SIGNAL Clock Calculus – An overview

Alexander Steen*

Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

Abstract: The compilation process of the synchronous programming language SIGNAL gives executable and efficient code that matches the input specifications. Several techniques are necessary for analyzing the given specifications, solving equation system of *clocks* and generating efficient code that controls the data-flow of the program. The collection of these techniques is called *clock calculus*. This paper surveys the principles of the compilation process and gives an brief overview over the implementation.

Contents

*a.steen@fu-berlin.de

1 Introduction

Synchronous programming languages allow automatic code generation out of specifications. While this enables the creation of safe executable code, the compilation process has been intensively developed with focus on code efficiency. Various synchronous languages have emerged, such as LUSTRE, ESTEREL, SIGNAL and SCADE. The first two languages generate automatons out of the input specifications, while SIGNAL and SCADE generate a sequential control structure.

This paper surveys the compilation process of signal, the so called *clock calculus*: A set of techniques for the analysis of given data-flow programs, synchronization synthesis, analysis of feasibility and code extraction. This paper will focus on the main ideas of the clock calculus and give intuitive explanations for the approach of the SIGNAL compilation process.
Since the implementation of the code-generations algorithms is very technical, only a coarse-grained overview is be given.

2 Preliminaries

This section recalls the building blocks of the synchronous programming language SIGNAL. It is assumed that the basic ideas of general synchronous notions are well-known to the reader; there are several introductions to different approaches [Hal10].

For the SIGNAL case, this section firstly presents *signals* and *clocks* [ABG94] as well as some examples. Based on the given definitions, the language primitives of SIGNAL are explained.

2.1 Signals

The variables of SIGNAL, called *signals*, are sequences of values of some domain $\mathcal{D}$ (e.g. Boolean, integer, ...). A signal $X = (x_i)_{i \in I}$ consists of values $x_i \in D$, where for each $i \in I$, X is said to be *present* at instant i. If X is not present at some instant i, X is called *absent* and carries no value (written $X = \perp$).

The set of all instants U, and therefore also $I \subseteq U$, is some countable set with total ordering $\leq$.

2.2 Clocks

The *clock* of a signal X is the set of all instants I at which X is present. This means that for the signal $X = (x_i)_{i \in I}$ the clock of X is exactly I.

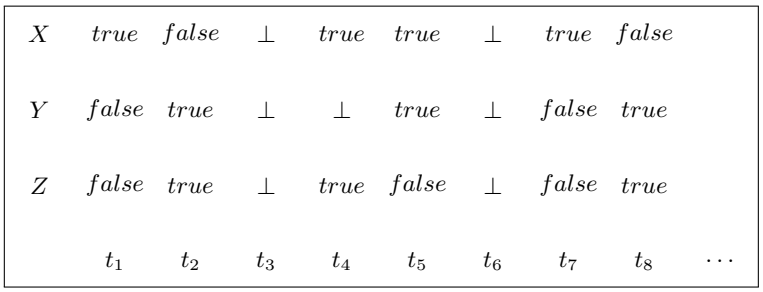

Figure 1: Clocks of signals X, Y, Z

In general, the clock of some signal X is denoted $\widehat{X}$. Since $\widehat{X}$ does not contain any information about the values of X, multiple signals can share the same clock if they are synchronous (i.e. present at the same set of instants). It follows directly that the property of synchrony induces an equivalence relation on the set of signals, $\widehat{X}$ being associated to the equivalence class of X with respect to the synchrony relation.

For any Boolean signal C, the expression $[C]$ denotes the clock that contains the instants of $\widehat{C}$ at which C carries the value $true$. Analogously, $[\neg C]$ denotes the clock that contains the instants of $\widehat{C}$ at which C carries the value $false$.
Hence, $[C] \cup [\neg C] = \widehat{C}$ and $[C] \cap [\neg C] = \mathbb{O}$, where $\mathbb{O}$ is the empty clock.

The relations over clocks can be formalized using the clock algebra $\mathcal{H}$ given by:

$$\mathcal{H} = (U, \cup, \cap, \setminus, \mathbb{O})$$

where U denotes the set of all instants and $\mathbb{O}$ the empty clock.

Example Consider the signals X, Y, Z of Boolean given by figure . It is apparent that (with correct continuation) $\widehat{X} = \widehat{Z} \subseteq \widehat{Y}$. ⌟

2.3 The SIGNAL kernel

This section recalls the basic statements of SIGNAL , statements being equations on signals. Amongst others, Benveniste et al. provided a complete description of the SIGNAL language and its formal semantics [BGJ91].

