

SynDEx Reference Manual

A graph oriented methodology and its system level CAD software for the optimization of distributed real-time embedded applications

Yves Sorel
INRIA Rocquencourt,
BP 105 - 78153 Le Chesnay Cedex, France
yves.sorel@inria.fr
www.syndex.org

Contents

1	Introduction	3
1.1	Context	3
1.2	Goals	4
1.3	Definitions	5
2	Application specification	6
2.1	Functionalities	6
2.2	Hardware	7
2.3	Constraints	7
3	The AAA methodology for optimized implementation	8
3.1	Algorithm model	8
3.1.1	Control and data flow graphs	8
3.1.2	Factorized conditioned data dependence graph	9
3.2	Architecture model	12
3.2.1	Multicomponent	12
3.2.2	Architecture characterization	13
3.3	Implementation model	14
3.3.1	Distribution and scheduling	14
3.3.2	Impact of the granularity and potential parallelism	17
3.4	Optimized implementation: adequation	19
3.4.1	Principles	19
3.4.2	Example of adequation heuristics	19
3.4.3	Resources minimization	22
3.5	Executives generation	22
3.5.1	From implementation graph to execution graph	24
3.5.2	From execution graph to macro-code	25
3.5.3	Macro-code to source files	28
3.5.4	Example of macro-code	28
3.5.5	Example of macro-definition	28
4	SynDEx: system level CAD software	29
5	Conclusion	30

1 Introduction

1.1 Context

The software in complex applications found in domains such as automobile, aeronautics, telecommunications, etc, is growing rapidly. On the one hand it increasingly replaces mechanical and analog devices which cost a lot and are too sensitive to failures, and on the other hand it offers to the end-users new functionalities which may easily evolve. These applications share the following main features:

- *automatic-control and discrete-event*: they include control laws, possibly using signal and image processing, as well as discrete events, in order to schedule these control laws through finite state machines;
- *critical real-time constraints*: they must satisfy input sampling rates (periodicity) or/and deadlines, otherwise the application may fail leading to a human, ecological or financial disaster, later on we shall use real-time by default;
- *embedding constraints*: they are mobile and rely on limited resources because of weight, size, energy consumption, and price limitations;
- *distributed and heterogeneous hardware architecture*: they are distributed in order to provide enough computing power through parallelism, but also for the purpose of modularity, and to keep the sensors and actuators close to the computing resources. Furthermore, fault tolerance imposes redundant architecture to cope with hardware components failures. They are also heterogeneous because different types of resources, processors, specific integrated circuits (ASIC, FPGA), and communications (link, bus), are necessary to implement the aimed functionalities while satisfying the constraints. The reader must be aware that distributed real-time systems are considerably more difficult to tackle than centralized ones (only one type of resource), it is the reason why the most significant results in the literature are given for this latter case.

Taking all these features into account is a great challenge, that only a formal (based on mathematics) methodology may properly achieve. Indeed, typical methods based on the one hand on specification graphical languages such as SADT (Structured Analysis and Design Technique), and on the other hand on C programming and RTOS (Real-Time Operating System) for the implementation, are not efficient enough to cope with the complexity of the target applications, mainly because there is a gap between the *specification* step and the *implementation* step. However, this does not mean that the application should not be carried out with respect to the constraints, but that the development cycle will have a too long duration, essentially due to the real-time tests which must cover as many cases as possible.

Therefore, we propose a two steps approach without any gap reducing significantly the development cycle time:

- a formal specification of the application, allowing verifications very early in the development cycle in order to eliminate logical errors, i.e. in terms of events order;
- an optimized implementation guaranteeing the formal properties proved during the specification. This approach relies on graph transformations from the specification up to the automatic code generation, noticeably reducing real-time tests.

In this paper we will focus only on the second step by presenting a summary of several research works carried out last past years on this subject. Concerning the first step, we rely on the well known denotational semantics of synchronous languages [1] such as Esterel, Lustre, Signal or Statecharts. They all offer a formal framework where it is possible to demonstrate useful properties when specifying applications with critical

real-time constraints. Nowadays, commercial tools providing modern GUI (Graphical User Interface), based on this semantics, are available on the market. More and more industries in the fields we are interested in, use this approach in order to specify complex applications. For example, it is well known that several car manufacturers use Statemate (the tool based on Statecharts) in order to specify their embedded systems, for sequencing control laws involved for controlling the engine or the brakes, as well as for managing the events triggered by the user when executing common tasks such as opening or closing a door, turning the ignition key, signaling direction modifications, etc . . . Similarly, Scade (the tool based on Lustre) is used to specify avionics applications. The crucial issue in both cases is actually a matter of ordering the different operations necessary to perform each specified functionality. The next step consists in implementing these operations through software and hardware. At this high level of specification, thus very early in the development cycle, it is possible to verify logical properties such that an event will never occur, or will occur only if another event occurred a specified number of times. In this paper the term *event* is used in a broad sense, no assumption is made whether it refers to a periodic or to an aperiodic signal, both types of signal are considered as a set of events. These formal verifications are based on “model-checking” techniques [2] using BDD (Binary Decision Diagram) [3] for solving these combinatorial problems. It is important to understand that only properties in terms of event ordering are demonstrated in this framework, and therefore it is not possible to say that the real-time constraints were satisfied. However, they prevent from a large amount of errors found in real-time applications. At this specification level, we may carry out a functional simulation where the hardware is not actually considered. It is worth noting that in the typical methods mentioned before, these logical errors are usually discovered during real-time tests, consequently it is very difficult to find their causes at the application specification level, mainly because of the gap between the specification and the real-time implementation. This has an heavy price that the manufacturers are tired to pay, it is the reason why they are ready to invest in new methods and their associated tools.

