# Automatic Analysis and Abstraction for Model Checking HW/SW Co-Designs modeled in SystemC

Timm Liebrenz
Technische Universität Berlin
Berlin, Germany
timm.liebrenz@tu-berlin.de

Verena Klös
Technische Universität Berlin
Berlin, Germany
verena.kloes@tu-berlin.de

Paula Herber
Technische Universität Berlin
Berlin, Germany
paula.herber@tu-berlin.de

## ABSTRACT

Embedded systems usually consist of deeply integrated hardware and software components. As a consequence, modular verification is not easily possible. One important step towards modular verification of integrated HW/SW systems is to automatically compute abstractions of components that influence the overall system behavior but are not relevant for a given property. In this paper, we present an automatic abstraction technique for HW/SW co-designs modeled in SystemC. The key idea is to use a variant of classical abstract interpretation that is tailored for the specific semantics of SystemC. Our main contributions are the following: First, we present an analysis that determines data-dependencies between variables and equivalent data values with respect to conditional branches while taking the timing behavior and scheduling policies of SystemC into consideration. Second, we use the results for slicing and variable abstraction to significantly reduce the semantic state space of a given SystemC design and again produce a valid abstract design. Our abstraction technique makes it possible to automatically verify properties for comparatively large designs with the UPPAAL model checker, which cannot be handled without our approach. We demonstrate this with two case studies from the SystemC reference implementation.

## CCS Concepts

•**Software and its engineering** → **Automated static analysis;** *Formal methods;*

## Keywords

Data Abstraction, Verification, Formal Analysis, Slicing

## 1. INTRODUCTION

Embedded systems are often used in safety critical applications, for example in airplanes, cars and transportation systems, for which the correctness and reliability is a crucial issue. At the same time, they typically consist of deeply integrated hardware and software components. As a consequence, modular verification is difficult to achieve.

Although there has been a lot of work on model-driven development and verification methods for embedded HW/SW systems, they typically do not scale well for industrial-sized systems. To overcome this problem, modular verification techniques, e.g., contract-based approaches, are needed. However, the deep integration of hardware and software in embedded systems makes it impossible to verify modules separately. As a consequence, a new notion of modularity is needed. In [21], we have presented a modular verification concept that relies on the idea that we can divide a given system design into submodels where each submodel concentrates on certain aspects, and abstracts from other aspects. For example, a submodel might focus on the hardware part, the software part, or the communication architecture of a given system. The main idea to achieve such submodels is to compute abstractions for all components that are not relevant for the current aspect.

As a first step towards this goal, we have developed an automatic abstraction technique for HW/SW co-designs modeled in SystemC, which takes a SystemC design and a property as input and computes an abstraction where all variables that are not relevant for the given property are removed and all variables that influence the given property are replaced by (abstract) variables with significantly smaller value ranges such that only the relevant behavior is preserved. Our abstraction technique directly targets SystemC designs and produces a valid abstract SystemC design. The main idea is to abstract values for data variables by calculating non-overlapping partitions of their value range and mapping the values to the corresponding partition. The partitions are equivalence classes according to the control flow of each process in a given design. This decreases the range of variable values and, thus, the semantic state space. While the general idea of mapping variable values to interval equivalence classes is not new (cf. abstract interpretation, predicate abstraction based on conditions), the main challenge in the context of SystemC is to derive the intervals from the control flow of SystemC processes. This requires to take the specific execution semantics of SystemC into account, including the semantics of ports and channels, event-triggered process execution, and delta-cycles.

Our abstraction technique can be automatically applied and also enables slicing, which corresponds to a complete abstraction of code blocks due to a strong independence of some given property. Our reduction of the semantic state

space is safe as it preserves the behavior with respect to a given property.

As we separate the abstraction process from later verification, our approach can be combined with various verification techniques. As a major advantage, we can improve the performance of the verification by using the abstracted system to verify properties. To evaluate our approach, we have implemented our abstraction technique in Java. Our abstraction technique generates abstract SystemC designs that can be further processed with any verification tool. To evaluate the effect of our abstractions, we have used the **S**ystemC to **T**imed **A**utomata **T**ransformation **E**ngine (STATE) [22] to transform the concrete and the abstract SystemC designs into UPPAAL timed automata, and the UPPAAL model checker to verify some crucial properties. To show the effectiveness of our approach in terms of reduced verification times, we have used two case studies: a simple producer-consumer example and a multicast helix packet switch, both taken from the SystemC reference implementation.

In the following, we first briefly introduce SystemC in Section 2 and discuss related work in Section 3. Then, we present the general idea of our abstraction technique in Section 4, our SystemC specific variable analysis in Section 5, and our slicing approach in Section 6. We present experimental results in Section 7 and conclude in Section 8.

## 2. SYSTEMC

SystemC [23] is a system level design language, which can be used to model hardware, software, and their interactions. It is implemented as a C++ class library and extends C++ with constructs for the description of hardware, e. g., time, reactivity, concurrency and hardware data types. The language clearly separates computation and communication and has a modular structure. Modules contain processes that encapsulate computations, ports that are connected to ports of other modules via channels, and sockets that enable a direct connection with other modules. Channels implement communication methods.

The execution of a SystemC design is controlled by the SystemC scheduler. It controls the simulation time and the execution of processes, handles event notifications, and updates primitive channels. SystemC introduces an integer-valued time model with arbitrary time resolution. (Deterministic) time is modeled in SystemC using timed event notifications. Like typical hardware description languages, SystemC supports the notion of delta-cycles, which impose a partial order on parallel processes.

