Towards Verification of Uncertain Cyber-Physical Systems

Carna Radojicic, Christoph Grimm
TU Kaiserslautern, Germany
radojicic|grimm@cs.uni-kl.de

Axel Jantsch, Michael Rathmair
TU Wien, Austria
jantsch|rathmaier@ict.tuwien.ac.at

Cyber-Physical Systems (CPS) pose new challenges to verification and validation that go beyond the proof of functional correctness based on high-level models. Particular challenges are, in particular for formal methods, its heterogeneity and scalability. For numerical simulation, uncertain behavior can hardly be covered in a comprehensive way which motivates the use of symbolic methods.

The paper describes an approach for symbolic simulation-based verification of CPS with uncertainties. We define a symbolic model and representation of uncertain computations: Affine Arithmetic Decision Diagrams. Then we integrate this approach in the SystemC AMS simulator that supports simulation in different models of computation. We demonstrate the approach by analyzing a water-level monitor with uncertainties, self-diagnosis, and error-reactions.

1 Introduction

Cyber-Physical Systems (CPS) consist of a large number of networked, co-operating and open sub-systems [20]. This is a blessing and a curse: On one hand, the high number of components and their open nature make it likely that some components fail, are changed, or get inaccurate sensor data. On the other hand, co-operation allows us to implement resilience that maintains dependable operation. As CPS often implement mission-critical ecosystems and services, e.g. autonomous driving, or aviation, it is mandatory to show that errors and deviations must not lead to a failure or to an unsafe state (fail safe). Even more: in CPS, the correct function shall be maintained under presence of uncertainties (fail operational). However, in CPS, the terms "error" and "correctness" require a new understanding.

The first reason for this is that faults, unforeseen changes, and deviations have to be considered as likely part of normal behavior. Therefore, we summarize all kinds of such events that cause deviations from the ideal behavior under the more general term *uncertainty*. Due to its higher probability, the propagation and interaction of multiple uncertainties of different kinds must be verified thoroughly. However, the complexity of dynamic behavior, and interactions of multiple uncertainties across different domains are often too complex to be handled by simulation or human imagination. It requires exhaustive methods such as symbolic simulation or model checking.

The second reason is that CPS are deployed in a more open environment than embedded systems. In consequence, correctness in the sense of fulfilling specified properties that stem from design-time analysis of well-defined use-cases might be too simple. CPS also have cope with changing requirements or unforeseen scenarios. This demands for the application of adaptive methods.

This paper pursues the following objectives: First, we describe how formal verification and symbolic simulation can contribute in an heterogeneous, industrial verification process for CPS. Second, we study the propagation and interaction of uncertainties in some models of computation (MoC) as a starting point for future work that shall provide a basis for the symbolic simulation in arbitrary combinations of different MoC. Implementation and examples can be downloaded from `http://cps.cs.uni-kl.de/AADD`.

EPTCS ??, 20??, pp. 1–17, doi:10.4204/EPTCS.??.??

1.1 Related work and contribution

For the understanding of interacting discrete and continuous subsystems of CPS with uncertain behavior, research on verification of hybrid systems provides valuable insights. In the continuous domain, the – due to propagation of uncertainties modeled by non-deterministic behavior – reachable state space is segmented by planes into convex geometric figures. Zonotopes [11, 2] and support functions [18] improve scalability in non-linear systems. Also, affine arithmetic [10] is used in particular in the verification of analog circuits and systems [12]. Its geometrical interpretation is similar to a zonotope, but it offers further useful properties to which we come back later in the paper. In order to yield high scalability also for discrete systems, functionally reduced AND-Inverter graphs are combined with models of linear continuous dynamics [8].

For the discrete subsystem, this work is very similar to methods for symbolic execution of software, where control-flow introduces path conditions as discrete states. To cope with the path explosion problem SAT and/or SMT solvers (e.g. [4, 16]) are used to determine the reachable paths. Affine arithmetic has been used in this context for the static analysis of rounding errors in DSP algorithms [9] or numerical programs and even hybrid systems in [22] based on splitting affine arithmetic forms (AAF) and joining them in an enclosing hull, in particular targeting stability and robustness.

For CPS, its networked, distributed, and heterogeneous nature poses additional challenges [20]. This includes in particular the use of domain-specific modeling formalisms in different parts of a CPS. In this context, the term *models of computation* also became popular in the domain of modeling/simulation. Lee and Sangiovanni-Vincentelli [19] introduced a meta-model in which different models of computation and different means for communication and synchronization can be uniformly represented and compared. A mostly similar, but more refined approach is proposed by Jantsch in [14, 15].

This paper contributes two main results: First, we introduce an efficient yet simple method to compute with uncertain values that combines affine arithmetic with binary decision diagrams: Affine Arithmetic Decision Diagrams (AADD; Sec. 3). Second, we deliberately distinguish between symbolic computation with uncertain values and concrete models of computation (Sec. 4). This allows us to model CPS in a variety of models of computation without being limited to a specific one such as hybrid automata. We demonstrate the approach by an example (Sec. 5).

2 Verification of uncertain CPS

For software and digital systems, the underlying digital synchronous hardware platforms allow us to abstract from all physical variations in e.g. temperature, supply voltage, and to focus on the 'ideal', intended behavior of algorithms. Hence, inside this domain we can easily specify properties, prove its correctness, and trust the results. This does not hold once physical domains or human interactions are involved, e.g. an autonomously driving vehicle in a city. Therefore, for verification of CPS we have to very carefully re-think basic assumptions, methodologies, and strip down verification to different, well-defined verification problems.

A particular challenge for verification of CPS in this context is the presence of uncertainty. Uncertainty can be defined as *"any deviation from the unachievable ideal of completely deterministic knowledge of the relevant system"* [26]. Formally, uncertainty can be modeled by non-deterministic or probabilistic choice of a value from a set or range. As uncertainties, we treat in particular unknown deviations due to modeling or implementation issues, variations in physical implementations, abstractions in models, faults, or errors. Furthermore, uncertainties can also come from the environment of CPS (external

uncertainties). External uncertainties can be operating parameters such as the temperature or humidity, but also inputs that are uncertain, e.g. jitter in clocks, but also unforeseen use case scenarios.