Functional expressions Function applications, also standard infix operators, can be used. $X := f(X_1, X_2, \ldots, X_n)$. The resulting signal X is present if and only if all the argument signals X_i are present. Then, at all instants $i \in \widehat{X}$, the value x_i of X is $x_i = f(x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(n)})$, where $x_i^{(j)}$ is the value of the signal X_j at instant i.

Delay Signals can be delayed by n instants using the delay operator:
$X := Y$ `$` n `init` $\overrightarrow{v}$, where $\overrightarrow{v} = (v_1, v_2, \ldots, v_n)$ is a vector of length n. It defines a signal $X = (x_i)_{i \in \widehat{Y}}$ which carries the values $v_1 \ldots v_n$ for the first n instants, and then the value of Y shifted by n values.

Downsampling The process $X := Y$ `when` C, where C is a Boolean signal, defines the signal X with $\widehat{X} = \widehat{Y} \cap [C]$. When X is present, it carries the same value as Y at that instant.

Deterministic merge Two signals Y, Z can be merged into a signal X by
$X := Y$ `default` Z. The resulting signal X carries the value of Y if Y is present and the value of Z otherwise. If both Y and Z are absent, X is also absent.

Parallel composition Processes can be executed in parallel with the composition operator `|`. The parallel composition of multiple processes defines a system of equations such that all equations of the composed processes hold at all instances.

3 Clock calculus

The compilation process of SIGNAL involves several steps of program analysis often called *clock calculus*: The synchronizations of each process (means of its clocks) are used to represent the *control* of the system; an equation system of relations over clocks. Additionally the data-flow of each process is analyzed, identifying the data-dependencies between multiple signals (e.g. *signal X must be calculated before signal Y can be assigned*). This section introduces the data structures and algorithms used by the SIGNAL compiler to infer information about the control and the dependencies of a program.

Generation of single loop code that matches the input specification and respects the stated synchronization constraints follows the analysis of the clock calculus.

3.1 Boolean equation systems

Each primitive SIGNAL process inherently defines a relation over the clocks involved. Consequently, a system composed out of multiple processes describes a system of these relations, which is, an equation system of clock variables that describe the synchronizations of the system. Note that these clock variables do not only consist of clocks of variables, but also of clocks synthesized by undersampling.
As an example, consider the statement $X := Y$ `when` C. Its synchronization can be described using the clock algebra of section 2.2 by the equation $\widehat{X} = \widehat{Y} \cap [C]$.

For each primitive SIGNAL process, the clock synchronizations are given by table 1.

Process	**Synchronisation**
$X := f(X_1, X_2, \ldots, X_n)$	$\widehat{X} = \widehat{X_1} = \ldots = \widehat{X_n}$
$X := U$ `when` C	$\widehat{X} = \widehat{U} \cup [C], \begin{cases} [C] \cup [\neg C] = \widehat{C} \\ [C] \cap [\neg C] = \mathbb{0} \end{cases}$
$X := U$ `default` V	$\widehat{X} = \widehat{U} \cup \widehat{V}$
$X := Y$ `$` n `init` $\overrightarrow{v}$	$\widehat{X} = \widehat{Y}$

Table 1: Synchronizations of SIGNAL processes [ABG94]

These equations can be identified by equivalent Boolean equations: Since any clock specifies the presence or absence of a signal, is appears intuitive to encode this with a Boolean variable. The translation of the set operations used by the clock algebra into the propositional calculus is given by the following morphism:

$$(U, \cup, \cap, \setminus, \mathbb{0}) \mapsto (true, \vee, \wedge, (x, y) \mapsto x \wedge \neg y, false)$$

Since for every Boolean signal the clocks $[C]$ and $[\neg C]$ also depend on the *value* of C, we need to encode this three-value logic $(\{true, false, \perp\})$ into a Boolean logic using two variables [Neb04].

3.2 Data-dependency

The analysis of the data-flow of an process can reduced to the calculation of the data dependencies of that process. Intuitively, a dependency can be understood as the necessity that the value of some signal must be known in order to calculate the value of some other signal. A program is compiled into a graph that described the data dependencies in the following sense: The edge

$$X \xrightarrow{h} Y$$

means that at each instant of the clock h, the value of X must be computed before the value of Y can be determined; we say that Y *depends on* X [ABLG95]. Table 2 lists the dependencies of each SIGNAL process.

