Multi-Thread Graph : A System Model for Real-Time Embedded Software Synthesis*

Filip Thoen, Jan Van Der Steen, Gjalt de Jong, Gert Goossens and Hugo De Man‡

IMEC, Leuven, B-3001, Belgium

‡ Professor at the Katholieke Universiteit Leuven, Belgium

Abstract

Software synthesis is a new approach which focuses on the support of real-time embedded multi-tasking software without the use of operating systems. A software synthesis system starts from a concurrent process system specification and maps this description automatically onto one or more processors.

In this paper, the internal system-level model which captures the embedded software and which is the backbone of our software synthesis methodology, is presented. The model captures the fine-grain behaviour of a system, and supports multiple threads of control (concurrency), synchronisation, data communication, hierarchy and timing constraints.

Keywords - real-time embedded software, multi-tasking, software modelling, software synthesis.

1 Introduction

Advanced real time information processing and personal communication systems are of increasing industrial importance. In this domain, there is a continuous evolution towards more complex applications, often integrating whole systems on a single chip. At the same time, the heavy competition on the market also pushes towards cheaper solutions and, even more important, shorter design times. Including a programmable processor on the chip is a viable solution to realize this, increasing the role of embedded software in contemporary systems.

In the software design process, mapping the many different interacting processes that form the system to one single thread of control (or at most a few, when using a multiprocessor system), taking into account timing constraints, becomes a major bottleneck. Presently, hand crafted solutions or real-time operating systems are used. The first solution, where the different processes are manually joined to one sequential (C-)program, is inflexible, time consuming and hard to debug. The second introduces a significant overhead, as well in extra execution time as in code size due to the inclusion of the kernel. Additionally, although they pretend to, real-time operating systems do not guarantee timing constraints.

In our software synthesis system [13], we aim at reducing the design time by automating the mapping described above, hereby taking into account the timing constraints, while still avoiding the overhead introduced by a real-time kernel. Therefore, a model is needed that can capture all relevant system aspects. We will show that the currently existing models do not satisfy this requirement. This paper presents a new model, called Multi-Thread Graph (MTG). For more information about our software synthesis methodology itself, we refer to [13].

In the next section, the requirements for a model, suitable for a software synthesis system, will be derived. Section 3 describes our MTG model itself. In section 4 we will illustrate by means of an industrial application the capability of the model to capture all information necessary for software synthesis. Finally, in section 5 we give some conclusions and indicate further work.

2 Requirements

From studies of industrial applications, of which an example is described in section 4, the following requirements for a model to be used for software synthesis were derived.

- the model has to support multiple threads of control. Threads may contain periodic as well as sporadic behaviour. Modelling of the synchronisation behaviour and interaction with the environment are also required.
- to perform software scheduling, a major task in the software synthesis script, at first control flow information is essential. Secondly, adequate timing information is necessary. This includes minimum and maximum latency constraints, response, inter-arrival and thread execution times, as well as fine-grain timing constraints (e.g. low-level IO-protocol timing).
- the model should include enough information about data communication to allow to generate compilable code for the complete integrated system, i.e. including

* This work was supported by the European Commission, under contract *Esprit-9138 (Chips)*

ED&TC '97 on CD-ROM

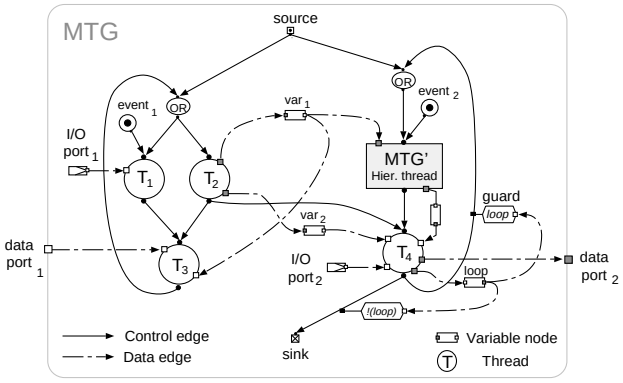

Figure 1: Example of a Multi-Thread Graph.

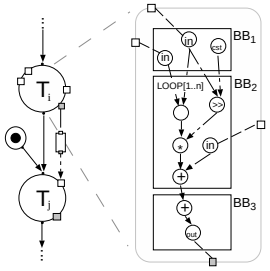

Figure 2: A program thread with its internal CDFG.

code for the interface between threads and processors. This means that not only the influence of data communication on thread dependencies, but also the binding of variables to memory or register locations must be available.

