
Thierry Grandpierre, Yves Sorel

To cite this version:


HAL Id: hal-01800622

https://hal.archives-ouvertes.fr/hal-01800622

Submitted on 27 May 2018

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.

Thierry Grandpierre (t.grandpierre@esiee.fr)  
ESIEE Paris, Cite Descarte BP99  
93162 Noisy Le Grand Cedex France  
Yves Sorel (yves.sorel@inria.fr)  
INRIA, Domaine de Voluceau BP 105  
78153 Le Chesnay Cedex France

Abstract

This paper presents a seamless flow of transformations which performs dedicated distributed executive generation from a high level specification of a pair: algorithm, architecture. This work is based upon graph models and graph transformations and is part of the AAA methodology. We present an original architecture model which allows to perform accurate sequencer modeling, memory allocation, and heterogeneous inter-processor communications for both modes shared memory and message passing. Then we present the flow of transformations that leads to the automatic generation of dedicated real-time distributed executives which are deadlock free. This transformation flow has been implemented in a system level CAD software tool called SynDEx.

1. Introduction

The increasing complexity of signal, image and control processing algorithms in embedded applications, requires high computational power to meet real-time constraints. This power can be achieved by distributed (parallel) hardware architectures which are often heterogeneous in embedded systems: they are made of different types of processors (DSP, RISC...) interconnected by different types of communication medias (FIFO bus, shared memory...). The real-time implementation of an application algorithm onto such architecture is a complex task since numerous problems (which are not necessarily independent) must be solved:

- Distributed architectures involve the distributing (mapping) and scheduling of each algorithm functionality onto each processor. This distribution requires to add costly (time-consuming) communication operations to the algorithm in order to transfer data between operations executed by different processors.
- The real-time constraints of the application must always be met whatever the processor load and the communication media load are. In these systems, as explained in [8], communications are too often neglected although they may tremendously decrease the actual performances of the aforementioned applications.
- Since the target architecture is embedded, it is often necessary to minimize its size: this is a resource allocation optimization problem.
- Performances, which are required for real-time embedded applications, are mostly obtained by mixing high level language and low level language. In heterogeneous architecture each processor may have its own compiler and this consequently increases the code generation complexity.
- For industrial production, development costs must always be reduced. Since the debugging of the distributed application represents a costly part of the development cycle, it should always be minimized. More generally, time-to-market must always be decreased, the development of applications must always be shorter, it is then important to keep a seamless development flow in order to reuse as much work as possible between prototyping and industrialisation, but also to improve traceability.

In order to address these issues, we improve the AAA\textsuperscript{1} rapid prototyping methodology proposed in [13]. AAA is based on graph theory in order to model hardware architectures, application algorithms as well as implementations which are obtained by graph transformations. Afterwards, for easier readability, we will use the term “algorithm” instead of “application algorithm”. Algorithms are scheduled off-line (statically) onto the hardware architecture in order to predict a corresponding real-time behaviour and to build an optimized implementation by automatic code generation. Even communications are routed off-line in order to ensure real-time response. In the case of real-time (critical) embedded applications, off-line scheduling is preferred for two main reasons. First, it induces a very low temporal and spatial overhead compared to resident OS (because such OS involves in-line scheduling and then requires time and memory for the execution of its scheduler). Secondly, the

\textsuperscript{1}Algorithm Architecture Adequation.
Figure 1. AAA methodology flow and paper organization

deterministic behaviour of a real-time application is much easier to prove with off-line scheduling, especially if communications are also scheduled off-line [1][5]. On the other hand, off-line implementations are rarely compatible with rapid prototyping since developers must be in charge of everything (allocation, communication usually left to the OS) in order to implement a distributed application. Then it is usually very costly to test different algorithms mapping on different distributed architectures since this requires rewriting all the code. The goal of this paper is to address these issues by combining rapid prototyping and off-line scheduling. The principle is to automatically generate a distributed dedicated executive using graph transformations of any validated algorithm specification. If each transformation is correct, the generated executive is necessarily correct and no other post-validation process is required.