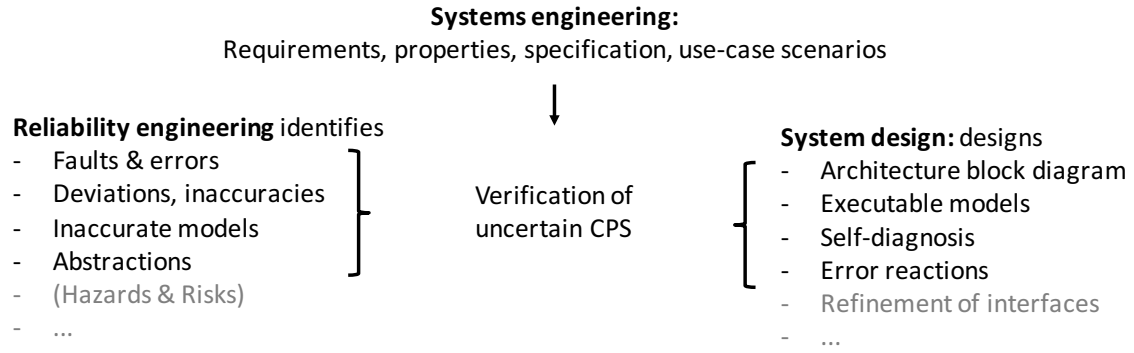

Figure 1: Overview of the development- and verification methodology.

For the verification of uncertainties in CPS we assume a development process as shown in Fig. 1: Systems engineering formulates, more or less formal, required properties, use case scenarios, and maybe a functional model. Verification objective is, at this stage, to show that the functional model fulfills the required properties and is free of inherent problems.

The following system development consists at least of system design and reliability engineering. System design refines the functional model to an architecture level block diagram, and reliability engineering. Reliability engineering identifies risks, hazards and initiating faults and deviations. Then it is investigated whether initiating faults or deviations can become an error or a hazard. If this is the case, methods for the self-diagnosis and error reactions are implemented. Objective of the verification of uncertain CPS is to support these tasks.

We make the following basic assumptions for verification of uncertain CPS:

- The system design is specified in a domain-specific language in the form of a block diagram.
- The components have parameters that model arbitrary kind of uncertainties such as the potential presence of continuous deviations or discrete faults.
- The inputs can have parameters that model external uncertainties.

The objective of verification of uncertain CPS is to show that for arbitrary chosen uncertain values the properties of the design are within its specified ranges. Our approach can be considered as a bounded-time symbolic simulation with assertion checking [24].

Obviously, verification of uncertain CPS does not show its general correctness. It must be complemented with other approaches that focus specifically, e.g. on concurrency issues. We deliberately limit our approach to one task to yield better scalability. In the following, we first formalize the representation of uncertainties and its propagation in computations, and then in different models of computations.

3 Uncertainties in computations

In the following, we first formalize the representation of uncertainties in symbolic computations. Informally, we represent uncertain values, short uncertainties, as quantities (e.g. $\tilde{x}, \hat{x}, \check{x}$) that depend from uncertain variables ε_i, χ_i. We consider these variables as the atomic, basic sources of uncertainty.

Definition 3.1 (Basic uncertainties) *A discrete basic uncertainty is a variable $\chi_i, i \in \{1,\ldots,m\} \subset \mathbb{N}$ which takes values in the set $\mathbb{B} = \{true, false\}$. Let $\mathbb{X} = \{\chi_1,\ldots,\chi_m\}$ be the set of all basic discrete uncertainties. A continuous basic uncertainty is a variable $\varepsilon_j, j \in \{1,\ldots n\} \subset \mathbb{N}$ which takes values in $[-1,1] \subseteq \mathbb{R}$. Let $\mathbb{E} = \{\varepsilon_1,\ldots,\varepsilon_n\}$ be the set of all basic continuous uncertainties.*

3.1 Propagated continuous and discrete uncertainties

Definition 3.2 (Continuous propagated uncertainty) *A continuous propagated uncertainty $\tilde{x}$ is a quantity $\tilde{x} : \mathbb{E} \to \mathbb{R}$ that describes the dependency of the real-valued result of a computation from the basic continuous uncertainties $\mathbb{E}$.*

We represent continuous propagated uncertainties by affine arithmetic forms (AAF, [10]):

$$\tilde{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i \qquad \text{with } \varepsilon_i \in [-1,1], x_0 \in \mathbb{R}, x_i \in \mathbb{R} \tag{1}$$

where each x_i models the sensitivity of $\tilde{x}$ to the basic uncertainty ε_i (1^{st} order effects). Let range$(\tilde{x})$ of an AAF $\tilde{x}$ be an interval $[lb, ub] \subseteq \mathbb{R}$ with:

$$\text{range}(\tilde{x}) = [x_0 - \sum_{i=1}^{n} x_i, \quad x_0 + \sum_{i=1}^{n} x_i] \tag{2}$$

The linear operations $(+,-,\cdot)$ where $\cdot$ corresponds to multiplication with a constant $c \in \mathbb{R}$ on AAF are:

$$c(\tilde{x} \pm \tilde{y}) = c(x_0 \pm y_0) + \sum_{i=1}^{n} c(x_i \pm y_i)\varepsilon_i \tag{3}$$

For non-linear operations, approximation schemes compute safe inclusions subject to optimization criteria. Non-linear operations $\tilde{z} = f(\tilde{x},\tilde{y})$ are handled by a linear inclusion. For this purpose $f(\tilde{x},\tilde{y})$ is over-approximated by an affine form $f^a(\tilde{x},\tilde{y}) = z_0 + \sum_{i=1}^{n} z_i\varepsilon_i$ and an additional term $z_{n+1}\varepsilon_{n+1}$:

$$f(\tilde{x},\tilde{y}) \subseteq f^a(\tilde{x},\tilde{y}) + z_{n+1}\varepsilon_{n+1} \tag{4}$$

The computation of $f^a(\tilde{x},\tilde{y})$ and z_{n+1} is a multi-criteria optimization problem that is solved by approximation schemes that make different trade-offs, e.g [10]:

- *Minimal range approximation* – The minimal range approximation eliminates over-approximation at interval bounds (Min: range$(f(\tilde{x},\tilde{y}))$) at the cost of an increasing z_{n+1}.
- *Chebychev approximation* – The Chebychev approximation minimizes z_{n+1} at the cost of increasing the range of $f^a(\tilde{x},\tilde{y})$.

To avoid an increasing number of variables $\varepsilon \in \mathbb{E}$ with the number of non-linear operations, we use only a single variable ε_{n+1} to which we add all approximation errors by assuming them as uncorrelated which guarantees safe inclusion.

Example As example consider two AAF $\tilde{a} = 1 + 2\varepsilon_1, \tilde{b} = 2 - 2\varepsilon_1 + \varepsilon_2$. Then $\tilde{a} + \tilde{b} = 3 + \varepsilon_2$, and $a \cdot b = 2 + 2\varepsilon_1 + \varepsilon_2 + [-6,2]$ where $[-6,2]$ ensures safe inclusion; to come to an AAF with a $\varepsilon_3\chi_3$ including the non-linear terms, we can either increase $[-6,2]$ to $[-6,6]$, or increase ε_1 and ε_2 while reducing x_3.

Note, that the geometrical interpretation of an AAF is equivalent to two zonotopes: one including $x_{n+1}\varepsilon_{n+1}$ that gives the enclosing hull (over-approximation), and another one without $x_{n+1}\varepsilon_{n+1}$ that gives the inner hull (under-approximation).

