HAL
open science

Profiling of SIGNAL programs and its application in the timing evaluation of design implementations

Apostolos Kountouris, Paul Le Guernic

To cite this version:

Apostolos Kountouris, Paul Le Guernic. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. IEE Colloquium on the Hardware-Software Cosynthesis for Reconfigurable, Feb 1996, Bristol, United Kingdom. pp.6/1-6/9, 10.1049/ic:19960225 . hal-00544253

HAL Id: hal-00544253

https://hal.science/hal-00544253v1

Submitted on 7 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.

Profiling of SIGNAL programs and its application in the timing evaluation of design implementations

Apostolos A. Kountouris
Paul Le Guernic

In this paper we present the tool currently under construction in order to enhance the SIGNAL environment with a facility that will allow the temporal validation of a system specification in respect to its R/T constraints while staying within the context of the SIGNAL language. By use of the so called temporal homomorphisms we express the temporal dimension of a functional specification as a SIGNAL program. This facility can be further extended to evaluate the temporal behavior of a system in respect to a chosen execution architecture, by modelling processor architectural features influencing execution time.

1.0 Introduction

Reasoning about the timing properties of a program is indispensable in the development of time critical systems where failure to meet deadlines can result in loss of life or material. There is much work done in the domain of functional specification where formal languages like SIGNAL, Esterel, Lustre [3] can be successfully used. There seems to be an inadequacy as far as the validation of temporal properties is concerned and this is mainly due to the fact that many different factors influence the execution time of a program making this problem quite complicated when viewed from a high abstraction level.

In the problem of finding execution time bounds of a system there are two types of pessimism involved. The first type has to do with the program execution flow, available parallelism etc. The second type is related to target architecture configuration, and choice of specific components that affect the system execution time by either constraining the potential parallelism or imposing different operation delays. Previous research efforts focus in finding safe and tight execution time bounds by limiting as much as possible the sources of pessimism. Several approaches are proposed in the literature [12][14]. In [14] the conditions necessary for a program so that this problem has a solution are set, making SIGNAL programs a good candidate since by definition they satisfy all of them. In [13] it is shown that reasoning about time at the higher level source language, is practicable and can yield interesting results, it is also shown that features like compiler optimizations can also be taken into account. From work in [16] it becomes apparent that modelling processor architectural features increases the quality of the results. In [16][17][18][19] it is shown that RISC and DSP processor pipelines can be adequately modelled and in some cases the models can also be extended to account for instruction caches. Finally in [20] it is demonstrated that it is possible to predict the performance of synthesized hardware at the behavioral specification level.

From the discussion above it is clear that a facility that evaluates at the specification level if R/T constraints are respected, when a system is implemented by a chosen architecture, is feasible and can be very useful. In he remainder of this paper we describe the facility for extracting the timing properties of SIGNAL programs by generating their temporal homomorphisms, the advantages of the SIGNAL graph in minimizing the pessimism in execution time estimation and finally how we plan to extend the use of homomorphisms to evaluate implementation alternatives.

2.0 Introduction to SIGNAL

The SIGNAL language[1] is a dataflow oriented language based on the synchrony hypothesis[2]. It belongs to the family of synchronous languages [3] and it is used for the functional specification of reactive R/T systems for control and DSP applications. Using the language expressions the user is programming in an equational style and thus each program is a system of equations. The

A. Kountouris and P. Le Guernic are with Institut de Recherche en Informatique et Systèmes Aléatoires - IRISA, in Rennes, France

SIGNAL compiler resolves these systems and proves that the control of a program is functionally safe. Around the SIGNAL language and its compiler exists a variety of tools that constitute the SIGNAL environment. There is a *Graphical User Interface* for program *(system design)* entry, C and Fortran code generators to generate code used for functional simulators, intermediate code generators to access formal verification [4] and other third-party development tools [9]. Finally there exists a VHDL code generator [5] that enables us to access hardware synthesis tools. The basic data structure of the SIGNAL environment is the *dynamic graph* (DG) which is constructed by the compilation process. During the compilation the SIGNAL compiler performs checks (see [1]) useful in discovering possible sources of error and functional unsafety. Briefly once compilation is over it is certain that the program is deterministic, does not exhibit contradictions, has no cycles (circular data/control dependencies), no constraints are set on the inputs and that desired functional properties (coded as SIGNAL equations) are satisfied.

