A Canonical Form for Affine Relations in Signal

Irina Smarandache, Paul Le Guernic

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Canonical Form for Affine Relations in SIGNAL

Irina Smarandache, Paul Le Guernic

N ° 3097

February 1997

THÈME 2

ISSN 0249-6399

INRIA
RENNES

A Canonical Form for Affine Relations in SIGNAL

Irina Smarandache, Paul Le Guernic*

Thème 2 — Génie logiciel
et calcul symbolique
Projet EP-ATR

Rapport de recherche n° 3097 — February 1997 — 19 pages

Abstract: In this paper we present affine transformations as an extension of the SIGNAL language for the specification and validation of real-time applications. A SIGNAL program is a system of equations which specify dependencies between program data and synchronization constraints on clock variables. In order to test if a program is functionally safe, the SIGNAL compiler resolves the clock constraints and verifies that the data dependency graph contains no cycles. By means of the new transformations, *affine relations* can be defined between clock variables and it gets necessary to enhance the compiler with facilities for the resolution of synchronization constraints on these clocks. To tackle these constraints, we propose an extension of the compiler based essentially on a canonical form of the affine relations. This extension removes some of the limits of the existing compiler and enlarges the range of applications that can be validated using SIGNAL.

Key-words: SIGNAL, real-time languages, clock calculus, affine transformations, canonical form

(Résumé : tsvp)

Paper accepted to the Fourth International AMAST Workshop on Real-Time Systems, Concurrent and Distributed Software (ARTS'97).

* Irina.Smarandache@irisa.fr, Paul.Leguernic@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 02 99 84 71 00 – Télécopie : (33) 02 99 84 71 71

Une Forme Canonique pour des Relations Affines en SIGNAL

Résumé : Cet article présente les transformations affines comme une extension de SIGNAL pour la spécification et la validation d'applications temps-réel. Un programme SIGNAL est un système d'équations qui représente des dépendances entre des données et des contraintes de synchronisation entre des variables d'horloges. Afin de tester si un programme est correct, le compilateur de SIGNAL résoud les contraintes de synchronisation et vérifie que le graphe des dépendances de données est acyclique. Les nouvelles transformations permettent de définir des *relations affines* entre des variables d'horloges et il s'impose de fournir les outils nécessaires pour la vérification des contraintes de synchronisation entre ces horloges. Une extension du compilateur basée sur une forme canonique des relations affines a été proposée : elle élargi la capacité de preuve du compilateur actuel et augmente le nombre d'applications qui peuvent être spécifiées et validées avec SIGNAL.

Mots-clé : SIGNAL, langages temps-réel, calcul d'horloges, transformations affines, forme canonique

1 Introduction

Real-time systems, and more generally reactive systems [1], are in continuous interaction with their environment. Therefore, they must respond *in time* to external stimuli. Moreover, real-time systems must be safe, thus one would wish to prove their correctness. Response time and safety are two important aspects in the design of real-time applications.

In order to deal with these aspects, a new family of languages has emerged – *the synchronous languages* [2]. The hypotheses of synchronous programming are: 1. All actions (communications and calculi) in a system are considered to be instantaneous; 2. Two or more actions can take place at the same instant (they are simultaneous). Thus, from the point of view of the temporal properties of a system, only succession and simultaneity of instants are of interest, their exact time values have no signification.

SIGNAL [3, 4] is a synchronous language developed for the specification, validation and implementation of real-time systems. Due to the synchronous approach, the verification of the logical properties of an application is possible before taking into consideration the physical properties (time, resources) of a particular implementation. The notion of *clock* is important in SIGNAL: with each SIGNAL variable is associated a clock that summarizes the temporal properties of the variable. A SIGNAL program is a system of equations which specify synchronization constraints on clocks and dependencies between values of variables (data). The SIGNAL compiler resolves these equations and verifies if the control of a program is functionally safe. Currently, proving the correctness of a SIGNAL program with respect to its clock constraints reposes on boolean equation resolution methods which appeared sufficient for the validation of real-time systems where control is an important aspect [5, 6]. Most of the real-time signal processing applications contain important numeric computations; in order to treat such applications, we define *affine transformations* on clock variables and we extend the existing compiler in order to treat synchronizations on clocks defined by affine transformations. The extended compiler has more powerful proof facilities which increase the specification and validation power of the SIGNAL language, either if SIGNAL is employed alone or in conjunction with other languages in a data flow framework.