Definition 3.3 (Discrete propagated uncertainty) *A discrete propagated uncertainty $\check{x}$ is a Boolean function $\check{x} : \mathbb{X} \to \mathbb{B}$ that describes the dependency of the ideal result of a computation from the basic discrete uncertainties $\mathbb{X}$.*

For example, discrete propagated uncertainties can describe the possible results of a Boolean function in the potential presence of discrete faults (i.e. the χ_i). As discrete propagated uncertainties are boolean functions, we represent them by (if needed reduced) ordered binary decision diagrams (ROBDD [7]). We assume the reader is familiar with ROBDD. In brief, a BDD is a DAG (V,E) whose terminal vertices are labeled $true$ or $false$, and whose internal vertices $v \in V$ are labeled with the variables $x_i, i \in \{1,\ldots,n\}$ of a boolean function $y = f(x_1,\ldots,x_n)$ and are connected with a vertex $v_t = true(v)$ and $v_f = false(v)$, depending of the value of a variable x_i. If the order of its variables x_i is the same on all paths, it is ordered (OBDD). If all redundancies in form of isomorphisms and double-edges are removed, it is reduced (ROBDD).

3.2 Hybrid uncertainties

Computations consist of both discrete and continuous parts. This is the case for algorithms where computations on real numbers are controlled by a discrete control flow, i.e. conditional statements and iterations. In the following we extend the definitions of uncertainties towards 'hybrid uncertainties'.

Definition 3.4 (Hybrid uncertainty) *A hybrid uncertainty $\hat{x}$ is a quantity $\hat{x} : \mathbb{X} \times \mathbb{E} \to \mathbb{R} \cup \mathbb{B}$ that describes the dependency of a real-valued or boolean result of a computation from the discrete and continuous basic uncertainties $\mathbb{X}, \mathbb{E}$.*

We structure computations as shown by Fig. 2 into a discrete and a continuous part. The parts interact via comparisons of continuous variables that can be uncertain ('uncertain conditions', later defined as $\mathbb{X}_c$), and via branches in the continuous part that can be uncertain. We consider uncertain conditions and uncertain branches as basic uncertainties for the discrete and continuous parts.

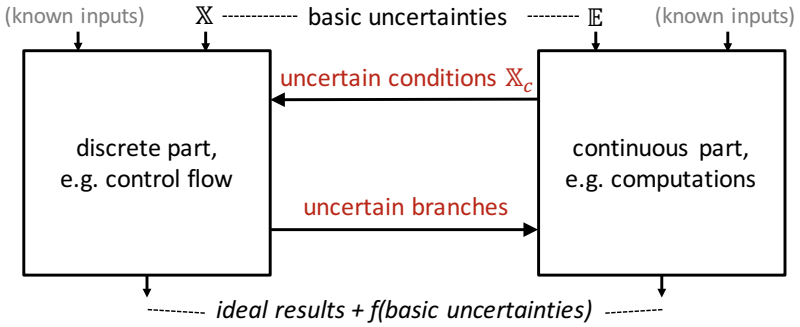

Figure 2: Interactions between discrete and continuous part in a computation.

However, the uncertainty in the interaction between discrete and continuous parts must be considered when computing the uncertainty of the overall computation ('hybrid uncertainties'). [22] handles uncertain branches by merging all paths into a single, convex region. However, this introduces overapproximation that we strive to avoid. To come to more accurate bounds, we use the information on the interaction between the discrete and the continuous part.

Example For example, consider $b = 3 + \varepsilon_1$ and the computation `if(b>3) b+=10; else b-=10;`. Then, we cannot evaluate $b > 3$ to either *true* or $false$. A safe inclusion of the result would be $3 + \varepsilon_1 + 10\varepsilon_2$; however, with over-approximation. A more accurate representation is:

$$(13 + \varepsilon_1 \mid (\varepsilon_1 > 0)) \vee (-7 + \varepsilon_1 \mid (\varepsilon_1 \leq 0))$$

The above representation is a Shannon expansion. This motivates the following representation based on ordered binary decision diagrams.

3.3 Affine Arithmetic Decision Diagrams

We represent hybrid uncertainties by affine arithmetic decision diagrams (AADD). The idea of AADD is shown in Fig. 3. We use OBDD to represent the selection of an AAF by the discrete uncertainties $\mathbb{X} \cup \mathbb{X}_c$. The OBDD can be reduced, if needed, to an ROBDD; however, we do not need it as a canonical representation. We label its leaf vertices with AAF in case we have an uncertain real-valued result, or with *true* resp. $false$ if the function has a boolean result.

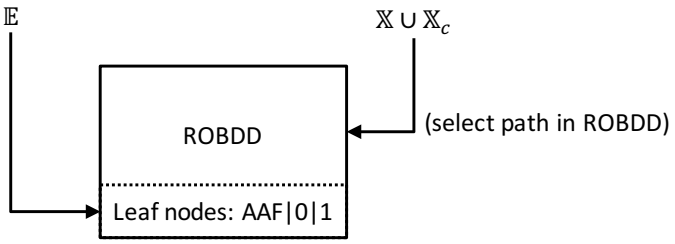

Figure 3: Overview of AADD and how they model hybrid uncertainties.

Definition 3.5 (Uncertain condition) *An uncertain condition is χ_c be a predicate whose variables are from $\mathbb{E} \cup \mathbb{X}$. Let $\mathbb{X}_c$ be the set of all uncertain conditions.*

For convenience, we do not distinguish between basic discrete uncertainties $\mathbb{X}$ and uncertain conditions $\mathbb{X}_c$ and abbreviate $\mathbb{X} \cup \mathbb{X}_c = \mathbb{X}_G = \{\chi_1 \ldots \chi_o\}$ (general discrete uncertainties).

Definition 3.6 (AADD) *An AADD is a directed acyclic graph $AADD = (Q, T, E_1, E_0, \mathbb{X}_G)$ with internal vertices Q, terminal vertices T, edges $E_1 \cup E_0$, discrete uncertainties $\mathbb{X}_G$, and it holds:*

- *Internal vertices $v \in Q$ have two leaving edges $true(v) \in E_1$ and $false(v) \in E_0$, and are labeled with $index(v) = i$; each i corresponds to an $\chi_i \in \mathbb{X}_G$.*
- *AADD are ordered: $index(v_1) < index(v_2)$ for all edges (v_1, v_2) from $v_1 \in Q$ to $v_2 \in Q \cup T$.*
- *Terminal vertices $v \in T$ are labeled with $value(v) \in \mathbb{B}$ for a boolean-valued hybrid uncertainty, or an AAF (Eq. 1) for a continuous-valued hybrid uncertainty.*

An AADD with root $r \in Q \cup T$ represents a hybrid uncertainty $\hat{x} = f(v)$; $f(v)$ is defined recursively:

- For $v \in T$: $f(v) = value(v)$,
- For $v \in Q$: $f(v) = \chi_{index(v)} f(true(v)) \vee \overline{\chi}_{index(v)} f(false(v))$ for $\chi_{index(v)} \in \mathbb{X}_G$

In the following we define arithmetic, logical and relational operations on AADD.

