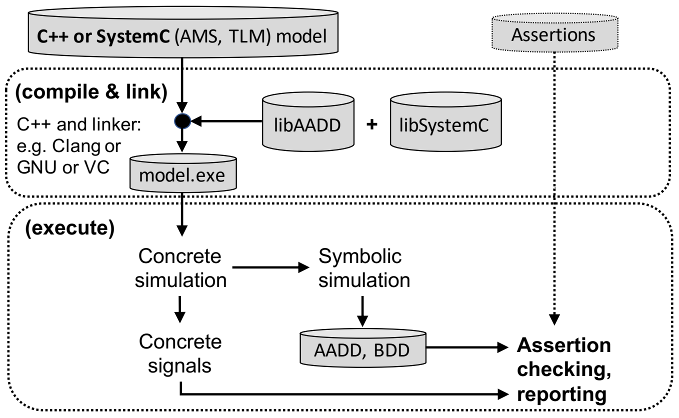

Fig. 1. Tool flow in which libAADD is applied.

The tool flow consists of the following steps:

1) Modeling: Models are specified in regular SystemC (AMS and/or TLM). By code-instrumentation a designer specifies which parts of a model are simulated symbolically, and which in a concrete way.
2) Compile & link: The model is compiled with a C++ compiler. After that, it must be linked with libSystemC (the SystemC simulator) and libAADD that provides symbolic extensions.
3) Execute: The resulting executable file is executed which starts a concrete simulation run. Where instrumented, the simulator will do a symbolic simulation run that
   a) Follows all feasible paths in a comprehensive way,
   b) Generates an AADD [11], [13] as formal model.
4) Check assertions: within (symbolic/concrete) simulation, assertions as described in [14] are checked, and reports are generated.

We use AADD (Affine Arithmetic Decision Diagrams) as a formal model because AADD are an extremely efficient representation of reachability for the given use case. However, the same approach can also be used to generate other internal representations such as control/data-flow graphs for the purpose of high-level synthesis.

The libAADD is available on GitHub under https://github.com/TUK-CPS/AADD. Some of the examples from this article are also provided there.

B. Internal representation: AADD

Affine Arithmetic Decision Diagrams (AADD) combine two representations: Affine forms that represent the reachability in continuous variables, and binary decision diagrams that represent discrete (path) conditions.

The intuition behind AADD is illustrated by the short C++ program below. We will use it as a didactic example throughout the paper.

```
#include "aadd.h" // symbolic types
doubleS a(0,100); // a is from [0,100]

int main() {
  if(a > 1)
    a = a + 2;
  else
    a = a - 2;
}                  // a is now an AADD.
```

To distinguish symbolic and concrete semantics, we denote types and keywords that take symbolic values by a capital S at the end. The variable a, in the example, is a symbol for an unknown value from the range $[0, 100]$. Depending on the path condition $(a > 1)$, the then- or the else-part is executed. As a is from $[0, 100]$, both the then- and the else-part are feasible. After termination, a is from either $[-2, -1]$ or $(3, 102]$, depending on the path condition $(a > 1)$.

An AADD represents this information in a (reduced, ordered) binary decision diagram [15] whose internal nodes are labeled with the path conditions, and whose leaf nodes are affine forms that model the ranges and dependencies. An affine form $\tilde{x}$ represents an unknown value x from a range by a linear model of its dependency:

$$\tilde{x} ::= x_0 + \sum_{i=1}^{n} x_i \varepsilon_i.$$

The coefficient x_0 is the *center value*, the coefficients $x_1 \dots x_n$ are the *partial deviations*. The *noise symbols* ε_i are symbolic variables that are restricted to $[-1, +1]$. Different affine forms (e.g. $\tilde{x} = x_0 + x_1\varepsilon_1, \tilde{y} = y_0 + y_1\varepsilon_1$) can share noise symbols. This represents the dependencies between them. Geometrically, the joint range of k affine forms is a zonotope in a k-dimensional space that is generated by the vectors of shared noise symbols, and centered around the vector of the center values.