## 3. RELATED WORK

There exist many approaches for the automated formal verification of SystemC designs. For example, in [15, 14], SystemC is transformed into state machines, Karlsson et al. [25] verify SystemC designs using a petri-net based representation and the PRES+ model checker, and in [11, 18], an encoding from SystemC into the verification toolbox CADP is proposed. In [13], SystemC designs are verified using k-inductive invariant checking with CBMC [7] and Boolector [27] as underlying SMT solver. In [20, 22], a transformation from SystemC into UPPAAL timed automata is proposed, in [21], a transformation from SystemC into the input language of the BLAST model checker [2], and in [24] an encoding of SystemC designs into the specification language of the UCLID SMT solver [26]. All of these approaches rely on a transformation from SystemC into the input language of some verification tool. While this conveniently enables automated formal verification of SystemC designs, the verification itself severely suffers from scalability issues in all cases. Although all of the verification tools include optimizations and some make use of sophisticated abstraction techniques, none of those optimizations is tailored for SystemC designs and, thus, none of them makes use of the specific semantics of SystemC to abstract from concrete data values.

There exist only a few approaches that apply abstraction techniques directly on a given SystemC design. In [10], the authors propose to transform SystemC designs to the input language of the software model checker SPIN. They focus only on the relevant parts of the model while leaving functional blocks untransformed. However, they still consider the complete state space including all concrete data values. In [4], [3], Chou et al. define symbolic behavioral states, that represent the system behavior between the beginning of an evaluation phase to the point at which simulation time advances. However, this abstraction prevents reasoning about values at a finer level, for example between delta cycles. Furthermore, they do not provide any data abstraction.

Widely used approaches to reduce the state space of systems by abstracting from concrete data values are abstract interpretation [9, 17] and predicate abstraction [12]. The former is based on the idea that concrete variables can be replaced by abstract variables with smaller data ranges. The latter is based on abstract interpretation and is performed by replacing concrete variables by predicates which are usually boolean variables or boolean formulas. It is often used in lazy abstraction [19], where predicates over-approximate the system and are refined if a spurious counterexample occurs during verification. Three approaches for the verification of SystemC that use lazy abstraction are [5, 6, 16, 21]. However, lazy abstraction means that the verification is in the loop of the abstraction, which is typically very costly. Furthermore, the abstraction technique is implemented within the verification engine and not interchangeable with other verification back-ends.

Our approach for the automatic abstraction of HW/SW co-designs modeled in SystemC is a variant of classical abstract interpretation especially tailored for the execution semantics of SystemC.

## 4. AN AUTOMATIC ABSTRACTION TECHNIQUE FOR SYSTEMC

The main contribution of this paper is an automatic abstraction technique for HW/SW co-designs modeled in SystemC, which can also be used for slicing of SystemC designs. Our abstraction is based on a novel variable analysis especially tailored for SystemC. It directly targets SystemC designs such that our approach can be combined with any existing verification technique for SystemC.

Our analysis operates on the SysCIR intermediate representation of SystemC designs which provides convenient access to all design elements including resolved port and socket connections. With that, we can easily traverse the complete SystemC design and its call graph, but we inherit the assumptions the SysCIR imposes on a SystemC design (see [22]). The most important restrictions are no dynamic creation of variables or processes, restriction to in-

```
1  void poll_sensor(){
2    int c0;
3    wait(100, SC_NS);
4    c0 = read_sensor();
5    port_C1.write(c0);
6  }
```

```
1  void controller2(){
2    int c2;
3    c2 = port_in_C1.read();
4    open_valve2();
5    if(c2 < 3){
6      wait(40, SC_NS);
7    } else {
8      wait(100, SC_NS);
9    }
10   close_valve2();
11 }
```

```
1  void controller1(){
2    int c1;
3    c1 = port_in_C0.read();
4    if(tick < 10){
5      wait(200, SC_NS);
6      tick++;
7    } else {
8      open_valve1();
9      if(c1 < 5){
10       wait(50, SC_NS);
11     } else {
12       wait(80, SC_NS);
13     }
14     close_valve1();
15   }
16   port_C2.write(c1);
17 }
```

Listing 1: Concrete design

teger, boolean, and enum as base types, no function pointers, no pointer arithmetic, no direct memory access, and no references to fields of structs.

We compute partitions of concrete value ranges such that each partition forms an equivalence class according to the execution of the program. In the following, we use integers as concrete variables and show how they can be abstracted to enumerations of non-overlapping integer intervals, without changing the behavior of a SystemC design. Note that the behavior-preservation means that no spurious counter examples may be produced.

To conserve the timing behavior of a design, we do not abstract variables that are used in `notify` or `wait` statements. We also do not abstract variables in some other cases, e.g. in case of arithmetic operations with more than one source variable or in case of increment operators (`x++`) (cf. Section 5) as this would lead to nondeterminism.

For all other variables, we construct an abstract type, which contains all non-overlapping intervals that are derived from conditional branches in the control flow and, furthermore, some additional intervals to handle data dependencies.

To illustrate our approach, we use a small temperature control application, as shown in Listing 1. Our SystemC-specific variable analysis, which can be used to automatically gather the necessary information to enable the abstraction, is introduced in Section 5.

Our temperature control application consists of one module that polls a temperature sensor and two controller modules. The module `poll_sensor` periodically reads a sensor and sends the sensor data to `controller1`. The first controller initially waits for some time to let the system heat up and then opens a valve for some time that depends on the current temperature. Subsequently, it sends the sensor data to the second controller, which also opens another valve for some time that depends on the current temperature.

The variables `tick`, `c0`, `c1`, and `c2`, are candidates for abstraction. However, as `tick` is incremented within the first controller and `c0` is initialized with an arbitrary sensor value, we do not abstract from these variables. However, we can compute abstractions for the variables `c1` and `c2`.

In the following, we first explain the construction of abstract types for arbitrary variables. Then, we show how these abstract types can be used to compute a safe over-

| Condition | Result |
|---|---|
| $x < c$, $x \geq c$ | $(-\infty, c-1]$, $[c, \infty)$ |
| $x > c$, $x \leq c$ | $(-\infty, c]$, $[c+1, \infty)$ |
| $x = c$, $x \neq c$ | $(-\infty, c-1]$, $[c, c]$, $[c+1, \infty)$ |

Table 1: Interval candidates for conditions

approximation of a given SystemC design and apply this to our temperature control application.