1.2 Goals

Assuming that an application specification has been performed with a language verifying the aforementioned semantics, and that some logical properties have been demonstrated, the goal of the AAA methodology is to optimize the implementation of this specification. That is to say, that the implementation will satisfy the specification in terms of functionalities, and will satisfy the real-time and embedding constraints, while the logical properties shown previously are maintained. This approach increases the dependability of the application, especially if fault tolerance is specified at the level of the application.

AAA stands for Algorithm Architecture Adequation, adequation meaning an efficient mapping of the algorithm onto the architecture.

In order to achieve our goals, we chose very soon in our research works the “off-line” approach for optimizing implementations. Indeed, the implementation of an application specification onto an hardware architecture corresponds to a resource allocation problem. There are two possible resource allocation policies : “on-line” or “off-line”. It is now generally admitted that “off-line” policies are better suited for critical real-time, that is to say, when it is mandatory that real-time constraints are met, because otherwise dramatic consequences may occur. These policies have two main advantages: first they are deterministic and second they induce very low executive overhead. Thus, even if these approaches are more difficult to implement and may be costly in resources, they must be applied in order to avoid these consequences which may have an higher price. For the rest of this paper we will assume to be in this case. Of course, when real-time constraints are not critical more simple policies are used.

1.3 Definitions

In order to avoid ambiguities, it is necessary to be precise about definitions such as application, physical environment, reactive system, algorithm, architecture, implementation, and finally adequation which will be used afterwards.

In the AAA methodology, an *application* is a system composed of two sub-systems in interaction. The first one called *physical analog environment*, is controlled by the second one called the *digital controller*, because it is assumed to be based on computers. This latter is a *reactive system* [4] meaning that it must mandatorily react to the variations $U(t)$ of the physical environment state (discrete input for the controller through the analog to digital converter (ADC) of a sensor, t is an integer) in order to produce a control $Y(t)$ for the physical environment (discrete output of the controller provided to the physical environment through the digital to analog converter (DAC) of an actuator) and a new state for the controller $X(t)$. $X(t)$ and $Y(t)$ define respectively input events and output events consumed and produced by the reactive system. Both $X(t)$ and $Y(t)$ are functions of the physical environment state and the previous state of the controller ($U(t)$, $X(t)$ and $Y(t)$ may be vectors) given by the equation 1.

$$(Y(t), X(t)) = f(U(t), X(t-1)) \quad (1)$$

Real-time systems are, first of all, reactive systems for which a maximum delay must be imposed between an input event arriving into the system and an output event produced by the system, in reaction to this input event. Usually, an output event is obtained from an input event processed by operations on which precedence constraints may be imposed.

There are two kinds of real-time constraints: the *latency* corresponds to the duration of a reaction between an input event and the output event the input triggered, the *cadence* corresponds to the periodicity of an input (i.e. the duration between two consecutive reactions). The latency refers to the elapsed time between an input and the resulting output, whereas the cadence refers to the sampling rate of an input. In the general case more than one latency or/and cadence constraints are specified.

The reactive system is composed of two parts, the hardware called *architecture* and the software called *algorithm*. We use the term architecture because we are mainly interested in the structure of the hardware. More precisely, we consider *multicomponent architecture* because its structure, which provides *physical parallelism*, usually includes sensor and actuator, “programmable components” or processors: RISC (Reduced Instruction Set), CISC (Complex Instruction Set), DSP (Digital Signal Processor), microcontroller (incorporating ADC/DAC, real-time clock, etc. . .), and “non programmable components” (application specific integrated circuit ASIC possibly reconfigurable like FPGA), all interconnected through communication resources. A multicomponent is heterogeneous due to these two types of components, but also different types of processors and integrated circuits may be used as well as different types of communication resources.

An algorithm is the result of the transformation of an application specification, which may be more or less formalized, in a software specification adapted to its digital processing by a computer or a specific integrated circuit. More precisely, as defined by Turing [5] an algorithm is a *finite sequence of operations* (total order) that must be processed in a finite time and with a finite hardware support. We need here to extend this notion of algorithm in two directions. On the one hand we have to take into account the infinite repetition of reactive systems, and on the other hand we have to take into account parallelism, which is necessary for the distributed implementation of an algorithm. However, for each reaction, the number of necessary operations to produce the control for the physical environment must be finite because real-time constraints must be satisfied. Consequently, instead of a total order (sequence of operations) we prefer a partial order which describes a *potential parallelism*, often called “inherent parallelism”. It is different from the physical parallelism provided by the hardware. It is worth noting that when we speak of an algorithm, it possibly means that it is a set of algorithms, rather than a unique algorithm.

