Modular Design Through Component Abstraction

David Berner, Jean-Pierre Talpin, Paul Le Guernic
INRIA project ESPRESSO, IRISA, France
FirstName.LastName@irisa.fr

Sandeep Kumar Shukla
Virgina Tech, USA
shukla@vt.edu

ABSTRACT

Growing design sizes and shrinking time to market windows can only be met with drastically increased productivity. One way to obtain this is a smart reuse of intellectual property. This paper presents a methodology for modular design with the help of component abstraction. It describes how imperative components can be transformed into a formal, synchronous description to provide behavioral types to the components. The synchronous composition of these abstracted components helps discover errors in the component composition. The presented methodology is illustrated by the detailed case study of a Finite Impulse Response filter. We transform initial SYSTEMC modules into an intermediate static single assignment representation which is used as a basis from which corresponding behavioral types are built.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques— *Modules and Interfaces*; D.2.10 [**Software Engineering**]: Design— *Methodologies*

General Terms

Design, Verification

Keywords

Behavioral Interfaces, Component Based Design, Synchronous Languages

1. INTRODUCTION

The ever increasing complexity of embedded systems coupled with aggravated time constraints makes it difficult for design houses to keep both costs and quality within bounds. Component design and IP reuse - though identified a long time ago as possible solutions for these challenges - have been missing a broad adoption. This is mainly due to inherent problems of component reuse. Components often are created for a special purpose, and even if they are sufficiently general, they behave differently in different environments. A lack of pertinent interface descriptions impedes an unambiguous reuse. In many cases subtleties that only the component's

original developers are familiar with are indispensable for efficient integration and testing. In this paper we address this problem by providing components with interfaces that not only comprise the components' input / output signals and their types but also causal and synchrony relations between signals. This enables more exhaustive checks for the compatibility of components. The description of interface signal dependencies can also be seen as a behavioral interface. Figure 1 shows the connection of two components. A normal type description only checks information about the data type of the common signals x, y, and z. As long as these types match, the type checker will approve the composition. If A produces x and y at the same rate but B consumes two values of x for each value of y, this would go undetected. Also there is no means to discover a combinational loop over the signal z. A behavioral type description that holds information about the synchronization of signals is able to detect these errors. It exhibits part of the internal functioning of the component in a data-flow synchronous formalism that makes it possible to formally reason about these interfaces. The synchronous composition of several of these behavioral interfaces automatically reveals intricate problems in the composition of the components just as a simple type checker would find a signal data type mismatch. Many of these errors would otherwise remain undetected and could cause great costs and delays later in the design process. The more behavior such an interface captures, the more errors can be found. A system built from components whose compositions all have been checked with the help of a behavioral type, can therefore be expected to function more reliably and have a much higher overall design quality than compositions with simple data type checks [6] - even after thorough testing.
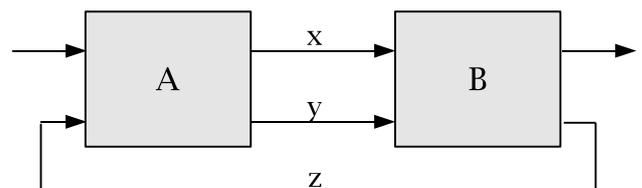

Figure 1: Two connected components

Previous Work. The proposed work and methodology arises from previous work on embedded systems design and verification in the Polychrony workbench [12], a tool-set based on a multi-clocked synchronous model of computation [13] and implemented by the data-flow notation SIGNAL [3] and its related model-checking tool SIGALI [14]. Using the Polychrony workbench for component-based design has been the subject of recent studies [2], yet not within the methodological framework of a behavioral type system for existing structural components, which we consider here.

In [19], the use of Polychrony to describe services of the real-time Java virtual machine is demonstrated and applied to rethreading multi-task real-time Java programs by using global program optimization algorithms provided in the workbench. In [20], the application of this technique to system-level design is developed by studying its application to checking behavioral conformance between embedded systems described in [9] and at heterogeneous levels of abstraction. In [18], a generic translation scheme of SystemC programs to the Polychrony workbench is presented by considering a static single assignment intermediate representation due to the GCC project [7].

Contribution. Our approach consists in putting the polychronous model of computation [13] to work in the context of system-level design languages such as SYSTEMC [16, 10]. We provide imperative system components with formal behavioral interfaces in different levels of detail. These interfaces can be used to prove formal properties of the components as well as on the composition of components. We propose a technique to generically obtain formal abstractions of SYSTEMC components and use the power of the synchronous paradigm to formally verify the correctness of component compositions. In particular we present a fully featured case study detailing all steps from the analysis and translation of SYSTEMC components to building behavioral types for these components and synchronously composing them to detect possible flaws. As a side note we show how these interfaces can be used to verify formal properties of the components and their composition.

Plan. We start with Section 2, a rationale on our behavioral type inference technique. In Section 3 we give an introduction to the polychronous model of computation and its supporting data-flow notation SIGNAL. Section 4 describes the general flow of our approach and the methodology as well as the principal steps and tools it involves. In Section 5 we show how the application of this flow with a case study on a Finite Impulse Response (FIR) filter. After this related work is discussed in Section 6 and then the paper concludes with Section 7.

2. RATIONALE

To allow for an easy grasp on the proposed behavioral type inference technique, we start with an outline of the analysis of imperative programs in Figure 2, and the construction of behavioral types, Figure 3. The left hand side of Figure 2 depicts an imperative code fragment consisting of an iterative program that counts the number of bits set to one in the variable idata. While idata is not equal to zero, it adds its right-most bit to an output count variable ocount and shifts it right in order to process the next bit.

Static Single Assignment. The static single-assignment (SSA) representation of this program is shown on the right hand side of Figure 2 and consists of three blocks. The block labeled L1 waits for the event start before initializing the local state variable idata to the value of the input signal data and the variable icount to 0. Label L2 corresponds to a loop that shifts idata right and adds its right-most bit to icount until termination (condition T2). Finally, in the block L3, icount is sent to the signal ocount. Before going back to L1 the done event is emitted.

All variables (idata and ocount) are read and written once per cycle. Label L2 is the entry point of the SSA block associated with the while loop. The first instruction loads the input variable idata into the register T1. The second instruction stores the result of its comparison with 0 in the register T0. If T0 is true, control is

```
wait (start);          | L1:wait (start);      | L2:T1 = idata;
idata = data;          |    idata = data;      |    T0 = T1 = 0;
icount = 0;            |    icount = 0;        |    if T0 then goto L3;
while (idata != 0) {   |    goto L2;           |    T2 = ocount;
   ocount = ocount     |                       |    T3 = T1 & 1;
           + (idata & 1); | L3:notify (done);  |    ocount = T2 + T3;
   idata = idata >> 1; |    ocount = icount;   |    idata = T1 >> 1;
}                      |    goto L1;           |    goto L2;
ocount = icount;
notify (done);
```

Figure 2: From C-like programs to static single assignment

passed to block L3. Otherwise, the next instruction is executed: the variable ocount is loaded into T2, the last bit of T1 is loaded into T3, the sum of T2 and T3 is assigned to ocount and the right-shift of T1 is assigned to idata. Finally, the block terminates with an unconditional branch back to label L2.

Behavioral Type Inference. Let us zoom on the block L2 in the example of Figure 3. The behavioral type of the block L2 on the right hand side consists of the simultaneous composition of logical propositions that form the behavioral type of the block. Each proposition is associated with one instruction: it is an equation that specifies its *invariants*. In particular, it tells when the instruction is executed, what it computes, when it passes control to the next statement, when it branches to another block.