Linear operations on affine forms and constants $c \in \mathbb{R}$ are defined as:

$$\tilde{z} \leftarrow c(\tilde{x} \pm \tilde{y}) ::= c(x_0 \pm y_0) + \sum_{i=1}^{n} c(x_i \pm y_i)\varepsilon_i.$$

Non-linear operations $\tilde{z} \leftarrow f(\tilde{x}, \tilde{y})$ are over-approximated by an affine form $f^a(\varepsilon_1, \dots, \varepsilon_n)$ that reasonably well models f, and a new term $z_k\varepsilon_k$ that guarantees inclusion of the "real" value of z:

$$\tilde{z} \leftarrow f(\tilde{x}, \tilde{y}) \subseteq f^a(\varepsilon_1, \dots, \varepsilon_n) + z_k\varepsilon_k.$$

For the non-linear operations, Chebychev approximations compute the coefficients while minimizing z_k (see [16] for details and other approximations).

The *fundamental invariant* of affine arithmetic [16] states that, at any instant between affine arithmetic operations, there is a single assignment of values from $[-1,1]$ to each of the noise variables in use that makes the value of every affine form $\tilde{x}$ equal to the true value of the corresponding ideal quantity x.

III. Symbolic execution of C++

A. Operator overloading and polymorphism

C++ and many other object-oriented languages support polymorphism and the overloading of operators and functions. For symbolic execution of C++ or symbolic simulation of SystemC, we provide a class `AADD`. This class can be used mostly like the C++ types `double`, `float`, and `int`. Furthermore, we provide the class BDD that can be used like the C++ type `bool`.

The class `BDD` implements an ROBDD [15]. In a symbolic execution run, an object of the class `BDD` holds all possible boolean values of a variable of the type `bool` at its leaves, and the path condition in the internal nodes. The class `BDD` provides, among others, the following methods:

- Constructors from values of types `bool` and `BDD`,
- Logic operators and functions as defined on `bool`, but with `BDD` as parameters and result.
- Overloaded assignment operators, see section IV.

The operators implement the known semantics of reduced ordered BDD.

The class `AADD` implements an AADD [11]. An object of the class `AADD` holds a sound abstraction of values of a variable of the types `int, float`, or `double` in a symbolic execution. It provides among others the following methods:

- Constructors from values of `int, float, double, AADD`, and from interval bounds.
- Arithmetic operators and functions as defined for `int, float,` and `double`.
- Relational operations with result of type `BDD`.
- Overloaded assignment operators, see section IV.

In addition, the class implements methods for printing by overloaded stream operators, and an interface to GLPK that is used to compute accurate interval bounds within symbolic execution. Details on operations on AADD are described in [11]. The overloaded assignment operator is important for the handling of conditional statements. We describe it in section IV. To hide the implementation, and to permit a more readable code we define:

```
typedef class AADD doubleS;
typedef class AADD floatS;
typedef class AADD intsS;
typedef class BDD  boolS;
```

The above definitions allow us to compute symbolically within a simulation run of an untouched SystemC simulator. We only have to include the header file that provides operators, functions, and methods with the appropriate signatures. Then, we can instantiate symbolic classes for symbolic execution, and use the expression- and compound-statements [17] of C++. For the didactic example we can execute:

```
1: #include "aadd.h" // for symb. execution
2: doubleS a(0,100); // a from range 0..100.
   ...
6:   a = a + 2;      // as for double
   ...
8:   a = a - 2;      // or other expression.
```

B. Concrete and symbolic execution

We allow the designer to select parts that are executed with concrete values, and parts that are executed with symbolic values. This is useful as symbolic execution by principle suffers from the path explosion problem. To deal with this issue, one must carefully select critical parts for which symbolic execution is done. Switching semantics to concrete execution is also useful for debugging and in case of limited support for symbolic semantics (e.g. pointer arithmetic, etc.).

Selection between concrete and symbolic execution is done globally by the type of variables, e.g. double for concrete and doubleS for symbolic execution. Furthermore, we provide the macros CONCRETE and SYMBOLIC. These macros give the designer a more fine-grain control:

- `CONCRETE(s, c)` assigns a symbolic variable s a concrete value $c \in s$.
- `SYMBOLIC(s, a)` assigns a symbolic variable s a safe abstraction $a \supseteq s$.

Refinements resp. abstractions can either be chosen by specifying a value resp. a range, or by giving it constrained random values. The above macros also do the following checks:

- The macro `CONCRETE` checks if a given concrete value is a valid refinement of an AADD.
- The macro `SYMBOLIC` checks if a given symbolic AADD is a safe abstraction of a concrete value.

This mechanism allows us to combine symbolic and concrete execution.