The AAA methodology should help the designer to rapidly obtain an efficient implementation (i.e. which meets real-time constraints and minimizes the architecture size) of these complex algorithms, and to simplify the implementation task from the specification to the final prototype.

Figure 1 shows the transformation flow of AAA as well as the paper organization: the first two sections are dedicated to our heterogeneous architecture model and to our algorithm model. Section 4 presents our implementation model of an algorithm onto an architecture, and presents our optimization principles. Section 5 is dedicated to the automatic executive generation from an implementation graph. The last section presents our software tool SynDE\textsuperscript{x} which implements AAA, and some applications developed with this tool.

2. Architecture model

In literature we find two main levels of hardware models: on the one hand there are high level models like PRAM, DRAM, BSP…[14]. They are known to be very useful when working on computing complexity, but they are based on a hypothesis which intentionally neglects the details of the underlying architecture. On the other hand, we found very accurate low level models implemented by HDL\textsuperscript{2} and ADL\textsuperscript{3}[10]. Such models are mainly used by designers of processors, simulators and compilers but rarely by high level programmers because of their high complexity.

In order to efficiently implement any algorithm onto any multiprocessor architecture, we need comparable models to specify the architecture with its available parallelism, and the algorithm with its potential parallelism. We can find such models in several CAD tools (Ptolemy[3], Casch\textsuperscript{[9]}, Trapper\textsuperscript{[12]}, VCC\textsuperscript{[15]}…), but they are often implicit and are also dedicated to a restricted subset of multi-processor machines. We need an architecture model allowing to specify different types of processors, accessing different types of memories and different types of media (each with its own characteristics: size, bandwidth, etc.).

Here we propose an architecture model which is at an intermediate level [7] between the two previously mentioned. The complexity of this model is sufficient to enable accurate optimization while it is not too fine and then does not lead to a combinatorial explosion. Based on graphs, this model highlights the available parallelism of any architecture. Its accuracy enables the performing of realistic real-time behaviour predictions of any algorithm implementation. This model enables different types of automatic optimizations of this implementation to be made possible, but it also enables automatic generation of dedicated executives.

The heterogeneous distributed architecture is specified as an oriented graph, called architecture graph, denoted by $G_{ar} = (V, E)$ where $V$ is the set of vertices of $G_{ar}$, and $E$ the set of edges.

The set of vertices $V$ is made up of four subsets corresponding to four kinds of Finite State Machines (FSM) called operator ($V_O$), communicator ($V_C$), memory ($V_M$) and Bus/Mux/Demux with or without arbiter ($V_B$): $V = V_O \cup V_C \cup V_M \cup V_B$, and $V_O \cap V_C \cap V_M \cap V_B = \emptyset$. Each edge $e \in E$ represents connections between the input and the output of FSMs, so that the hardware graph forms a network of FSMs.

There are two types of memory vertices, RAM (Random Access Memory, $S_{RAM} \in S_M$) and SAM (Sequential Access Memory, $S_{SAM} \in S_M$):

- RAM memories may be used to store operations of the algorithm graph (Cf. algorithm section 3), in this case

\textsuperscript{2}Hardware Description Language.

\textsuperscript{3}Architecture Description Language.
we call them RAM \(_{P}\) (program memory). When RAM store only data we call them \(RAM_{D}\) (data memory). We call them \(RAM_{DP}\) when they store both program and data. A RAM may be shared (i.e. connected to several operators and/or communicators), it then may be used for data communications. In 5.1 we will see how we avoid problem due to concurrent access by adding automatically synchronization operations,

- SAM memories are always shared since they are only used to communicate data using the message passing paradigm. In a SAM, data must be read in the same order as it has been written (as FIFO), all access is then said to be synchronized whereas in a RAM it is not synchronized since it is possible to read data independently of the order of the write operation. SAM may be peer to peer or multipoint and support or not hardware broadcasting. Each memory is characterised by its size and its access bandwidth.