Before introducing the basic elements of the SIGNAL language let us give some basic definitions. In SIGNAL terminology a *signal* is an infinite sequence of data where each element is implicitly indexed by time. At any given logical instant a signal may be absent or present. Presence is denoted by the signal's value at that instant. The *clock* of a signal is the set of the logical instants that it is present (and thus carries a value). Clocks can be considered as equivalence classes between signals. When two signals are present at the same logical instants they possess the same clock; otherwise their clocks are different.

The SIGNAL language is defined by a small kernel of statements. Each statement has formally defined semantics and defines a clock equation and the data dependencies of the participating signals. Its small size allows for mathematical manipulation of these equations enabling formal verification of program properties. The basic language constructs are shown in the table below along with an informal description. For a more detailed description of the language, its semantics and applications the reader is referred to [1] and [4]. A last point of interest is the SIGNAL language features that extend its use in the domain of DSP applications.

3.0 Temporal property extraction

In the paragraphs that follow we present the idea of SIGNAL program homomorphisms and their application for the extraction of the temporal properties of a SIGNAL specification. Another important aspect is the context of their use that permits not only the extraction of information, but also the experimentation with alternatives influencing execution time (e.g. partitioning, scheduling).

An homomorphism is a SIGNAL program generated by applying a set of *transformation* rules on

Language Construct	SIGNAL syntax	Description
stepwise extensions	X := A op B	where op: arithmetic/relational/boolean operators
delay	ZA := A $x	memorization of the xth past value of A
extraction	R := A when B	R equal to A when B is present and true
priority merging	R := X default Y	if X present R:=X else if Y present R := Y else R absent
process composition	(| P | Q |)	processes are composed common names correspond to shared signals
useful extensions		
	when B	the clock of the true instants of B
	event X	the presence instants of X
	synchro {A, B}	clock of A equal with clock of B

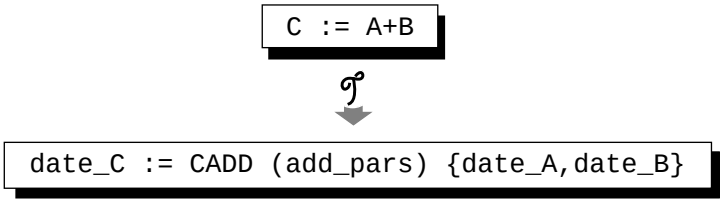

FIGURE 1. translation rules for temporal homomorphism generation

an initial SIGNAL program. The structure of the homomorphic program is essentially the same but its computations expose another aspect of the information contained in the program graph. Homomorphism generation can be fully automated provided that the transformation rules are defined. Depending on the type of information we wish to extract from a SIGNAL program we may define an appropriate homomorphism. As an example in figure 1 we present the basic translation rules that when applied to a SIGNAL program yield its temporal homomorphism. Roughly these rules are: substitute each signal by its corresponding date (e.g. date_C for C), substitute each operator by an appropriate SIGNAL process (e.g. `CADD(){}` for the addition) that acts as a component library front-end, insert parameters values that affect operation execution time (e.g. `add_pars`). These parameters reflect information that is either found in the DG or is provided by the user and it is the topic of a later section relating to the target architecture modelling. Another useful homomorphism is the one for operation counting. It can be used in identifying computationally intensive parts that may require parallelizing, or algorithmic bottlenecks in the specifications that may prompt necessary modifications.

3.1 Temporal homomorphism

To present the basic ideas we start by considering an ideal case. As we proceed we relax our assumptions so that in the end we consider a realistic case of system implementation. At each step we introduce the necessary additions to the SIGNAL *Dynamic Graph* in such a way that the same model always applies for date calculations.

3.1.1 Ideal parallel case

In this case we assume that an unlimited number of processors is available, so that at every instant of program execution all the potential parallelism inherent in the system specification can be exploited. We assume zero communication delays and finally to make things even simpler, constant operation delays. Even though this scheme may be far from real it can yield some useful results like an upper limit of system performance capabilities. If performance constraints are not satisfied even under such an ideal execution, that prompts the designer for algorithmic modifications at the functional specification level, before proceeding to later development phases.

Pure data programs: In pure data programs there is no control so in the DG dependencies are always active at every iteration of the system. The nodes represent operations. At every node we have incoming arcs representing the operation arguments and outgoing arcs representing the operation results. The graph of the temporal homomorphism has exactly the same structure but its arcs represent dates and its nodes the actions that compute the result dates as a function of the argument dates. These actions consist of finding the maximum of incoming dates and adding to it the delay corresponding to the operation in the original graph. This is demonstrated in the example given in figure 2. In the center we have drawn the graph that corresponds to the program on the left. On the right we give the graph corresponding to the temporal homomorphism of the program, which is produced by applying the transformation rules. In each node we give the Δop which is the delay of the operation *(op)*. The bottom box contains the actions taking place at each node in order to compute the node's result date. In this way starting with the system input dates and by traversing the graph we obtain the system output dates.

