Modular Design Through Component Abstraction
David Berner, Jean-Pierre Talpin, Paul Le Guernic, Sandeep Shukla

To cite this version:

HAL Id: hal-00542139
https://hal.archives-ouvertes.fr/hal-00542139
Submitted on 1 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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.

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 accept 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 the 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.

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 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.

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 ocount to 0. Label L2 corresponds to a loop that shifts idata right and adds its right-most bit to an output count variable ocount and shifts it right in order to process the next bit.

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

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 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 not T2.
T2 = T1 == 0;
T3 = 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 T2 is not 0, see Figure 3).

L2: T1 = idata;
T2 = T1 == 0;
T3 = true when T2
T3 = icount;
T3 ::= icount$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 xL1. 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.

Figure 5: Modeling communication protocols

Completion. All entry clocks xL are simultaneously present when a block is executed. Each signal xL holds the value 1 iff the block
Olychrony role during modeling as they represent clocks, i.e. the data-structure
adaptation and transformation. As a result, traceability is easily pro-
vided 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
\[ T = (T, \leq, 0) \]

of tags. A tag \( t \) denotes a symbolic period in time. The re-
lation \( \leq \) denotes a partial order. Its minimum is noted \( 0 \). We note \( T \subseteq \mathbb{Z} \) a chain of tags (a totally ordered subset of \( T \)). We define:

- an event \( e \) in \( E = T \times V \) by the pair of a value and a tag,
- a signal \( s \) in \( S = C \rightarrow V \) by a function from a chain to values,
- a behavior \( b \) in \( B = X \rightarrow S \) by a map from names to signals,
- a process \( p \) in \( P \) by a set of behaviors of the same domain.

Figure 6 depicts a behavior 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 syn-
chronous to odd occurrences of events along \( y \) and share the same
moment of \( x \) and \( y \) are present; \( x \) is present and that is absent otherwise. The
signal \( y \) and its delayed copy \( x := y \textrm{?init} v \) are synchronous: they share the same set of tags \( t \), \( t_2 \), \( t_3 \)...
Initially, at \( t_1 \), the signal \( x \) takes the declared value \( v \) and then, at tag \( t_n \), the
value of \( y \) at tag \( t_n - 1 \).

\[(x := y \text{?init} v)\]

\[\begin{align*}
  x &\overset{t_1}{\rightarrow} y &\overset{t_2}{\rightarrow} z &\overset{t_3}{\rightarrow} \\
\end{align*}\]

Figure 6: A multi-clocked behavior

In the remainder, we write \( \text{tags}(s) \) for the tags of a signal \( s \),
\( b_X \) for the projection of a behavior \( b \) on \( X \subseteq X' \) and \( b/X = b|_{\text{vars}(b) \cap X} \) for its complementary, \( \text{vars}(b) \) and \( \text{vars}(p) \) for the
domains of \( b \) and \( p \). We write \( B|_X \) for the set of all behaviors defined
on the set of variables \( X \). Synchronous composition is noted \( p \circ q \) 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 \( I = \text{vars}(p) \cap \text{vars}(q) \), are equal.

\[ p|q = \{ b \cup c | (b, c) \in p \times q, I = \text{vars}(p) \cap \text{vars}(q), b|_I = c|_I \} \]

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 com-
position of simultaneous equations \( x := f(y, z) \) or \( x := y \# z \)
over input signals \( y, z \) and output signals \( x \). A signal \( x \) is a possibly infinite flow of values \( v \in V \) sampled at a clock noted \( \hat{x} \).

\[ P, Q ::= x := y \# z \mid P \mid \{P\} \]

An equation \( x := y \# z \) denotes a relation between the input sig-
als \( y \) and \( z \) and an output signal \( x \) by a combinator \( f \). An equation
is usually a ternary or infix relation noted \( x := y \# z \) but it can in general be an \( m + n \)-ary relation noted \( (x_1, \ldots, x_m) :=
\sum_{i} y_i \# z_i \). SIGNAL requires three primitive combinators to per-
form delay \( x := y \text{?init} v \), sampling \( x := y \# z \), and merge \( x = y \# z \). The equation \( x := y \text{?init} v \) initially de-
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 \( x := y \text{?init} v \) are synchronous: they share the same set of tags \( t_1 \), \( t_2 \), \( t_3 \)...
Initially, at \( t_1 \), the signal \( x \) takes the declared value \( v \) and then, at tag \( t_n \),
the value of \( y \) at tag \( t_n - 1 \).

\[\begin{align*}
  (x := y \text{?init} v) \]

\[\begin{align*}
  x &\overset{t_1}{\rightarrow} y &\overset{t_2}{\rightarrow} z &\overset{t_3}{\rightarrow} \\
\end{align*}\]

The equation \( x := y \text{?init} \) 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 \( (t_2, t_3) \) then \( x \) holds its value
whether \( z \) is present (at \( t_2 \)) or not (at \( t_3 \)).