Definition 3.7 (Arithmetic and binary logical operations) *Let $\hat{x}$, $\hat{y}$ be an AADD with root vertices v_x, v_y. Operations $\hat{x} \bigodot \hat{y}$ with $\bigodot : AADD \times AADD \to AADD$ are defined recursively:*

1. *For $v_x, v_y \in T$, the operations result in an AADD that is a terminal vertex v labeled $value(v) = value(v_x) \bigodot value(v_y)$. For $value(v_x), value(v_y)$ of type AAF (Eq. 1), $\bigodot$ is given by Eq. 3,4. For $value(v_x), value(v_y) \in \mathbb{B}$, the operation $\bigodot$ are boolean functions.*
2. *For $v_x \in T, v_y \in Q$ the result is an AADD with root v and $index(v) = index(v_y)$, $false(v) = v_x \bigodot false(v_y)$ and $true(v) = v_x \bigodot true(v_y)$.*
3. *For $v_x, v_y \in Q$ the result is an AADD with root v and*
   - *For $index(v_x) = index(v_y)$:*
     $index(v) = index(v_x)$, $false(v) = false(v_x) \bigodot false(v_y)$, $true(v) = true(v_x) \bigodot true(v_y)$.
   - *For $index(v_x) < index(v_y)$:*
     $index(v) = index(v_x)$, $false(v) = false(v_x) \bigodot v_y$, $true(v) = true(v_x) \bigodot v_y$.
   - *For $index(v_x) > index(v_y)$:*
     $index(v) = index(v_y)$, $false(v) = v_x \bigodot false(v_y)$, $true(v) = v_x \bigodot true(v_y)$.

In the following we define relational operations. We define the comparison of an AADD as with 0 as the right operand. Comparisons of two AADD can be transformed into this representation by subtracting the right side of the relational operator.

Definition 3.8 (Relations) *Let $\hat{x}$ be an AADD with root v_x. Then, the inequality operation $\hat{x} \oslash 0$ with $\oslash \in <, >, ==, \leq, \geq$ and $\oslash : AADD \times \{0\} \to AADD$, is defined recursively as:*

1. *For $v_x \in T$ the result is an AADD that consists of one vertex v_z.*
   - *For $0 \notin \text{range}(value(v_x))$, the result v_z is a terminal vertex with $value(v_z)$ defined by Table 1.*
   - *For $0 \in \text{range}(value(v_x))$, the result v_z is an internal vertex. $\chi_{m+1} = value(v_x) \oslash 0$ is added to $\mathbb{X}_G$, and $index(v_z) = m+1$, $false(v_z) = 0$, and $true(v_z) = 1$.*
2. *For $v_x \in Q$ the result is an AADD with root v_z and $index(v_z) = index(v_x)$, $false(v_z) = false(v_x)$ and $true(v_z) = true(v_x)$.*

Table 1: Relational operators.

$value(v_x) < 0$	$value(v_z) = 1$	$: ub(value(v_x)) < 0$
	$value(v_z) = 0$	$: lb(value(v_x)) \geq 0$
	v_z	$: otherwise$
$value(v_x) \leq 0$	$value(v_z) = 1$	$: ub(value(v_x)) \leq 0$
	$value(v_z) = 0$	$: lb(value(v_x)) > 0$
	v_z	$: otherwise$
$value(v_x) > 0$	$value(v_z) = 1$	$: lb(value(v_x)) > 0$
	$value(v_z) = 0$	$: ub(value(v_x)) \leq 0$
	v_z	$: otherwise$
$value(v_x) \geq 0$	$value(v_z) = 1$	$: lb(value(v_x)) \geq 0$
	$value(v_z) = 0$	$: ub(value(v_x)) > 0$
	v_z	$: otherwise$
$value(v_x) == 0$	$value(v_z) = 1$	$: value(v_x) == 0$
	$value(v_z) = 0$	$: (ub(value(v_x)) < 0) \vee (lb(value(v_x)) > 0)$
	v_z	$: otherwise$

The definition of the inequality operation $\neq$: $AADD \times AADD \rightarrow \mathbb{R}$ is straight forward since it holds that $(\hat{x} \neq 0) = \neg(\hat{x} == 0)$.

The AAF at the terminal nodes are representing the correlations correctly, but over-approximate the range. Accurate bounds are required for the evaluation of relational operators according to Tab. 1. The over-approximation is mostly due the uncertain conditions that impose additional constraints. To compute the exact range of $\hat{x}$ in a terminal node $v_x \in T$ with value $\tilde{x}$ we set up a system of inequations (for all indices k of the nodes along the path to v_x):

$$lb(\tilde{x}), ub(\tilde{x}) \text{ subject to the uncertain conditions for all internal vertexes along the path to } v_x \quad (5)$$

In the concrete implementation of AADD, this is the LP problem that is solved using GLPK LP solver.

Example In Fig. 4 we show a computation with uncertain values `AAF a(0,2), b`. The representation by AAF is shown to the right of the C++ code. The statements in the conditional statement introduce conditions; the AADD for `b` after the conditional statement is shown at the right hand side of the figure.

```
#include <AADD.h>
void example()
{                    // Trace of b:
  AAF a(0,2), b;     // b=0
  b = a+2;           // b=3+ε1
  if (b>3) b+=10;    // b>3:  b=13+ε1
  else b -= 10;      // b<=3: b=-7+ε1
  b = b*2;           // no merge=>AADD
}
```

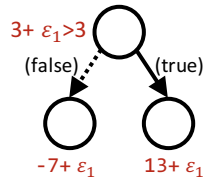

Figure 4: Example for an AADD.

4 Uncertainties in Models of Computation

So far, we have neglected communication and synchronization and focused on the pure computational kernel of processes. In this section we study the impact of uncertainties on synchronization and process execution mechanisms that define Models of Computations (MoCs).

4.1 Processes and signals

Lee and Sangiovanni-Vincentelli introduced the tagged signal model [19] as a meta-model based on which the interaction of models of computation such as dataflow, discrete and continuous timed and or synchronous languages can be modeled. A tag is drawn from a set T that models time and that may be a partially ordered set (untimed models), the integers (timed models), or the set of real numbers (continuous timed models). Hence, MoCs are distinguished only by the structure of signals and, to some extent, by the execution mechanics of processes.

In the following we briefly describe the model from [14, 15] which is equivalent to the tagged signal model but represents time implicitly by the order of events rather than by explicit time stamps. Processes communicate with each other by writing to and reading from signals. Given is a set of values V, which represents the data communicated over the signals. V may be a discrete set or a continuous set. *Events* are the basic elements of signals; they consist of values and time tags. In the special case of untimed signals, the tags may be omitted and ordering of events is determined by the ordering within a signal. We distinguish between three different kinds of events.