Control/Data programs: The case of programs that also contain control demonstrates the advan-

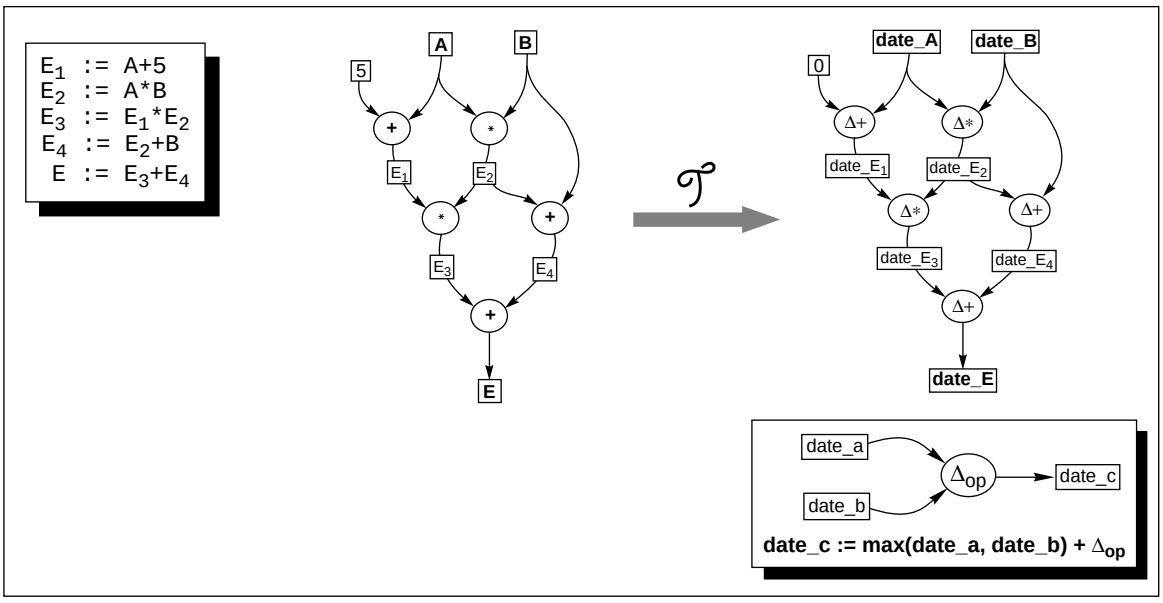

FIGURE 2. date calculations for pure data programs

tages of the SIGNAL graph in accounting for the longest execution paths. The date computation model is exactly the same as before, only now we have to take into account the clocks of arcs and nodes. In the SIGNAL graph every element (arc or node) is tagged by the clock that defines the logical instants of its presence. Thus considering the temporal homomorphism of a node at the left of figure 3 the result of the *max* is controlled by the clocks of the incoming arcs, as it is shown in the bottom box. When both arcs are present at the same logical instant we take the maximum date of the two or else we take the date of whichever arc is present. In the resulting date we add the delay of the operation in order to compute the date of the node output.

Using the SIGNAL clocks it is easy to account only for the active paths in the graph during successive system iterations, and even more, exclude processing paths that can never exist (e.g. paths containing mutually exclusive parts). Such paths contain dependencies whose clocks are mutually exclusive[1]. This information is discovered during clock calculus. These advantages of the DG in respect to traditional graphs are illustrated in the example given in figure 4. In this example thick lines correspond to program data and thin ones to control data that affects the execution flow. Next to the graph we give the corresponding pseudocode. It is easy to see that P_2 and Q_1 are