For illustration, we use the didactic example. As we have not yet introduced the symbolic execution of selection statements, we may use the SYMBOLIC and CONCRETE macros in lines 4b and 8b. With these macros, we can model the didactic example as follows:

```
1: #include "aadd.h" // symbolic extensions
2: doubleS a(0,100); // a is from [0,100];
3:
4: int main() {
4b:  CONCRETE( a, 50.0 ); // concrete test
5:     if(a > 1)        // is executed
6:       a = a + 2;     // a is 52
7:     else
8:       a = a - 2;     // this not.
8b:  SYMBOLIC( a, doubleS(2,102) );
9: } //continues with [2,102] if a in [2,102]
```

IV. Block condition tracking

A. Code instrumentation

The challenge with control flow statements is that for symbolic simulation we have to execute all feasible paths, while the concrete C++ control flow statements execute exactly one path. For example, in a selection statement (if-then-else), C++ will execute either the if or the then part, but not both.

In functional languages, where selection is a function, we can overload it by a function that

- Computes the condition (1^{st} parameter)
- Computes the then-part (2^{nd} parameter)
- Computes the else-part (3^{rd} parameter)
- Returns an AADD or a BDD that has a new level with the 1^{st} parameter as condition of a new internal node, and the 2^{nd} and 3^{rd} parameters as its child nodes.

However, procedural languages like C++ don't allow us to overload selection or iteration statements. Even worse, selection statements and iterations are no functions, which requires additional precautions to consider possible side effects. In the following, we overcome this limitation with a method we call "block condition tracking".

Selection statements with symbolic semantics: Selection statements in C++ have the following syntax [17]:

```
if (condition)
  statement  // then-part, arbitrary stmt.
[else        // optional:
  statement] // else-part
```

For block condition tracking, we instrument this statement. The instrumentation can be done manually or by a simple preprocessor provided with the AADD toolkit. An instrumented selection statement has the following syntax:

```
ifS (condition)
  statement
[elseS
  statement] endS
```

where

- `condition` is a C++ expression of type BDD.
- `ifS(condition)` is a macro that is executed at the beginning of a selection statement,
- `elseS` before the start of the else-part, and
- `endS` after the end of a selection statement.

We explain the function of the macros in the next section.

Iterations with symbolic semantics: Iteration statements are instrumented in a similar way as selection statements. In C++, a (while) iteration statement has the following syntax [17]:

```
while (condition)
  statement // arbitrary stmt.
```

By manual instrumentation, or by a preprocessor we translate it to an instrumented iteration statement that uses the macros `whileS` and `endS`. Again, these macros introduce code-fragments for symbolic execution of the iteration:

```
whileS (condition)
  statement; endS
```

B. Block condition tracking

In order to generate AADD and BDD by the instrumented selection and iteration statements, we track block conditions that are a subset to the path conditions.

A *path condition* is the conjunction of all conditions in a program run's conditional statements and iterations, from the program start to the current point of execution. In symbolic execution, we represent the path conditions by the decision tree in BDD or AADD. The tree is reduced immediately, if a path condition can be evaluated to either $true$ or $false$.

A *block condition* is the conjunction of all conditions in nested selection or iteration statements. After an assignment, a block condition becomes part of the path condition of a variable.

We compute block conditions by pushing all conditions from selection and iteration conditions on a stack (stack of BDDs). Statements in a block are feasible (reachable) if the conjunction of block conditions is not false.

The instrumentations `ifS`, `elseS`, and `endS` track the block conditions:

- `ifS` pushes its parameter `cond` on a stack.
- `elseS` negates the condition on top of this stack, and
- `endS` pops the condition from the stack.

The block condition is then always the disjunction of all conditions on the stack. Note, that the instrumentation does not implement a selection statement. It only tracks the block condition. In consequence, both the statements in the in-part and in the else-part will be executed sequentially.

As an example, consider the selection statement of the didactic example from section II-B, with instrumentation:

```
5: ifS(a > 1)
6:   a = a + 2;
7: elseS
8:   a = a - 2; endS
```

The macro in line 5, the condition $a > 1$ is put on the stack. The macro does nothing else; it does not start an if-statement. Line 6 is therefore executed, independent from the condition. The macro in line 7 negates the condition on top of the stack to $!(a > 1)$. Then, line 8 is executed, independent from the condition, and `endS` pops the condition from the stack. After line 8, a is an AADD that has the condition $a > 1$, a true-leaf $a + 2$, and a false-leaf $a - 2$.