There are two types of sequencer vertices, operator and communicator:

- each operator sequentially executes a finite subset of the algorithm’s operations stored in a \(RAM_{P}\) (or \(RAM_{DP}\)) which must be connected to the operator. The worst-case execution duration of an operation \(o\) on an operator \(s_o\) is denoted \(\delta(o, s_o)\). We use worst-case execution duration because we deal with real-time applications where it is crucial to satisfy the constraints whatever happens. This duration depends on the operator and memory characteristics. An operation executed by an operator reads input data stored in a connected \(RAM_{D}\) (or \(RAM_{DP}\)) and produces output data which is written in the \(RAM_{D}\) (or \(RAM_{DP}\)) connected to it,

- each communicator sequentially executes communication operations stored in their connected \(RAM_{P}\) (or \(RAM_{DP}\)). These operations transfer data from one memory (\(SAM, RAM_{P}, RAM_{DP}\)) connected to the communicator into another memory connected to the same communicator. The execution duration of a communication depends on the size of data to be transmitted but also depends on the available bandwidth computed from each parameters of the edge. A communicator is equivalent to a DMA channel for which we build a sequence of transfers (Cf. 5.3). It allows to decouple computation and communication and then to take advantage of the parallelism between computation and communication.
ture:

- when a single sequencer (operator or communicator) requires to access more than one memory, a BMD vertex must be inserted between the sequencer and each memory,

- when a memory is shared by several sequencer vertices, a BMD including an arbiter (a BMDA) must be inserted between all sequencers and the shared memory. Its characteristics are stored in tables which enable the computing of available bandwidth at any time. Arbiter are considered to have a deterministic behaviour. If it is not the case, we will use the worst case behaviour again because we deal with real-time applications.

In our model, a processor corresponds to an architecture sub-graph made of one operator and optionally one or several communicator(s) and BMD(s). The whole architecture model is given in [7] as well as the set of connexion rules that enable to build a valid architecture graph.

2.1. Example of architecture graphs

Figure 2-a depicts a very simple graph representing a mono-processor hardware architecture based on only one processor connected to two memory banks: the unique operator Opr1 is connected to RAM vertices R1 and R2 (which store both data and operation instructions) by a BMD vertex.

Figure 2-b represents a hardware architecture based on two processors communicating by a serial link. The two operators Opr1 and Opr2 are connected by two RAMDP (R1,R2) to their communicators (Com1,Com2) that respectively share a unique SAM memory (S). For example, an operation executed by Opr1 reads its data from R1, produces and writes a data into R1. Then a pair of communication operations executed on Com1 and Com2 cooperates to write this data into R2. Finally an operation B executed on Opr2 is able to read this data in order to use it in turn.

Figure 3 shows the architecture diagram of a PENTEK 4290 Quad C6201 board4 based on four Texas Instrument DSP (TMS320C6211). Figure 4 represents the corresponding architecture graph of this board. Each DSP is able to access three types of local memories (SB-SRAM,SDRAM,DPRAM) modeled by three memory vertices (RSB,RS,D). Each DSP is able to communicate data to two neighbors through two BI-FIFO modeled by Sbiifo vertices. Each empty circle represents a BMDA vertex. This board has been used to implement real-time image processing applications with AAA/SynDEx.

4www.pentek.com

Figure 5. Architecture graph of Mephisto SoC

Figure 5 shows the architecture graph of the System on Chip “Mephisto” made by Thales. This SoC [2] is based on a SIMD processor (Opr1), a DSP processor (Opr2) and an ARM7 core cpu (Opr3). Each pair of communicator shares a SAM (S1,S2,S3,S4) or a RAM (R0,R2,R3) in order to communicate. Each operator is connected to its own memory (R1,R4,R5).

3. Algorithm Model

An algorithm is a finite sequence (total order) of operations directly executable by a FSM. If we want to use efficiently a multiprocessor architecture, composed of several FSM giving some available parallelism, algorithms should be specified with at least as much potential parallelism as the available parallelism of the architecture. Notice that it is always preferable to specify the algorithm with much more potential parallelism than available parallelism in order to take advantage of possible choices. Moreover, since we want to be able to compare the implementation of an algorithm onto different architectures, the algorithm graph must be specified independently of any architecture graph. Thus, we extend the notion of algorithm to an oriented hyper-graph G_al of operations (graph vertices O), whose execution is partially ordered by their data-dependences (oriented graph edges D) so that \( D \subseteq O \times 2^O \), \( G_al = (O,D) \).