Untimed events $\dot{E}$ are just values without further time information beyojnd the ordering within the signal, $\dot{E} = V$. *Discrete timed events* $\hat{E}$ include a pseudo value $\perp$ in addition to the normal values,

hence $\bar{E} = V \cup \{\perp\}$. *Continuous timed events* $\tilde{E}$ use the real numbers as time tags and thus, they can be considered as a (v,t) pair with $v \in V, t \in \mathbb{R}$.

We use $E = \dot{E} \cup \hat{E} \cup \tilde{E}$ and $e \in E$ to denote any kind of event.

Signals are sequences of events. Sequences are ordered and we use subscripts as in e_i to denote the i^{th} event in a signal. E.g. a signal may be written as $\langle e_0, e_1, e_2 \rangle$. In general signals can be finite or infinite sequences of events and S is the set of all signals. We also distinguish between three kinds of signals and $\dot{S}$, $\hat{S}$ and $\tilde{S}$ denote the untimed, discrete timed and continuous timed signal sets, respectively, and $\dot{s}$, $\hat{s}$ and $\tilde{s}$ designate individual untimed, discrete timed, and continuous timed signals. A particular type of signal is used in the corresponding Model of Computation, e.g. an untimed MoC contains only untimed signals.

Processes are defined as functions on signals

$$p : S \to S,$$

which means process p consumes the events on its input signal and produces the events on its output signal. Since a process is a function, it is deterministic and will produce always the same output signal for a given input signal. They may have internal state, which means that the generated event at the output depends on the input event and the internal state of the process at that time. A *process network* is constructed by connecting processes via their input and output signals.

4.2 Models of Computation

Now we are in a position to define several Models of Computation (MoC) that are popular in hardware, software or embedded systems design. A MoC determines execution mechanics that activates a process based on specific conditions, e.g. availability of inputs.

Definition 4.1 (Untimed MoC) *An untimed MoC is the set of all process networks where all processes communicate with each other with untimed signals $\dot{s} \in \dot{S}$ only.*

Processes can be executed once all inputs are available. Outputs can be written depending on values computed by processes. An example is the Kahn Process Networks (KPN) MoC [17].

Definition 4.2 (Static Dataflow) *Static Dataflow is an untimed MoC where each process consumes and produces always the same number of events in each evaluation cycle.*

Different processes may consume and produce different number of events. In the literature this static dataflow is often also called synchronous dataflow. Process execution is done in a static schedule that can be computed before execution.

Definition 4.3 (Discrete Time MoC) *Discrete Time MoC is the set of all process networks where all processes communicate with each other with timed signals $\hat{s} \in \hat{S}$.*

Processes can be activated by arbitrary conditions of time, inputs, or internal states. Examples for discrete time MoC are the discrete-event simulation semantics of VHDL or SystemC.

Definition 4.4 (Continuous Time MoC) *Continuous time MoC describes processes by differential and algebraic equations (DAE). Signals are continuous timed signals $\tilde{s} \in \tilde{S}$ that represent the solutions of the DAEs.*

In the continuous-time MoC, the process execution is controlled by a 'solver'. The solver can select discrete time steps in order to approximate the ideal, continuous time signals while minimizing the error due to discretization. Values of events for arbitrary times can be computed by interpolation if needed.

Note, that the set V of values transported by signals and processed by processes is irrelevant for the definition of MoCs. V may be an arbitrary set, binary, discrete, continuous, structured, etc. This obliviousness allows us to extend these MoCs to values with uncertainties. However, uncertain values might have impact on the concrete behavior of models in different MoCs.

4.3 Uncertain values and their impact on MoCs

In the case of uncertain values, the set V becomes a set of continuous, discontinuous, or hybrid (propagated) uncertainties. Then, the computation of the function p that defines the behavior of processes can be computed and represented as described in Sec. 3. However, as conditions on values can trigger the execution of processes, we have to study the impact of uncertain values. For this purpose we distinguish static and dynamic MoC activation:

Definition 4.5 (Static/Dynamic MoC) *MoC are called static if the order of all events and their times are independent from the values of the computations. MoCs that are not static are dynamic.*

Static dataflow is a static MoC: by definition 4.2, a constant number of events is produced and consumed at each process activation.

The continuous time MoC can also be considered as a static MoC. In continuous time MoC, there are events for every time drawn from $\mathbb{R}$; we consider the way how they are computed as an implementation detail.

Untimed MoC and discrete timed MoC are dynamic MoC, if they are not further restricted. As proof we give one example of a process (in pseudo-code) that shows it is not static:

```
if(condition_on_value) // can be uncertain
  write(signal_event_that_activates_process);
else
  do_nothing;
```

Nevertheless, many models and implementations in these MoC can be considered as static if they avoid activation of processes depending on conditions on values.

If we have static MoCs or models that can fulfill the requirements for a static MoC, we can do symbolic simulation by replacing the set of values V with symbolic representations such as AADD. Static MoCs (or static models) pave the path for a very easy implementation in design and modeling languages that support abstract data types and operator overloading; examples for such languages are C++, VHDL, SystemC. Then, we can replace pre-defined data types with an AADD-based data type and leave the symbolic simulation to overloaded operators.

For dynamic MoC in the general case we have to consider that process execution can become uncertain. This leads to a number of problems that we have not yet completely solved:

- First, a process, at the same time, must be executed and not executed. We can solve this by adding a symbol $\sqcup$ to the set of reachable values represented by an AADD. Then, a process, when executed can compute outputs also for this case (that is, doing nothing) in addition to the other symbolic manipulations on AADD; however this is seen as future work.
- A fundamental issue is the interface between continuous-time MoC and other MoC that leads to a non-computable problem in general. Here, uncertainty of values in conditions leads to an uncertain time of process activation; to cope with this issue is rather a modeling challenge and can be avoided in many cases.

4.4 Turning the SystemC AMS simulator into a symbolic simulator

SystemC AMS ([6, 25], IEEE Std 1666.1-2016) targets the modeling of highly complex mixed discrete/continuous systems with a focus on a high simulation performance. It extends SystemC's timed (discrete-event) MoC with support for the static dataflow MoC, and a continuous time MoC. By the interaction of the untimed static dataflow with the timed signals of the other MoCs its samples get timed semantics; therefore, in [6, 25] we used the term *timed dataflow (TDF)*. SystemC AMS uses TDF as a coordinating MoC that controls the interactions between different MoC and executes them in constant time steps. Its processes can be specified by

- C++ code that models e.g. computations of embedded software.
- Transfer functions, linear differential equations, or static nonlinear functions.
- Linear circuits and switches that model sensors and actuators.