C. Building AADD and BDD by overloaded assignments

To build AADD and BDD we use the ITE function. The function `ITE(cond, t, e)` adds new levels to the decision diagrams. It has the following parameters:

- *cond* of type BDD,
- t, an AADD or BDD; the result for $cond == true$,
- e, an AADD or BDD; the result for $cond == false$.

The ITE function is implemented as follows (pseudo-code):

```
FUNCTION: ITE(cond, t, e: BDD) returns BDD
    if (cond == true) return t;
    if (cond == false) return e;
    return (cond & t) | (!cond & e);
```

Note, that the parameters, including the condition, are AADD or BDD. The conjunction and disjunction functions for BDD [15] merge the conditions for all tree parameters. The resulting BDD has all levels of the parameters and a new level for the condition `cond`. The algorithm is similar for AADD. However, we use (arithmetic) multiplication with 0 for false and 1 for true instead of conjunction, and addition instead of disjunction.

We are now in a situation where we globally know all block and path conditions. Block conditions are on a stack of all possible "branches": so far, conditions in selection and iteration statements; later we handle symbolic process activations in the same way. Path conditions are represented in the decision diagrams of BDD and AADD.

We use this information in overloaded assignment operators. For an assignment `lval := rval`, concrete execution semantics will create a clone of rval and assign it to lval. In symbolic execution semantics, we have to merge the block conditions to the path conditions. This is done by the following method (in pseudocode):

```
0: global: stack of conditions of type BDD.
1: METHOD assign (rval, lval: AADD, BDD):
2:    bc := AND(all conditions on stack);
3:    rval := ITE(bc, lval, rval);
4:    return rval;
```

For illustration, we come back to the didactic example from section II-B. Lines 5-8, after expansion by the preprocessor become:

```
5:{ blkConds().thenBlock(a > 1);
6:  a = a + 2; // calls a=ITE(a>1,a+2, a);
7:  blkConds().elseBlock(__LINE__,__FILE__);
8:  a = a - 2; // calls a=ITE(!a>1,a-2, a);
8b: blkConds().endBlock(__LINE__,__FILE__);}
```

Instead of an if-keyword in line 5, the condition $(a > 1)$ is put on the stack for the block condition. As there is no if-statement anymore, no matter which result the condition has, all following statements line 6, 7, 8 are executed and an AADD is generated.

Iteration statements are handled in a mostly similar way. However, a real iteration is executed, and the block condition is put on the stack of block conditions in each new iteration. Note, that for termination of the loop we explicitly compare the condition of type BDD with false. When leaving the iteration, the conditions are popped from the stack. The code generated by the macro `whileS` is then:

```
while ( (cond)!=false)
{
  blkConds().whileBlock(cond);
  statement;
} blkConds().endBlock(); // pops all cond.
```

Limitations: We have implemented and tested the selection statement and the iteration statement as described above. The above approach is not limited to these control-flow statements. Implementations of other syntactic forms of iteration and selection like do .. while, case .. select, for (...) are straightforward.

A bit ugly is the following issue: The transformations $if \rightarrow ifS$, $else \rightarrow elseS$ and $while \rightarrow whileS$ can be done easily e.g. by the C-preprocessor. However, inserting of $endS$ at the end of selection statements requires a parser that recognizes selection statements. We implemented such a preprocessor based on ANTLR by adding approx. 10 lines to a listener class of its CPP14 grammar.

A more fundamental issue is that AADD use safe abstractions that are meaningless for e.g. pointer arithmetic and representation of bit-vectors.

V. Symbolic Simulation of SystemC (AMS)

A. Symbolic signals and process activation

For symbolic simulation of SystemC (AMS), we have to consider the impact of symbolic representations on the signal representations, and on the activation of processes. Therefore, we first formalize some aspects of signals and the underlying models of computation (MoC).

Let a signal be a sequence of samples $\langle v(t_1), v(t_2), \ldots, v(t_n)\rangle$, where the $t_i, i \in \mathbb{N}$ are elements of the simulated time, and $v(t_i)$ is the value of the sample at time t_i. Then, we call a signal concrete (or deterministic), if each $v(t_i)$ is a concrete value. We call a signal symbolic (or uncertain [13]) if at least one sample is symbolic, e.g. a BDD or an AADD, and represents more than one value.