Embedding constraints correspond to the number of processors and communication resources, the amount

of memories for a multicomponent, the number of combinatorial functions in an integrated circuit, its surface, or its power consumption, etc. . .

The *implementation* of a given algorithm onto a given multicomponent architecture consists in allocating the architecture resources to the operations defining the algorithm. Architecture resources are mainly the sequencer of a processor and of a communication resource if it has one (otherwise the processor sequencer is borrowed), and the memories (program and data). Then after compiling, resetting the different processors and loading the different programs, after resetting the specific integrated circuits (note that it is only necessary to allocate their memory because they are not programmable, i.e. they have been designed only to perform a specific operation), the application may be run. The implementation of a given algorithm onto a specific integrated circuit architecture which is to determine, also consists in allocating the architecture resources to the operations defining the algorithm. In this case architecture resources are combinatorial and sequential circuits created from the algorithm specification seeking for a compromise between the surface occupied by these circuits and the real-time constraints. The implementation of an algorithm on a multicomponent corresponds to an *hardware/software codesign* where the part of the algorithm distributed onto the processors and the part distributed onto the integrated circuits, corresponding to the partitioning, are decided a priori by the user.

Finally, an *adequation* consists in searching, among all the possible mappings of the algorithm onto the architecture, for the one which corresponds to an *optimized implementation*. We use this notion of optimized implementation although it is impossible to guarantee that an optimal solution has been found for this kind of problems (multicomponent or integrated circuit) which complexity is said NP-hard (i.e. non polynomial relatively to the number of algorithm operations and architecture resources). Hence, it is preferable to obtain rapidly an approximate solution than an optimal solution which may take too much time compared to the human life. The search for an optimized implementation is oriented by, on the one hand the real-time constraints (latency and cadence), and on the other hand the embedding constraints (hardware resources). If the real-time constraints are impossible to satisfy while the potential parallelism is completely exploited relatively to the physical parallelism, it is necessary to modify the algorithm itself in order to increase its potential parallelism. Note that the adequation is an iterative process where the architecture influences the algorithm and vice versa.

The document is organized as follows: we first present how to specify an application, that is, the functionalities it is supposed to perform, corresponding to our notion of algorithm, the hardware components that can be used, corresponding to our notion of architecture, and the real-time and embedding constraints. Then, we present the AAA methodology based on graphs models for the algorithm, the architecture, and on graph transformations for the possible implementations and the executable codes. We present the optimization techniques corresponding to the adequation. Finally, before concluding, we briefly present the system level CAD software SynDEx associated to the AAA methodology.

2 Application specification

In order to specify an application it is necessary to describe its functionalities, the hardware which may be used in order to implement these functionalities, and finally the real-time and embedding constraints the application has to satisfy.

2.1 Functionalities

Functionalities stand for the operations the application has to perform, but also when it is useful, for the data transfers between operations and/or the informations about the relative execution order of the operations and of the data transfers.

Usually, high level languages, often said “application domain oriented”, are used in order to specify the functionalities the application must perform. Such a language is the “entry point” of a programming environment (workshop) usually based on a graphic user interface (GUI) which simplifies the user’s work. There are several possibilities for these languages, but presently modeling languages based on object oriented approach, are the most popular. UML [6] is the best known of these object oriented languages, and several commercial programming environments (tools) are proposed, which are more or less application domain oriented. AIL (Automobile Architecture Implementation Language) is an example of such a programming environment defined by the French car manufacturers and providers, and based on a specialization of UML.

However, although these languages are very useful for specification purposes because of modularity, reutilization, genericity, etc, they do not offer a “denotational semantics” allowing formal verifications, and as it will be underlined later on, optimizations. On the other hand, even though synchronous languages are not object oriented languages they have a denotational semantics, allowing to verify properties in terms of events ordering, very early in the development cycle. This is the reason why in the AAA methodology we chose that the algorithms, directly issued from the application specification, have this semantics. Nevertheless, there are works in progress which aim to interface UML with the synchronous languages Esterel and Signal, in order to associate in a unified framework the best of both worlds.

2.2 Hardware

We address two kinds of hardware: the programmable components and the non programmable components. The first kind of components corresponds to general purpose processors of type RISC and CISC, to processors oriented towards signal and image processing (DSP), or to microcontrollers, used in complex computers (parallel machines, multiprocessors) when they are connected through a shared memory or a network using message passing, and thus providing physical parallelism. Each processor executes a program performing a part or the whole specified application. The second kind corresponds to ASIC (Application Specific Integrated Circuit), a potentially infinite set of logic gates connected together in order to perform the specified application, or to FPGA (Field Programmable Gate Array), a limited set of logic gates the interconnection of which may be configured more or less rapidly in order to perform the specified application, or only a part of the application if the number of gates of the FPGA is not sufficient. ASICs and FPGAs both provide physical parallelism at the level of each logical gate. Both kinds of components may be mixed leading to a multicomponent. The communications between the different components, whatever their kinds are, must be carefully taken into account in order to offer the best performances because they are crucial in complex multicomponent architectures. Indeed, it is well known that nowadays performances of parallel architectures strongly depend on the performances of their communication mechanisms.