1. mutually exclusive clocks are never present at the same instant

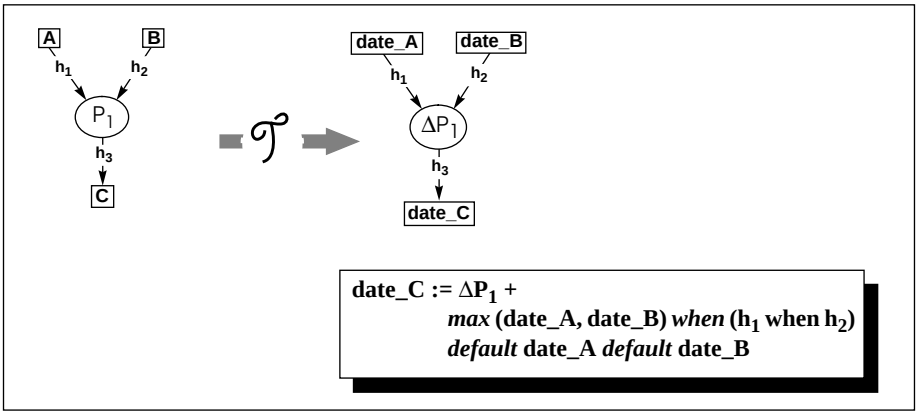

FIGURE 3. date calculations when SIGNAL clocks are taken into account

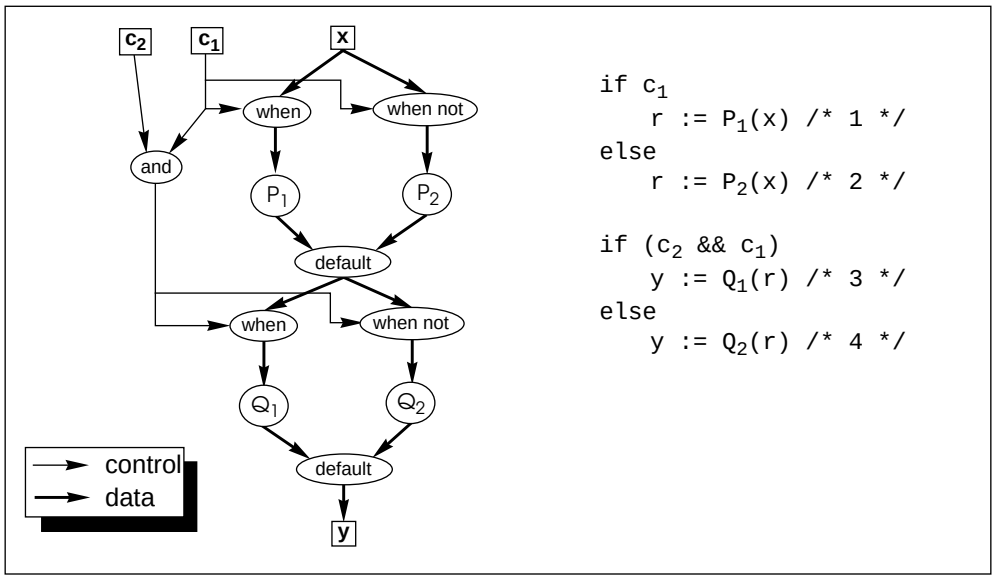

FIGURE 4. finding longest paths in the SIGNAL dynamic graph

mutually exclusive so the path containing both (1, 3) should not be considered in time consumption counting. This is important if one considers that in many approaches for the worst case execution time calculation, a lot of pessimism is attributed to the fact that many paths that are infeasible cannot be excluded from the computations. In [13] [14] [15] [16] user annotations are used to indicate such paths and thus reduce pessimism but there is no guarantee that these annotations are always correct especially in the case of large complex programs. In the context of SIGNAL these annotations are clock relationships that thanks to clock calculus become explicit and are validated during compilation and thus they can be considered to be safe and correct.

The case of *control/data* programs requires some special care in the generation of their homomorphisms. Many control branches depend on boolean signals that are either inputs of the program or they are internally computed by the evaluation of relational operations. Since in the homomorphic program no computations contained in the original program take place, we have to provide the booleans that define such clocks as extra inputs to the homomorphisms in order to preserve the same model for date calculations.

3.1.2 Sequential case

Since the graph corresponding to the program is a partial order, to achieve sequential execution we have to enforce it with additional dependencies between potentially parallel operations. The main preoccupation is to preserve the same model for date calculations as before. The example in figure 5 demonstrates our approach. For this simple case there are no dependencies between operations P_2 and P_3 so there are two possibilities for sequential execution: P_1, P_2, P_3, P_4 or P_1, P_3, P_2, P_4. A first step is to add a dependency between P_2 and P_3. This dependency is conditional in order to be active only at the instants that both operations are to be executed at the same time. The clock of this *added* dependency is the intersection (common instants) of the clocks of the two operations and it is noted $[s_1]$ which is the clock defined by the instants that the boolean s_1 carries the *true* value. When s_1 is *true* the dependency of P_2 to P_3 is active meaning that P_2 must execute before P_3. In order to make our scheduling scheme more flexible we also add an inverse conditional dependency from P_3 to P_2 with $[\neg s_1]$ as its clock so that when a scheduling dependency is active its inverse dependency is inactive. To demonstrate that the same model of date calculation is preserved let us assume that the clocks of P_i's (in figure 5) are equal and that we choose s_1 to be *true*. The production date of P_4 is calculated as follows:

$$\begin{aligned} \text{date_}P_4 &= \mathit{max}\,(\text{date_}P_2, \text{date_}P_3) + \Delta P_4 \\ &= \mathit{max}\,((\text{date_}P_1 + \Delta P_2), \end{aligned}$$

$$(max\ (date_P_1, date_P_2) + \Delta P_3) + \Delta P_4$$
$$= max\ ((date_P_1 + \Delta P_2), (max\ (date_P_1, (date_P_1 + \Delta P_2)) + \Delta P_3) + \Delta P_4$$
$$= date_P_1 + \Delta P_2 + \Delta P_3 + \Delta P_4$$

which corresponds to the chosen sequential execution path.

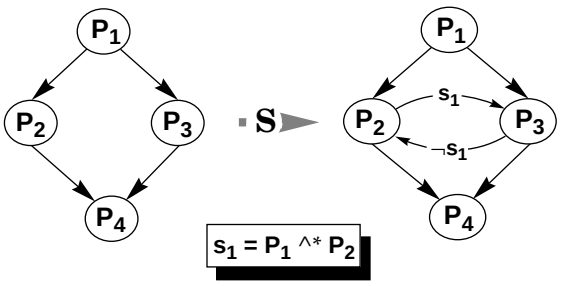

FIGURE 5. operation scheduling for sequential execution

3.1.3 General case: constrained parallelism

In a more realistic case an execution architecture consists of a finite number of *processing elements* (PE's) where the communications between these PE's take time and are no longer for free. At this point we consider the PE's as abstract processing elements. The architecture configuration (PE's and communication links) defines a hardware graph on which the software graph mapped. This mapping is dictated by a graph partitioning, that assigns each graph node to a PE. This partitioning can be generated in many ways either manually [6] or automatically by using a partitioning algorithm. Execution on each PE may be sequential, so ordering dependencies are added (where needed) between the nodes assigned to the same PE. So far in order to treat this general case we combine the ideas already mentioned in the ideal parallel and sequential cases. The last thing that remains is to present how inter-PE communications are handled so that we can still apply the same model for date calculations. Communications are represented in the DG as operation nodes added [6] in the graph during partitioning. For each communication node its clock has to be found in such a way that the functionality of the initial graph is preserved and control errors (cycles) are not introduced. Once all the above steps are performed we have a DG for which its temporal homomorphism can be generated as before. Furthermore communication links can be modelled as SIGNAL processes in order to account for different execution rates between PE's.

3.1.4 Obtaining results using homomorphisms

There are two ways to use the temporal homomorphism in order to obtain results regarding the temporal behavior of a system and reason whether or not time constraints are satisfied. The first is a sort of approximate temporal simulation where the homomorphism is used on a stand-alone basis. In the second by combining the original SIGNAL program with its temporal homomorphism we obtain its *profiled* version.

Approximate temporal simulation: In order to simulate the temporal response of a system in respect to defined operation delays, we generate its homomorphism which is the main element of such a simulator whose configuration is shown in figure 6. In the bottom box it is indicated that an output date is a function of subsets of input dates, condition booleans and scheduling dependencies. For a simulation we have to provide test vectors $[c_1,...c_q]$ which are the clock defining booleans. We can either provide a set of vectors that covers all the possible combinations or use a smaller representative set of test vectors. The input generator (process IN) may perform a statistical emulation of the external environment in order to get an approximate temporal response of the system. A statistical emulation consists in providing input date values (environment emulation) and $[c_1,..., c_q]$ test vectors (program control flow emulation). We also have to provide the scheduling booleans s_i as inputs to the homomorphism. To do this we add a process *(sched)* that assigns them a value according to a scheduling strategy. For a *static* scheduling these boolean values are kept constant throughout the execution. In this way we can also evaluate the results of a scheduling algorithm that when applied to a program graph gives a combination of s_i values. For a *dynamic* scheduling s_i values are assigned according to a chosen scheduling strategy that may depend on *input/output* or *intermediate* signal dates. In this case the *sched* process includes logic simulating the scheduling strategy. Finally the post-processing process may calculate,