The instruction T1 = idata on the left hand side is associated with the partial equation T1 ::= idata$1 when L2$1 on the right. It means that, if the label L2 is being executed, then T1 is equal to idata$1 (the operator $1 refers to the value of the variable during the previous cycle). Next, consider instruction if T2 goto L3. It corresponds to the partial equation L3 ::= true when T2. This means that control is passed to L3 when T2 is true. Instructions that follow are conditioned by the negative not T2 which means: "in the block L2 and not in its branch going to L3".

```
L2: T1 = idata;      |  T1 ::= idata$1 when L2$1
    T2 = T1 == 0;    |  T2 ::= T1 = 0 when L2$1
    if T2 goto L3;   |  L3 ::= true when T2
    T3 = icount;     |  T3 ::= icount$1 when not T2
```

Figure 3: From static single assignment to data-flow equations

Figure 4 depicts the translation of operations in block L2. The assignment of icount to the local variable T3 is translated by the partial equation T3 ::= icount$1 when not T2 which assigns the previous value of icount to the temporary variable T3 at the clock not T2 (i.e. when T1 is not 0, see Figure 3).

```
T3 = icount;         |  T3     ::= icount$1 when not T2
T4 = T1 & 1;         |  T4     ::= T1 & 1 when not T2
icount = T3 + T4;    |  icount ::= T3 + T4 when not T2
idata = T1 >> 1;     |  idata  ::= T1 >> 1 when not T2
```

Figure 4: Translation of primitive operations

Types for Predefined Protocols. Consider the wait-notify protocol at blocks L1 and L3, Figure 5. The wait instruction receives control at the clock x_{L1}. If the value of start changes (i.e. when not T0) then icount and idata are initialized and control is passed to the block L2. Otherwise, at the clock when T0, a transition back to L1 is scheduled.

```
L1: wait (start);  │   T1 ::= start=start$1 when L1$1
    ...            │ | L1 ::= true when T1
                   │     ...
L3: notify (done); ││  done ::= not done$1 when L3$1
    ...            │     ...
```

Figure 5: Modeling communication protocols

Completion. All entry clocks x_L are simultaneously present when a block is executed. Each signal x_L holds the value 1 iff the block L is active during a transition currently being executed. Otherwise, x_L is set to `L ::= defaultvalue false`. The same holds for local variables T with `T ::= defaultvalue T$1`. The SIGNAL compiler guarantees the completion of the next-state logic by aggregating partial equations.

```
  L1 := true when (T1 default L3$1) default false
| L2 :=          true when (L1b$1 default not T3)
      default false
| L3 := true when T3 default false
```

Additional Remarks. The proposed inference technique is modular (block-wise), conceptually simple (one equation per instruction) and language independent (SSA is the input formalism). The host formalism SIGNAL supports a scalable notion and a flexible degree of abstraction. Notice that the structure of the original program is represented by program labels L, which play an essential role during modeling as they represent clocks, i.e. the data-structure used by the POLYCHRONY workbench to represent the control flow of programs. This information is propagated during modeling, verification and transformation. As a result, traceability is easily provided by this information to relate an error to its original block, in addition to the name of all variables it implies.

3. A MODEL OF POLYCHRONY

We start with a brief overview of the polychronous model of computation [13]. We consider a partially-ordered set $(\mathcal{T}, \leq, 0)$ of tags. A tag $t \in \mathcal{T}$ denotes a symbolic period in time. The relation $\leq$ denotes a partial order. Its minimum is noted 0. We note $C \in \mathcal{C}$ a *chain* of tags (a totally ordered subset of $\mathcal{T}$). We define:
- an *event* $e \in \mathcal{E} = \mathcal{T} \times \mathcal{V}$ by the pair of a value and a tag,
- a *signal* $s \in \mathcal{S} = C \to \mathcal{V}$ by a function from a *chain* to values,
- a *behavior* $b \in \mathcal{B} = \mathcal{X} \rightharpoonup \mathcal{S}$ by a map from names to signals,
- a *process* $p \in \mathcal{P}$ by a set of behaviors of the same domain.

Figure 6 depicts a behavior b over three signals named x, y, and z in the domain of polychrony. Two frames depict timing domains formalized by chains of tags. Signals x and y belong to the same timing domain: x is a down-sampling of y. Its events are synchronous to odd occurrences of events along y and share the same tags, e.g. t_1. Even tags of y, e.g. t_2, are ordered along its chain, e.g. $t_1 < t_2$, but absent from x. Signal z belongs to a different timing domain. Its tags, e.g. t_3 are not ordered with respect to the chain of y, e.g. $t_1 \not\leq t_3$ and $t_3 \not\leq t_1$.
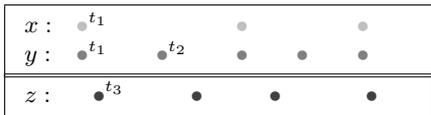

Figure 6: A multi-clocked behavior

In the remainder, we write $\mathrm{tags}(s)$ for the tags of a signal s, $b|_X$ for the projection of a behavior b on $X \subset \mathcal{X}$ and $b/X = b|_{\mathrm{vars}(b) \setminus X}$ for its complementary, $\mathrm{vars}(b)$ and $\mathrm{vars}(p)$ for the domains of b and p. We write $\mathcal{B}|_X$ for the set of all behaviors defined on the set of variables X. Synchronous composition is noted $p \,|\, q$ and is defined by the union of all behaviors b (from p) and c (from q) that are synchronous: all signals they share, i.e. in $I = \mathrm{vars}(p) \cap \mathrm{vars}(q)$, are equal.

$$p \,|\, q = \{ b \cup c \mid (b, c) \in p \times q, I = \mathrm{vars}(p) \cap \mathrm{vars}(q), b|_I = c|_I \}$$
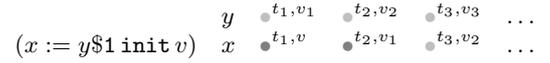
A Polychronous Programming Model. In the POLYCHRONY workbench, the polychronous model of computation is implemented by the multi-clocked synchronous data-flow notation SIGNAL [3]. It serves as the specification formalism used for the case study of the present article. In SIGNAL, a process P consists of the composition of simultaneous equations $x := f(y, z)$ or $x := y\,f\,z$ over input signals y, z and output signals x. A signal $x \in \mathcal{X}$ is a possibly infinite flow of values $v \in \mathcal{V}$ sampled at a clock noted $\hat{\ }x$.

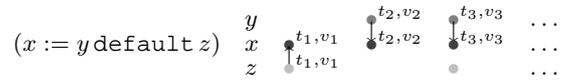$$P, Q ::= x := y\,f\,z \mid P/x \mid P\,|\,Q \qquad (\text{SIGNAL process})$$

In the polychronous model of computation, Section 3, the denotation of a clock $\hat{\ }x$ is the domain of the signal associated to x: a chain of tags. We note $\llbracket P \rrbracket$ for the denotation of a process P. The synchronous composition of processes $P\,|\,Q$ consists of the simultaneous solution of the equations in P and in Q. The process P/x restricts the signal x to the lexical scope of P.

$$\llbracket P\,|\,Q \rrbracket = \llbracket P \rrbracket \,|\, \llbracket Q \rrbracket \text{ and } \llbracket P/x \rrbracket = \llbracket P \rrbracket / x$$
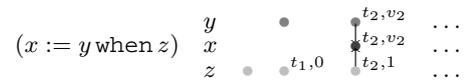
An equation $x := y\,f\,z$ denotes a relation between the input signals y and z and an output signal x by a combinator f. An equation is usually a ternary and infixed relation noted $x := y\,f\,z$ but it can in general be an $m + n$-ary relation noted $(x_1, \ldots x_m) := f(y_1, \ldots y_n)$. SIGNAL requires three primitive combinators to perform delay $x := y\$1\,\mathtt{init}\,v$, sampling $x := y\,\mathtt{when}\,z$ and merge $x = y\,\mathtt{default}\,z$. The equation $x := y\$1\,\mathtt{init}\,v$ initially defines the signal x by the value v and then by the previous value of the signal y. The signal y and its delayed copy $x := y\$1\,\mathtt{init}\,v$ are synchronous: they share the same set of tags $t_1, t_2, \ldots$. Initially, at t_1, the signal x takes the declared value v and then, at tag t_n, the value of y at tag t_{n-1}.