## 4.1 Equivalence Classes as Abstract Types

To replace variables with concrete data ranges with abstract variables with smaller data ranges in a way that safely preserves the behavior of the design under verification, we compute *equivalence classes*, that is, data ranges of each variable for which the design behaves equivalently. To achieve this, we analyze the conditions of each conditional branch in a given design and partition the data values of each variable into intervals such that all conditions in a given design evaluate equivalently for all values in each partition. The partitions (i.e., equivalence class candidates) that are derived for different kinds of conditions are listed in Table 1.

In our example, three conditions are present. Since we determined that the variable tick remains untouched, we only obtain partitions for c1 and c2. The condition $(c1 < 5)$ generates the partitions $(-\infty, 4]$, $[5, \infty)$ for c1 and from $(c2 < 3)$ we obtain $(-\infty, 2]$, $[3, \infty)$ for c2. Each partition contains all values of the variable that result in the same control flow behavior.

However, partitions that are solely derived from conditions are not sufficient as equivalence classes for the corresponding variables as the variable may be used in assignments, arithmetic operations, and method calls, where the value of one variable is written to another. As a consequence, data dependencies between all variables must be computed and the influence of each variable on the relevant intervals of all dependent variables must be propagated into the computation of equivalence classes of the former. To take assignments, arithmetic operations and method calls where a source variable $s$ influences a target variable $t$ into consideration, we compute equivalence classes $I_s$ for each source variable that enable an unambiguous mapping to the abstract domain of each target variable $I_t$. This means that each partition of $I_s$ has to be contained in a partition of $I_t$. We ensure that there exists a mapping function for all abstract domain pairs $(I_s, I_t)$ that are connected with a data dependency such that

$$\forall i_s \in I_s \; \exists i_t \in I_t :: b_l(i_t) \leq b_l(i_s) \wedge b_u(i_t) \geq b_u(i_s) \quad \text{with}$$

boundary access operators $b_l([l,u]) = l$ and $b_u([l,u]) = u$.

To achieve this, we compute the equivalence classes for source variables by starting on a leaf in the dependency graph and propagating the intervals from the target to the source node. If the dependency results from an assignment with an arithmetic operation, we modify the target intervals according to the rules in Table 2 before we add them to the source node. The rules represent the inverse function of the arithmetic operations. In the case of division, we also include the truncation behavior (e.g., if we want to determine equivalent values for $x$ for $y = x/3$ and we know that e.g., $5/3 = 4/3 = 3/3 = 1$, it follows that the values in $[3, 5]$ are equivalent regarding this statement). We only consider operations between a variable and a constant value

here. For operations between two variables, it is not generally possible to find an unambiguous inverse function on intervals. Therefore, we do not abstract the source variables in this case. The target variable can be abstracted and a mapping of the resulting integer value to the target domain can be used for the assignment.

Note that there might be cyclic dependencies between variables. Such dependencies can exist between local variables in a function as well as between variables in different modules. We have to resolve all cyclic dependencies before we can apply our propagation algorithm. Otherwise, it would not terminate. To resolve cyclic dependencies, we replace each cycle in the dependency graph by a master node that includes all variables that are part of the cycle. We calculate the interval information for this master node by merging all interval information inside the cycle as all included variables share the same information. For edges inside the cycle that include arithmetic operations, these operations could be executed arbitrarily often. Hence, we apply a simple widening that sets the corresponding interval information to $\perp$, which means that we do not abstract the affected variables (e.g., in the case of `x++`). As all variables inside the loop are affected, all of them are not abstracted. The resulting non-cyclic dependency graph can be processed as described before.

As a result, we get equivalence classes as non-overlapping sets of intervals for each abstractable variable.We use these as abstract types for the corresponding variables. As the resulting sets are compatible for dependent variables, we can create unambiguous mapping functions between their abstract domains as well as mappings between concrete integers and all abstract domains. The latter take an integer value $x$ and return a value of the target domain for which holds that $b_l(i_t) \leq x \wedge b_u(i_t) \geq x$. With that, we are able to replace abstractable integer variables by variables of the corresponding abstract type. Our abstraction is safe, as we only abstract variables where we could find a non-overlapping partition of the value range (in all other cases our analysis result is $\perp = \{[l, l] \mid l \in \mathbb{Z}\}$, which means that the variable is not abstractable).

In our temperature control application, we have simple data transfer from c0 to c1, and from c1 to c2. The previously determined intervals of c1 and c2 overlap and a mapping is ambiguous, since the element $(-\infty, 4]$ of the source variable c1 contains values of the interval candidates $(-\infty, 2]$ as well as $[3, \infty]$) of the target variable c2. To enable an unambiguous mapping, we split the interval $(-\infty, 4]$ of c1. The algorithm to partition intervals is given in Section 5.

As resulting intervals, which in this case also represent the equivalence classes, we obtain

$$I_{c1} : (-\infty, 2], [3, 4], [5, \infty)$$

$$I_{c2} : (-\infty, 2], [3, \infty)$$

For our running example, we need two mapping functions: First, the concrete integer values of c0 are passed from the module that reads the sensor data to the first controller and are used in `controller1`. Therefore, we add a function that takes concrete integer values and returns the appropriate element of $I_{c1}$. The second mapping function takes an element of $I_{c1}$ and returns an element of $I_{c2}$.

In the following, we describe how our abstraction can be applied to a SystemC design such that the result is still a valid SystemC design.

## 4.2 Abstract Domains in SystemC

The main goal of our approach is to compute an abstraction of a given SystemC design that can again be represented as a valid SystemC design to allow for further analysis and verification with various verification tools. Therefore, we use only constructs that are provided by the SystemC language to abstract the necessary parts of a given system design. The abstraction process consists of two steps:
1) create abstract types for each abstractable variable and change its definition
2) transform statements that include abstracted variables.

### *Variable Abstraction*

We use enumerations to represent the abstract domains of each abstracted variable in SystemC. Each element in the enumeration stands for a partition (interval) of values. We use the boundary values to name the elements. Then, the type of each abstractable variable is changed to the enumeration representing its abstract domain. Variables that are marked as *not abstractable* remain unchanged.