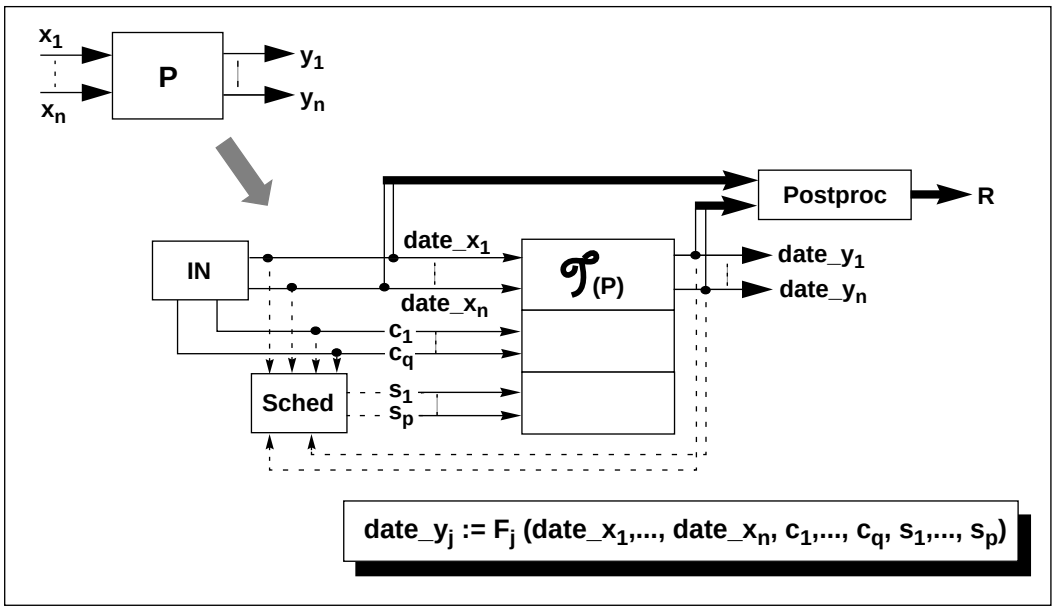

FIGURE 6. temporal behavior simulator configuration

among other things, maximum, minimum and average dates for the outputs. All these processes can be automatically generated.

Temporal profiling: In software development profiling of a program is the instrumentation of a source code so that information and statistics are gathered during execution. This information is later processed in order to study program behavior, debug, test etc. In the configuration shown in figure 7 a SIGNAL program is instrumented for temporal property extraction when composed with its temporal homomorphism. To refer to the combined execution of a SIGNAL program and its temporal homomorphism we introduce the term *temporal profiling*. The original program is transformed in such a way the conditional booleans computed inside are produced as outputs connected to the corresponding inputs of the temporal homomorphism process. Scheduling and date post-processing processes follow the same model as in the approximate simulation case. The basic application of this *profiling* is a high level simulation environment where we can simulate the execution of a SIGNAL program on a variety of target architectures and contrast the performance of each one against the system's R/T constraints. In this environment we can experiment with various scheduling strategies and with different types of processing elements in order to refine the design choices. The necessary ingredients for such a scheme to work is the topic of the next section.

3.2 Extending homomorphisms for timing evaluation of system implementations

Homomorphisms is the tool that permits to extract such timing information and in the paragraphs that follow we present how its use can be extended in order to evaluate at a higher level how the choice of a specific execution platform influences the temporal response of a system. The first step is to parametrize the temporal homomorphism and for that we identify the parameters that influence operation delays and the building blocks *(components)* of system architectures. The second step is to make available the timing model of each component and by accessing these models with the appropriate parameter values get the operation delays when executed on a specific component. Finally, using the parametrized homomorphisms a high-level *(co-)*simulation environment can be defined, in which the designer can simulate the execution of a system on a target architecture[1]. This facility permits fast and cheap design exploration as evaluation occurs early in the development process and entirely in software.

1. if a target architecture contains hardware such a simulation can be seen as a sort of co-simulation.