- in order to keep the analysis of systems tractable, non-deterministic timing locations (e.g. waiting for a user command) should be isolated. Also for this reason, details which are not relevant for software synthesis should be hidden. Isolation of these timing locations allows to schedule other behaviour while waiting, in this way maximizing processor utilization.
- the model should be target independent, as many different architectures are used for system implementation : both different types of processors (standard, DSP or application specific) as different number of processors (single versus multiprocessors) are implementation candidates.

Our Multi-Thread Graph model is designed to meet these requirements. It is important to note that it is the combination of all these elements rather than any single one that makes our model unique.

3 The MTG System Model - Concepts

Figure 1 graphically illustrates all the elements of which a MTG consists. These elements will be discussed in the next paragraphs. However, due to space limitation this discussion focuses on the main principles only. For an in depth formal discussion, we refer to [14].

We assume that the application can be specified in a concurrent process description, capturing operation behavior, data dependencies, concurrency and communication. From this specification, we derive our MTG model, which contains sufficient information for the software synthesis problem. Currently, an automatic extraction of the MTG model from a behavioural VHDL system specification is being implemented [6].

Below, we subsequently discuss basic control flow concepts, timing information, data communication and additional features of the MTG model.

3.1 Basic Control Flow Concepts

The basic control flow concepts in a MTG are the operation nodes and the control edges. Several attributes are used to capture additional information. Operation nodes can be : program threads[1], events, or-nodes, a source and a sink (see fig. 1).

We define a **program thread** (or shortly: a thread) *as a set of operations with a deterministic execution latency.* This means that, once started, it can be executed fully to its end without any synchronisation with the environment or with other threads. In our model, the underlying (fine-grain) behaviour of a thread is captured by a Control Data Flow Graph (CDFG) [11], as depicted in figure 2. Note that a thread contains only statically compilable code (and can therefore be handled with existing compilers), while all system level aspects are captured in the MTG layer. Remark that a thread can consist of multiple basic blocks [2].

Operation nodes have single control $entry$ and/or $exit$ points. A **control edge** $e_{i,j}$ between an $exit$ point of an operation node o_i and $entry$ point of an operation node o_j enforces the start of execution of o_j to be after the completion of o_i. In this way, control precedences and data dependencies are modelled.

For the operational semantics of the model, *token flow semantics* is defined. The *firing rule* of a node is as follows: a node can only start execution when *all* its input edges carry a token. An exception to this is the or node, which only has to carry a single token on *one* of its input edges. The execution of a node o_i takes $\delta(o_i)$ time units,[3] after which a single token is placed on all outgoing control edges of the node. This execution time is a node attribute in the model.

At startup, an initial token marking is rendered by placing a single token on each of the outgoing edges of the source node. The sink node can be used to detect termination of

[1] There are also hierarchical threads (see section 3.4) but for this moment they can be considered as ordinary threads.

[2] a basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility for branching except at the end.

[3] $\delta(o_i)$ is only different from zero for a thread.

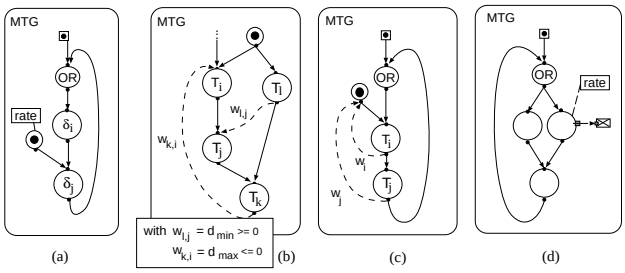

Figure 3: Execution latencies (a) and latency (b), response time (c) and execution rate (d) timing constraint

a MTG. We put the restriction on the graph structure that no tokens may be alive in the graph when the sink node is reached.

Interaction with the external environment is modelled by **event** nodes, which are bound to external events, such as a "ready" signal from a peripheral or the arrival of new data. Conform to the token flow semantics, an external event will cause the corresponding event node to inject a new token. In this way, *external synchronisation* can be realized. On a processor, external synchronisation can be implemented by an interrupt or a polling flag.

Non-deterministic timing delays are related to external synchronisation (e.g. wait for communication, wait for peripherals, …). Therefore, as this external synchronisation is always isolated in event nodes, also the isolation of uncertainties related to execution delay is realized.