\[\begin{align*}
  (x := y \text{?init} z) \]

\[\begin{align*}
  x &\overset{t_1}{\rightarrow} y &\overset{t_2}{\rightarrow} z &\overset{t_3}{\rightarrow} \\
\end{align*}\]

The equation \( x := y \# z \) defines \( x \) by \( y \) when \( y \) 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 \( v_1 \) at \( t_1 \) 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 \( y_2 \rightarrow x_2 \rightarrow z_2 \).

\[\begin{align*}
  (x := y \# z) \]

\[\begin{align*}
  x &\overset{t_1}{\rightarrow} y &\overset{t_2}{\rightarrow} z &\overset{t_3}{\rightarrow} \\
\end{align*}\]

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. \( y := x \equiv \lnot y := (x = x) \)). Referring to the polychronous
model of computation, it represents the set of tags at which the Boolean signal
\( x \) is present and true (i.e. \( y := x \equiv \lnot y := 1 \) when \( x \)). The clock
[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).

\[ e ::= \text{true} \mid [\text{not } x] \mid e + e' \mid e - e' \mid e^* e' \mid 0 \]

A clock constraint \( E \) is a \texttt{SIGNAL} process. The constraint \( e^* e' \) synchronizes the clocks \( e \) and \( e' \). It corresponds to the process \((x:=(e+e'))/x\). Composition \( E|E' \) 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 \( e \).

\( E ::= () \mid e^* e' \mid e' < e' \mid x \rightarrow y \) when \( e \mid E | E' | E/x \)

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

\[
\begin{align*}
x & ::= y\text{init } v ::= \text{true} \mid y \\
x & ::= y \text{when } z ::= x = y [x] \mid y \rightarrow x \text{ when } z \\
x & ::= y \text{def } z ::= x = y^* + z \rightarrow x \text{ when } (z - y) \\
\end{align*}
\]

\[ P : E \mid Q : E' \]

\[ P | Q : E | E' \]

\[ P / x : E/x \]

\textbf{Figure 7: Clock inference system}

\textbf{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-clock synchronous specification by a technique of hierarchization, proposed in [1]. It can be outlined by considering a simple \texttt{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 \texttt{SIGNAL} operation defined by:

\[
x ::= y \text{cell } z \text{ init } v \text{ def } m ::= x\text{init } v \\
\]

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 \( b \).

\[
\text{process buffer = (? i ! o) } \\
\quad | \text{ alternate (i, o) } | o ::= \text{current (i)} \\
\text{where} \\
\text{process alternate = (? i, o !) } \\
\quad | \text{ zb := b\text{init }true} \\
\quad | \text{ b ::= not zb} \\
\quad | \text{ o ::= when not b} \\
\quad | i ::= \text{when b} \\
\quad | /, b, zb; \\
\text{process current = (? i ! o) } \\
\quad | \text{ zo := i cell } \text{ o init false} \\
\quad | \text{ o ::= zo when } \text{o} \\
\quad | /, zo; \\
\]

\textbf{Figure 8: Polychronous specification of a buffer}

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.

\[
\begin{align*}
\{ c_b ::= b & | c_o ::= o \\
b ::= zb & | i -> zo \text{ when } i \\
zb ::= zo & | zb -> b \\
c_i ::= i & | zo -> o \text{ when } o \\
c_o ::= \text{not } b & | /, zb, zo, c_b, c_o, c_i, b; \\
\}
\]

\textbf{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 \( c_b, c_o \), and \( c_i \) are calculated from \( zb \). At the sub-clock \( b \), the input signal \( i \) is read. At the sub-clock \( c_i \) the output signal \( o \) is written. Finally, \( zb \) is determined. Notice that the sequence of instructions follows the scheduling constraints determined during clock inference.

\[
\text{buffer_iterate ()} \\
\quad | \text{ if (c_o) } \\
\quad | \quad | b ::= !zb; \\
\quad | \quad | c_o ::= !b; \\
\quad | \quad | w_buffer_o(o); \\
\quad | \quad | if (b) \{ \\
\quad | \quad \quad | \text{ if (!r_buffer_i(i)) } \\
\quad | \quad \quad \quad | \text{return } \text{FALSE;} \\
\quad | \quad \quad \quad | \text{return } \text{TRUE;} \\
\}
\]

\textbf{Figure 10: Buffer code generation}

\textbf{Some More Concrete Syntax.} In addition to the core syntax of \texttt{SIGNAL} presented so far, we make extensive use of process declarations and partial equations for the purpose of modeling our case study. In \texttt{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 \( x ::= \text{defaultvalue} \) 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 \( v \):

\[
x ::= x_1 \text{ when } e_1 \mid \ldots \mid x ::= x_n \text{ when } e_n \mid x ::= \text{defaultvalue} \\
\]