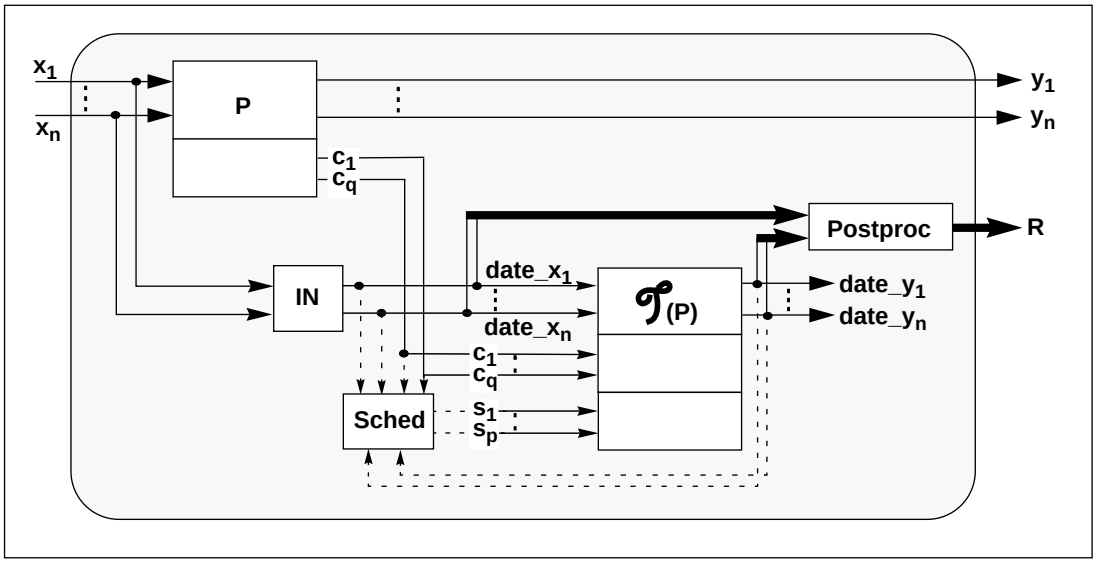

FIGURE 7. profiling of SIGNAL programs for exact modelling of temporal behavior

Homomorphism parametrization: Briefly we can introduce parameters in the component library process front-ends (e.g. add_pars of CADD in figure 1) whose values indicate among other things the PE executing an operation so that the appropriate component library file can be used, the operation class and the argument types involved. To account for compiler optimizations and architectural features like pipelined or superscalar processors it is necessary to include parameters giving the operation context (in terms of surrounding operations, argument sources and result sinks etc.). The parameter values can be either found in the DG or be provided by the user.

Component modelling: A target architecture might contain a various number of processing elements and interconnects between them. A processing element can be either an off-the-shelf processor or a hardware ASIC (or ASIP) but at a high level of abstraction we can refer to them as processors. For the off-the shelf processors different ones impose different levels of modelling complexity depending on their architectural features. In [17] and [18] it is shown that RISC processor pipelines can be adequately modelled and in [16] and [18] the method is also extended to account for instruction caches. In [16] it is clearly shown that memory hierarchy cannot be neglected since it affects execution time considerably. Modelling of DSP processors is the subject of [19] where things seem to be less complicated since in their design features introducing unpredictability, like data caches for example, are usually avoided.

Since there are many similarities in machine code generation for off-the-shelf processors and netlist generation for hardware ones it would be very useful if we could use the same modelling approach for all types of components. In [20] the feasibility of extracting performance estimations from behavioral specifications is encouraging as far as modelling hardware at a high level is concerned. Finally in the case of interconnects accurate analytical models, that we can use to obtain communication delays, already exist. In our view it is possible to create high-level models for each basic processor architectural feature and then by combining them obtain complete models of existing processors that can be used under our framework.

4.0 Conclusions

In this paper we presented the ideas for augmenting the SIGNAL environment with a facility that permits the validation of functional as well as timing properties of R/T control and DSP systems. The SIGNAL language has many desirable features permitting control-safe specifications and it also satisfies the prerequisites for the extraction of safe execution time bounds. The internal representation of the *dynamic graph* contains information that permits to safely minimize the pessimism for making those bounds tighter by

excluding infeasible paths. The concept of homomorphisms has been introduced, in order to expose the temporal dimension of SIGNAL programs expressed as SIGNAL processes. The parametrization of homomorphisms will permit reasoning about the timing properties of specific implementations by use of high-level component models. This extension will enable us to evaluate implementation alternatives early in the development lifecycle by building versatile implementation simulators that can also serve as experimentation platforms assisting in fine tuning the design choices. Accurate component models will further minimize the pessimism in estimating even tighter execution time bounds.

In respect to the potential applications of our method two domains are of particular interest. First, design evaluation schemes for hw/sw codesign methods and second, development framework aspects.Currently the evaluation of design choices (e.g. hw/sw partitioning) is implicit in the algorithms and bound to either inaccurate component modelling information or quite restrictive target architectures [7][8][9], at the cost of limiting the design space and making re-iterations in the design process, costly. The high-level approach we propose will increase the quality of results and moves the evaluation early enough so that design space exploration is quicker and more flexible.

As future directions in our work we can briefly mention the use of homomorphisms to obtain other types of information like for example operation counting. Also, since the temporal aspect of a system is expressed as a SIGNAL process that means that a system of temporal equations can be extracted and formal calculi tools might be used. From a SIGNAL program we can generate correctly annotated C code in order to access other existing timing tools.