For a programmable component, we are mainly interested in its sequencer because it will execute sequentially the set of the application operations that have been distributed onto this component. This means that the potential parallelism of the algorithm must be locally reduced to match with the physical parallelism of the given architecture. Similarly, the set of data transfers between operations distributed onto different sequencers, is going to be executed sequentially by the communication sequencer, if it exists, belonging to a communication resource, or otherwise by borrowing the operations sequencer. In the first case operations and communications may be executed in parallel whereas in the second case they may not.

Regarding the development process of the application, it is worth noting that the first kind of component induces flexibility and low cost, whereas the second one induces performance but high cost.

2.3 Constraints

As mentioned before, two kinds of constraints may be specified: real-time and embedding ones. Usually, application specification languages do not provide such possibilities, so these constraints are specified at the

level of the implementation process. Nevertheless, there are works in progress aiming at specifying at least real-time constraints at the level of the application specification [7]. For example, in the AIL language it is possible to specify latency constraints between a sensor and an actuator. Generally these constraints are called “end-to-end”. Similarly, it is possible to specify a period for each sensor. Embedding constraints are usually taken into account in the CAD tools for the specific integrated circuits. There are only few approaches which allow to take into account accurately all types of hardware resources in the case of multi-component architectures.

3 The AAA methodology for optimized implementation

The AAA methodology is based on graphs in order to model the algorithm as well as the architecture. Therefore, a possible implementation of an algorithm onto an architecture may be specified as a graph transformation. The adequation amounts to choose among all the possible implementations (graphs transformations), the one which satisfies real-time and embedding constraints, corresponding to the optimized implementation. Finally, the code generation is an ultimate graph transformation leading to a distributed real-time executive for the multicomponents and to a structural hardware description, e.g. structural VHDL, for the specific integrated circuits. This graph oriented approach relies on a formal framework where it is possible to describe and verify all the steps from the specification to the real-time execution of the application. This allows to ensure a high level of dependability because there is actually no gap between these steps.

Moreover, if fault tolerance is specified at the level of the application by the user who describes the redundant computation and communication resources, then it is also possible to automatically add redundant operations and data dependences to the algorithm graph, which are taken into account during the implementation and the optimization, guaranteeing the behavior of the application if the specified hardware components fail. This issue will not be addressed here, but the interested reader may consult [8, 9].

3.1 Algorithm model

3.1.1 Control and data flow graphs

There are two main approaches for specifying an algorithm: the control flow and the data flow. In both cases the algorithm may be modeled by a directed graph [10]. It is the meaning given to the edges, which will differentiate both approaches. Briefly, we remind the reader that a graph (V, E) is a pair of sets, the set of vertices V and the set of edges E , each edge $e = (v, w)$ being a pair of vertices, and then $E \subseteq V \times V$. Directed means that the order of the vertices in the pair (v, w) is considered, whereas in non directed graphs it is not.

A “program flow chart” is a typical example of control flow graph, usually used before programming with an imperative language like C. Each vertex of such a graph represents an operation which consumes from, and produces data into variables during its execution, and each edge represents an execution precedence relation between the two operations. Actually, an edge is a sequence control which corresponds either to an unconditional (back or forth) or to a conditional branching. This latter is the basic mechanism when an operation or a subgraph of operations must be conditioned for example by the result of a test (“if . . . then . . . else . . .”). The notion of iteration or loop (“for $i=1$ to n do . . .”), related to unconditional back branching, corresponds to a *temporal repetition* (in opposition to *spatial repetition* used later on) of an operation, or of a subgraph of operations. A “state chart” is another commonly used control flow graph, where each vertex represents one of the possible states, and each edge represents a transition, from one state to another one, triggered by the arrival of an event. Each transition leads to execute an operation which consumes from and produces data into variables. In both cases the set of all edges defines a total order on the execution of all operations. There is no potential parallelism directly specified in this model, although it might be extracted

from a data dependence analysis through the variables in which the operations read and write data. However, in the general case this analysis is very complicated and may conclude that no potential parallelism is available in this control flow graph. Moreover, there is no relationship between the order in which the operations must be executed, and the order in which these operations consume (read) from and produce (write) their data into the variables. Another way to specify potential parallelism consists in composing several control flow graphs which will communicate through shared variables. This approach is similar to CSP (Communicating Sequential Processes) [11].