A symbolic signal represents all possible signal trajectories. As consequence of the fundamental invariant of affine arithmetic, it holds that that there is a single assignment of values from $[-1, 1]$ to each of the noise variables that makes the trajectory of the symbolic signal equal to the trajectory of a 'real', concrete signal. This is a fundamental difference to the flow-pipe representation of signals that only represents an enclosing hull.

Let a process be specified by $P = (I, O, proc, a)$ with

- input signals I and output signals O,
- $proc$, a processing method,
- a, an activation condition.

$proc$ uses samples from the input signals I to compute samples of the output signals O, maybe using internal states. The execution of $proc$ is done at points in simulated time, when a is $true$.

An activation condition is concrete (deterministic), if its values are either $true$ or $false$. An activation condition is symbolic (uncertain), if at least one of its values is a symbol, i.e. represented by a BDD or AADD.

B. Symbolic simulation with concrete activation condition

In many relevant cases, we have concrete activation conditions. This depends on the models of computation, and is the case for:

- *Timed data flow (TDF).* The TDF model of computation of SystemC AMS is based on the static data flow MoC. TDF/Static data flow defines a static schedule before start of simulation. It depends only on the rates of ports and modules, but not on other values. If rates are concrete, TDF (and SDF) have static activation conditions.

- *Continuous-time models (CT).* The selection of analog solution points depends on the simulator. The objective of CT models is to minimize the quantization error between the unreachable ideal of infinitely many analog solution points, and a discrete number thereof. Hence, from a functional point of view, CT has a static activation condition.
- *Discrete-event (DE).* The activation condition in DE MoC is the sensitivity list of SC_METHOD or wait statements of SC_THREAD. The activation condition is concrete, if the respective signals or events are based on concrete values (e.g. a concrete clock).

For concrete activation conditions, the process activation mechanism is not influenced by symbolic simulation results. For SystemC AMS, symbolic simulation requires hence no further modifications. For the SystemC DE MoC, we have concrete activation conditions if the sensitivity list consists of concrete signals. As an example, consider a counter that counts if a symbolic input `th` is above a threshold:

```
sc_in<bool>    clk; // concrete signal
sc_in<doubleS> th;  // symbolic signal
sc_out<intS>   cnt; // symbolic signal

void count() {
  if (th>2) cnt += 1;
}

SC_METHOD(count) sensitive << clk;
```

The above example has a concrete activation by the concrete signal `clk`. Unfortunately, SystemC's event detection mechanism is part of all DE signals, even if they are not part of the sensitivity list. We handle this by overloading a direct template instantiation. Support for the general case of symbolic process activations is subject of future work.

C. Symbolic simulation with symbolic activation conditions

In the DE MoC, we can have the general case of a symbolic activation condition. This case is subject of ongoing research, and results are not yet stable enough for publication. In principle symbolic activation conditions can be treated like those for selection statements: We activate the process, but all assignments are made only under the activation condition a. The process activation condition is a block condition of type BDD. An implementation would then be:

- Push a on the stack of block conditions.
- Execute the process; use ITE function for all assignments.

However, we still lack integration of this approach in SystemC.

VI. Examples

A. Simple example: water level monitor

As a simple example, we model a water level monitor in SystemC AMS. We use the TDF model of computation. We model the water level by a variable `wlevel` of type `doubleS`. The water tank has two sensors that signal whether the water level is lower than 5 (port `l5` of type `boolS`) or greater than 10 (port `g10` of type `boolS`). The processing method called in each time step is for the water tank model:

```
void processing() {
  if(pump)    // dynamics of water level
    wlevel+=(1.+uncertainty1)*timestep;
  else
    wlevel+=(-2.+uncertainty2)*timestep;

  if(wlevel < 5) l5 = true; // sensors
  else l5 = false;

  if(wlevel > 10) g10 = true;
  else g10 = false;
}
```

The controller's processing method switches a pump on, depending on the sensor's values (in-ports of type `boolS`):

```
void processing() {
   if( l5 )  pump = true;
   if( g10 ) pump = false;
}
```