REFERENCES

[1] “Programming Real Time Applications with SIGNAL”, Paul Le Guernic, Michel Le Borgne, Thierry Gautier, Claude Le Maire, Proceedings of the IEEE, vol.79, no.9, pgs 1321-1336, Sep. 1991

[2] “The Synchronous Approach to Reactive and Real-Time Systems”, Albert Benveniste, Gerard Berry, Proceedings of the IEEE, vol.79, no.9, Sep. 1991, pgs 1270-1282

[3] Special section: R/T Programming, Proceedings of the IEEE, vol.79, no.9, Sep. 1991

[4] “The SIGNAL dataflow methodology applied to a production cell”, T.P. Amagbegnon, P. Le Guernic, H. Marchand, E. Rutten, Lecture Notes in Computer Science LNCS No 891, Springer Verlag, Jan. 1995

[5] “Using VHDL for Link to Synthesis Tools”, M. Belhadj, North Atlantic Test Workshop, Jun. 1994

[6] “Synchronous distribution of SIGNAL programs”, Pascal Aubry, P. Le Guernic, S. Machard, Proceedings 29th Hawaii Intl. Conference on System Sciences, Jan. 1996, pgs 656--665

[7] “System Synthesis via Hardware-Software Codesign”, R.K.Gupta, Giovanni De Micheli, Computer Systems Laboratory Technical Report, CSL-TR-92-548

[8] “A Global Criticality / Local Phase Driven Algorithm for the Constrained Hardware / Software Partitioning Problem”, A.Kalavade, E.A.Lee, IEEE Proceedings, Intl.Conf. on HW/SW Codesign, pgs 42-48, 1994

[9] “The SynDEx software environment for real-time distributed systems design and implementation”, C. Lavarenne, O. Seghrouchni, Y. Sorel, M. Sorine, ECC91 European Control Conference, Grenoble, France Jul.91, pgs 1684-1689

[10] “A Hardware-Software Codesign Methodology for DSP Applications”, Asawaree Kalavade, Edward A. Lee, IEEE Design & Test of Computers, pgs 16-28, Sep. 1993

[11] “Synthesis Steps and Design Models for Codesign”, Tarek Ben Ismail, Amine Jerraya, IEEE Computer, pgs 44-52, Feb. 1995

[12] “Reasoning about time in higher-level language software”, A.C. Shaw, IEEE Transactions on Software Engineering, vol.15, no.7, July 1989, pgs 875-889

[13] “Experiments with a Program Timing Tool Based on Source-Level Timing Schema”, Chang Yun Park, Allan C. Shaw, IEEE Computer, May 1991, pgs 48-57

[14] “Calculating the Maximum Execution Time of Real-Time Programs”, P. Puschner, Ch. Koza, RR-01-89, Institut fur Technische Informatik, Technische Universitat Wien, April 1989

[15] “ A Tool for the Computation of Worst Case Task Execution Times”, P. Puschner, A. Schedl, RR-04-93, Institut fur Technische Informatik, Technical University of Vienna

[16] “Efficient Microarchitecture Modelling and Path Analysis for Real-Time Software”, Y-T.S. Li, Sh. Malik, A. Wolfe, Proceedings of the IEEE Real-Time Systems Symposium, Dec. 1995

[17] “Predicting Worst Case Execution Times on a Pipelined RISC Processor”, S.J. Bharrat, K. Jeffay, Technical Report TR94-072, Dept. of CS, Univ. of North Carolina at Chapel Hill, April 1994

[18] “An Accurate Worst Case Timing Analysis Technique for RISC Processors”, Sung-Soo Lim et al., IEEE Real-Time Systems Symposium 1994, Puerto Rico, Dec. 1994, pgs 97-108

[19] “Software performance estimation of DSPs for HW/SW partitioning”, M. Augin, C. Belleudy, G. Gogniat, C. Kieffer, Intl. Workshop on Logic and Architecture Synthesis, IFIP TC10 WG10.5, Grenoble-France, Dec. 1995, pgs 273-282

[20] “Estimating Architectural Resources and Performance for High-Level Synthesis Applications”, Alok Sharma, Rajiv Jain, IEEE Transactions on VLSI systems, vol.1, no.2, June 1993, pgs 175-190

[21] “Generating Machine Specific Optimizing Compilers”, Roger Hoover, Kenneth Zadeck, Research Report, IBM Research Division

[22] “A Retargetable Compiler for ANSI C”, C.W. Fraser, D.R. Hanson, SIGPLAN Notices 26, pgs 29-43, Oct. 1991