$$(x := y\$1\,\mathtt{init}\,v) \quad
\begin{array}{llll}
y & {}^{t_1, v_1} & {}^{t_2, v_2} & {}^{t_3, v_3} \quad \ldots \\
x & {}^{t_1, v} & {}^{t_2, v_1} & {}^{t_3, v_2} \quad \ldots
\end{array}$$

The equation $x := y\,\mathtt{default}\,z$ defines x by y when y is present and by z otherwise. If y is absent and z present with v_1 at t_1 then x holds (t_1, v_1). If y is present (at t_2 or t_3) then x holds its value whether z is present (at t_2) or not (at t_3).

$$(x := y\,\mathtt{default}\,z) \quad
\begin{array}{llll}
y & & {}^{t_2, v_2} & {}^{t_3, v_3} \quad \ldots \\
x & {}^{t_1, v_1} & {}^{t_2, v_2} & {}^{t_3, v_3} \quad \ldots \\
z & {}^{t_1, v_1} & & \quad \ldots
\end{array}$$

The equation $x := y\,\mathtt{when}\,z$ defines x by y when z is true (and both y and z are present); x is present with the value v_2 at t_2 only if y is present with v_2 at t_2 and if z is present at t_2 with the value true. When this is the case, one needs to schedule the calculation of y and z before x, as depicted by $y_{t_2} \to x_{t_2} \leftarrow z_{t_2}$.

$$(x := y\,\mathtt{when}\,z) \quad
\begin{array}{llll}
y & & {}^{t_2, v_2} & \ldots \\
x & & {}^{t_2, v_2} & \ldots \\
z & {}^{t_1, 0} & {}^{t_2, 1} & \ldots
\end{array}$$

Relating Polychronous Signals with Clocks. In SIGNAL, the presence of a value along a signal x is the proposition noted $\hat{}x$ that is true when x is present and that is absent otherwise. The syntax of clock expressions e and clock relations E is a particular subset of SIGNAL that is defined by the induction grammar e. The clock expression $\hat{}x$ can be defined by the Boolean operation $x = x$ (i.e. $y := \hat{}x \stackrel{\text{def}}{=} y := (x = x)$). Referring to the polychronous model of computation, it represents the set of tags at which the signal holds a value. Clock expressions naturally represent control, the clock $[x]$ represents the time tags at which the Boolean signal x is present and true (i.e. $y := [x] \stackrel{\text{def}}{=} y := 1$ when x). The clock $[\text{not } x]$ represents the time tags at which the Boolean signal x is present and false. We write 0 for the empty clock (it has no tags).

$$ e ::= \hat{}x \mid [x] \mid [\text{not } x] \mid e\,\hat{}\!+\!e' \mid e\,\hat{}\!-\!e' \mid e\,\hat{}\!*\!e' \mid 0 $$

A clock constraint E is a SIGNAL process. The constraint $e\,\hat{}\!=e'$ synchronizes the clocks e and e'. It corresponds to the process $(x := (e = e'))/x$. Composition $E \mid E'$ corresponds to the union of constraints and restriction E/x to the existential quantification of E by x. A transitive scheduling constraint $x \rightarrow y$ when e specifies the order of execution between x and y at the clock e.

$$ E ::= () \mid e\,\hat{}\!=e' \mid e\,\hat{}\!<e' \mid x \rightarrow y \text{ when } e \mid E \mid E' \mid E/x $$

Each process P corresponds to a clock constraint E defined by the clock inference system $P : E$ of Figure 7.

$$ x := y\$1 \text{ init } v : \hat{}x\,\hat{}\!=\hat{}y $$
$$ x := y \text{ when } z : \hat{}x\,\hat{}\!=\hat{}y\,[z] \mid y \rightarrow x \text{ when } z $$
$$ x := y \text{ default } z : \hat{}x\,\hat{}\!=\hat{}y\,\hat{}\!+\hat{}z \mid z \rightarrow x \text{ when } (\hat{}z\,\hat{}\!-\hat{}y) $$
$$ \mid y \rightarrow x \text{ when } \hat{}x $$

$$ \frac{P : E \quad Q : E'}{P \mid Q : E \mid E'} \qquad \frac{P : E}{P/x : E/x} $$

Figure 7: Clock inference system

Code Generation via Hierarchization. The clock constraints E of a process P hold the necessary information to generate a sequential control-flow graph starting from a multi-clocked synchronous specification by a technique of hierarchization, proposed in [1]. It can be outlined by considering a simple SIGNAL program, Figure 8. Process `buffer` implements two functionalities. One is the process `current`. It defines a `cell` in which values are stored at the input clock `^i` and loaded at the output clock `^o`. `cell` is a predefined SIGNAL operation defined by:

$$ x := y \text{ cell } z \text{ init } v \stackrel{\text{def}}{=} (\mid m := x\$1 \text{ init } v $$
$$ \mid x := y \text{ default } m $$
$$ \mid \hat{}x\,\hat{}\!=\hat{}y\,\hat{}\!+\hat{}z $$
$$) / m $$

The other functionality is the process `alternate` that desynchronizes the signals `i` and `o` by synchronizing them to the true and false values of an alternating Boolean signal `b`.

Clock inference (Figure 9) applies the clock inference system of Figure 7 to the process `buffer` to determine three synchronization classes. We observe that `b`, `c_b`, `zb`, and `zo` are synchronous and define the master clock synchronization class of `buffer`. There are two other synchronization classes, `c_i` and `c_o`, that correspond to the true and false values of the Boolean flip-flop variable `b`, respectively.

```
process buffer = (? i ! o)
  (| alternate (i, o) | o := current (i)
  |) where
  process alternate = (? i, o ! )
    (| zb := b$1 init true
     | b := not zb
     | o ^= when not b
     | i ^= when b
    |) / b, zb;
  process current = (? i ! o)
    (| zo := i cell ^o init false
     | o  := zo when ^o
    |) / zo;
```

Figure 8: Polychronous specification of a buffer

```
(| c_b ^= b          | c_o ^= o
 | b   ^= zb         | i -> zo when ^i
 | zb  ^= zo         | zb -> b
 | c_i := when b     | zo -> o when ^o
 | c_i ^= i          |) / zb, zo, c_b,
 | c_o := when not b       c_o, c_i, b;
```

Figure 9: Clock analysis of the buffer

This defines three nodes in the control-flow graph of the generated code shown in Figure 10. At the main clock `c_b`, `b`, and `c_o` are calculated from `zb`. At the sub-clock `b`, the input signal `i` is read. At the sub-clock `c_o` the output signal `o` is written. Finally, `zb` is determined. Notice that the sequence of instructions follows the scheduling constraints determined during clock inference.

```
buffer_iterate () {          if (c_o) {
  b = !zb;                       o = i;
  c_o = !b;                      w_buffer_o(o);
  if (b) {                     }
    if (!r_buffer_i(&i))       zb = b;
      return FALSE;            return TRUE;
  }                          }
}
```