In our running example, we introduce an enumeration with the elements C1_NINF_2 (where NINF denotes negative infinity), C1_3_4 and C1_5_PINF (where PINF denotes positive infinity). Each element represents one of the intervals that were obtained in the previous step. The enumeration for c2 contains the elements C2_NINF_2 and C2_3_PINF.

### *Transformation of Assignments and Conditions*

If values are passed between abstract variables of the same abstract type, and the value is not changed in the statement, no further transformation is necessary. For all other cases, we introduce mapping functions from one enumeration to another. With our construction of abstract types, we ensure that there exists a mapping function for all abstract domain pairs $(I_s, I_t)$ that are connected with a data dependency. We add these mapping functions, as well as a mapping from the domain of integers into all abstract domains, as global methods in a given SystemC design. For all statements with simple value passing (without arithmetic operations) between different source and target domains, we extend the code by a call of the corresponding mapping function. To handle value passing with arithmetic operations, we introduce a different kind of mapping function that performs the arithmetic operation on the interval. As we already compute the inverse operation when calculating the source intervals (as depicted in Table 2), the mapping function only has to apply the operation *op* on the boundaries of the source element to get the interval boundaries in the target domain:

$$b_l(i_t) = op(b_l(i_s), c) \wedge b_u(i_t) = op(b_l(i_s), c)$$

We also adjust conditional statements that include abstracted variables. Here, we replace all conditions that operate on concrete values with checks on the abstract values. Note that in order to check whether an abstract variable satisfies a given condition, we need to compute all intervals from the corresponding equivalence classes for which the original condition evaluates to true. The resulting abstract condition statement is constructed as a disjunction of these intervals (i.e., if the abstract value is in one of the equivalence classes for which the original condition evaluates to true, the abstract condition also evaluates to true). Table 1 shows which equivalence classes evaluate to true according to the evaluation of different condition types. Note that our transforma-

| Mod. | generated boundaries for x | | | |
|---|---|---|---|---|
| $y =$ | lower | | upper | |
| $x + c$ | $b_l(i_y) - c$ | | $b_u(i_y) - c$ | |
| $x - c$ | $b_l(i_y) + c$ | | $b_u(i_y) + c$ | |
| $c - x$ | $c - b_u(i_y)$ | | $c - b_l(i_y)$ | |
| $x \cdot c$ | $\lceil b_l(i_y)/c \rceil$ | for $c > 0$ | $\lfloor b_u(i_y)/c \rfloor$ | for $c > 0$ |
| | $\lceil b_u(i_y)/c \rceil$ | for $c < 0$ | $\lfloor b_l(i_y)/c \rfloor$ | for $c < 0$ |
| | $-\infty$ | for $c = 0$ | $\infty$ | for $c = 0$ |
| $x/c$ | $b_l(i_y) \cdot c$ | for $c > 0, b_l(i_y) > 0$ | $b_u(i_y) \cdot c + (\lvert c \rvert - 1)$ | for $c > 0, b_u(i_y) \geq 0$ |
| | $b_l(i_y) \cdot c - (\lvert c \rvert - 1)$ | for $c > 0, b_l(i_y) \leq 0$ | $b_u(i_y) \cdot c$ | for $c > 0, b_u(i_y) < 0$ |
| | $b_u(i_y) \cdot c - (\lvert c \rvert - 1)$ | for $c < 0, b_u(i_y) \geq 0$ | $b_l(i_y) \cdot c$ | for $c < 0, b_l(i_y) > 0$ |
| | $b_u(i_y) \cdot c$ | for $c < 0, b_u(i_y) < 0$ | $b_l(i_y) \cdot c + (\lvert c \rvert - 1)$ | for $c < 0, b_l(i_y) \leq 0$ |

Table 2: Interval boundaries for interval modification

tion of conditional statements is unambiguous because the original condition was used to build the abstract domain.

Listing 2 shows the result of our abstraction for our running example of a small temperature control application. The functions *mapINTtoC1* and *mapC1toC2* map a value of one domain into a value of the respective abstract domain. In this example *mapINTtoC1* takes a concrete integer value and returns its respective representation as one of the three possible elements in the abstract domain of $c1$. The condition `(c1 < 5)` in Listing 1, Line 6 in *controller1()*, evaluates to true for all elements of the intervals $(-\infty, 2], [3, 4]$.

```
1  void poll_sensor(){
2    int c0;
3    wait(100, SC_NS);
4    c0 = read_sensor();
5    port_C1.write(
6      mapINTtoC1(c0));
7  }
```

```
1  void controller2(){
2    c2 = port_in_C1.read();
3    open_valve2();
4    if(c2 == C2_NINF_2){
5      wait(40, SC_NS);
6    } else {
7      wait(100, SC_NS);
8    }
9    close_valve2();
10 }
```

```
1  void controller1(){
2    c1 = port_in_C0.read();
3    if(tick < 10){
4      wait(200, SC_NS);
5      tick++;
6    } else {
7      open_valve1();
8      if(c1 == C1_NINF_2
9           || c1 == C1_3_4){
10       wait(50, SC_NS);
11     } else {
12       wait(80, SC_NS);
13     }
14     close_valve1();
15   }
16   port_C2.write(
17     mapC1toC2(c1));
18 }
```

Listing 2: Abstract design

In this section, we have shown how we compute an abstraction of a given SystemC design using data dependency information and the results from a variable analysis. The main idea is to map integer variables to the abstract domain of sets of non-overlapping intervals and to use this to create a valid SystemC design. The abstract design has a significantly smaller semantic state space (if the value range of integers can be safely divided into non-overlapping partitions) and can be further processed by any verification tool for SystemC. Our SystemC-specific interval and data dependency analysis is presented in the following section.