In the basic data flow graph [12] each vertex represents an operation which consumes data before its execution and produces data after its execution, thus introducing a relationship between the order in which the operations must be executed and the order in which these operations consume and produce their data. The data produced by an operation and consumed by another one corresponds to a data transfer. Note that the notion of variable does not exist in this model; it is replaced by the notion of data transfer or “data flow”. This approach is also called “unique assignment” avoiding the problem related to shared variables. Each edge represents a *data dependence* inducing an execution precedence relation between two operations. An operation which does not need to transfer data to another operation is not connected by an edge to this operation. Consequently, the set of edges defines a partial order relation on the execution of all the operations [13, 14], defining in turn the potential parallelism of the data flow graph. The *level of potential parallelism* of a data flow graph depends on the lack of data dependences relatively to all the possible data dependences in the graph. There are two kinds of potential parallelism, *data potential parallelism*, usually called “data parallelism” when the operations without data dependences are the same (i.e. the same operation is applied to different data), and *operation potential parallelism*, usually called “task parallelism” when operations are different. Furthermore, because edges represents data transfers, hyper-edges (n-uples of vertices) are needed rather than edges (pairs of vertices) when it is necessary to specify that a data produced by an operation is consumed by more than one operation, corresponding to a data diffusion. This category of graph is called “hyper-graph”. Finally, a data flow graph is “acyclic” meaning that any path in the graph, formed by a succession of vertices and edges, must not have the same extremities which would produce a cycle. Cycles must be avoided because they introduce indeterministic behavior in the graph execution.

3.1.2 Factorized conditioned data dependence graph

The algorithm model [15, 16, 17] used in the AAA methodology allows to specify potential parallelism, to ensure coherence between the execution order of the operations and the way they consume and produce data, to avoid shared variables which are the source of numerous errors. This model is an extension of the basic data flow model in three directions. First we need to repeat infinitely and finitely a data flow graph pattern in order to take into account respectively the reactive aspect of the real-time systems and potential data parallelism. Second we need to specify “states” when data dependences are necessary between infinite repetitions. Third we must be able to condition the execution of several alternative data flow graphs according to the value transferred by a *control dependence*. Moreover, this model follows the synchronous language semantics [1], that is, physical time is not taken into account. Regarding one reaction of the system, that is, one data flow graph pattern of the infinite repetition, this means that each operation produces its output events instantaneously with the consumption of its inputs events which must be present altogether. Consequently it means that this data flow graph pattern is instantaneously executed. The successive executions of the data flow graph pattern introduces a notion of “logical instant”, using an additional precedence dependence (without data) between each repetition of the data flow graph pattern which ensures that a reaction will complete before another one begins. Each input or output of an operation carries an infinite sequence of events taking values which is called a “signal”. The union of all the signals define a “logical time”, such that physical time elapsing between events is not considered. Finally in order to limit the complexity of the graph, our model is *factorized*, that is, only the repeated data flow graph pattern is represented instead of all

its repetitions (infinite or finite), leading to a *factorized conditioned data dependence graph*.

When an application is running, the reactive system (controller of equation 1) is infinitely repeated interacting with a physical environment through, let us say to simplify, one sensor and one actuator. In order to specify the maximum of potential parallelism, it is possible to “unroll” this infinite temporal repetition (iteration) in an infinite spatial repetition, assuming that it exists an infinity of sensors and actuators. This allows to model the algorithm corresponding to the controller as an infinitely repeated data flow graph pattern. However, in order to simplify its large specification, it is only necessary to describe one instance of this data flow graph repetition, and then it is said *infinitely factorized*. When an instance of the factorized data flow graph needs a data produced by an operation belonging to a previous instance corresponding to an inter-repetition data dependence, this induces a cycle which must be mandatory avoided (acyclic graph) by introducing a specific vertex called *delay*. It is equivalent to the well known z^{-n} used in control and signal processing theory. The set of all the delays memorizes the algorithm state. Actually a delay is equivalent to two vertices containing a memory: one vertex without predecessor and one vertex without successor. The value contained in the memory of the latter vertex is copied in the other memory at the end of each reaction (data flow graph pattern execution). Moreover, another advantage of the data flow approach, is that the algorithm state is clearly localized in the delay vertices, whereas in the control flow approach it is spread out in all the variables. This issue is especially important when dealing with control and signal processing algorithms where state must be carefully mastered. It is also possible to repeat spatially an operation, or a subgraph of operations, N times (finitely), but represented as a single graph with a label indicating the number of repetitions, and then is said *N times factorized*. When each operation, or subgraph of operations, concerns different data, this spatial repetition provides data potential parallelism. This is the data flow equivalent of unconditional back branching in control flow graphs (“for $i=1$ to n do ...”). If data dependences between the consecutive repetitions are necessary (inter-repetition data dependences), this would cause cycles when factorizing, therefore a specific vertex called *iterate* must also be introduced, equivalent to the delay necessary for infinite repetition seen previously. In this case there is no potential parallelism because each inter-repetition data dependence induces an execution precedence. Note that factorized specification does not change anything about the semantics of the specification, it is only a way of simply represent complex graphs but potentially with parallelism. Later on during the implementation process, it will sometimes be necessary to transform a spatial repetition in a temporal repetition, or vice versa depending on the type of optimization the designer aims at. Finally, a vertex may be conditioned by the value transfered on its control dependence if it owns one. This conditioned vertex is specified as a set of alternative data flow graphs and the value transfered on the control dependence indicates the one to execute during the reaction. This is the data flow equivalent of conditional branching in control flow graph (“if ... then ... else ...” or more generally “case ... of ...”).