To model *internal synchronisation*, the implicit AND semantics of edges connected to the same operation node *entry* is used, which avoids the need for a separate model construct. An example is the synchronisation between T_2 and T_4 in figure 1. Also the re-synchronisation, i.e. join, of concurrent behaviour can be done in this way, e.g. between T_1 and T_2 in the same figure.

Multi-rate transitions are supported by associating two integer numbers, $rp_{i,j}$ and $rc_{i,j}$, respectively called *production* and *consumption* rate, to each control edge.

3.2 Timing information

Due to the real-time nature of the target application domain, capturing timing information and supporting timing constraints is crucial. All timing information is expressed by attaching attributes to edges, threads and events (see fig.3).

A **thread execution latency** is associated to each thread. Accurate estimations can be generated with an existing compiler, as threads can be statically compiled. This results in worst case execution times, but, as we have the thread behaviour available in a CDFG, nothing prevents us to take into account other numbers.

With each control edge, a **latency constraint** is associated as a "weight", allowing to constrain the temporal distance between two different threads. A control edge $e_{i,j}$ with weight $w_{i,j}$ between the $exit$ of T_i and the $entry$ of T_j represents a minimum latency constraint between T_i and T_j, i.e. the requirement that the start time of T_j must occur *at least* $w_{i,j}$ units of time later than the end time of T_i. Similarly, a negative weight models a *maximum* latency constraint. representing the requirement that the end time of T_k must occur no later than $\mid w_{k,l} \mid$ units of time later than the begin time of T_l.

Response time constraints are a subclass of the above, specifying a latency constraint with respect to an *event* node.

Execution rates (associated to threads or events) and **minimum inter-arrival times** (associated to an event) model timing information of periodic and sporadic processes respectively. They are especially useful for I/O operations. Both minimum and maximum rates can be specified.

3.3 Data Communication

With respect to data communication, by means of the control edges, only their influence on thread dependencies and precedences can be modelled, as in ordinary (data) flow graph models. For more sophisticated data dependent control and synchronisation behaviour, other data communication primitives are needed, too. The same applies for our goal to generate compilable code for the whole system,

All data communication in the MTG model is based on the *shared memory* paradigm. This is done because all other communication paradigms (e.g. message passing) finally have to be implemented using physical memory anyhow. Techniques are available to do the refinement [6].

To model communication, a program thread has a number of shared memory **data ports**, either inputs or outputs, which are respectively linked to in and out operations in the underlying CDFG, as depicted in figure 1. Additionally, the model contains **variable nodes**, and **data edges**. These directed edges connect the thread data ports with the variable nodes, indicating that both are bound to the same physical location.

Communication modelled in this way is *unsynchronised*, i.e. reading and writing can be done at any time, without waiting for the next value to arrive or the previous value to be consumed. For modelling *synchronised* communication additional control edges must be used to express the synchronisation explicitly. An example of both types is depicted in figure 1: var_1 is used for unsynchronised communication between the two subgraphs, while the control edge between T_2 and T_4 ensures that communication through var_2 happens in a synchronised way.

Data communication with the environment is done via system **I/O ports**, which can be bound to a memory location (to model memory mapped I/O) or to a register (for communication via dedicated I/O registers or periph-

eral registers). This communication, also based on shared memory, can be unsynchronised or synchronised with the environment (by means of events). The latter is illustrated in figure 1 ($I/O\ port_1$).

3.4 Concurrency, decisions and hierarchy

In this section, some additional concepts are treated, necessary for an easy modelling of complex systems.

Concurrency Operation level (i.e. fine grain) concurrency can be handled by a compiler and is therefore captured by the CDFG. Thread level concurrency, implicitly supported by allowing multiple control edges to leave and arrive at an operation node, can capture both process-level and internal process (typically described by par_begin/par_end [9]) control concurrency of the system-language. An example is given in figure 1: the two subgraphs capture the process concurrency, and internal process concurrency is present in the left subgraph for $thread_1$ and $thread2$.

Guarded control edges To model state machines and decisions in the MTG model, the concept of a *conditional guard* is introduced as a control edge attribute. This attribute must be linked to a shared memory variable, produced by another thread (or by the environment). The execution semantics of an operation node whose $exit$ point is connected to a guarded edge is changed: when the node finishes execution, tokens are only placed on the edges that have a guard which evaluates to true. An example of conditional guards can be found in figure 1, where the variable *loop* is used to choose between starting a next iteration or not.