The \texttt{SIGNAL} compiler processes this definition by first checking the clock expressions \( e_1, \ldots, e_n \) mutually exclusive and then handling the definition as the equivalent equation: \( x ::= (x_1 \text{ when } e_1) \ldots (x_n \text{ when } e_n) \text{ default } v \). The declaration of a process \( P \) of name \( f \), input signals \( x_1, \ldots, x_m \), output signals \( x_{m+1}, \ldots, x_n \) is noted \texttt{process } \( f ::= (x_1, \ldots, x_n) \{ P \} ; \). Once declared, process \( f \) may be called with its actual parameters \( y_1, \ldots, y_n \) by \((y_{n+1}, \ldots, y_n) ::= (y_1, \ldots, y_m) \) and behave as \( P \) with \( x_1, \ldots, x_n \) substituted by \( y_1, \ldots, y_n \). 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 \texttt{SIGNAL} by declaring its interface and by declaring an abstraction \( E \) of its behavior, which consists of scheduling and clock constraints.

\[
\text{process } f ::= (x_1, \ldots, x_n) \text{ spec } (\{ E \}) \\
\text{pragma C CODE}^{\text{CE}} &<& xx = f(\&x_1, \ldots, \&x_n)" \\
\text{end pragma;}
\]
4. METHODOLOGY AND TOOLS

Our modeling and verification methodology starts off with a SystemC 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 SystemC programs are written in C++, they can contain very complex constructs, which make it difficult to obtain a corresponding SIGNAL 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 SIGALI [14] can be used. Given a formal model of a system in SIGNAL, SIGALI can verify formal properties of the model. It is an interactive tool specialized on algebraic reasoning in Z/3Z 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 Z/3Z 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).

Figure 11: Structure of the FIR filter with testbench

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 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 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.

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.

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.

Figure 13: Clock and scheduling relations for the FIR

Figure 13 displays this information for the FIR in the form of a transition system (STS, as in [18]). It consists of propositions on clocks $x_i$ guarded by block input clocks $x_iL$ to for implications $x_iL \rightarrow x_i$ and of proposition on state transitions of the form $e \rightarrow x_i$ to mean that if $e$ is present then $x_i$ 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, \( x_0 \Rightarrow \text{shift} \) means that whenever block L0 is entered, the signal \( \text{shift} \) 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 \( x_{L9} \Rightarrow x'_{L9} \), 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 small loops, such as the one between L1 and L2, representing the manipulation of an array of values.

\[
\begin{align*}
ex_{L0} & \Rightarrow \text{input_valid} \\
ex_{L3} & \Rightarrow \text{clk_pos} \\
ex_{L4} & \Rightarrow \text{out_ready}
\end{align*}
\]

**Figure 14: Control flow of the FIR filter**

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 input value. Then the calculations are executed and it subsequently waits for the next positive clock edge before resuming execution at block L3.

\[
\begin{align*}
ex_{L1} & \Rightarrow \text{sample} \\
ex_{L3} & \Rightarrow \text{coefs} \\
ex_{L9} & \Rightarrow \text{out_ready}
\end{align*}
\]

**Figure 15: Data Flow of the FIR**

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 \( a \rightarrow b \) 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 variable 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 \( x_{L1} := \text{xL1}_1 \text{init} \text{false} \). Here \( \text{xL1} \) represents the next value and \( \text{xL1} \) 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:

\[
\begin{align*}
\text{out_ready} & := \text{false when xL3} \\
\text{out_ready} & := \text{true when xL9}
\end{align*}
\]

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:

\[
\begin{align*}
\text{out_ready} & := \text{false when xL3} \\
\text{out_ready} & := \text{true when xL9} \\
\text{default} & := \text{true when xL9}
\end{align*}
\]

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 SIGNAL of the POLYCHRONY workbench. Usually, it is not necessary to transform the whole model to a binary form as we might not be interested 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 \(\). This is the input file for SISIALI. It contains a model of the SIGNAL program expressing polynomial differential equations, the data structure manipulated by SIGNAL.

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.

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

**Figure 17: Verification of Properties using SISIALI**

**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.

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.

```plaintext
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 :=
| default (xfir$ init true)
| xL3 :=
| default true when xL9
| default true when (clk1
| default true when not (clk1 = clk1$)
| default false when xL9a
| default when not (clk1 = clk1$)
| default false
| xL9a :=
| default (clk1 = clk1$)
| default false
| xL9 :=
| default true when xL9
| default true when (not clk1)
| default (not clk1$)
| default false when xL9a
| default false when xL9
| default (xfir$ = xL3 = xL9 = clk1
| " = xL3a = xL9a = out\_ready
| ) where integer i;
| boolean xL3, xL3a, xL9a, xfir, xL9,
| xL9a, xL9a;
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 $\pi$-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 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].

8. REFERENCES


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
xL2 :=: when cond2 when xL1 default false
| xL2 :=: xL0$
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 == xL2 == xL3 == sample
|) where integer send_value; boolean cond1, cond2, xstim, xL0, xL1, xL2, xL3$
end;

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

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