Therefore, the algorithm which corresponds to the decomposition of the controller (equation 1) in a set of data dependent operations, is modeled by a factorized conditioned data flow graph where each atomic (impossible to distribute on several resources) vertex is, either an operation performing computations (calculations) without side effect (the output only depends of the input, no internal state, no internal sensor or actuator), or a *factorizer*. There are four types of factorizer. For each instance of the spatial repetition, the *Fork* (F) provides separately each element of the vector it has in input. The *Diffuse* (D) operates like a fork but all the elements of the output vector are identical since there is a unique data in input. The *Join* (J) takes the result of each instance of the spatial repetition and provides as output the vector composed of the separate elements. Finally the *Iterate* (I) provides inter-repetition data dependences (temporal repetition equivalent to a finite iteration).

The figure 1 presents the algorithm graph performing an *iterative matrix-vector product* corresponding to the infinite repetition of a matrix-vector product MV . The input vector e_t which has three elements, is produced by a sensor, and the input 3×3 matrix is produced via a delay by the result of the matrix-vector product $z s_{t-1}$ performed during the previous $t - 1$ infinite repetition. The result of each matrix-vector

product is also sent to an actuator (data diffusion). The figure 2 presents the subgraph performing one of the three matrix-vector products. It corresponds to three repetitions of the scalar product V . The figure 3 presents the subgraph performing one of the three scalar products. It corresponds to three repetitions of the operations multiply-accumulate. This subgraph has three inter-repetition data dependences in order to perform an accumulation from the result of the sum performed during the previous repetition, locally preventing from potential parallelism specification.

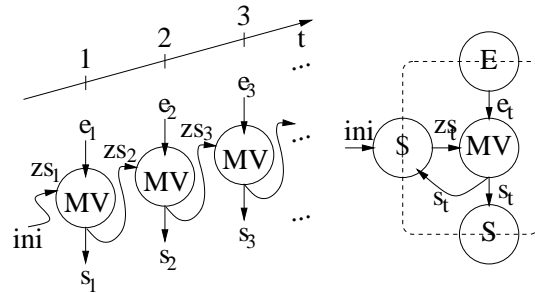


Figure 1: Algorithm graph (infinite repetition of the matrix-vector product)

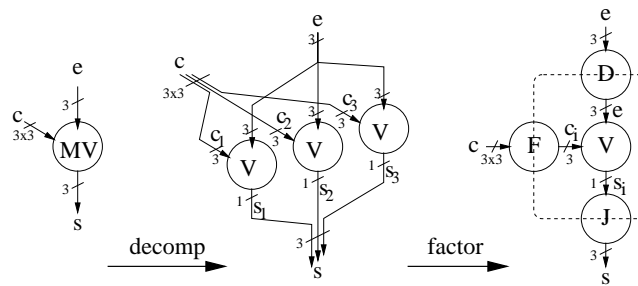


Figure 2: Subgraph MV (3 times repetition of V)

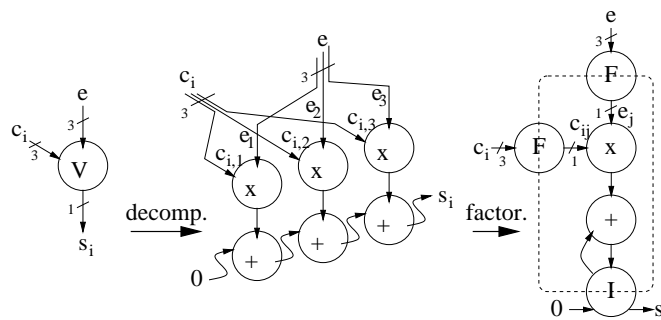


Figure 3: Subgraph V (3 times repetition of multiply-accumulate)

There are two ways for obtaining such an algorithm specification. On the one hand the user may directly input the factorized conditioned data flow graph through the graphical interface of the system level CAD software SynDEx as explained in section 4. On the other hand he may import this graph from one of the application specification languages interfaced with SynDEx, like presently one of the synchronous languages

Esterel and Signal, SyncCharts a state diagram language close to Statecharts, Scicos a free software control theory oriented language close to Simulink, CamlFlow and Avs two image processing languages, and AIL close to Titus.