Hierarchy In the build-up of complex systems, hierarchy is highly desirable since it allows to hide lower level details and enhances modularity. Therefore **hierarchical threads** were introduced. An hierarchical thread contains a link to another MTG and can freely be substituted by it. The firing of a hierarchical thread removes the tokens from its input edges, and places tokens on all edges connected to the source node of the instantiated MTG. Similarly tokens are removed from the edges connected to the sink node of the instantiated MTG and placed on the outgoing edges of the hierarchical thread, in this way terminating the thread.
Data communication happens through *hierarchical data ports*, which correspond in a one-to-one fashion to data ports of the underlying MTG. The model supports *local* variables associated with the scope of defined by the hierarchy, and both *private* (i.e. local to the instantiation) and *shared* (i.e. accessible to the different instantiations) variables inside a hierarchical thread.

3.5 Properties of the MTG model

Much attention was given to choosing the correct abstraction level, in order to keep analysis as simple as possible, while still capturing all relevant information. Here, we mention our 2-level approach, i.e. MTG versus CDFG, that hides the details which are not relevant for software synthesis and which can be handled by existing compilers. In fact, the MTG captures the system-level, consisting of interacting code segments, and the CDFG captures the operation level. This granularity level is automatically extracted and works across the process boundaries as defined by the user. However access to the fine-grain operation level is still possible (e.g. for further optimisations, such as detecting points suited for context switching).

Another characteristic of the model is the isolation of all non-determinism in event nodes, which eases timing analysis. As such, all non-deterministic locations of the software are explicitly exposed.

The formally defined operational semantics in terms of token flow allow to perform checks for deadlocks, dead-code, race-conditions, etc. In fact, these semantics make our model very close to a (hierarchical) interpreted (timed) Petri net.

Realistic and varied sets of timing constraints as well as functional timing information (execution latencies, event occurrence rates) can be captured in the model by just a few different numbers.

The property of isolation of ND timing points and the token flow, giving a clear view on the external and internal synchronisation respectively, combined with the extensive support for realistic timing constraints, makes the model especially suited for performing the software scheduling task. The control edges between threads indicate the interrelation between these code segments, making performance analysis possible. Additionally, the availability of enough additional information such as data communication allows to perform the complete software synthesis, resulting in generation of compilable code.

A wide range of specification styles can supported, from FSM to concurrent, communicating processes (because of the generality of the underlying Petri net).

3.6 Related work

Nearly all system models are based on directed graphs, and so is our MTG model. However, only our MTG model can capture all information necessary for the complete software synthesis trajectory.

The *sequencing graph model* [10], a flow graph model, supports timing constraints adequately, but is at a too low abstraction level, as the model entities are individual operations. Moreover, there is the restriction that a sequencing graph must be a single connected graph, not able to capture process concurrency. Data communication is not supported and neither is the concept of rate transitions. Synchronisation with the environment is modeled by anchor nodes.

In [8] an extended version of the sequencing graph model is presented. A higher abstraction level is introduced

with the concept of threads, extracted from the flow graph. Our model however works directly on the thread level, being able to capture transformations on these threads, and the operational semantics are formally defined in terms of token flow between threads. [8] also supports external synchronisation (by the introduction of a *wait* operation). Process level concurrency is captured by encapsulating the sequencing graph in a process dependency graph (PDG). Temporal spacing between the process start times is specified by weighted control edges between the processes. However, we judge this process level as being artificial and unnecessary since it is arbitrarily defined by the designer. Our model works across process boundaries.

The internal model in [5] is based on the constraint graph of [10], focussing at the operation level, however with an additional concept of system *modes*. Latency, rate and response timing constraints are supported, both on operations and on modes.

The model in [1] is based on *CSP*, modeling process communication by message passing. It consists of multiple processes, each containing the following operations: task invocations, conditional branches, loops and send and receive operations. Control and data edges capture the control and data flow between these operations inside a process. Process interaction is modeled by connections between send and receive operations inside the different process. A task is a clustered group of (arithmetic) operations extracted from the input specification, representing the atomic units of functionality in the synthesis process. For the moment, tasks are equal to basic blocks (BBs), making the model more fine grain than ours since a thread can contain multiple BBs. No timing constraints are supported.