# 5. VARIABLE AND DATA DEPENDENCY ANALYSES FOR SYSTEMC

Our abstraction technique for SystemC is based on a variable analysis that is especially tailored for SystemC. As explained in Section 4, the main idea is to collect a set of non-overlapping intervals for each integer variable, which represents a partition of the concrete values into equivalence classes according to the control flow of the design. This set is used to build an abstract interval type that is used instead of the integer type in the abstract design. The main challenge of our SystemC specific variable analysis is to derive the intervals from the control flow of SystemC processes, which requires to take the specific execution semantics of SystemC into account, including the semantics of ports and channels, the execution semantics of event-triggered process execution, and delta-cycles. In this section, we first describe how we adapt the existing concept of abstract interpretation [8, 9] and explain how we handle special SystemC constructs. Note that most data types used in SystemC designs can be converted to integers.

Besides control flow information, we also have to consider the data flow. If a variable is assigned to another variable, their abstract types have to be compatible. A challenge are dependencies across modules and channels, and cyclic dependencies. To detect and handle these, we collect data flow information and build a (SystemC-specific) data dependency graph. Hence, our analysis consists of two parts, an interprocedural variable analysis that collects a set of possible intervals for each variable from conditional branches, and a data flow analysis that builds a data dependency graph.

## *Variable Analysis*

For our abstraction, we need to know which integer values are equivalent according to the control flow of the given design.Therefore, we define a variable analysis that walks through the call graph of each process of a SystemC design and collects partition boundaries from conditional statements. The analysis is an interprocedural, data flow sensitive forward analysis that builds the minimal split of all previous equivalence classes at merging points. With that, we can traverse the call graph very efficiently. If the analysis encounters a loop, the condition is used to build new partitions once and on the second traversal of the entering block we already have a fixpoint as all collected information remain in the analysis result, which is not statement specific. Thus, we obtain a globally valid partition of each variable.

Our analysis operates on sets of non-overlapping interval partitions of $\mathbb{Z}$ and starts with the most general interval $(-\infty, +\infty)$ for each variable. The formal data structure, the analysis operates on, is a lattice $\mathbb{D}_{\mathbb{I}^*}$, which maps all variables of a design ($\mathbf{Var}_\star$) to sets of non-overlapping intervals ($\mathbb{I}^*$). Let $\mathbb{I} = \{[l, u] \mid l, u \in \mathbb{Z}, l \leq u\} \cup \{(-\infty, u] \mid u \in \mathbb{Z}\} \cup \{[l, \infty) \mid l \in \mathbb{Z}\} \cup \{(-\infty, \infty)\}$ be the set of all integer inter-

vals. Then we define $\mathbb{I}^*$ as the set of elements of the power set of $\mathbb{I}$ that do not overlap:

$$\mathbb{I}^* = \{S \in \mathcal{P}(\mathbb{I}) \mid \forall I_1, I_2 \in S.\ I_1 \cap I_2 = \emptyset \wedge \left(\bigcup_{I \in S} I\right) = \mathbb{Z}\}$$

Now, we can construct the lattice

$$\mathbb{D}_{\mathbb{I}^*} = ((\mathbf{Var}_\star \to \mathbb{I}^*), \sqsubseteq), \text{ with}$$

$$f \sqsubseteq g \text{ iff } \forall v \in \mathbf{Var}_\star.\ f(v) \sqsubseteq_{\mathbb{I}^*} g(v), \text{ where}$$

$$S_1 \sqsubseteq_{\mathbb{I}^*} S_2 \text{ iff } \forall I_2 \in S_2.\ \exists X \subseteq S_1.\ \left(\bigcup_{I \in X} I\right) = I_2.$$

$S_1 \sqsubseteq_{\mathbb{I}^*} S_2$ means that all intervals in $S_2$ can be constructed by a union of intervals from $S_1$. In other words $S_2$ may join together partitions of $S_1$. During the analysis, we gain information at branching points and split our collected intervals accordingly. To ensure a minimal split, we define the greatest lower bound $S_1 \sqcap_{\mathbb{I}^*} S_2$ of two sets $S_1$ and $S_2$ so that for each interval $I_2$ from $S_2$, we take the smallest set $X$ of continuous intervals from $S_1$ whose union includes $I_2$ and collect the intersections of all $I \in X$ and $I_2$. To illustrate this, consider the following example: $\{(-\infty, 5], [6, 7], [8, \infty)]\} \sqcap_{\mathbb{I}^*} \{(-\infty, 5], [6, 9], [10, \infty)\} = \{(-\infty, 5], [6, 7], [8, 9], [10, \infty)\}$. The result is again a set of non-overlapping intervals that contains the minimal split of all previous intervals. This can be formally defined as

$$S_1 \sqcap_{\mathbb{I}^*} S_2 = \{Y \mid Y = I \cap I_2, \text{ with } I_2 \in S_2 \wedge I \in X,$$

$$\text{where } X \subseteq S_1, \text{ such that } I_2 \subseteq \left(\bigcup_{I \in X} I\right) \wedge\ \nexists X' \subseteq S_1,$$

$$\text{with } X' \subset X \wedge\ I_2 \subseteq \left(\bigcup_{I \in X'} I\right)\}.$$