The SystemC model with its instrumentations for symbolic simulation is compiled by the regular C++ compiler (on OS X, LLVM), and linked against the SystemC, SystemC AMS and AADD libraries. After running the executable, we get the well-known outputs, and in addition some reporting from libAADD:

```
     AADD lib -- Symbolic execution is enabled.
        AADD library (c) TU Kaiserslautern,
              C. Zivkovic, C. Grimm.

    SystemC 2.3.0-ASI --- Apr 27 2017 16:27:00
   Copyright (c) 1996-2012 by all Contributors,
               ALL RIGHTS RESERVED

      SystemC AMS extensions 2.0 Version: 1.0
            Copyright (c) 2010-2013  by
              Fraunhofer-Gesellschaft
        Institut Integrated Circuits / EAS
   Licensed under the Apache License, Version 2.0

Info: SystemC-AMS:
        3 SystemC-AMS modules instantiated
        1 SystemC-AMS views created
        3 SystemC-AMS synchronization
       objects/solvers instantiated

Info: SystemC-AMS:
        1 dataflow clusters instantiated
        cluster 0:
    3 dataflow modules/solver, contains e.g. module: wtank
    3 elements in schedule list, 100 ms cluster period,
      ratio to lowest:  1 e.g.module: wtank
      ratio to highest: 1 sample time e.g. module: wtank
 0 connections to SystemC de,
 0 connections from SystemC de

Symbolic simulation took: 3.17225 seconds.
```

The result of symbolic simulation is a sequence of AADD/BBD that represent all possible trajectories of the signals. To get a result that can be plotted and reasonably well understood we plotted the minimum and maximum values in a file. This is shown by Fig. 2.

B. Analog/mixed-signal example: delta-sigma modulator

As a more complex example we model a 3^{rd}-order delta-sigma modulator. Like in the first example we use the timed data flow model of computation in SystemC AMS. The block diagram of the modulator is shown in Fig. 3.

The values of the coefficients are taken from [18]:

$$b_1 = 0.0444; b_2 = 0.2881; b_3 = 0.7997$$

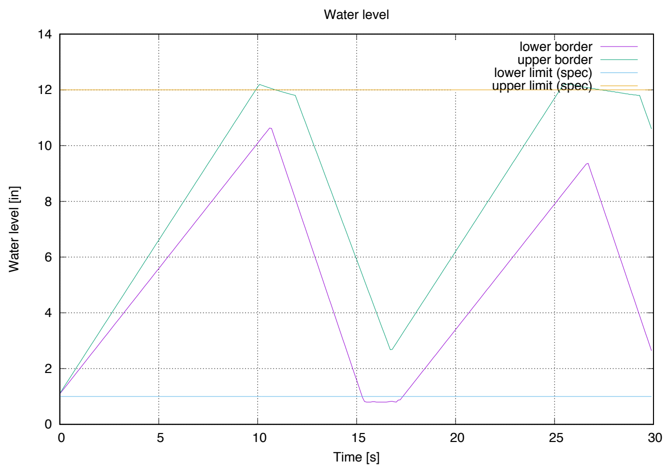

Fig. 2. Minimum and maximum of water level trajectories over time.

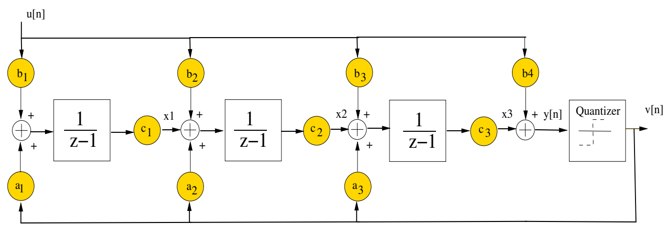

Fig. 3. Block diagram of 3rd order Delta-Sigma modulator [18].

$$a_1 = -0.0444; a_2 = -0.2881; a_3 = -0.7997$$

$$c_1 = c_2 = c_3 = 1; b_4 = 1.$$

The integrator outputs x_1, x_2, x_3 are computed by discrete-time integration of the integrator input signals.

The SystemC AMS model consists of a timed data flow cluster that implements the 3^{rd} order integrator part and the quantizer. For example, the processing method of the SystemC AMS model of a single discrete-time integrator stage is:

```
void processing() {
  x_1=x_1+0.0444*(u-v);
}
```

Implementation of the other integrators is straight-forward. The output signal of the third integrator $x_3[n]$ is added to the modulator input signal $u[n]$ and forwarded to the input of the one-bit quantizer $y[n]$. The quantizer sets all positive values of y to 1 and negative values and 0 to -1. As a SystemC AMS module:

```
SCA_TDF_MODULE(quantizer) {
  sca_tdf::sca_in<doubleS> y;
  sca_tdf::sca_out<doubleS> v;
  void processing() {
    if(y>0) v=1;
    else v=-1;
  }
  quantizer(sc_module_name nm){}
}
```