3.2 Architecture model

3.2.1 Multicomponent

The most typically used models for the specification of parallel or distributed computer architectures, are PRAM (Parallel Random Access Machine) and DRAM (Distributed Random Access Machine) [18]. The first model corresponds to a set of processors communicating data through a shared memory, whereas the second model corresponds to a set of processors with its own data memory (distributed memory), communicating through message passing. Although these models would be sufficient for describing the distribution (allocation) and the scheduling of the algorithm operations in the case of homogeneous architecture, they are not precise enough for dealing with heterogeneous architectures and with the distribution and the scheduling of the communications which are, as mentioned before, crucial for real-time performances. Furthermore, we also need to take into account specific integrated circuits considered as non programmable components possibly communicating with other components whatever their kinds are. The main difference between a programmable component and a non programmable component, lies in the fact that only a unique operation may be distributed (allocated) on a non programmable component, whereas on a programmable component a set of operations, which must be scheduled, may be distributed.

Thus, our heterogeneous multicomponent model [19] is an oriented graph, where each vertex is a sequential machine (automaton with output) and each edge is a connection between the output of a sequential machine and the input of another sequential machine, thus forming a network of automata [20]. There are five types of vertices: the *operator* for sequencing computation operations, the *communicator* for sequencing communication (DMA channel), the *memory* for memorizing data or program, and finally the *bus/mux/demux* (BMD) with or without *arbiter* for selecting from and diffusing data toward a memory. When there is an arbiter in a bus/mux/demux/arbiter, this one is also a sequential machine deciding which resource will access to a memory, which is, in this case, a shared resource. The bus/mux/demux and the memory are considered as degenerated automata. There are two types of memories: RAM memory with random access for storing data or program, and SAM with sequential access for storing data, maintaining their order, when they must be communicated from an operator or a communicator to another operator or a communicator. The different types of vertices may not be connected in whatever manner, a set of connection rules must be verified [19]. For example two operators must not be directly connected and identically for two communicators. In order to communicate data, an operator must be at least connected to a RAM or a SAM, connected in turn to another operator. When computations and communications must be decoupled, communicators must be inserted, between operator and memory, whatever its type is. Heterogeneous architecture does not only mean that vertices have different characteristics, for example different execution durations for a given operation executed on an operator or a data transferred through a communicator, but also for example, that some operations may be executed only by some specific operators, or some data must be transferred only by some specific communicators. This allows, among other possibilities related to the architecture characterization described later on in section 3.2.2, to take into account specific integrated circuits, which are only able to execute a unique operation.

A basic processor may be specified as a graph containing one operator, one data RAM, and one program RAM, all interconnected. If this processor has been designed for parallel architecture, it may also contain one or several communicators with the corresponding data RAM or SAM for communications. A direct (without routing) communication resource between two processors, may be specified as a linear graph composed of the vertices n-uple (bus/mux/demux/arbiter, RAM or SAM, bus/mux/demux/arbiter). Typically, the RAM vertex is used to model a communication by shared memory, whereas the SAM vertex is used to

model a communication by message passing through a FIFO. When the computations must be decoupled from communications, some communicators must be added leading to the n-uple (bus/mux/demux/arbiter, communicator, RAM or SAM, communicator, bus/mux/demux/arbiter). A *route* is a path in the architecture graph connecting two operators. It is composed of a list of pairs (edge, vertex) plus an edge. A communicator allows data to cross through a processor without requiring its operator (“store and forward”). Several parallel routes may be specified in order to transfer data in parallel, but these routes may be of different length (number of elements in the route).

This model, well adapted to the optimizations presented later on in the paper, allows to specify architectures with more or less details. But it is important to be aware that the more detailed the architecture will be, the more the solution of the optimization problem will take time to be found.

The figure 4 presents the detailed model of the DSP TMS320C40 from Texas Instrument, obtained from the Data-Book [21]. Here all the connections between the vertices are bi-directional, then for the sake of readability we have represented each pair of arrows by a simple segment. The CPU, including its sequencer, its memory controller, and its arithmetic and logic unit, are represented by an operator. Since it is able to simultaneously access two internal ($R0$ and $R1$) and/or external (R_{loc} and R_{glob}) memories modeled by RAM vertices, it is connected to two bus/mux/demux ($b7$ and $b8$) which select the appropriated memory. Because these memories may also be accessed by one of the six DMA channels modeled by communicators ($C1$ to $C6$), each communicator is connected to the memories by a bus/mux/demux/arbiter ($b9$) which arbitrates among the communicators. Each point-to-point communication port is modeled by a SAM which may be accessed either by a DMA or by the CPU, here the operator. The operator and the communicators may access the external RAM R_{loc} using the arbiter of the bus/mux/demux/arbiter ($b11$). The operator and the communicators may access the external RAM R_{glob} using the arbiter of the bus/mux/demux/arbiter ($b12$). Each of the six bus/mux/demux ($b1$ to $b6$) selects either a SAM or the external RAM (R_{glob}), for each of the six communicators ($C1$ to $C6$). The bus/mux/demux ($b10$) selects one of the six SAM accessed by the CPU.

The figure 5 presents the model of a complex architecture composed of four TMS320C40 communicating, on the one hand by point-to-point links, and on another hand by a shared memory (R_{glob}). Then, although the same type of processor (operator) is used in this example, this is an heterogeneous architecture relative to the communications which are of different types. Each processor has its own local memory (R_{loc}).