Figure 10: Buffer code generation

Some More Concrete Syntax. In addition to the core syntax of SIGNAL presented so far, we make extensive use of process declarations and partial equations for the purpose of modeling our case study. In SIGNAL, a partial equation $x ::= y\,f\,z$ when e is the partial definition of the variable x by the operation $y\,f\,z$ at the clock denoted by the expression e. The default equation $x ::= \text{default value } v$ defines the value of the variable x when it is present but no corresponding partial equation $x ::= y\,f\,z$ when e applies (because e is absent). Let x be a variable defined using n partial equations and a default value v:

$$ x ::= x_1 \text{ when } e_1 $$
$$ \vdots $$
$$ \mid x ::= x_n \text{ when } e_n $$
$$ \mid x ::= \text{default value } v $$

The SIGNAL compiler processes this definition by first checking the clock expressions $e_1, \dots e_n$ mutually exclusive and then handling the definition as the equivalent equation: $x := (x_1 \text{ when } e_1)$ default $\dots (x_n \text{ when } e_n)$ default v. The declaration of a process P of name f, input signals $x_1..x_m$, output signals $x_{m+1}..x_n$ is noted process $f = (?\, x_1, \dots x_m \,!\, x_{m+1}, \dots x_n) (\mid P \mid);$. Once

declared, process f may be called with its actual parameters $y_1..y_n$ by $(y_{m+1}, \ldots y_n) := f(y_1, \ldots y_m)$ and behave as P with $x_1..x_n$ substituted by $y_1..y_n$. A variant declaration is that of a foreign function f, accessible, e.g. from a separately compiled C library. Its call can be wrapped into SIGNAL by declaring its interface and by declaring an abstraction E of its behavior, which consists of scheduling and clock constraints.

$$\texttt{process } f = (?\, x_1, \ldots x_m\, !\, x)\, \texttt{spec}\, (|\, E\, |)$$
$$\texttt{pragmas C\_CODE}"\&x = \texttt{f}(\&\texttt{x}_1, \ldots \&\texttt{x}_\texttt{m})"$$
$$\texttt{end pragmas;}$$

4. METHODOLOGY AND TOOLS

Our modeling and verification methodology starts off with a SYS-TEMC model of a system. The goal is to provide all system components with a formal behavioral type that can be used to discover errors in the composition of components. The formal type can also be used to formally verify properties of the components and their composition. The methodology consists of several steps. First the SYSTEMC code is analyzed in a preprocessing step and some types are replaced for better conversion results. Then a static single assignment intermediate representation is generated. From this representation, clock and scheduling relations are extracted that serve as a basis for the generation of SIGNAL code. The compilation of the signal code performs static checking for types, dependencies, and clock constraints. This results in a highly reliable connection of components as the synchronous composition - once successfully performed - rules out many sources of error that are not checked for in a common type checking system.

As the SIGNAL program represents a formal model, also dynamic properties can be checked for, reaching into formal verification with reasonably small additional effort. For the model checker we use, the model has to be abstracted or transformed into a Boolean version. The remainder of this section describes some of the tools used throughout the process. In Section 5 all of these steps can be followed more in detail for the example of an FIR filter.

Static Single Assignment Form and GIMPLE. As SYS-TEMC programs are written in C++, they can contain very complex constructs, which make it difficult to obtain a corresponding SIG-NAL representation. It is obvious that it would be much easier to make a translation from a more low level representation or from a C++ subset that adheres to some rules of simplicity. Not wanting to restrict the input language, we are using an intermediate representation that fits our needs.

GIMPLE [15] is a simple intermediate representation developed at McGill University [11] and has now been adopted in the Gnu Compiler Collection (GCC) [8]. GIMPLE is a three address C-like language with no high level control structures. Some of its particularities are that GIMPLE statements - with the exception of function calls - contain not more than three operands and have no side effects, intermediate values are stored in temporary variables, and all control structures are lowered to conditional gotos.

Most of the GCC optimization passes use the data flow information provided by the static single assignment form (SSA) [4]. This is an intermediate representation in which every variable is assigned exactly once. It is particularly used for high level compiler optimization. This makes it particularly useful for our purpose, since static single assignments have a very regular structure, can be easily manipulated by automatic program analysis tools, and find a quite natural translation into the SIGNAL synchronous formalism.

GIMPLE and SSA are part of the Tree-SSA [7] changes that have been integrated into the Gnu Compiler Collection (GCC) starting from version 3.5 that - at the time of writing - has yet to be released. These changes allow for language independent, higher level optimization passes. By using GIMPLE-SSA as an intermediate representation for the behavioral type generation, we can therefore benefit from all current and future optimization passes implemented in the GCC. GIMPLE-SSA code can be dumped with compiler options corresponding to different levels of optimization, such as `-fdump-tree-ssa`, `-fdump-tree-gimple` and `-fdump-tree-optimized`. The last option gives the best results. When further automating the type extraction process, the tree representation of GCC can be used directly in shared memory without having to dump it to disc and to read it again.

Formal Verification of Component Properties. Before code from a SIGNAL program is generated, the SIGNAL compiler checks for static problems such as contradictory clock constraints, cycles, and zero clocks. However, in order to check dynamic properties of the system, the SIGNAL companion model checker SI-GALI [14] can be used. Given a formal model of a system in SIG-NAL, SIGALI can verify formal properties of the model. It is an interactive tool specialized on algebraic reasoning in $\mathbb{Z}/3\mathbb{Z}$ logic.

SIGALI transforms SIGNAL programs into sets of dynamic polynomial equations that basically describe an automaton. It can analyze this automaton and prove properties such as liveness, reachability, and deadlock. The fact that it is solely reasoning on a $\mathbb{Z}/3\mathbb{Z}$ logic constrains the conditions to the Boolean data type (true, false, absent). This is practical in the sense that true numerical verification very soon would result in state spaces that are no longer manageable, however it requires, depending on the nature of the underlying model, major or minor modifications prior to formal verification.

For many properties numerical values are not needed at all and can be abstracted away thus speeding up verification. When verification of numerical manipulations is sought, an abstraction to several Boolean values suffices in most cases to satisfy the needs.

5. CASE STUDY OF AN FIR FILTER

This section exemplifies the presented approach with the design of a finite impulse response filter (FIR). It details the decomposition of a full featured SYSTEMC specification into an SSA representation. The different analysis steps are demonstrated down to the final typed SIGNAL representation.

As a starting point, we use the SYSTEMC model of the FIR from the SYSTEMC 2.0.1 distribution [16] and translate it into SSA code. We show how this SSA code is analyzed and how clock and scheduling information can be extracted. In Section 5 the corresponding SIGNAL type is presented and it is shown how to obtain it with the preceding information.

The SystemC Model. In the SYSTEMC model, the filter itself consists of one functional block surrounded by a testbench consisting of a Stimulus that generates input values and a Display module that receives the output and displays it on the screen (Figure 11).

The FIR unit is implemented as an SC_THREAD that is triggered on the positive clock edge. The other blocks are SC_METHODs. The left hand side of Figure 12 displays the SYSTEMC code of the entry function for the FIR block. The first 10 lines just handle the initialization of variables. Then there is an infinite *while* loop that contains the actual filter functionality.