By symbolic simulation and assertion checking we check the model for all inputs $u[n]$ in the range $[-0.5, 0.5]$, and all initial states $x_1[0], x_2[0], x_3[0]$ in $[-0.1, 0.1]$. To guarantee accurate function of the modulator, integrator saturation, and quantizer overload are checked. This means that the respective outputs of the integrators, and inputs of the quantizer must be in a range $[-2, 2]$. We verify this by symbolic simulations, where we assign the inputs a range (`doubleS(-0.5, 0.5)` resp. `doubleS(-0.1, 0.1)`.

After the symbolic simulation values of all signals are a sequence of AADD. Fig. 4 plots minimum and maximum values of the integrator output x_3 for 20 time steps in one symbolic simulation run. The symbolic simulation for the given example took around $10.3s$. This includes the time for writing the results into a file which requires a solver call to compute upper/lower bounds for each point to be plotted.

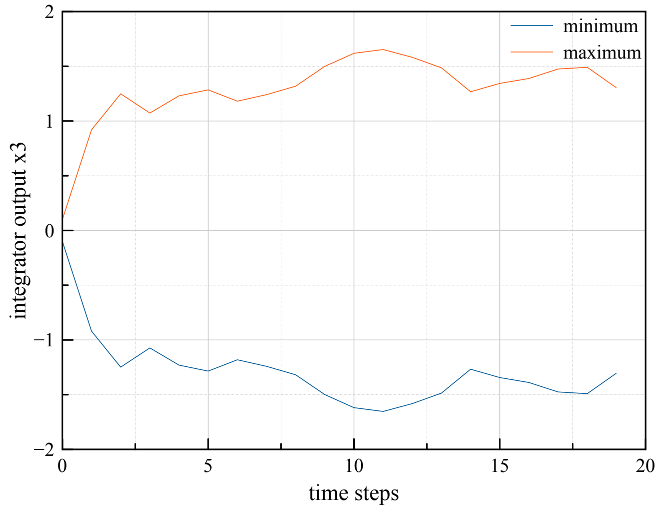

Fig. 4. Worst case signal values for x_3 at 20 time steps.

For comparison, a single, concrete simulation run takes 2 ms. However, using random inputs, and given strongly nonlinear dynamic behavior of a quantizer, it is extremely unlikely to find the unknown corner cases.

VII. Summary and Conclusion

A. *Symbolic simulation, without another compiler?*

The objective of the work was to avoid the implementation of a C++/SystemC compiler, and to permit a close interaction of concrete simulation with symbolic simulation.

By polymorphism and operator overloading, we can completely go without another C++ compiler for expression statements including function and method calls, and compound statements. For these kinds of statements the classes `doubleS, floatS, intS, boolS` provide sufficient functionality. The instrumentation of these types is necessary to distinguish concrete and symbolic semantics, e.g. by giving variables ranges of possible values.

Iteration and selection statements require macros that modify the control flow and track conditions. This has no semantic need. However, it is easy to introduce these macros via the keywords `if, else, while` by the preprocessor. Unfortunately, adding the macro `endS` the end of an iteration or selection statement requires a simple compiler. We implemented it with 10 lines of code and ANTLR. Table I gives an overview.

TABLE I
C++: WHAT IS NEEDED WITHOUT ANOTHER COMPILER.

C++ construct	Requires
expression (statement), compound statement, declarations	operator overloading, polymorphism
selection statement, iteration statement	instrumentation or own preprocessor adding ENDS, block condition tracking
goto statement try statement (exceptions)	likely as above, not done

Regarding simulation semantics of SystemC (AMS), concrete process activations are supported without any limitations. This is sufficient for the TDF MoC, and DE processes activated by a deterministic clock. Symbolic process activations are not yet supported; they are subject of future research. Table II gives an overview of required changes.

TABLE II
SYSTEMC: WHAT IS NEEDED WITHOUT ANOTHER COMPILER.