This data-dependence graph, also called directed acyclic graph (DAG), exhibits a potential parallelism: two operations which are not in data-dependence relation, may be executed in any order by the same operator or simultaneously by two different operators. We need a hyper-graph model (an example is given in figure 6) because each data-dependence may have several extremities but only one ori-
1. spatial allocation of the operations onto the operators, leading to inter-operator edges (data-dependence between two operations belonging to two different operators),

2. each of these edges is replaced by a linear sub-graph (sequence of edges and vertices) that will be detailed in 4.3. We will see that this sub-graph includes communication operations allocated to communicators,

3. for each operation allocated to an operator, we will see in 4.3 that new “memory allocation vertices” are added and allocated to one of the RAM memories connected to the operator.

### 4.2. Scheduling

The scheduling is modeled by the relation sched applied to a pair \((G_{al}, G_{ar})\) so that \((G_{al}, G_{ar}) \xrightarrow{\text{sched}} (G'_{al})(\cdot)\). For each operator (resp. communicator) the scheduling is a temporal allocation of the operations (resp. communication operations) allocated to this sequencer. This amounts to the adding of “precedence without communication edges” \((e_p \in E_p)\) in order to transform the partial order associated to the operations allocated to an operator (resp. communication operations allocated to a communicator). This is necessary because operators (resp. communicators), which are FSM, require an execution order between the operations (resp. communication operations) allocated to them. This order must be compatible with the precedences required by the data-dependences. Then, if operations executed by a same sequencer are not directly or transitively ordered, this execution order is specified by adding precedence edges \(e_p\) between them. Valid implementation graphs are obtained using the following rules, afterwards we will see how to build optimised implementation graphs.

### 4.3. Distribution and scheduling rules

In order to execute an operation (resp. a communicating operation), an operator (resp. a communicator) needs to read all of the instructions in its connected \(RAM_p\): we model this by adding an allocation vertex in the algorithm graph. Such vertex, denoted \(alloc_p \in V_{alloc}\) is associated to \(RAM_p\) and connected to the operation by a non-oriented edge. If the operation requires some local variables to perform computation, we add a second allocation vertex denoted \(alloc_{lo} \in V_{alloc_{lo}}\) associated to one \(RAM_D\) connected to the operation's operator.

The execution of each operation by an operator consists in reading the operation's input data from one of the operator memories (\(RAM_D\), \(RAM_D^{pp}\)), then in combining them to compute the output data, which is finally written into one memory connected to the operator. Therefore, when two
by construction the total order of operations in data-dependence (edge of algorithm graph) are executed by the same operator, the operation producing the data must be executed in sequence before the operation consuming the data. We model the allocation into one of the connected memories by adding an allocation vertex to the algorithm graph. This vertex, denoted allocD ∈ VallocD, must be associated to the allocated memory.

These three types of allocation vertex are connected to their producing and consuming operations by non-oriented hyper-edges ealloc ∈ EA. As explained hereafter, such allocation vertices will be used in order to predict and minimize memory allocation as well as the duration of the execution of operations.

When two data-dependent operations are executed by two different operators, the data must be transferred, from the RAM memory of the operator executing the producing operation (after its execution), into the RAM memory of the operator executing the consuming operation (before its execution). Such a data-dependence is called an inter-operator data-dependence. In order to support it, a route (path in the architecture graph) must be chosen between the two memories connected to the operators. For each communicator (resp. BMD) composing this route, a communication operation ecom ∈ Ec (resp. identity vertex eι ∈ Ev) must be inserted into the algorithm graph. For each memory between the producing and the consuming operations an allocation vertex allocD is also added.

Finally, from a pair of algorithm and architecture graphs we get the set of the possible implementation by the composition of the two previous relations: (Galg, Garch) → distsched (G′ arch)(G′ al)(1) where G′ arch = (O ∪ VallocD ∪ VallocD ∪ VallocD ∪ Vc ∪ Vr ∪ D) and G′ al has been transformed in a total order. The partial order of the algorithm graph G′ al is also added. The partial order of the algorithm graph Ga creates a total order and is used to order the execution of operations on the architecture.