Roughly speaking, it waits until there is a valid input available, reads this input, processes it, writes it to an output, and then notifies its environment that the result is available. At the end of each
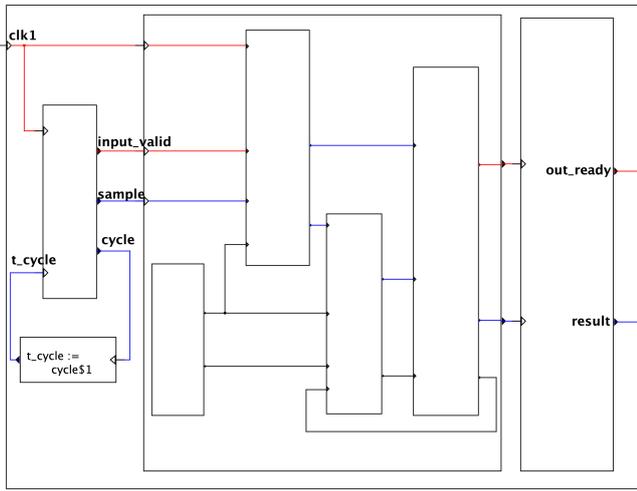
Figure 11: Structure of the FIR filter with testbench

while loop it suspends itself until the next positive clock edge. The FIR result is the sum of the last 15 input values weighted with 15 coefficients. This is done in two *for* loops. The first one does the weighting and the second one is shifting the buffer array containing the last inputs.

Communication with the environment is done via *enable* signals. The Stimulus indicates with the signal *in_valid* that a new value is available. In the same way, the Display is sensitive to the variable *output_data_ready* that is set when a new output value is available.

Obtaining a GIMPLE-SSA Representation. The right hand side of Figure 12 shows the GIMPLE-SSA code that corresponds to the SYSTEMC FIR. For the generation of a clean GIMPLE-SSA representation we follow three steps. First, preprocessing of the SYSTEMC code, second, translation to GIMPLE-SSA with GCC, and third, post processing of the generated GIMPLE-SSA code. The direct generation of GIMPLE-SSA from SYSTEMC can be done, but it results in very large and hardly readable code. A closer look reveals that most of this bloating is due to the SYSTEMC types and statements, which are implemented as macros and get translated as well. If we replace the SYSTEMC types by corresponding C++ types, e.g. *sc_int* is changed to *int* or *unsigned*, in a simple pre-processing step, the size of the generated code shrinks drastically.

More complex statements such as *wait(signal)*, however, still cause a considerable increase of the code size compared to the original SYSTEMC code. We decide to simply comment these out in the SYSTEMC source so they are ignored by the compiler and can later be taken care of separately in a post processing step.

During post processing we replace the *wait(signal)* statements by corresponding SSA statements. Logically a *wait* statement is similar to an *if* branch. Depending on a condition something is executed, otherwise something else. The condition is the signal that we are waiting for (e.g. *in_valid == true*. If there is no signal given, the process waits for the signal it is sensitive to (this is the positive edge of the clock in this example).

In order to be able to execute the *wait* statement separately, we have to introduce a separate label for it. As we can see on the right hand side of Figure 12, for *L1*, *L1a* is introduced since the wait statement is not at the beginning of the block, and for *L3*, there are two additional labels, *L3a* and *L3b* because this wait statement is in the middle of a block.

```
void fir::entry() {
    sc_int<8> tmp;
    sc_int<17>pro;
    sc_int<19>acc;
    sc_int<8>shift[16];
    result.write(0);
    out_ready.write(false);

    for (int i=0; i<=15; i++)
        shift[i] = 0;
    wait();

    while(1) {
        out_ready.write(false);
        wait_until
            in_valid.delayed()==true;
        tmp = sample.read();
        acc = tmp*coefs[0];
        for(int i=14; i>=0; i--) {
            pro = shift[i]*coefs[i+1];
            acc += pro;
        };
        for(int i=14; i>=0; i--)
            shift[i+1] = shift[i];
        shift[0] = tmp;
        // write output values
        result.write((int)acc);
        out_ready.write(true);
        wait();
    };
}
```

```
void fir::entry() {
        int shift[16], i, acc, tmp;
        i=0; goto L1;
        this → result = 0;
        this→output_data_ready = 0;
L0:  shift[i]= 0
        i = i + 1;
L1:  t_i = (i<=15)
        if (t_i) goto L0;
        else goto L1a;
L1a:wait (clk1_pos);
L3:  this→output_data_ready = 0;
L3a:wait_until(in_valid == true);
L3b:tmp = this→sample;
        acc=this→coefs[0]*tmp;
        i = 14;
        goto L5;
L4:  acc=acc+shift[i]
            *this→coefs[i+1];
        i = i - 1;
L5:  if (i >= 0) goto L4;
        else goto L6;
L6:  i = 14;
        goto L8;
L7:  shift[i + 1] = shift[i];
        i = i - 1;
L8:  if (i>=0) goto L7;
        else goto L9;
L9:  shift[0] = tmp;
        this→result = acc;
        this→output_data_ready = 1;
L9a:wait (clk1_pos);
        goto L3;
}
```

Figure 12: SystemC and SSA code for the FIR core

Extracting Clock and Scheduling Information. Though slightly bigger in size, the SSA representation has several advantages with respect to automated analysis and conversion: it consists of very simple and repetitive statements, it is separated into sequential blocks without branches and where variable are assigned once. The extracted behavioral type information can be separated into two parts, control and data flow.

Figure 13 displays this information for the FIR in the form of a synchronous transition system (STS, as in [18]). It consists of propositions on clocks $\hat{\ }x$ guarded by block input clocks x_L to for implications $x_L \Rightarrow \hat{\ }x$ and of proposition on state transitions of the form $e \Rightarrow x'_L$ to mean that if e is present then x_L is the next block to be executed.

In order to understand how these clock relations are obtained, we have to take a look at the SSA form in Figure 12. For instance, $x_{L0} \Rightarrow \hat{\ }shift$ means that whenever block L0 is entered, the signal *shift* has to be present. Transitions from one block to another are represented like this: $x_{L4} \Rightarrow x_{L5}$. However, if in the following block a signal is assigned that has already been assigned in the current block, it cannot be executed in the same cycle. The time has to be advanced, this is expressed in $x_{fir} \Rightarrow x'_{L1}$, where the ′ indicates the next value for this signal. For *if* statements - such as in block L1 - the value of a Boolean signal decides which of the two targets is taken. Figure 14 graphically details this control flow. There are several the small loops, such as the one between L1 and L2, representing the manipulation of an array of values.

The big loop between L3 and L9 represents the actual program execution loop. Everything before that deals with initialization. After initialization the program waits for the next positive clock edge. At the beginning of the execution it is waiting for a valid
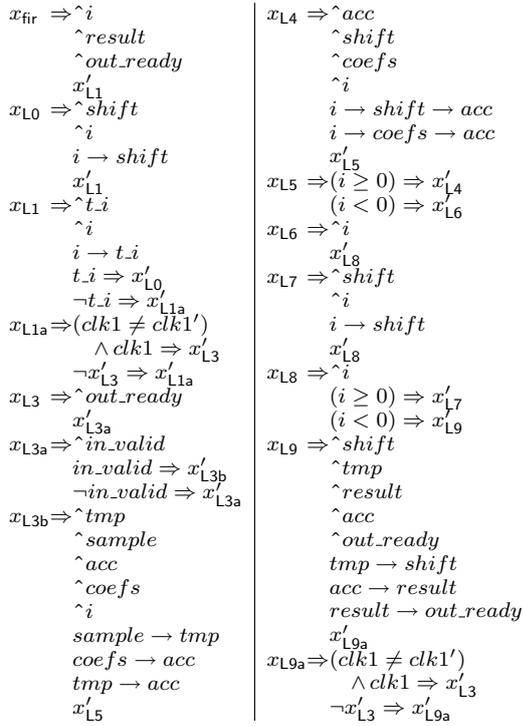
Left column:

$$x_{\text{fir}} \Rightarrow {}^{\wedge}i$$
$${}^{\wedge}result$$
$${}^{\wedge}out\_ready$$
$$x'_{\text{L1}}$$
$$x_{\text{L0}} \Rightarrow {}^{\wedge}shift$$
$${}^{\wedge}i$$
$$i \rightarrow shift$$
$$x'_{\text{L1}}$$
$$x_{\text{L1}} \Rightarrow {}^{\wedge}t\_i$$
$${}^{\wedge}i$$
$$i \rightarrow t\_i$$
$$t\_i \Rightarrow x'_{\text{L0}}$$
$$\neg t\_i \Rightarrow x'_{\text{L1a}}$$
$$x_{\text{L1a}} \Rightarrow (clk1 \neq clk1')$$
$$\wedge\, clk1 \Rightarrow x'_{\text{L3}}$$
$$\neg x'_{\text{L3}} \Rightarrow x'_{\text{L1a}}$$
$$x_{\text{L3}} \Rightarrow {}^{\wedge}out\_ready$$
$$x'_{\text{L3a}}$$
$$x_{\text{L3a}} \Rightarrow {}^{\wedge}in\_valid$$
$$in\_valid \Rightarrow x'_{\text{L3b}}$$
$$\neg in\_valid \Rightarrow x'_{\text{L3a}}$$
$$x_{\text{L3b}} \Rightarrow {}^{\wedge}tmp$$
$${}^{\wedge}sample$$
$${}^{\wedge}acc$$
$${}^{\wedge}coefs$$
$${}^{\wedge}i$$
$$sample \rightarrow tmp$$
$$coefs \rightarrow acc$$
$$tmp \rightarrow acc$$
$$x'_{\text{L5}}$$

Right column:

$$x_{\text{L4}} \Rightarrow {}^{\wedge}acc$$
$${}^{\wedge}shift$$
$${}^{\wedge}coefs$$
$${}^{\wedge}i$$
$$i \rightarrow shift \rightarrow acc$$
$$i \rightarrow coefs \rightarrow acc$$
$$x'_{\text{L5}}$$
$$x_{\text{L5}} \Rightarrow (i \geq 0) \Rightarrow x'_{\text{L4}}$$
$$(i < 0) \Rightarrow x'_{\text{L6}}$$
$$x_{\text{L6}} \Rightarrow {}^{\wedge}i$$
$$x'_{\text{L8}}$$
$$x_{\text{L7}} \Rightarrow {}^{\wedge}shift$$
$${}^{\wedge}i$$
$$i \rightarrow shift$$
$$x'_{\text{L8}}$$
$$x_{\text{L8}} \Rightarrow {}^{\wedge}i$$
$$(i \geq 0) \Rightarrow x'_{\text{L7}}$$
$$(i < 0) \Rightarrow x'_{\text{L9}}$$
$$x_{\text{L9}} \Rightarrow {}^{\wedge}shift$$
$${}^{\wedge}tmp$$
$${}^{\wedge}result$$
$${}^{\wedge}acc$$
$${}^{\wedge}out\_ready$$
$$tmp \rightarrow shift$$
$$acc \rightarrow result$$
$$result \rightarrow out\_ready$$
$$x'_{\text{L9a}}$$
$$x_{\text{L9a}} \Rightarrow (clk1 \neq clk1')$$
$$\wedge\, clk1 \Rightarrow x'_{\text{L3}}$$
$$\neg x'_{\text{L3}} \Rightarrow x'_{\text{L9a}}$$

Figure 13: Clock and scheduling relations for the FIR

Figure 14: Control flow of the FIR filter

Figure 15: Data Flow of the FIR

able within one block and at the same instant to conform with a purely synchronous execution. Whenever there is the need to advance time before a move from one block to another, the execution of the next block is delayed using a signal delay statement `xL1 := xnL1$1 init false`. Here `xnL1` represents the next value and `xL1` its current value. A Variable that gets assigned a value in more than one block would be renamed in SSA.

In SIGNAL it does not have to be renamed, instead we can use partial equations, designated with the ":=". A partial equation defines a variable for a certain number of instants. A second or third partial equation then can define additional instants for this variable, as long as it is not defined twice on any instant. Since the instants of the blocks are temporarily disjoint, a variable can be defined once per block with the help of partial equations instead of once per program. Two partial equations for the variable `out_ready` could be distributed anywhere in the program:

```
  out_ready ::= false when xL3
| out_ready ::= true when xL9
```

Still, partial equations are a source of errors since it is difficult to make sure that they are conflict-free, and to have parts of a variable defined in different parts of the program can also obstruct legibility. This is why we often combine these partial equations afterward into full equations, adding a default statement that otherwise is implied by the compiler:

```
  out_ready :=        false when xL3
               default true when xL9
               default false
```

In the code given Figure 19, six partial equations for variable i have been combined. Assembled at one location, it is more clear to see that the definitions do not conflict. In the FIR example, we do not have any complex data manipulations. Would that be the case, it would probably be unreasonably complicated to describe them in SIGNAL. In such cases, they can be wrapped as external functions using SIGNAL's `pragma` directive. With `pragma` statements external code can be used directly. When the type is provided with appropriate signal dependency and clock synchronization relations, these functions are not completely black boxes to the system. This `pragma` mechanism permits the handling of data flow intensive applications without much additional cost.

Making a Boolean Model. As explained Section 4 the FIR model has to be abstracted to a Boolean model in order to check dynamic properties with the symbolic model checker SIGALI of the POLYCHRONY workbench. Usually, it is not necessary to transform the whole model to a binary form as we might not be interested
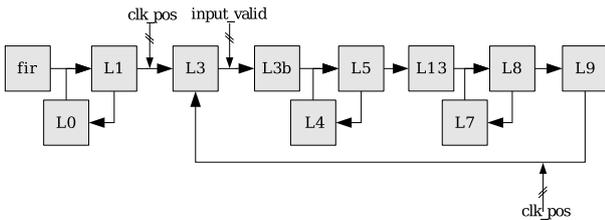
input value. Then the calculations are executed and it subsequently waits for the next positive clock edge before resuming execution at block L3.

Data flow dependencies for the FIR are displayed on the right hand side of Figure 13. The structure of these dependencies is very simple, the arrow (a→b) showing that *a* has to be present before *b* can be evaluated. Overall we see that for the FIR example, the control part largely outweighs the data flow part. Figure 15 illustrates the data flow of the example. We see that for the execution loop, the major data activity takes place in blocks L3b, L4, L7, and L9. L3b reads the external inputs *coefs* and *sample*, L9 eventually produces the outputs *out_ready* and *result*.

The Equivalent SIGNAL Program. As described earlier, the combination of control and data flow can be expressed by SIGNAL equations. In order to obtain such equations, it is helpful to have the clock and scheduling information particularized earlier, but they can also be obtained directly from the SSA representation to reflect all control and data flow information. Figure 19 details the SIGNAL equation giving the corresponding abstraction of the FIR behavior.

Translation can be done block-wise, and mostly line by line. The SIGNAL language strictly prohibits multiple assignments of a vari-
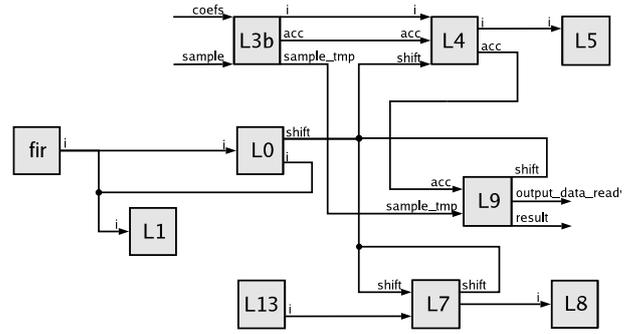
in all numerical details of the model, but rather some higher level properties such as liveness or deadlock-freedom.

In the case of the FIR, however, we rewrote a binary version of the initial model. Obviously, this blows up the model in size, so we cannot show the whole binary model here. For this transformation we first check where we have integer or float variables and how they are used. In the FIR, no float variables are used. However, the actual values of the FIR are integers. They are generated in the stimulus, fed into the FIR and stored in a 15 stage pipeline. The result is calculated numerically and then the output is an integer again.