We implemented this with a method $mergeAndSplit(S_1, S_2)$ that first, initializes a set $S_{Res}$ to $S_1 \cup S_2$. Then, for two overlapping intervals $I_1, I_2 \in S_{Res}$, we calculate $I_3 = I_1 \cap I_2$, $I_1' = I_1 \backslash I_3$, $I_2' = I_2 \backslash I_3$, and replace $I_1$ and $I_2$ by $I_3$, $I_1'$, and $I_2'$ in $S_{Res}$. This is repeated until all intervals in $S_{Res}$ are non-overlapping. In contrast to classical interval analyses, we are interested in intervals that form possible partitions of variable ranges during execution. Therefore, we only use control flow information gained from conditional branches. To collect interval candidates, we define a transformation function $f_{trans}$ for each type of conditional statement or assignment, for array accesses and for the SystemC specific methods `wait` and `notify`, as shown in Table 3. For each block, we compute the information at the exit of the block by $I_{out} = f_{trans}(I_{in})$. We handle assignments and method calls in a subsequent step on the data flow graph to ensure compatibility between types of dependent variables in the sense that we can easily map from the type of the source to the target variable. Note that there are some statements where the concrete value of a variable is needed. In this case, we mark the variable as *not abstractable* by using the least element of our lattice as analysis result: $\perp = \{[l, l] \mid l \in \mathbb{Z}\}$. This is the case for arithmetic operations where we do not abstract the source variables if the operation contains more than one source variable, as we cannot ensure an unambiguous abstraction. Also, for array accesses, the array variable

```
1  analyzeMethod(Method m, ClassInstance ci)
2    i_m = new methodInformation(m);
3    for each stmt ∈ m do
4      if isBranchingPoint(stmt) then
5        ... // update i_m for each variable according to Table 3
6      if isMemberMethodCall(stmt,ci) then
7        m_m := getMethod(stmt,ci)
8        i_m ← i_m ⊓_I* analyzeMethod(m_m,ci)
9      if isMethodCalledOverPort(stmt,ci) then
10       ch = getChannel(stmt)
11       m_c = getChannelMethod(stmt,ch)
12       i_m ← i_m ⊓_I* analyzeMethod(m_c,ci)
13     if isCalledOverSocket(stmt,ci) then
14       s = getSocket(stmt)
15       m_s = getSocketMethod(stmt,s)
16       i_m ← i_m ⊓_I* analyzeMethod(m_s,ci)
17     ... // update i_m for other statements according
18     ... // to f_trans (e.g. wait, notify)
19   return i_m
```

Listing 3: Call Graph Analysis

can be abstracted, but the index variable not, as it is important to know which array position is accessed. Variables that are used inside a wait statement or event notification are also not abstracted, as we do not want to abstract from timing behavior. These statements are detected with a simple analysis on the statement structure.

Our variable analysis is applied to all methods that influence the system behavior. These are all methods that are called by a process during system execution. To get these, we start with all class instances, determine all processes, and traverse the call graph of each process. To enrich our analysis with interprocedural data flow information, we keep track of the data flow in case of method calls or SystemC specific port or socket accesses. Method calls are handled as assignments of values to the parameters. The method body is analyzed and local analysis results are stored for each method (*methodInformation(m)*), then the analysis returns back to the caller. Ports and sockets are communication anchors in SystemC that are used to connect modules. To handle this, we resolve the binding of ports and sockets and analyze the accessed element similarly to method calls. This is shown in Listing 3. Note that by analyzing each process in a given SystemC design along its complete call graph (including resolved communication via ports and sockets), and by not abstracting variables that are used for `wait` and `notify`, we can cope with the event-triggered execution semantics of SystemC designs. As we disallow any abstraction of the behavior at `wait` statements, we are also able to cope with delta-cycles and their zero-delay semantics.

### *Data Dependency Analysis*

Our variable analysis for SystemC focuses on conditional branches in the control flow. This is very efficient and enables us to construct equivalence classes without considering each statement in a block. However, to make sure that our analysis is safe, we additionally consider data dependencies for our abstraction. To this end, we collect all data dependencies between variables inside a module and across module borders. To achieve this, we construct a set of data flow objects $df = (V_s, v_t, s)$, where $V_s$ is the set of source variables, $v_t$ is the target variable and $s$ is the statement that con-

| basic block | | transfer function $f_{trans}$ |
|---|---|---|
| condition | $x < c$ | $I_{out}(x) = \{(-\infty, c-1], [c, \infty)\} \sqcap_{\mathbb{I}^*} I_{in}(x)$ |
| | $x > c$ | $I_{out}(x) = \{(-\infty, c], [c+1, \infty)\} \sqcap_{\mathbb{I}^*} I_{in}(x)$ |
| | $x \leq c$ | $I_{out}(x) = \{(-\infty, c], [c+1, \infty)\} \sqcap_{\mathbb{I}^*} I_{in}(x)$ |
| | $x \geq c$ | $I_{out}(x) = \{(-\infty, c-1], [c, \infty)\} \sqcap_{\mathbb{I}^*} I_{in}(x)$ |
| | $x = c$ | $I_{out}(x) = \{(-\infty, c-1], [c], [c+1, +\infty)\} \sqcap_{\mathbb{I}^*} I_{in}(x)$ |
| | $x \neq c$ | $I_{out}(x) = \{(-\infty, c-1], [c], [c+1, +\infty)\} \sqcap_{\mathbb{I}^*} I_{in}(x)$ |
| $x := y \otimes z$ | | $I_{out}(y) = I_{out}(z) = \bot$, $x, y, z \in \mathbf{Var}_\star$, arithm. operation $\otimes$ |
| array access A[x]:= y | | $I_{out}(x) = \bot$ |
| wait(x), e.notify(x) | | $I_{out}(x) = \bot$ |

Table 3: Abstract Transfer Function (only changes to $I_{out}$ are listed)

tains the data flow. If we analyze a statement that contains data flow (assignments, comparisons, method calls, port and socket accesses) during our variable analysis, we create a corresponding data flow object. At the end of our analysis, all data flow objects are merged into a data dependency graph with the variables as nodes and the statements as labels on the edges. This graph is used in our data abstraction.

# 6. SLICING FOR SYSTEMC DESIGNS

First introduced by [28], slicing is a technique to remove program parts that do not influence the program behavior in regard to a property. This property is called a slicing criterion. The resulting subsystem is behaviorally equivalent to the original system with regard to a property of interest. Therefore, the verification results on a slice of the system also hold for the overall system.

With the results of our analysis, described in Section 5, we can easily extend our approach by slicing. In the analysis step, the data dependencies of all variables are determined. This information can be used to generate a system slice where all variables that are not relevant for a given slicing criterion are removed, as well as all statements that only influence irrelevant variables and system parts. We define our slicing criterion to be a set V of variables that are relevant for some property. Variables in V can be global variables, local variables, class member variables, and return values of function calls. Additionally all variables that influence the value of such a relevant variable are added to this set. These influences can be determined by a data-dependency analysis as explained in Section 5. Subsequently, system parts that do not influence relevant variables are removed.

## *Relevance of statements*

We infer the relevance of all statements from the included variables. Statements that we mark as important in this step are included in the system slice and all statements that are marked as unimportant are not included.