SystemC construct	Requires
AMS extensions, TDF MoC	./.
AMS extensions, CT or LSF MoC	modified solver, e.g. [19]
DE w/ concrete activation	== operator overloaded or sc_signal<AADD/BDD>
DE in general	instrumentation of processes block condition tracking (future work)
TLM	(future work)

B. Only for symbolic simulation?

Although we worked with SystemC (AMS) and AADD in this paper, the approach is not limited to this particular use case at all. Just for example, control-data-flow graphs are a good starting point for high-level synthesis, or for timing analysis. The central idea of block condition tracking and overloaded assignments as a rather abstract method can easily be modified to generate control-data-flow graphs. For this purpose, the code that creates the internal representations has to be rewritten.

REFERENCES

[1] M. Barnasconi, K. Einwich, C. Grimm, and A. Vachoux, Eds., *Standard SystemC® AMS extensions 2.0 Language Reference Manual.* OSCI, 2013. [Online]. Available: http://accellera.org

[2] A. Vachoux, C. Grimm, and K. Einwich, "Extending SystemC to support mixed discrete-continuous system modeling and simulation," in *2005 IEEE International Symposium on Circuits and Systems*, May 2005, pp. 5166–5169 Vol. 5. [Online]. Available: http://dx.doi.org/10.1109/ISCAS.2005.1465798

[3] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/6560709/

[4] P. Herber, J. Fellmuth, and S. Glesner, "Model checking SystemC designs using timed automata," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '08. New York, NY, USA: ACM, 2008, pp. 131–136. [Online]. Available: http://doi.acm.org/10.1145/1450135.1450166

[5] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign*, ser. MEMOCODE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 113–122. [Online]. Available: https://doi.org/10.1109/MEMCOD.2010.5558643

[6] K. Marquet, M. Moy, and B. Karkar, "A theoretical and experimental review of systemc front-ends," in *Forum on Specification and Design Languages 2009*, 2009. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00495886

[7] A. Kaushik and H. D. Patel, "Systemc-clang: An open-source framework for analyzing mixed-abstraction SystemC models," in *Proceedings of the 2013 Forum on specification and Design Languages, FDL 2013, Paris, France, September 24-26, 2013*, 2013, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/6646649/

[8] K. Marquet and M. Moy, "Pinavm: A systemc front-end based on an executable intermediate representation," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '10. New York, NY, USA: ACM, 2010, pp. 79–88. [Online]. Available: http://doi.acm.org/10.1145/1879021.1879032

[9] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[10] T. Ball and J. Daniel, "Deconstructing dynamic symbolic execution," in *Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering*, 2014. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/dse.pdf

[11] C. Radojicic, C. Grimm, A. Jantsch, and M. Rathmair, "Towards Verification of Uncertain Cyber-Physical Systems," *Electronic Proceedings in Theoretical Computer Science*, vol. 247, pp. 1–17, Apr. 2017. [Online]. Available: https://doi.org/10.4204%2Feptcs.247.1

[12] C. Grimm, W. Heupke, and K. Waldschmidt, "Analysis of mixed-signal systems with affine arithmetic," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 1, pp. 118–123, 2005. [Online]. Available: https://ieeexplore.ieee.org/document/1372667/

[13] C. Grimm and M. Rathmair, "Dealing with uncertainties in analog/mixed-signal systems: Invited," in *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, 2017, pp. 35:1–35:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3072949

[14] C. Radojicic, C. Grimm, F. Schupfer, and M. Rathmair, "Verification of mixed-signal systems with affine arithmetic assertions," *VLSI Design*, vol. 2013, 2013. [Online]. Available: http://dx.doi.org/10.1155/2013/239064

[15] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986. [Online]. Available: http://dx.doi.org/10.1109/TC.1986.1676819

[16] J. Stolfi and L. H. de Figueiredo, "An Introduction to Affine Arithmetic," *TEMA Trend. Mat. Apl. Comput.*, vol. 4, pp. 297–312, 2003. [Online]. Available: https://tema.sbmac.org.br/tema/article/view/352

[17] A. Marchetti, "Hyperlinked c++ bnf grammar." [Online]. Available: http://www.nongnu.org/hcb

[18] G. A. Sammane, M. H. Zaki, S. Tahar, and G. Bois, "Constraint-Based Verification of Delta-Sigma Modulators using Interval Analysis," in *50th Midwest Symposium on Circuits and Systems*, 2007, pp. 726–729.

[19] D. Grabowski, C. Grimm, and E. Barke, "Semi-Symbolic Modeling and Simulation of Circuits and Systems," in *IEEE International Symposium on Circuits and Systems (ISCAS)*. Washington, DC, USA: IEEE, 2006, pp. 983–986. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1692752