Figure 7 shows a simple implementation example of the algorithm graph presented in figure 6 onto the architecture presented in figure 2-b. Such an implementation graph is automatically and quickly generated from the results of the optimization heuristic given in the next section. In this example we want A, B, and C to be executed by Opr1 and D executed by Opr2. Consequently two pairs of communication operations (sendBD, receiveBD and sendCD, receiveCD) must be inserted and associated to com1 and com2 in order to realize data transfers on the shared SAM S (7-a). Allocation vertices (allocAB, allocBD, allocAC . . . ) have also been added in order to model all required memory allocations (7-b). Since operations B and C, which are not dependent, are allocated to the same operator, they may be executed in parallel if allocated to different operators because there is no data-dependence edge between them: an execution order may be chosen between them7. Thus, we add a dependence edge between B and C and symmetrically between the communication operations (bold arrows of figure 7-c).

4.4. Optimization of the implementation

When several operators are able to execute an operation, one of them must be chosen in order to execute it. Consequently, for a given pair of algorithm and architecture graphs, there is a large but finite number of possible implementations, among which we need to select the most efficient one, i.e., the one which satisfies real-time constraints and uses the least possible architecture resources. This optimization problem, as most resource allocation problems is known to be NP-hard, and its size is usually huge for realistic applications. This is why we need to use heuristics.

The one we chose is based on a fast and efficient greedy list scheduling algorithm, with a cost function based on the “critical path” and the “schedule flexibility” of the implementation graph: it takes into account the execution dura-

---

7Since space is lacking, it is not possible to illustrate an implementation on a more complex architecture (such as the four DSP presented in figure 2-b). The one we chose here (figure 2-b) is sufficient and will be used to illustrate all transformations (including code generation).

8Notice that in this example, for pedagogical reason, we do not take advantage of the potential parallelism between operation B and C.
not require any resident real-time executive since we generate all the necessary services from a generic library that we provide, and which is small enough to be easily translated to support different processors as explain later. The executives that we will be automatically generated are said to be dedicated because they are perfectly tailored for each real-time embedded application. Since no real-time operating system (RTOS) is required, we can guarantee the deterministic behaviour of our real-time execution. All off-line optimizations which have been made by the distribution and scheduling heuristic are then necessarily implemented efficiently.

Executive generation corresponds to the final graph transformation of our methodology. This generation is performed following four steps [7]: (1) transformation of the optimized implementation graph into an executive graph, (2) transformation of the executive graph into as many sequences of macro-executive as there are of processors, (3) transformation of each sequence of macro-executive into a source file, (4) compilation, downloading and execution of each source file.

5.1. From implementation graph to execution graph

This transformation is modeled by the relation exec: \((G^I, G^E) \xrightarrow{\text{exec}} (G^E)'\). It consists in adding new types of vertex: Loop, EndLoop, pre-full/suc-full, pre-empty/suc-empty vertices. This is done in two steps:

1. since we deal with reactive applications (i.e. applications in constant interaction with the environment that they control) we need to make the sequence of operations allocated to each operator repetitive: in each operator partition we add and connect a Loop vertex before the first operation and an EndLoop vertex after the last operation (Cf. figure 11),

2. since operator and communicator are independent sequencers, it is necessary to synchronize the execution of data dependent operations executed by different operators and/or communicators, i.e. this corresponds to the implementation of inter-partition edges, drawn with bold arrows on figure 10 and 8. In order to realize these synchronisations we replace each inter-partition edge by a linear sub-graph made of an edge connected to a pre-full vertex which is itself connected to a suc-full vertex (right part of figure 10). The pre-full (resp. suc-full vertex is allocated to the same partition as the producing (resp. consuming) operation. Pre-full and suc-full vertices model the operations which are able to read-modify-write a binary semaphore. This semaphore is allocated into the memory shared by the sequencers of the producing and consuming operations.
which must be synchronized. If suc-full (which precedes the consuming operation) is executed before the connected pre-full (which follows the producing operation) then the suc-full waits until the end of the pre-full execution. This ensures a correct execution order. If pre-full is executed before the corresponding suc-full it just changes the state of the semaphore without waiting: the operation following pre-full is executed in sequence. We also need to avoid a producing operation overwrites the content of a buffer which has not yet been sent: then we inserted a pair of pre-empty/suc-empty vertices. Pre-empty is inserted after the consuming operation while suc-empty is inserted before the producing operation. Notice that synchronisation transformations are not required if the sequencers share a SAM since this type of memory ensures a write-read synchronisation in hardware.