To enable industrial application, SystemC AMS provides a framework that that yields predictable behavior and scalability. [1] Predictable behavior means that the behavior of models is intuitive; this is achieved by using TDF as a coordinating MoC in hierarchical models. For the sake of scalability, SystemC AMS by default:

- Uses MoC that are scalable and numerically robust: data-flow, and linear continuous-time models.
- Does not compute threshold-crossing of continuous quantities; continuous quantities are sampled.

However, SystemC AMS gives users the choice where, and how to solve problems that are known to lack scalability. For this purpose, SystemC AMS provides the user with interfaces to allow them to select time steps in a dynamic way, to compute thresholds accurately, and to add solvers for non-linear differential equations if needed.

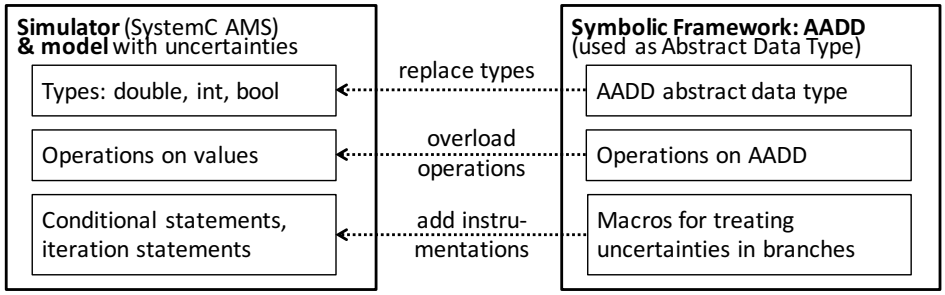

Figure 5: Overview of implementation based on the SystemC AMS reference simulator.

We extended Coseda's implementation of the numeric SystemC AMS simulator [1] to a symbolic simulator. The underlying MoC are static MoC provided the C++ code does not fork additional processes or uses shared variables. Like the SystemC AMS standard we give the user the choice where to use symbolic simulation. For this purpose, we provide an abstract data type `AADD` that provides AADD, overloaded operations on it, and some macros that support the interactive choice between symbolic and numeric execution for branches in loops and conditional statements. Figure 5 gives an overview.

[1] The standardization was clearly industry-driven with focus on scalability. Hybrid automata in contrast, with good reason, focus on the fundamental issues with a penalty in terms of scalability.

4.5 Scalability: is symbolic simulation with AADD worth the effort?

The use of AADD imposes some overhead. In the following we discuss the overhead to a single, numerical simulation run and the fidelity of the results, and compare complexity with multi-run simulation based approaches that strive to yield similar results (Table 2).

Table 2: Scalability and fidelity of numerical vs. AADD-based symbolic simulation.

	Multi-run simulation, continuous		Symbolic with AADD	
	MC analysis	**WC/EVA analysis**	**continuous**	**discrete**
Scalability	quadratic(confidence), const(#cont. uncert.)	Worst case: $>$ exp. Typical case: linear	linear (#cont. uncert.)	exp. (#disc. uncert.)
Fidelity	under-approximation	under-approximation	over-approximation	over-approximation

For numerical simulation-based approaches of continuous systems, Table 2 gives the number of numerical simulation runs that are needed to achieve comparable, yet not comprehensive results by semi-formal approaches. We compare Monte Carlo analysis (MC), or Worst-Case (WC) analysis by Extreme Value Analysis (EVA). MC analysis has the advantage that the number of simulation runs does not grow with the number of uncertainties; it grows quadratically with the desired confidence. WC/EVA analysis tries to find worst-case corners of properties by checking possible combinations of corners of uncertainties. Various heuristics find under-approximations more efficiently in particular for linear systems, but it is impossible to get comprehensive results in the general case. For mixed discrete/continuous systems resp. uncertain CPS, we have to visit each reachable discrete mode and do MC, WC or EVA analysis in it for comprehensive results.

For AADD-based symbolic simulation the overhead depends on the number of uncertainties. In the continuous domain, AADD-based symbolic simulation introduces a constant overhead that only grows with the number of uncertainties; however, over-approximation can be an issue for non-linear systems. In the discrete domain, we in general have the problem that there is an exponential growth of possible execution paths (path explosion problem). For AADD, the size (and the run-time) grows with the size of the (R)OBDD. It *can* grow exponentially in the worst case with the number of basic discrete uncertainties; however, it does not in many cases which is a well-known issue in the research on ROBDD.

A problem is that reduction of OBDD to ROBDD costs a lot of time. Therefore, we use OBDD or even a simple array structure as implementation in [23] that we call XAAF. This is still efficient for some thousand reached discrete states (leaf nodes of AAF). Scalability using the array-based implementation is shown by examples of a $\Sigma\Delta$-converter in [23] and a PLL in [5].

5 Analysis of an uncertain water-level monitor with error-reactions

5.1 Modeling the water level monitor in SystemC AMS

We have characterized cyber-physical systems in Sec. 1 by its open and networked nature that, due to the increased likelihood of deviations or faults, demands for means to compensate such uncertainties. For demonstration, we extend the water level monitor model from [3] in that direction. The water level monitor consists of a tank of which the water level falls with a rate of $falling = 2$ in./sec. A pump can, if switched on, add an incoming flow so that in sum the water level increases with a rate of $rising = 1$ in./sec. Two sensors indicate whether a level of 5 in. (empty) or 10 in. (full) is reached. The water level must not fall below 1 in. or rise above 12 in.

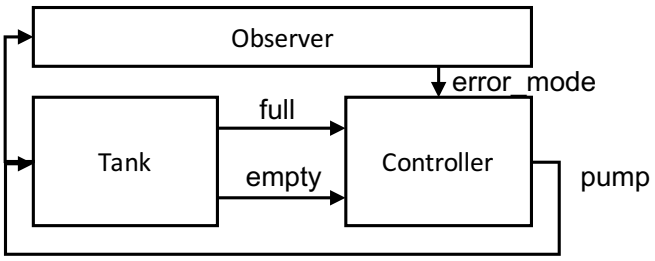

Figure 6: The cyber-physical water level monitor with observer and a controller with fail-safe mode.

Compared with the water level monitor from [3], we added in Fig. 6 the following uncertainties:

- Continuous uncertainties in the rates of the water tank: $falling = 2 + \varepsilon_1$, $rising = 1 + \varepsilon_2$.
- A discrete uncertainty: the possibility that the sensor in the tank does not indicate, when 10 in. are reached as an initiating fault: $full = \chi_1$.