The *Extended Syntax Graph* model in [7] overlays a data flow graph, called *basic scheduling blocks*, on top of a syntax graph (SG), each giving a different view on the specification. They argue that this combination is necessary since data flow graphs are inappropriate for representing dynamic data structures and parallel processes. The only advantage of having the SG is that the original syntactical structure of the original description in C^x can be kept, facilitating the generation of HardwareC and ANSI-C. Intra-process timing constraints are specified between two SG labels. No inter-process constraints can be captured. Each process can have a single execution rate. It is not clear how the task concept is captured in the ESG, neither how inter-task communication is modeled.

In [2] software synthesis starts starts from the imperative synchronous language ESTEREL to derive a single finite automaton from a collection of initially concurrent modules. Timing in this model however is virtually absent.

In [4] a software synthesis system is developed starting from *extended asynchronous Finite State Machines*, called Co-design Finite State Machines (CFSMs). This model is however mainly oriented towards small control applications. As most FSM-based models, it is not targeted towards computing-intensive tasks. For large systems, complexity problems may arise, as the abstraction level is very low and the number of nodes can become excessive. In the MTG model in contrast, data as well as control can be handled, the complexity being hidden in the underlying CDFGs. Timing constraints can not be captured in the CFSM model.

Task graph models are too coarse-grain, and may hide ND timing points and data dependent loops, complicating timing analysis and software scheduling. Often timing constraints are restricted to constraints at the process level only, as in [16] which only allows periodic processes with a single deadline. The granularity level, which is determined by the user only, is too high to do code restructuring, as we can in the MTG model.

Data flow oriented systems as Ptolemy [3] and Grape [12] use a hierarchical flow graph model, with nodes that are very similar to our threads. These models contain data and control flow information and support rate transitions. However, these systems are mainly oriented towards simulation (and rapid prototyping), and do not intend to check timing. Consequently, the models do not contain information about timing constraints. Non-determinism cannot be captured neither.

4 Industrial example

In this section we will illustrate how the MTG concepts were used to describe an industrial satellite paging system (about 1500 lines of C code). First, a short description of the pager functionality will be given.

Pager functionality The pager is a receiver for a continental paging system, using a satellite link. It consists of two main parts. The first one, implemented in hardware, is a set of very fast decorrelators. The second part is implemented in software on an ARM processor. It consists of algorithms for synchronisation with the (fixed) base station, hereby using the decorrelation results, and a man-machine interface. This second part will be modelled in the MTG model.

The synchronisation is performed as follows : by the start-up of the system, an *acquisition* algorithm is run, providing a first, raw synchronisation ("locking"). Then, every time a set of decorrelation values comes in, a fine tuning is done by *tracking* subroutines. When for any reason the synchronisation is lost, the acquisition algorithm is run again.

The tracking and acquisition algorithms must be able to follow the speed of the decorrelators, i.e. they have to run at a very high rate, making high demands upon the processor and the software.

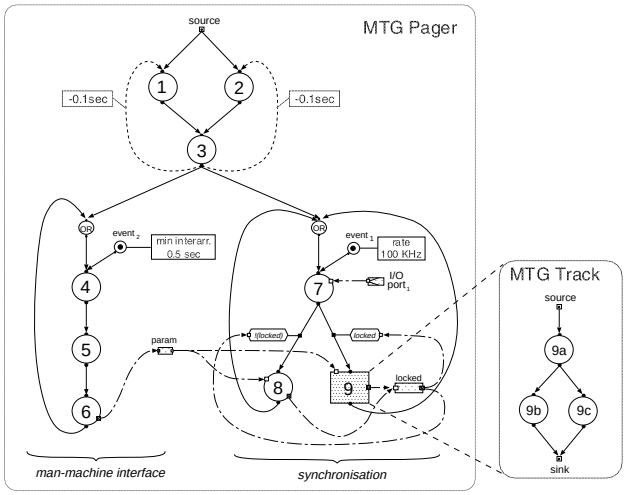

Figure 4: The MTG description of the pager.

Modelling the pager Figure 4 gives a MTG description of the pager. The content of the different threads are given in the figure. It should be mentioned that $thread_9$ is a hierarchical thread, as the tracking process consists of three different routines. There are two external synchronisation points : $event_1$, representing the arrival of new data from the decorrelators, and $event_2$, which is fired when a user command is issued. Although the application is rather large, the number of threads is still limited : details are hidden inside the threads, to be handled by a compiler. In this way, we keep the analysis tractable.