Synchronizations operations are fundamental in distributed systems since they guarantee that each data-dependence of the algorithm graph is implemented correctly. They guarantee that all buffers are always accessed in the order specified by the data-dependences in a way that this order is satisfied at runtime independently of the execution durations of the operations. Therefore the implementation optimization, even if it may be biased by inaccurate architecture characteristics, is safe in the sense that it cannot induce runtime synchronization errors (such as deadlocks, or lost data). This certitude allows big savings in application coding and at debugging times. Such synchronizations are often hand-written in usual design: deadlocks may then occur if the designer misses one of them or does not write them in the correct order.

Finally, since synchronizations operations are added in order to guarantee the partial execution order specified in the initial algorithm graph, and because the implementation of our synchronization reflects exactly our models, we do not have to consider any run-time overhead (as consensus waiting problem) induced by synchronization.

Figure 11 depicts a complete example of the execution graph obtained after the transformation of the implementation graph of figure 8. Loop/EndLoop vertices has been added on Opr1, Com1, Com2 and Opr2 operations. In order to lighten the graph allocation vertices are not drawn.

5.2. From execution graph to macro-executives

Once the executive graph has been built, we transform the sub-graph allocated to each operator (processor) of the architecture graph into a sequence of macro-instructions. The use of a macro code enables to mix easily different programming languages (C, asm, Fortran, SystemC... ) that can be found in heterogeneous architecture.

The structure of a macro-executive sequence of an operator opr is sequentially composed of 3 sections:

- a list of macros allocating memory buffers (lines 2-5 of fig.12): for each allocation vertex allocated to each RAM connected to opr we generate an alloc(name, type, size) macro. Where name is name of the operation producing the data, type the data format and size the number of data,

- as many communication sequences (lines 6 to 14) as existing communicators connected to opr (only one communicator is connected to each operator in our example). This sequence is generated between a pair of com_thread, end_thread macros. Such a sequence is built by exploration of the sequence of totally ordered vertices allocated to the communicator partition. For each vertex of the sequence we generate a corresponding macro (send, receive, read, write, pre-empty...). The arguments of these macros are computed from the edges connected to their corresponding vertices,

- a unique computation sequence (line 16 to 29). This sequence is generated between a pair of main, end_main macros. Such a sequence is also built by exploration of the sequence of totally ordered vertices allocated to the operator partition. The spawn(com1) macros (line 17) has to run the communication thread com1. This thread is executed under DMA interrupt (end of transfers interrupt) of the main computation thread which have a lower priority in order to be able to be interrupted. In the computation sequence suc macros are implemented by “active waiting” (polling) while in the communication sequences they are implemented by “passive waiting”.

In order to generate an implementation code whose partial order is identical to the implementation graph, it is im-
Figure 12. Macro code from graph of figure 8

important to remind that the translation/print process follows the exact order given by intra-sequencer vertices.

5.3. From macro-executives to source code

Each sequence of macro-instruction is translated by a macro-processor (we use gnu-M4) into a source code written in the best suited compilable language for each target operator. A macro is translated either into a sequence of in-lined instructions, or into a call to a separately compiled function. These macros are classified into two sets corresponding to two kind of libraries. The first one is a fixed set of system macros, which support code downloading, memory management, sequence control, inter-sequence synchronization, inter-operator transfers, and runtime timing (in order to characterise algorithm operations and to profile the application). The second one is an extensible set of application dedicated macros, which support the algorithm operations.

Once the executive libraries have been developed for each type of processor, it takes only few seconds to automatically generate, compile and download the deadlock free code for each target processor of the architecture. It is then easy to experiment different architectures with various interconnection schemes.