3.3 Code generation

In order to generate executable code, the dependency graph created out of the program specification is transformed into a control structure that handles all the reactions of the system. The code is generated by the following scheme:
Since variables only change their value during reactions, every access to variables is

Process	**Dependency**
for each signal X	$\widehat{X} \xrightarrow{\widehat{X}} X$
for $X := f(X_1, X_2, \ldots, X_n)$	$X_i \xrightarrow{\widehat{X}} X$, for all $1 \leq i \leq n$
for $X :=$ U `when` C	$U \xrightarrow{\widehat{U} \cap [C]} X$
for $X :=$ U `default` V	$U \xrightarrow{\widehat{U}} X \xleftarrow{\widehat{V} \setminus \widehat{U}} V$
for $X :=$ Y `$` n `init` $\overrightarrow{v}$	none

Table 2: Data dependencies of SIGNAL processes [ABG94]

Listing 1: Control of equation (1)

```
if b_C then
  b_[C] := C
  b_X := b_Y and b_[C]
  if b_X then
    X := Y
  end
end
```

Listing 2: Control of equation (2)

```
1 b_X := b_Y or b_Z
2 if b_X then
3   if b_Y then
4     X := Y
5   end
6   if b_Z and not b_Y then
7     X := Z
8   end
9 end
```

Figure 2: Example: The control of SIGNAL process

guarded by a test over its clock, enforcing the presence of the clock [Neb04]. The translation of clock equations into the propositional calculus, see section 3.1, is utilized to encode the actual testing into Boolean values. This described pattern can easily be recognized in the following two examples. $X := Y$ `when` C and $X := Y$ `default` Z. The synchronizations of these statements are given by equation (1) and (2) respectively. Their code is materialized by Listing 1 and 2 respectively shown in figure 2.

$$\widehat{X} = \widehat{Y} \cap [C] \tag{1}$$

$$\widehat{X} = \widehat{Y} \cup \widehat{Z} \tag{2}$$

Since the code contains actual tests over the Boolean values associated to the clocks, they must be chosen a definition that matches their synchronization specification. This is one of the main goals of the clock calculus. As we have seen in the examples of figure 2, some clocks can be inferred using a combination (intersection, union, difference) of other clocks (listing 2, line 1). Those rewritings can be extracted using the synchronizations of the processes presented in section 3.1.

In general, the transformation is not as easy as presented here, the problem is considered NP-hard [ABLG95]. The general approach for solving Some clocks cannot be rewritten in the above sense, they are *free variables* an considered as input that must be provided by the execution environment.

3.4 Solving the equation system

From the previous section it is apparent that an efficient method for checking the presence or absence of clocks is required. For that purpose, the Boolean equation system to be solved is extracted from the system by applying the rules of table 1. The general approach for solving this equation system is given by *triangularization*:
The equation system is transformed into a system of so called *directed* definitions, that is, a system of equations of the form $a = b \circ c$, where a, b, c are clocks and $\circ$ is some operator on clocks.

The main benefit of this structural representation is that the dependency graph of the program contains no cycles. Finding this representation involves numerous problems:

- Multiple definitions: If the equation system contains more than one equation with clock a on the left side, the equality of their respective right sides must be proven.
- Cycles: If the equation system contains cyclic dependencies, they have to be eliminated.
- Complex relations: If equations are not of the form $a = b \circ c$, e.g. given by $a \circ b = c \tilde{\circ} d$, they need to be transformed into triangular form.

The SIGNAL compilation process involves solving this problems, mainly by using a rewriting system plus some heuristics, in order to achieve triangular form. But since this problem is hard, the compiler is not complete: If no appropriate rewriting rule can be applied, an input program may be refused although the system could be solved. In any case, if some equalities cannot be proven or if some cycle cannot be eliminated, an input program is considered *temporal incorrect* and refused [ABG94].

The generated control structure is optimized by using extra knowledge about clock inclusions and clock equivalences. This allows to reduce the number of clocks and also reduces the number of tests over the clock variables. Additionally, tests can be nested efficiently by factorizing.