Most of the data communication is left out of the figure to enhance readability: the number of data edges and variable nodes is rather large. However, the complexity of timing analysis and scheduling is not influenced by this number, as the data communication information is only used during the mapping of variables on addresses. In the figure, only the choice between tracking and acquisition, based on the variable *"locked"* and the setting of a parameter by a user command are retained.

All timing constraints can be expressed by a very limited set of numbers. The execution rate attributed to $event_1$ implicitly imposes a latency constraint on $thread_7$ and $thread_8$ or $thread_9$, for they must have finished execution before the next event occurrence. The – 0.1 sec. latency constraint limits the time available for initialisation. The *minimum inter-arrival time,* an attribute to $event_2$, is gives information to the software scheduling tools about the speed at which user commands can arrive. This information, although used in the scheduler, does not guarantee a certain response time. To achieve this, an explicit response time constraint should be imposed.

5 Conclusions

In this paper, we presented the Multi-Thread Graph model (MTG) which is developed as a backbone for our software synthesis system. Our model is powerful enough to capture control as well as data flow, decisions, concurrent behaviour, interaction with the environment and synchronisation, and can therefore handle complex industrial systems, such as the pager, which was described in the MTG model during a software synthesis experiment.

In the future, software synthesis will be integrated in the CoWare design framework [15], developed at IMEC. This will offer simulation possibilities of the concurrent processes in the MTG model. Future work on the model includes automating the expansion of high-level communication constructs, such as message passing, to the MTG shared memory paradigm and bringing hardware features that are relevant for the software synthesis, such as the time needed for a context switch, into the model. Also, the existing software synthesis mapping tools will be developed further. Automatic interface synthesis with hardware components will be investigated.

References

[1] J.K. Adams, D.E. Thomas, "Multiple-Process Behavioral Synthesis for Mixed Hardware-Software Systems," Proc. of ISSS'95, 1995.

[2] G. Berry, et al.,"The Synchronous Approach to Reactive and Real-Time Systems," IEEE Proc., 1991.

[3] J. Buck, et al., "Ptolemy : A Framework for Simulating and Prototyping Heterogeneous Systems," Int. J. Comp. Simulation, 4, 1994.

[4] M. Chiodo, et al., "Synthesis of Software Programs for Embedded Control Applications," Proc. DAC-95, 1995.

[5] P. Chou, G. Boriello, "Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems," Proc. ACM/IEEE DAC, 1994.

[6] F. Curatelli, et al., "Internal Representation of System Behaviour for Software Synthesis from VHDL," CHIPS project deliv. DC3, 1995.

[7] R. Ernst, et al., "Hardware-Software Cosynthesis of Microcontrollers," IEEE Design & Test of Comp., 10(3), pp. 64-75, 1993.

[8] R.K. Gupta, "A Framework for Interactive Analysis of Timing Constraints in Embedded Systems," Proc. Codes/CASHE, 1996.

[9] C.A.R. Hoare, "Communicating Sequential Processes (CSP)," Int. Series in Comp. Sc., Prentice Hall, 1985.

[10] D. Ku, G. De Micheli, "Relative Scheduling Under Timing Constraints," Proc. DAC-90, Orlando, FL, 1990.

[11] D. Lanneer, "Design Models and Data-Path Mapping for Signal Processing Architectures," Ph.D., IMEC, Leuven, Belgium, 1993.

[12] R. Lauwereins et al., "GRAPE-II.: A Tool for the Rapid Prototyping of Multi-Rate Asynchronous DSP Applications on Heterogeneous Multiprocessors," Proc. Workshop on Rapid System Prototyping, 1992.

[13] F. Thoen, et al., "Real-Time Multi-tasking in Software Synthesis for Information Processing Systems," Proc. of ISSS'95, 1995.

[14] F. Thoen, "Multi-Thread Graph (MTG) - A System-Level Representation Model for Software Synthesis," Techn. Report, IMEC, Leuven, Belgium, Apr., 1996.

[15] K. Van Rompaey, et al., "CoWare – A Design Environment for Heterogeneous Hardware/Software Systems," Proc. of EURO-DAC, 1996.

[16] T. Yen, W. Wolf, "Sensitivity-Driven Co-Synthesis of Distributed Embedded Systems," Proc. ISSS'95, 1995.