Large grain data flow programming [7] has been considered an intuitive and convenient paradigm for describing digital signal processing systems, i.e. systems with important numeric treatments. Nevertheless, when such systems must respect real-time constraints, the runtime overhead induced by dynamic scheduling has to be eliminated. Data flow models and languages have been adapted to tackle this aspect. In [8, 9] *Synchronous Data Flow* (SDF) graphs are particular data flow graphs in which the amount of data produced and consumed by a data flow node is specified a priori for each input and output. This property makes SDF graphs good candidates for verifying statically the existence of sequential and parallel schedules: consistency properties of an SDF graph can be checked at compile time. Moreover, in [10] the SDF model is enlarged in order to tackle with a larger range of signal processing applications by introducing boolean conditions in a generalized SDF model.

Another example of restricted data flow graphs is that of *reduced dependence graphs* which are specifications of systems in terms of periodic acyclic precedence graphs, where only one period is illustrated, and its dependence on previous periods is specified by indexing [11]. Reduced dependence graphs have optimal schedules which can be found at compile time. However, their application domain is more restricted than for SDFs: reduced dependence graphs are generally used to describe regular iterative algorithms, which can be systematically mapped onto processor arrays.

Based on this approach, the ALPHA language has been developed as a functional language based on the formalism of affine recurrence equations [12]. ALPHA allows the description of regular algorithms, like multiplication of matrices or vectors, and the transformation of programs for optimization. A transformational environment has been developed around ALPHA for the synthesis of parallel and regular architectures [13, 14].

Initially designed for use in the specification and validation of real-time signal processing systems, SIGNAL is based currently on a control-oriented formalism [6]. However, due to its data flow flavor, SIGNAL has been used in the specification of periodic synchronous VLSI systems modeled by SDF graphs [15, 16]. The modeling and simulation of SDF graphs in SIGNAL have pointed out two problems: the need to define a communication interface between SDF nodes and the important time and space resources necessary to verify if an application is functionally correct. Moreover, SIGNAL does not treat adequately computations on regular data like matrices or vectors. Thus, SDF nodes describing treatments on multidimensional data have been expressed by external C processes. In order to obviate these problems and due to the complementary properties of the languages SIGNAL and ALPHA, we have projected a hybrid framework [17] in which SIGNAL and ALPHA are combined for the specification, validation and implementation of applications. ALPHA provides optimal scheduling on hardware or software for regular iterative algorithms, while SIGNAL specifies the general control context in which ALPHA algorithms execute. For the modeling and validation of the SIGNAL-ALPHA cooperation, affine transformations on SIGNAL clock variables are introduced. In order to treat synchronization constraints on

clocks obtained by affine transformation, we must extend the current proof system implemented in the SIGNAL compiler.

The paper is organized as follows: in Sect. 2 we introduce the SIGNAL language and its compiler and present the context of our work. In Sect. 3, the new features introduced in SIGNAL are described, namely the affine transformations and a canonical form of the affine relations induced thereby. We also present the additional proof system and its main properties. Finally, in Sect. 4, we conclude with applications and future directions of our work.

2 The SIGNAL language

SIGNAL as a synchronous and equation-based language, has proved very useful for programming correct real-time systems. The synchronous approach enables the abstraction of the specification from the details of a given implementation, offering the possibility to verify logical properties and to guarantee the functional safety of an application [2]. Moreover, programming in SIGNAL consists simply of writing systems of equations on program variables [4]. The SIGNAL compiler resolves these systems and produces executable code whenever the program is correct.

A variety of tools exist around the SIGNAL language and its compiler. They constitute the SIGNAL environment. There are a *Graphical User Interface* for high-level system design [18], C and Fortran code generators to produce code for functional simulators, an intermediate format generator for the access to formal verification tools [19] and a VHDL code generator to access hardware synthesis tools [20]. The basic data structure of the SIGNAL environment is the *synchronized data-flow graph* (SDG) which is constructed during the compilation from the clock equations and elementary data dependencies graphs associated with each SIGNAL equation [5]. The construction of the SDG proceeds in parallel with the verification if the program is deterministic, has no cycles, if no constraints are imposed on its inputs and if the coded functional properties are guaranteed.

From all operations performed by the SIGNAL compiler, we will particularly insist on those methods involved in the resolution of the synchronization constraints on clocks, i.e. the *clock calculus*. Therefore, we give a more detailed presentation of SIGNAL and its clock calculus and we discuss an example of a SIGNAL program in detail. We conclude this section with a presentation of the context of our work and an introduction to the affine transformations in SIGNAL.

2.1 Basic concepts

Before introducing the basic language constructs and the clock equations, we give some definitions. The SIGNAL language operates on variables called *signals* which are infinite sequences of values, each value being indexed by an integer (positive) temporal index. The values of the index domain define the logical instants where the signal is present. A signal can be present or absent at a given logical instant from the positive integer domain. We define the *clock* of a signal as the set of instants where the signal is present (and has a value). Two signals have the same clocks if they are present (respectively absent) at the same instants.