Obviously, one can find and add more uncertainties. Furthermore, there is a process (Observer), written in C++, that shall detect the discrete fault and signal the controller to go into a fail-safe error mode. It is invoked every second and checks whether the pump is on for longer than 10 sec. (self-diagnosis) and then signals via *error_mode* that the controller shall go into a fail-safe state:

```
void sca_processing()
{
  timer += 1; // It is activated every 0.1 sec.
  if (error_mode == false)
  {
    if (pump==true)
      if (timer > 10) error_mode = true;
    else timer = 0;
  }
}
```

In the fail-safe state, the controller limits the time the pump is on. We specified the water level model using the TDF MoC of SystemC AMS. The controller process (also written in C++) is activated every 0.1 sec. and samples the water level sensors. The observer process is activated every 1.0 sec.

For symbolic simulation we have to modify the SystemC AMS model, supported by macro definitions. This affects in particular branches in control flow. Here, we have to consider that the result of a branch condition can be *true*, *false*, but as well uncertain. In the first two cases, the code remains unchanged. The third case occurs for comparisons that depend on at least one AADD. Then have to apply a code instrumentation that adds an uncertain condition; details on the code instrumentations are described in [23].

5.2 Symbolic simulation of the uncertain water level monitor

Objective of the verification of uncertain CPS is to show that in the presence of the uncertainties (inaccuracies, initiating faults), self-diagnosis and error reaction ensure that no unsafe state (here: a level above 15 in.) is reached. For this purpose we used symbolic simulation bounded to 40 sec. time with assertion checking.

In figure 7 we show the results of symbolic simulation of the water level monitor for the cases with and without the discrete error. The total time required for one symbolic run of the water level monitor

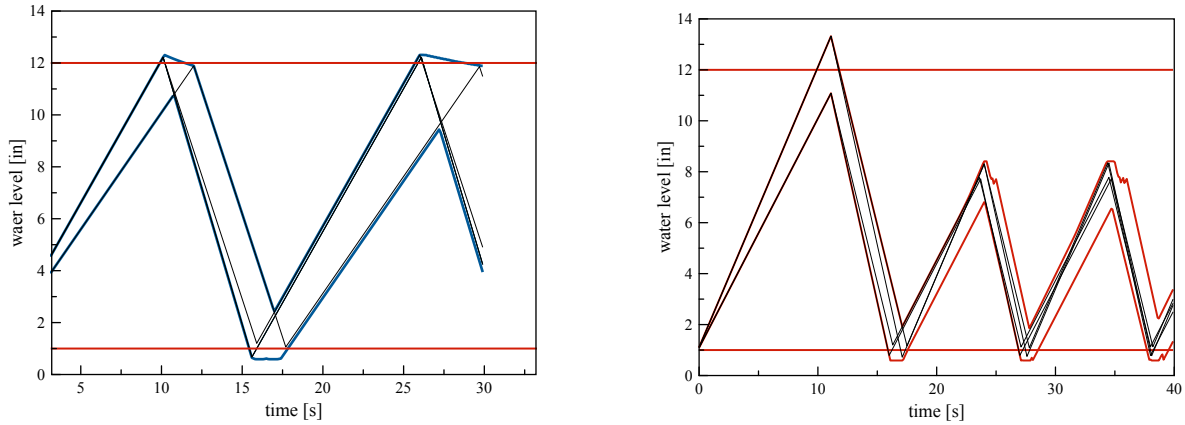

Figure 7: Symbolic simulation results (worst case borders plotted) compared with numeric worst case (by extreme value analysis) analysis results.

without observer is 6.8 sec.; it resulted in an AADD with 387 leaf nodes. The result is shown in Fig. 7a, left. As it can be seen, the inaccuracies in the rates can lead to an error. The corner-case simulation results (that my be an under-approximation) are plotted black, the range of the water level is plotted in blue. For the water-level monitor and the observer, with injecting a deterministic error in a sensor, we get a run-time of 30 sec. and an AADD with 945 leaf nodes: once the error is detected, the controller goes in a state in which he limits the on-time of the pump (Fig. 7b, right only shows this case).

5.3 Discussion

The example above does not show scalability to large problem sizes; this was not the intention. It shows that an how the approach contributes to scalability towards applicability: first, we can (re-)use existing models or C-code. Second, it shows that it is useful to solve the problem of analyzing the robustness of CPS under the presence of uncertainties including discrete faults. This is a domain, where pure numerical simulation fails as it needs an exponential number of simulation runs.

Lessons learned from the concrete implementation shows that run-times of symbolic simulation runs very much on details of models; e.g. we used timers that were not necessary (not wrong, but simply not needed) which doubled the run-time. Leaving modeling issues aside, the operations on AADD are implemented in an inefficient way and clearly need further research, while the LP problem we identified significantly contributes to keep over-approximation very little. Interesting would in that context be the use of a SAT/SMT solver to represent the functional dependabilities while maintaining the LP problem and the solver at the interface between discrete and continuous components.

6 Conclusion

We have described the problem of verification of uncertain CPS. As two main results we have introduced first AADD as a model that allows us compute with uncertainties, independent from a particular model of computation. Second, we have shown how this allows us to implement a symbolic simulator that is not strictly bound to a particular model of computation. The following aspects contribute to scalability towards larger problem sizes and, in particular, application in industrial development flows:

1. The approach to separate computation with uncertainties from simulation using signals and processes in different models of computation allows us to bring symbolic simulation easily into pre-existing frameworks. The re-use of existing models, simulators, and other verification infrastructure from numerical simulation becomes easier, at least for static models of computation. Currently we support SystemC and its AMS extensions; in student's work also Labview has been enabled for symbolic simulation.
2. The flexibility gained by not being bound to a particular model of computation (e.g. hybrid automata) makes it easier to formulate models that avoid known issues for scalability. In the model of the example, we could hence make the following abstractions:
   - The implementation of the controller and the sensors is discrete; we have hence chosen an interface between continuous and discrete parts that uses sampling – this avoids the need to determine threshold crossing.
   - We used the static data flow model of computation (together with continuous-time) for the overall structure of the model.
3. AADD combine affine arithmetic with (R)OBDD. Both are know to be efficient representations in the discrete and continuous domain. In particular the proposed use of an LP solver as described in Sec. 3 significantly reduces over-approximation even for larger numbers (i.e. 1000s) of leaf nodes (reached states). To foster higher scalability, operations on AADD must be further optimized to proceed to complex software systems. While (R)OBDD and affine arithmetic are each maybe not the most efficient representations, they are at least common and well-understood, and vast research is available to increase efficiency.

6.1 Future work

Currently, we limit the implementation of the symbolic simulation to static MoCs. This is useful as it allows us the integration of AADD in existing simulators without having to change their execution semantics. However, this limitation is too strict if one would like to check whether deadlocks or race conditions are due to uncertainties. A first idea to allow symbolic simulation also of dynamic MoCs would be to integrate a symbol for $\sqcup$ for that case; operators on AADD could then simply do nothing for that case.

We currently only support uncertainties that are modeled by non-deterministic choice of a set or a range. However, in particular when modeling many possible discrete errors one is also interested in the probability of a resulting hazard. Ongoing work targets the extension towards probabilistic uncertainties. We have described a first approach in that direction in [13]. Also, Olbrich's distribution arithmetic [21] would be a useful starting point in that context.