We reduce the model to binary values in several stages. At first, we reduce the pipeline from fifteen to three stages, representing the stages not by an integer variable, but by three Boolean variables. The input values for the FIR are then reduced to $(0, 1, 2)$ and also represented by Boolean variables. Finally, the most tricky part is the numerical calculation of the result. With the current reduction of values and coefficients, the output of the FIR can never be greater than 15, so we use four Boolean variables as output, interpreted as the bits of a four bit binary number. While the total size of the binary model in number of lines nearly doubled, the state-space for verification only represents a fragment compared to the integer version and is now small enough to be used for formal verification.

Using the Model Checker. Once we have a binary model of the FIR, the model checker can be used to verify formal properties on it. In order to do this, we have compile the SIGNAL program with the option -z3z, which results in the generation of a file with the extension .z3z. This is the input file for SIGALI. It contains a model of the SIGNAL program expressed using polynomial dynamical equations, the data structure manipulated by SIGALI.

As an example on how to define a property for verification, signal *test3*, Figure 16 is a Boolean property describing the situation that once the system reaches block *L3* and the signal *input_valid* is true, it will reach block *L9* in at most three steps. We define *test2* as an auxiliary variable that is only true between *L3* and *L9*. The state variable *test3* will be true as soon as test2 has been true for more than three cycles and there was no new value in between.

```
test2 :=            true when xL3b
        default false when xL9
        default test2$1 init false
test3 := true when (test2 and test2$1 and
                    test2$2 and test2$3)
            when (input_valid$2 = false)
        default test3 init false
```

Figure 16: Example of a formal property definition

Figure 17 depicts the interaction with the model checker. In the first statement, the z3z file of the design is read. Then internal libraries are loaded. Finally we check for liveness, if property 'test3' is reachable and if it is always false. If *test3* is not reachable and always false, then the property holds true.

Abstraction of the SIGNAL Model. The SIGNAL program described in appendix, Figure 19, is the model that implements the FIR filter. It is an exact SIGNAL mirror of the original SYSTEMC implementation. For many purposes, however, all functionality is not needed in order to evaluate the validity of a condition. An abstracted model that does not contain data manipulations is much lighter and still can serve to check conditions such as deadlocks, termination, and race-conditions.

```
read("top.z3z");
read("Creat_SDP.lib");
read("Verif_Determ.lib");
read("Property.lib");
Alive(S);
Reachable(S,B_True(S,test3));
Always(S,B_False(S,test3));
```

Figure 17: Verification of Properties using Sigali

Figure 18 depicts the code for a possible abstraction for the FIR model, reduced to the description of control-flow transitions between blocks. The light weight of this model allows for much faster verification of properties, and, therefore makes it possible to check for these properties on a higher design level, possibly comprising the whole system. For detailed checks including correctness of data manipulations or range-checks, the complete type can be used.

```
process fir (? boolean input_valid, clk1
             ! boolean out_ready)
(| out_ready :=         false when xfir
            default false when xL3
            default true when xL9
            default out_ready$ init false
 | xfir   :=         false when xL3
        default (xfir$ init true)
 | xL3    :=         true when xfir$
            when clk1
            when not (clk1 = clk1$)
        default true when xL9a
            when clk1
            when not (clk1 = clk1$)
        default false
 | xL3a   :=         true when xL3
        default xnL3a$1 init false
 | xL9    :=         true when xL3a
            when input_valid
        default false
 | xnL3a :=          true when xL3a
            when not not input_valid
        default false
 | xnL9a :=          true when xL9
        default true when xL9a
            when ((not clk1)
                    default (clk1 = clk1$))
        default false
 | xL9a := xnL9a$1
 | input_valid ^= xfir ^= xL3 ^= xL9 ^= clk1
            ^= xnL3a ^= xnL9a ^= out_ready
 |) where integer i; boolean xL3, xnL3a, xL3a,
        xL6, xfir, xL9, xL9a, xnL9a;
end;
```

Figure 18: Abstract SIGNAL model

Status and Current Work. There are still some obstacles to get this process smoothly to work. As we have seen in Section 5, it is not obvious to obtain clean SSA code. Therefore substantial effort has to be put into the generation of clean and reasonably short SSA code. This can be done with compiler optimizations on the one hand and pre/post processing on the other hand. When generating clock dependencies and the SIGNAL type respectively, the crucial point is the advancement of time. It has to be made sure that if blocks assign the same variable, there has to be an advancement of time in between. As the whole control tree has to be considered, this problem breaks down to graph coloring and is not trivial.

The presented approach illustrates its applicability for both C++ and JAVA. However, for SYSTEMC there are some additional issues to consider. As for now, we treat only the entry functions of SYSTEMC programs. In order to type a whole SYSTEMC application consisting of several modules in the same way, multiple entry functions would have to be treated as well as the architecture and connectivity between them. This is ineffective to do in SSA because the change to the lower level will obstruct the higher level hierarchy structures. Consequently, the pre processing step has to be designed to be more sophisticated in order to handle the structure correctly. Also, adequate SIGNAL statements equivalent to certain SYSTEMC constructs such as *sc_fifo* or *sc_semaphore* have to be defined. These can then form a library that would significantly simplify the conversion process.

6. RELATED WORK

The capture of the behavior of a system through a type theoretical framework relates our technique to the work of Rajamani et al. [17], and many others, on abstracting high-level and concurrent specifications, e.g. the π-calculus, by using a formalism, e.g. Milner's CCS, in which, primarily, checking type equivalence, e.g. bisimulation, is decidable.

Our contribution contrasts from related studies by the capability to capture scalable abstractions of the type-checked system. In our type system, scalability ranges from the capability to express the exact meaning of the program, in order to make structural transformations and optimizations on it, down to properties expressed by Boolean equations between clocks, allowing for a rapid static-checking of design correctness properties. Our system allows for a wide spectrum of design abstraction and refinement patterns to be applied on a model, e.g. abstraction of states by clocks, abstraction of existentially quantified clocks, hierarchic abstraction, in the aim of choosing a better degree of abstraction for faster verification.

We share the aim of a scalable and correct-by-construction exploration of abstraction-refinement of system behaviors with the work of Henzinger et al. on interface automata [5]. Our approach primarily differs from interface automata in the data-flow formalism used in the Polychrony workbench where partial clock and scheduling relations express the multi-clocked synchronous behavior of the system. Compared to an automata-based approach, our declarative approach allows to hierarchically explore abstraction capabilities and to cover design exploration with the methodological notion of refinement along the whole design cycle of the system, ranging from the early requirements specification to the latest sequential and distributed code-generation [20, 13].

7. CONCLUSION

The approach presented shows how to obtain a behavioral SIGNAL type from SYSTEMC components. The passage through the GIMPLE-SSA form allows for a straightforward translation to a formal synchronous model. When used for SYSTEMC components, this methodology can significantly help to detect problems in the connection with other components. If a synchronous composition of several SIGNAL types is successful the connection of the corresponding SYSTEMC components is very likely to work. Additional confidence can be gained by verifying formal properties of the components as well as of any composition of components thus increasing certainty on the correctness of the whole system design. If the methodology is systematically applied, a constantly growing library of verified IP components is obtained that helps to substantially reduce development cycles and makes it possible to develop larger scale systems.
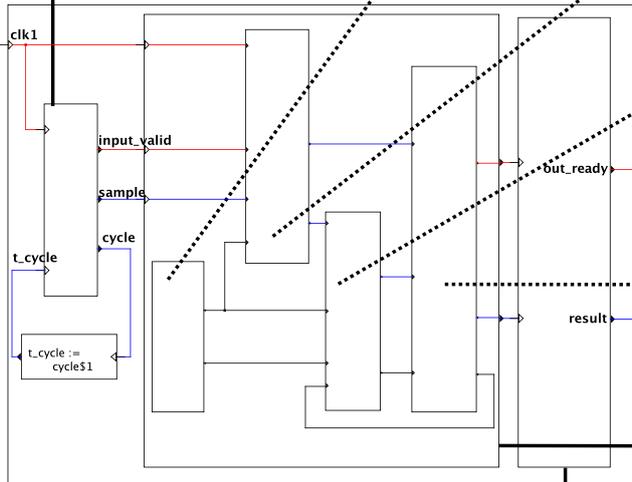
8. REFERENCES

[1] T. P. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Conference on Programming Language Design and Implementation*. ACM Press, 1995.