The SIGNAL kernel is defined by a small number of elementary processes (also called statements). Each process has a formal semantic that introduces a clock equation and a data dependencies graph on the signals involved [21, 22]. We present briefly the elementary processes and the extensions of SIGNAL in Table 1 (for a more detailed presentation refer to [4, 19]). The clock equations associated with the statements are shown in the right-hand column of Table 1.

By $\hat{X}$ we denote the clock of the signal X. The clock $[C]$ defines the instants where the boolean-valued signal C is present and has the value *true* (similarly $[\neg C]$ is the clock of the *false* values of C). The classical operations on sets $(\wedge, \vee, \setminus)$ can be performed on clocks. The inclusion relation between sets defines a partial order on the set of clocks; moreover the clock *poset*[1] is a boolean lattice [23]. In order to resolve the clock equations of a SIGNAL program, we use, in addition to the well-known properties of the boolean lattice, the special relation that exists between $[C]$ and $[\neg C]$, given by the following *partition* equations:

$$[C] \vee [\neg C] = \hat{C} \qquad [C] \wedge [\neg C] = \emptyset$$

where $\emptyset$ is the empty set of instants. These equations state that the pair $([C], [\neg C])$ defines a *partition* of the clock $\hat{C}$.

[1] A *poset* is a set with a partial order on it.

	Syntax	Description	Clock calculus
Elementary processes			
stepwise extensions	X := A op B	op: an arithmetic, relational or boolean operator	$\hat{X} = \hat{A} = \hat{B}$
delay	ZA := A $ N	ZA: the N^{th} past value of A	$\hat{ZA} = \hat{A}$
down-sampling by boolean condition	Y := X when C	Y: X when C is present and true	$\hat{Y} = \hat{X} \wedge [C]$
merge with priority	Z := X default Y	Z: X if X is present; else Y if Y is present; else Z is absent	$\hat{Z} = \hat{X} \vee \hat{Y}$
process composition	(| P | Q |)	The SIGNAL processes P and Q share their identically named signals	
process restriction	P/a	The signal a is not visible to the exterior of the process P	
SIGNAL extensions			
	X := when C	The clock of the true instants of C	$\hat{X} = [C]$
	Y := ^X	The clock of X	$\hat{Y} = \hat{X}$
	A ^= B	Explicit synchronization constraint: the clock of A is equal to the clock of B	$\hat{A} = \hat{B}$

Table 1: **Basic SIGNAL constructs and extensions .**

2.2 The SIGNAL compiler

The SIGNAL compiler resolves the clock equations using an algorithm that synthesizes the most frequent clock *(master clock)* of a SIGNAL program, if such a clock exists. If not, the result of the clock calculus is a set of clocks which are local maxima for the SIGNAL program. If there is a unique master clock, the program is said to be *endochronous* and the master clock defines its timing rate. Executable code for functional simulation can be generated for endochronous programs. Otherwise, SIGNAL processes are said to be *exochronous*. They do not dispose of all necessary information for their execution, i.e. the synchronizations of variables are not completely specified.

The resolution method uses a triangularization strategy and a tree representation in order to synthesize the most frequent clock of a program and to verify the synchronization constraints on clocks [6]. The idea is to derive definitions for all clocks in terms of already defined ones and to verify the equivalence between multiple definitions. The properties of a boolean lattice and the partition equations are rewriting rules largely used during the compilation. The tree structure speeds up the rewritings and incrementally builds the master clock of the program.

2.3 Example of a SIGNAL program

In Fig. 1 we give an example of a SIGNAL program and show the associated clock calculus. More details on the clock calculus are provided in Sect. 3.3.

```
(1) process OVERSAMPLE =
(2)  ( ? integer FB;  % FB is an input signal %
(3)    ! integer N )  % N is an output signal %
(4)   (| N := FB default (ZN - 1)
(5)    | ZN := N $ 1
(6)    | FB ^= when (ZN <= 1)  |)
(7)  where integer ZN init 1
(8)  end
```

Clock equations

(4) $\hat{N} = \hat{FB} \vee \hat{ZN}$

(5) $\hat{ZN} = \hat{N}$

(6) $\hat{FB} = [ZN \leq 1]$

Figure 1: **A SIGNAL program and the associated clock calculus.**

The SIGNAL program in Fig. 1 implements an oversampling: for each input value FB, the OVERSAMPLE process produces FB output values for N (cf. the timing diagram in Fig. 2). ZN is the previous value of N; ligne (4) of the SIGNAL program states that, for each logical instant, the value for N is equal to FB, if the signal FB is present, or to (ZN − 1), if FB is absent. The input signal FB is accepted only at the *true* value instants of (ZN ≤ 1) (see ligne (6)).