Figure 12 is a simplified example of code generation obtained by transformation of the execution graph given in figure 11. This example will focus on processor 1 (the code of processor 2 is generated symmetrically):

- the unique communicator sequence is generated between a pair of com_thread(com1) and end_thread(com2) macros. Then we build the contents of the communication sequence (line 6 to 14) of the communicator com1 of figure 12. Each communication vertex scheduled on com1 is translated into a send (line 9) or a receive macro, each synchronization vertex is translated into the corresponding macros pre-empty/full and suc-empty/full in order to synchronize the communication sequence with the operator sequence (pair of macro on line 10/21, 23/8, 13/25, 27/11 synchronize locally the operator sequence with the communication sequence),

- the unique operator sequence is generated between a pair of main and end_main macros (line 16 to 29): each operation vertex is translated into a macro with the same name (A, B and C for opr1, line 19, 22 and 26) taking the allocation vertex name as an argument.

Theses files are then translated into the language of the target processor by the m4 macroprocessor with the help of a generic processor specific library (containing the definitions of each “system” macro, and the application library). For example, if the compiler of the target processor accepts the C language, the translation of a alloc(ac, int, 1) macro will be int alloc_ac; . The translation of a send macro may be DMA_send(alloc BD, sizeof(alloc BD), com2) with com2 the address of a media writable by a DMA channel of the processor. The implementation of synchronization macros are often coded in assembly language since performance and context switching minimization between communications and computation sequence are required, i.e. context switches only occur between the communication sequences (which are composed of system macros only) and the computation sequence, then only few registers need to be saved and restored. The size of the current C executive library for DSP TMS320C6211 from Texas Instrument is 23Ko of source code, the size of the executive library dedicated to the DMA/FIFO communications of the Pentek4290 board is less than 10ko of source code. It is not a complex task to support a new architecture since skeleton of each library may be identical.

Finally, since the execution order of the generated code is coherent with the partial order of the algorithm graph, this ensures a correct real-time execution.

6. Related Work

The whole methodology is implemented in the system level CAD software SynDEx®[8]. Its graphical user inter-
face enables the user to specify both the algorithm and the architecture graphs, to execute the aforementioned heuristic and then to display the resulting distribution and scheduling on a timing diagram. When the user is satisfied by the predicted timing, SynDEx can automatically generate the deadlock free executive for the real-time execution of the algorithm onto the multiprocessor. Real-time distributed executive libraries have been developed for networks based on DSPs (TMS320C6x, TMS320C40, ADSP21060), micro-controllers (MPC555, i87C196KC, MC68332), and general purpose processors (PC and UNIX workstations). SynDEx has been used to develop several real-time heterogeneous applications, among which [11]: a semi-autonomous urban electric vehicle (controlled by five Motorola PowerPC MPC555 micro-controllers and a CAN bus), image processing application on multi-DSP [6], digital signal processing on a System On A Chip [2].

7. Conclusion

As it has been mentioned in section 3 our algorithm model supports more complex specification with loops and conditioned operations which has not been detailed in this paper since this does not modify the transformation flow.

The implementation task, from the high level specification to the code execution, has been expressed in terms of graph transformations and lead to a seamless development flow which improves traceability and which can be automatized. Thanks to these transformations, the partial order given by the automatic executive generation is coherent with the algorithm partial order, this guarantees a deadlock free distributed execution. Thanks to our implementation model, and although the algorithm graph is a unique assignation model, off-line memory re-allocation is achievable and enables memory optimizations. Thanks to our architecture model, it is possible to cover a large amount of architectures based on various memory and communication networks. Thanks to the chosen level, this model enables the accurate performing of behaviour prediction. It is then well suited for resources optimization. Communications, which are crucial in real-time embedded application, are carefully taken into account during optimization and executive generation. Finally, these models and transformations rules make possible the generating of off-line optimized distributed executives even for the rapid prototyping step.

Efforts are now made in order to specify and implement an equivalent seamless transformation flow in order to support architectures based on reconfigurable circuit [4].

References