[2] A. Benveniste. Some synchronization issues when designing embedded systems from components. In *Conference on Embedded Software, EMSOFT'01*, volume 2211, pages 32–49. T. Henzinger and C. Hirsch, Eds, LNCS, 2001.

[3] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, 1991.

[5] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *First International Workshop on Embedded Software*, pages pp. 148–165. Lecture Notes in Computer Science 2211, Springer-Verlag, 2001.

[6] F. Doucet, S. Shukla, and R. Gupta. Typing abstractions and management in a component framework. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE Press, Jan. 2003.

[7] Free Software Foundation. The GCC tree-ssa documentation. http://gcc.gnu.org/onlinedocs/gccint/Tree-SSA.html.

[8] Free Software Foundation. The GNU compiler collection. http://gcc.gnu.org

[9] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Mar. 2000.

[10] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[11] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420. Springer-Verlag, LNCS 757, 1993.

[12] IRISA, project ESPRESSO. The polychrony workbench. http://www.irisa.fr/espresso/Polychrony.

[13] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(1), Apr. 2003.

[14] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, Oct. 2000.

[15] J. Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *GCC Developers Summit*, Ottawa, Canada, May 2003.

[16] Open SystemC Initiative. The OSCI systemc website. http://www.SystemC.org/, 2004.

[17] S. K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In *Proceedings Static Analysis Symposium (SAS'01)*, Paris, July 2001. Springer Verlag.

[18] J.-P. Talpin, D. Berner, S. Shukla, A. Gamatié, P. Le Guernic, and R. Gupta. A behavioral type inference system for compositional system-on-chip design. In *International Conference on Application of Concurrency to System Design (ACSD)*, Hamilton, Canada, June 2004.

[19] J.-P. Talpin, A. Gamatié, D. Berner, B. Le Dez, and P. Le Guernic. Hard real-time implementation of embedded systems in java. In *International Workshop on Scientific Engineering of Distributed JAVA Applications, Lectures Notes in Computer Science*. Springer Verlag, Nov. 2003.

[20] J.-P. Talpin, P. Le Guernic, S. K. Shukla, R. Gupta, and F. Doucet. Polychrony for formal refinement-checking in a system-level design methodology. In *Special Issue of Fundamenta Informaticae on Applications of Concurrency to System Design*. IOS Press, Aug. 2004.

```
process stimulus =
 (? boolean clk1; integer cycle
  ! boolean input_valid; integer sample, t_cycle)
(| xstim  := clk1
 | t_cycle:=cycle + 1 when xstim default t_cycle$
 | cond1  := cycle <=3 default false
 | xL0 := when cond1 when xstim$ default false
 | xL1 := when not cond1 when xstim$ default false
 | input_valid := false when xL0
    default false when xL1
    default true when xL2 default false
 | xL3 := when xL0
    default when not cond2 when xL1
    default when xL2 default false
 | cond2:=(cycle modulo 19 == false) default false
 | xnL2 := when cond2 when xL1 default false
 | xL2 := xnL2$
 | sample := send_value when xL2 default sample$
 | write("Sample: ", sample when input_valid)
 | send_value := (send_value$ init 0) +1 when xL2 default
send_value$
 | clk1 ^= input_valid ^= send_value ^= t_cycle
   ^= cycle ^= cond1^= cond2^= xstim ^= xL0
   ^= xL1 ^= xnL2 ^= xL3 ^= sample
 |) where integer send_value; boolean cond1,
   cond2, xstim, xL0, xL1, xL2, xnL2, xL3;
end;
```

```
process fir=(? boolean input_valid,clk1;
            integer sample
           ! integer result;boolean out_ready)

(|coefs:=[16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
 | xfir := false when xL1
     default (xfir$ init true)
 | i ::= 0 when xfir
 | i ::= i$ +1 when xL0
 | result ::= 0 when xfir
 | out_ready ::= false when xfir
 | xnL1 := xfir
 | xL1 := when xL0 default xnL1$1 init false
 | zshift := shift$1
 | t_i :=  (i < 15) when xL1 default false
 | xnL0:= t_i
 | xL0 := xnL0$1
 | xnL1a := when not t_i when xL1 default false
 | xL1a  := xnL1a$1

 | xL3 := when (xL1a or xL9a)
       when clk1 when clk1 = clk1$ init 0
       default false
 | xnL1a := when not t_i when xL1 default false
 | xL3a  := when xL3 default xnL3a$1 init false
 | xL3b  := when xL3a when input_valid
        default false
 | xnL3a := when xL3a when not input_valid
         default false
 | sample_tmp := sample when xL3b
      default sample_tmp$ init 0
 | i ::= 15 when xL3b
 | out_ready ::= false when xL3
 | acc ::= coefs[0] * sample_tmp  when xL3b

 | acc ::=  acc$ + shift[i]* coefs[i+1] when xL4
 | xL5 := when xL3b default xL4
 | xnL4:= when i>0 when xL5 default false
 | xL4 := xnL4$1
 | xnL13:= when i<=0 when xL5 default false
 | i ::= i$ -1 when xL4

 | xL13 := xnL13$1
 | xL8 := when xL13 default xL7
 | xnL7 :=  when i>0 when xL8 default false
 | xL7 := xnL7$1
 | xL9 :=  when i=0 when xL8 default false
 | xnL9a := when xL9 default when xL9a
     when not clk1 default clk1 = clk1$ init 0
     default false
 | xL9a := xnL9a$1
 | i ::= 15 when xL13
 | i ::= i$ -1 when xL7
 | result ::=  acc when xL9
 | out_ready ::= true when xL9
 | array k to 15 of
   shift[k] := (k  when xL0)
    default (shift[k-1]  when (k=i+1) when xL7)
    default (sample_tmp when k=0 when xL9$)
    default zshift[k]
   end

|t_i^= sample ^= input_valid ^= sample_tmp ^= acc
 ^=shift ^= coefs ^= result ^= xfir ^= xL1 ^= xL3
 ^=xL3b ^= xL9 ^= xnL1a ^= xnL3a ^= xnL4 ^= xnL13
 ^=xnL7 ^= xnL9a ^= clk1 ^= out_ready|)
 where integer i, sample_tmp, acc;
  [16] integer shift , coefs, zshift;
  boolean xL3b, xnL1a, xL3, xL13, xL7, xL4, t_i,
  xfir, xL0, xL1, xL1a, xnL0, xnL1, xnL3a, xnL4,
  xnL13, xL5,  xL8, xL9, xL9a, xL3a, xnL7, xnL9a;
end;
```



```
process display =
 ( ? boolean out_ready; integer result)
 (| o_result := result when out_ready
  | message1 := "Result: "
  | write(message1, o_result)
  | o_result ^= message1
  |) where string message1; integer o_result;
 end;
```

Figure 19: Block view of the SIGNAL type for the FIR filter