Example Consider the left equation system (equations (3) – (7)) as an example for an equation system for a SIGNAL system by Amagbegnon et al. [ABLG95]. As mentioned above, we now try to build a system of directed equations. The equations (3) and (5) – (7) translate directly into equations (8) – (12) by reordering and using equalities. Equation (13) is achieved by substitution of $\widehat{C'}$ by $\widehat{C}$ in (4).

$$\begin{cases} \widehat{C} = \widehat{C'} & (3) \\ \widehat{C'} = [D] \cup [C_1] \cup \widehat{C} & (4) \\ [C] = \widehat{C_1} = \widehat{C_2} & (5) \\ [\neg C] = \widehat{D} & (6) \\ \widehat{C_3} = \widehat{C_1} = \widehat{C_2} & (7) \end{cases} \qquad \begin{cases} \widehat{C'} = \widehat{C} & (8) \\ \widehat{C_1} = [C] & (9) \\ \widehat{C_2} = [C] & (10) \\ \widehat{C_3} = [C] & (11) \\ \widehat{D} = [\neg C] & (12) \\ \widehat{C} = [D] \cup [C_1] \cup \widehat{C} & (13) \end{cases}$$

All equations but the last are in triangular form. To transform equation (13) into triangular form, we use that $[C_1] \subseteq \widehat{C_1} \overset{(9)}{=} [C] \subseteq \widehat{C}$. This way, we can simplify $[C_1] \cup \widehat{C}$ to $\widehat{C}$. Since $[D] \subseteq \widehat{D} \overset{(12)}{=} [\neg C] \subseteq \widehat{C}$, equation (13) yields $\widehat{C} = \widehat{C}$, so that it can be removed from the equation system. The resulting equation system has triangular form. The clock $\widehat{C}$ is a free variable of the system. ⌟

The compilation process must apply those rewriting rules effectively to create the triangular system. The next section gives a brief overview over the technical details of the implementation.

3.5 Hierarchical representation of equations

In order to efficiently apply rewriting and keep track of the triangular structure of the equation systems, a tree-based representation of the equations is used.

For a Boolean signal C, the known partition $[C] \subseteq \widehat{C} \supseteq [\neg C]$ is represented by a *partition tree*, whose edges are the set inclusion relations. An example is given by figure 3. The representation of the whole equation system of synchronizations starts with representing all directed definitions by a forest of clock trees. Then, two clock trees are iteratively fusioned by inserting one tree into another:
Let T, T' be two clock trees with roots a, b (respectively), where b is defined by the directed equation $b = c \circ d$, and c and d are subtrees of T. Then the fusion of T' into T is applied by adding the tree T' to the immediate children of the least common ancestor of c and d in T.
This way, the subtree T', which is defined by the operands c and d, is placed directly under the least common ancestor of these operands. This preserves the structural property of triangularization and gives credit to clock inclusion relations which yield more efficient nested if-tests [ABG94].
Intuitively, children in this trees are subset clocks of its parent.

The algorithm repeatedly tries to rewrite the equations to that they match the criteria of the fusion step and then executes the fusion until this cannot be done anymore. The implementation of the rewriting system was described by Besnard [Bes92].
The resulting tree-based representation of the equation systems over clocks are optimized in the sense that the insertion step during the fusion chooses a parent with greatest depth.

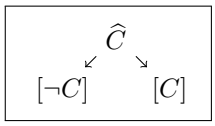

Figure 3: Partition tree of Boolean signal C

This algorithm is described by Amagbegnon et al. [ABG94]. It involves the transformation of the tree of clocks by a tree of BDDs and then applying factorization on the resulting Boolean functions.

4 Conclusion

In this paper the SIGNAL clock calculus is overviewed: The set of techniques for transforming a program of specifications into sequential single-loop code is outlined. It is explained how equation systems of synchronization constraints are extracted out of the code and how they are encoded into a Boolean equation system.

A coarse grained description of algorithms is given, involving the application of rewriting rules and the representation of equation systems by trees of clocks. In particular, tests over clocks are materialized by tests over Boolean. Due to tree-based encoding of the equation systems, these tests are generated efficiently: Redundant tests are avoided by nesting.

References

[ABG94] Tochou Amagbegnon, Loc Besnard, and Paul Le Guernic. Arborescent Canonical Form of Boolean Expressions. Technical report, 1994.

[ABLG95] Pascalin Amagbégnon, Loïc Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language SIGNAL. *SIGPLAN Not.*, 30(6):163–173, June 1995.

[Bes92] L. Besnard. *COMPILATION DE SIGNAL: HORLOGES, DEPENDANCES, ENVIRONNEMENT*. 1992.

[BGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103 – 149, 1991.

[Hal10] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Springer-Verlag, Berlin, Heidelberg, 2010.

[Neb04] Mirabelle Nebut. An Overview of the Signal Clock Calculus. *Electron. Notes Theor. Comput. Sci.*, 88:39–54, October 2004.