An assignment statement consists of a left hand side, which contains the target variable, and a right hand side. To determine the relevance of an assignment statement, we consider the relevance of the target variable. Assignments to important variables $v \in V$ are marked as important and therefore are preserved in the slice. Assignments to variables $v \notin V$ are removed. However, if the right hand side changes the value of a relevant variable, then we include this right hand side. This can include functions calls and accesses to ports.

A function call can be removed from a slice if the called function does not contain a relevant statement. To evaluate the relations across function calls, we again traverse the SystemC specific call graph of each SystemC process as described in Section 5.

For conditional branches, each branch is analyzed separately. If a branch contains no relevant statement, we get an empty branch. If-statements where all branches are empty are removed from the slice. Variables that are used in the condition for a branch that includes a relevant statement are also marked as relevant.

An important aspect of SystemC is the timing behavior of processes. All `wait` statements that represent timed sensitivity are considered important and are not removed in the slice. This ensures that function calls and branches that change the system behavior by suspending a process using the `wait` statement are considered important.

## *Combining Slicing and Abstraction*

With our slicing approach, we reduce the state space of a given system. However, we can further reduce the state space by combining our data abstraction and slicing. Since slicing can remove some conditional branches and statements that are used in the generation of equivalence classes, the resulting code contains less statements that must be considered during the creation of the necessary abstract types. Each statement could only add additional information to the necessary equivalence classes and not remove it. Therefore, the amount of elements of the abstract domain are reduced. Together, slicing and abstraction significantly reduce the state space of a given system while preserving all behavior that is relevant for a given property of interest.

# 7. EVALUATION

To demonstrate the practical applicability of our approach, we have implemented our slicing and abstraction technique for SystemC in Java. A given SystemC design is first translated into the SystemC intermediate representation SysCIR [22]. Then, our analysis and abstraction engine generates an abstract design, which is again represented as a SysCIR model. The resulting model can be transformed into the input languages of various verification tools, e. g., UPPAAL [1], BLAST [2], or UCLID [26].

To show the effect of our abstraction by means of reduced verification time, we have used the **S**ystemC to **T**imed **A**utomata **T**ransformation **E**ngine (STATE) [22] to transform both concrete and abstracted SystemC designs into UPPAAL timed automata models. We have then formally verified the resulting UPPAAL models using the UPPAAL model checker [1] and compared the verification times.

In our experiments, we use two case studies from the SystemC reference implementation: a producer-consumer with a *first-in first-out* (FIFO) buffer, and a multicast he-

| Property | concrete | abstract | abstract slice |
|---|---|---|---|
| **no deadlocks** | 18:11 | 0:16 | 0:02 |
| **consumer receives data** | $< 1$ | $< 1$ | n.a. |
| **no buffer overflow** | 11:56 | 0:10 | 0:01 |
| **received data always in expected range** | 11:56 | 0:10 | n.a. |

Table 4: Verification times for the producer-consumer example in [hh]:mm:ss

| Property | 1x1 | | 2x2 | |
|---|---|---|---|---|
| | concrete | abstract | concrete | abstract |
| **no deadlocks** | 0:47 | 0:12 | $> 24$ h | 24:12 |
| **data in receiver in correct intervals** | 0:30 | 0:07 | $> 24$ h | 17:12 |
| **receiver only gets correct packets** | 0:30 | 0:08 | $> 24$ h | 17:52 |
| **switch puts packets in correct output** | 0:30 | 0:08 | $> 24$ h | 16:57 |
| **packets arrive in time** | 2:10 | 0:13 | $> 24$ h | 2:03:49 |

Table 5: Verification times for the packet switch in [hh]:mm:ss

lix packet switch in two variants, with one sender and one receiver (1x1) and with two of each (2x2). The size of the producer-consumer example is approximately 130 lines of code (LOC) and it consists of two modules, two processes and one channel. The size of the packet switch example is approximately 400 LOC. It contains 5 modules, 5 processes and 6 channels in case of one sender and one receiver, and 7 processes and 10 channels in case of two senders and two receivers. In both case studies, we have added branches in the consumer resp. receiver that depend on the data that is send. We send random 4 bit data values via the FIFO resp. the packet switch to enable verification for various input scenarios. For the producer-consumer, we have verified the following properties: 1) No deadlock. 2) The consumer eventually receives data. 3) No buffer overflow within the FIFO. 4) All received data is in the expected range.

For the packet switch, we have verified the following properties: 1) No deadlock. 2) All received data is in the expected range. 3) Each receiver only gets packets addressed to it. 4) The switch puts packets into the correct output queue. 5) Packets arrive within a given time limit.

Note that for this last property, it was necessary to extend the automatically generated UPPAAL models with observer automata that keep track of individual packets, in order to be able to express the corresponding property in the CTL subset supported by UPPAAL. For (2x2), we have split the verification in four runs (one for each S-R combination) and built the sum of the resulting verification times. Note that the computational effort for our analysis and automatic abstraction was negligible in all cases (less than a second). All experiments were performed on a 3.4 GHz quad-core CPU with 32 GB RAM running Ubuntu 14.4. Table 4 shows the verification times for the producer-consumer. The computational effort for the verification is reduced by a factor of 70. Still, as our analysis performs a safe overapproximation, the results are also valid for the original design. For properties (1) and (3) a slice of the overall system is sufficient and the verification effort could be reduced even further by only verifying the automatically computed slice. The slicing criterion contains all variables of the FIFO buffer and it was chosen with the intention to verify only properties for the FIFO buffer. To verify properties (2) and (4), which consider the consumer, another slicing criterion would be necessary.

In Table 5 the results for the packet switch are shown. Again, the computational effort is significantly reduced for the verification of all properties. The concrete version of the 2x2 packet switch could not be verified within a given time limit of 24 hours. In the abstract design, properties (1) - (4) could be verified in less than half an hour and the verification of property (5) took slightly more than two hours. The higher computational effort for the verification of the timing property is due to the observer automata, which are necessary to track the timing of individual packets. They add additional clocks to track the timing of individual packets and thus significantly increase the semantic state space. The computation of a slice was not possible for the packet switch, as all components influence the verified properties.