Acknowlegement

This work is funded, in part, within the ANCONA project (16ES021) within the program IKT 2020 by the German Ministry of Education and Research (BMBF) and by Robert Bosch AG, Intel AG, and Mentor Graphics GmbH.

References

[1] *SystemC AMS Proof-of-Concept Download*. `http://www.coseda-tech.com/systemc-ams-proof-of-concept`. Accessed: 2017-03-21.

[2] Matthias Althoff, Akshay Rajhans, Bruce H. Krogh, Soner Yaldiz, Xin Li & Larry Pileggi (2011): *Formal Verification of Phase-Locked Loops Using Reachability Analysis and Continuization*. In: *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 659–666.

[3] R. Alur, C. Coucoubetis, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis & S. Yovine (1995): *The algorithmic analysis of hybrid systems*. *Theoretical Computer Science* 138, pp. 3–34, doi:10.1016/0304-3975(94)00202-T.

[4] Saswat Anand, Patrice Godefroid & Nikolai Tillmann (2008): *Demand-Driven Compositional Symbolic Execution*. In C.R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 4963, Springer Berlin Heidelberg, pp. 367–381, doi:10.1007/978-3-540-78800-3_28.

[5] Erich Barke, Andreas Fürtig, Georg Gläser, Christoph Grimm, Lars Hedrich, Stefan Heinen, Eckhard Hennig, Hyan-Sek Lukas Lee, Wolfgang Nebel, Gregor Nitsche, Markus Olbrich, Carna Radojicic & Fabian Speicher (2016): *Embedded Tutorial: Analog-/Mixed-Signal Verification Methods for AMS Coverage Analysis*. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE'16)*, pp. 1102–1111, doi:10.3850/9783981537079_1010.

[6] Martin Barnasconi, Karsten Einwich, Christoph Grimm & Alain Vachoux, editors (2013): *Standard SystemC® AMS extensions 2.0 Language Reference Manua*. OSCI. Available at `http://accellera.org`.

[7] Randal E. Bryant (1986): *Graph-based algorithms for Boolean function manipulation*. *IEEE Transactions on Computers* C-35(8), pp. 677–691, doi:10.1109/TC.1986.1676819.

[8] Werner Damm, Stefan Disch, Hardi Hungar, Jun Pang, Florian Pigorsch, Christoph Scholl, Uwe Waldmann & Boris Wirtz (2006): *Automatic Verification of Hybrid Systems with Large Discrete State Space*. In: *Automated Technology for Verification and Analysis, 4th International Symposium, ATVA 2006*, pp. 276–291.

[9] Fang Fang, Tsuhan Chen & Rob. A. Rutenbar (2002): *Floating-Point Bit-Width Optimization for Low-Power Signal Processing Applications*. In: *IEEE International Conference on Acoustic, Speech and Signal Processing*, 3, pp. 3208–3211.

[10] Luiz Henrique De Figueiredo & Jorge Stolfi (2004): *Affine Arithmetic: Concepts and Applications*. *Numerical Algorithms* 37(1-4), pp. 147–158, doi:10.1023/B:NUMA.0000049462.70970.b6.

[11] Antoine Girard (Springer, 2005): *Hybrid Systems: Computation and Control*, chapter Reachibility of Uncertain Linear Systems Using Zonotopes, pp. 291–305. LNCS 3414, doi:10.1007/978-3-540-31954-2_19.

[12] Christoph Grimm, Wilhelm Heupke & Klaus Waldschmidt (2005): *Analysis of mixed-signal systems with affine arithmetic*. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 24(1), pp. 118–123, doi:10.1109/TCAD.2004.839469(410) 24.

[13] Christoph Grimm & Michael Rathmair (2017): *Dealing with Uncertainties in Analog/Mixed-Signal Systems*. In: *Proceedings of the 54th Design Automation a Conference, 2017*, pp. 1–6.

[14] Axel Jantsch (2003): *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[15] Axel Jantsch & Ingo Sander (2005): *Models of computation and languages for embedded system design*. *Computers and Digital Techniques, IEE Proceedings* - 152(2), pp. 114–129, doi:10.1049/ip-cdt:20045098.

[16] Jinseong Jeon, Kristopher K. Micinski & Jeffrey S. Foster (2012): *SymDroid: Symbolic Execution for Dalvik Bytecode*. Technical Report CS-TR-5022, Department of Computer Science, University of Maryland, College Park.

[17] Gilles Kahn (1974): *The semantics of simple language for parallel programming*. In: *Proceedings IFIP74*, North-Holland, Amsterdam, p. 471475.

[18] Colas Le Guernic & Antoine Girard (2009): *Reachability Analysis of Hybrid Systems Using Support Functions*, pp. 540–554. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-02658-4_40.

[19] E. A. Lee & A. Sangiovanni-Vincentelli (2006): *A Framework for Comparing Models of Computation*. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 17(12), pp. 1217–1229, doi:10.1109/43.736561.

[20] E.A. Lee (2008): *Cyber Physical Systems: Design Challenges*. In: *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pp. 363–369, doi:10.1109/ISORC.2008.25.

[21] Markus Olbrich & Erich Barke (2008): *Distribution Arithmetic for Stochastical Analysis*. In: *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, ASP-DAC '08, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 537–542, doi:10.1109/ASPDAC.2008.4484009. Available at `http://dl.acm.org/citation.cfm?id=1356802.1356932`.

[22] Sylvie Putot (2013): *Lecture Notes, Digicosme Spring school 2013: Static Analysis of Numerical Programs and Systems*. MEASI Laboratory, CEA LIST.

[23] Carna Radojicic & Christoph Grimm (2016): *Formal Verification of Mixed-Signal Designs Using Extended Affine Arithmetic*. In: *12th Conference on PhD Research in Microelectronics and Electronics*, doi:10.1109/PRIME.2016.7519482.

[24] Carna Radojicic, Christoph Grimm, Florian Schupfer & Michael Rathmair (2013): *Verification of Mixed-Signal Systems with Affine Arithmetic Assertions*. *VLSI Design* 2013, doi:10.1155/2013/239064.

[25] Alain Vachoux, Karsten Einwich & Christoph Grimm (2005): *SystemC Extensions for Heterogeneous and Mixed Discrete/Continuous Systems*. In: *International Symposion on Circuits and Systems 2005 (ISCAS '05)*, IEEE.

[26] Warren E Walker, Poul Harremoës, Jan Rotmans, Jeroen P van der Sluijs, Marjolein BA van Asselt, Peter Janssen & Martin P Krayer von Krauss (2003): *Defining uncertainty: a conceptual basis for uncertainty management in model-based decision support*. *Integrated assessment* 4(1), pp. 5–17, doi:10.1076/iaij.4.1.5.16466.