Overall, our results show that our abstraction technique successfully reduces the semantic state space, and, consequently, the verification effort.

# 8. CONCLUSION

In this paper, we have presented an approach for fully-automatic abstraction and slicing of hardware/software co-designs modeled in SystemC. The core of our abstraction technique is a variable and data-dependency analysis, which is tailored to SystemC designs and their specific semantics. We use the analysis results to construct equivalence classes for data variables that are used for abstract interpretation of the design. With this, the semantic state space of SystemC designs can be significantly reduced. The reduction is safe, as it preserves the behavior with respect to a given property, and precise, as we do not abstract in cases where we cannot ensure an unambiguous abstraction. The resulting abstracted design is a valid SystemC design that can be further processed with any verification technique.

We have demonstrated the effectiveness of our approach with two case studies, both taken from the SystemC reference implementation and representing typical embedded controller designs where large data ranges occur, but do not influence the control logic. Our experiments show that the verification effort can be crucially reduced using our automatic abstraction and slicing techniques.

In future work, we will consider further case studies and investigate the impact of our data abstraction on other model checkers and verification approaches. Furthermore, we plan to embed our automatic abstraction technique into our framework for the modular verification of integrated HW/SW systems [21]. In particular, we plan to use our abstraction technique to divide a given complex model into smaller submodels which can then be delegated to various verification

engines, ranging from model checking based approaches as with UPPAAL to satisfiability modulo theory solvers like UCLID. Furthermore, we will extend the technique itself, e. g., to support more complex data types, and to derive more detailed dependencies from complex expressions.

# 9. REFERENCES


[1] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*, LNCS 3185, pages 200–236. Springer, 2004.

[2] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: Applications to software engineering. *Intl. Journal on Software Tools and Technology Transfer*, 2007.

[3] C.-N. Chou, C.-K. Chu, and C.-Y. R. Huang. Conquering the scheduling alternative explosion problem of systemc symbolic simulation. In *Intl. Conference on Computer-Aided Design (ICCAD)*, pages 685–690. IEEE, 2013.

[4] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. R. Huang. Symbolic model checking on SystemC designs. In *Design Automation Conference (DAC)*, pages 327–333. ACM, 2012.

[5] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. Kratos - A Software Model Checker for SystemC. In *Computer-Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 310–316. Springer, 2011.

[6] A. Cimatti, I. Narasamdya, and M. Roveri. Software model checking systemc. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 32(5):774–787, 2013.

[7] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[8] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Intl. Symposium on Programming*, pages 106–130, 1976.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of programming languages*, pages 238–252. ACM, 1977.

[10] M. Elshuber, S. Kandl, P. P. Puschner, C. Choppy, and J. Sun. Improving System-Level Verification of SystemC Models with SPIN. In *French Singaporean Workshop on Formal Methods and Applications (FSFMA)*, pages 74–79. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013.

[11] H. Garavel, C. Helmstetter, O. Ponsini, and W. Serwe. Verification of an industrial SystemC/TLM model using LOTOS and CADP. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 46 –55. IEEE Computer Society, 2009.

[12] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[13] D. Große, H. M. Le, and R. Drechsler. Proving Transaction and System-level Properties of Untimed SystemC TLM Designs. In *Formel Methods and Models for Codesign (MEMOCODE)*, pages 113 – 122. IEEE Computer Society, 2010.

[14] A. Habibi, H. Moinudeen, and S. Tahar. Generating Finite State Machines from SystemC. In *Design, Automation and Test in Europe*, pages 76–81. IEEE, 2006.

[15] A. Habibi and S. Tahar. An Approach for the Verification of SystemC Designs Using AsmL. In *Automated Technology for Verification and Analysis*, LNCS 3707, pages 69–83. Springer, 2005.

[16] N. Harrath, B. Monsuez, and K. Barkaoui. Verifying SystemC with predicate abstraction: A component based approach. In *Intl. Conference on Information Reuse and Integration*, pages 536–545. IEEE, 2013.

[17] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, Nov 1998.

[18] C. Helmstetter. TLM.open: a SystemC/TLM Frontend for the CADP Verification Toolbox. *Leibniz Transactions on Embedded Systems*, 1(1), 2014.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages (POPL)*, pages 58–70. ACM, 2002.

[20] P. Herber, J. Fellmuth, and S. Glesner. Model Checking SystemC Designs Using Timed Automata. In *Intl. Conference on Hardware/Software Codesign and Integrated System Synthesis (CODES+ISSS)*, pages 131–136. ACM press, 2008.

[21] P. Herber and B. Hünnemeyer. Formal Verification of SystemC Designs using the BLAST Software Model Checker. In *Intl. Workshop on Model-Based Architecting and Construction of Embedded Systems*, volume 1250, pages 44–53. CEUR, 2014.

[22] P. Herber, M. Pockrandt, and S. Glesner. STATE – a SystemC to Timed Automata Transformation Engine. In *Intl. Conference on Embedded Software and Systems (ICESS)*. IEEE Computer Society (to appear), 2015.

[23] IEEE Standards Association. IEEE Std. 1666–2011, Open SystemC Language Reference Manual. IEEE Press, 2011.

[24] L. Jaß and P. Herber. Bit-Precise Formal Verification for SystemC using Satisfiability Modulo Theories Solving. In *Intl. Embedded Systems Symposium (IESS)*. Springer, 2015.

[25] D. Karlsson, P. Eles, and Z. Peng. Formal verification of SystemC Designs using a Petri-Net based Representation. In *Design Automation and Test in Europe (DATE)*, pages 1228–1233. IEEE Press, 2006.

[26] S. K. Lahiri and S. A. Seshia. The UCLID Decision Procedure. In *Computer-Aided Verification (CAV)*, LNCS 3114, pages 475–478, 2004.

[27] A. B. R. Brummayer. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *LNCS*. Springer, 2009.

[28] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 4(SE-10):352–357, 1984.