The figure 6 presents a less detailed version of the previous architecture. It is obtained by encapsulating in a unique operator the graph given in figure 4, leading to a more simple description of the architecture.

3.2.2 Architecture characterization

The optimization process described in detail in section 3.4, is based on the multicomponent architecture characterization, meaning that to each operator and communicator, is associated the set of operations it is able to execute. Furthermore, to each operation is associated, its execution duration, the amount of memory, the power consumption, etc. . . , it requires. For example the CPU of a DSP is able to execute a multiply-accumulate operation in one clock cycle, and a FFT (Fast Fourier Transform) in several cycles. Similarly, the DMA of a DSP associated to a point-to-point link is able to transfer data in a specific time, and an array of the same data in a time proportional to the number of data to transfer, plus a set-up time. The arbiter of a bus/mux/demux/arbiter has a crucial role, it is characterized by a table of priorities and a table of bandwidths which has as much elements as connected edges. The values of these elements are used to determine during the optimizations which of the operators and/or communicators will access the memory and with which bandwidth.

Each integrated circuit of the architecture is characterized separately by associating to each vertex and edge the execution duration relative to the chosen technology.

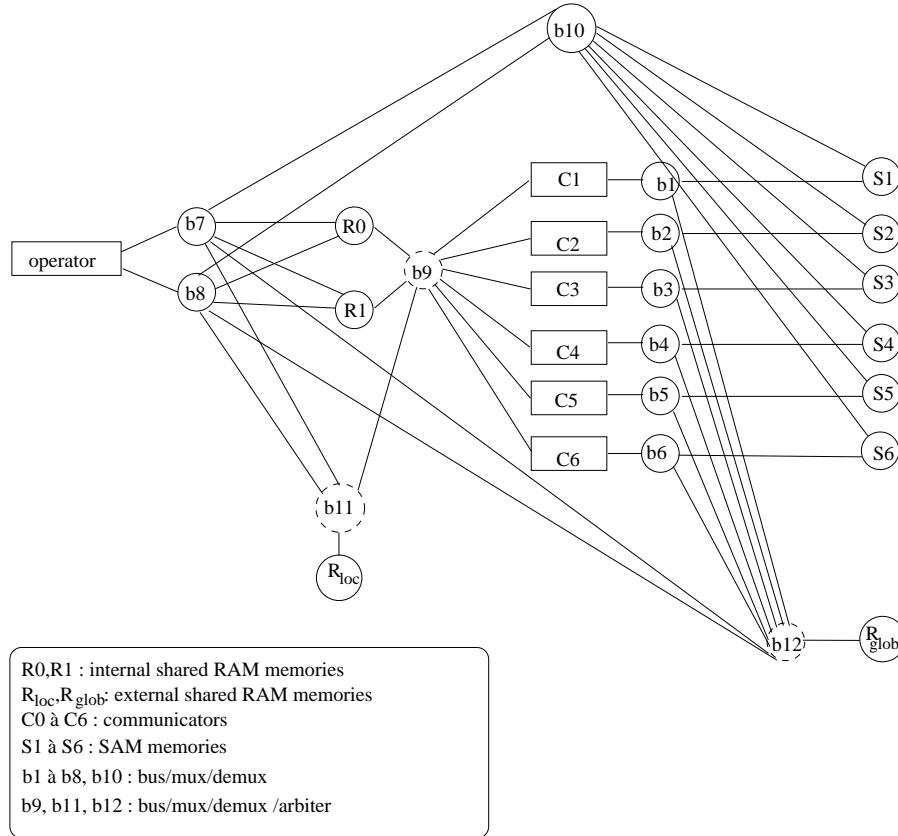


Figure 4: TMS320C40 architecture graph

3.3 Implementation model

In this section we present how to describe all the possible implementations of a given algorithm onto a given multicomponent, in intention rather than in extension, using graph transformations. Performing an implementation is mainly a matter of reducing the potential parallelism of the algorithm according to the physical parallelism of the architecture. More precisely, it consists in *distributing* and *scheduling* the operations of the algorithm onto the architecture which has been already characterized. We use the term distribution instead of “placement” or “allocation”, which are commonly employed, in order to refer to distributed systems.

3.3.1 Distribution and scheduling

The distribution and the scheduling are formally detailed in [16]. The distribution consists in performing a partition of the initial algorithm graph, in as much elements of partition as there are of operators in the architecture graph. Then, each element of partition, i.e. each corresponding subgraph of the algorithm graph, is distributed onto an operator of the architecture graph. This amounts to label each subgraph with the name of the operator it has been distributed onto. Remember that only a unique operation may be distributed onto an operator representing a specific integrated circuit in the considered multicomponent. Then, a partition of the data dependences of the algorithm graph between operations belonging to two different elements of operations partition must be performed, in as much elements of partition as there are of routes in the architecture graph. Each element of partition, i.e. each set of corresponding data dependences of the algorithm graph, is distributed onto a route of the architecture graph. This amounts to label each set

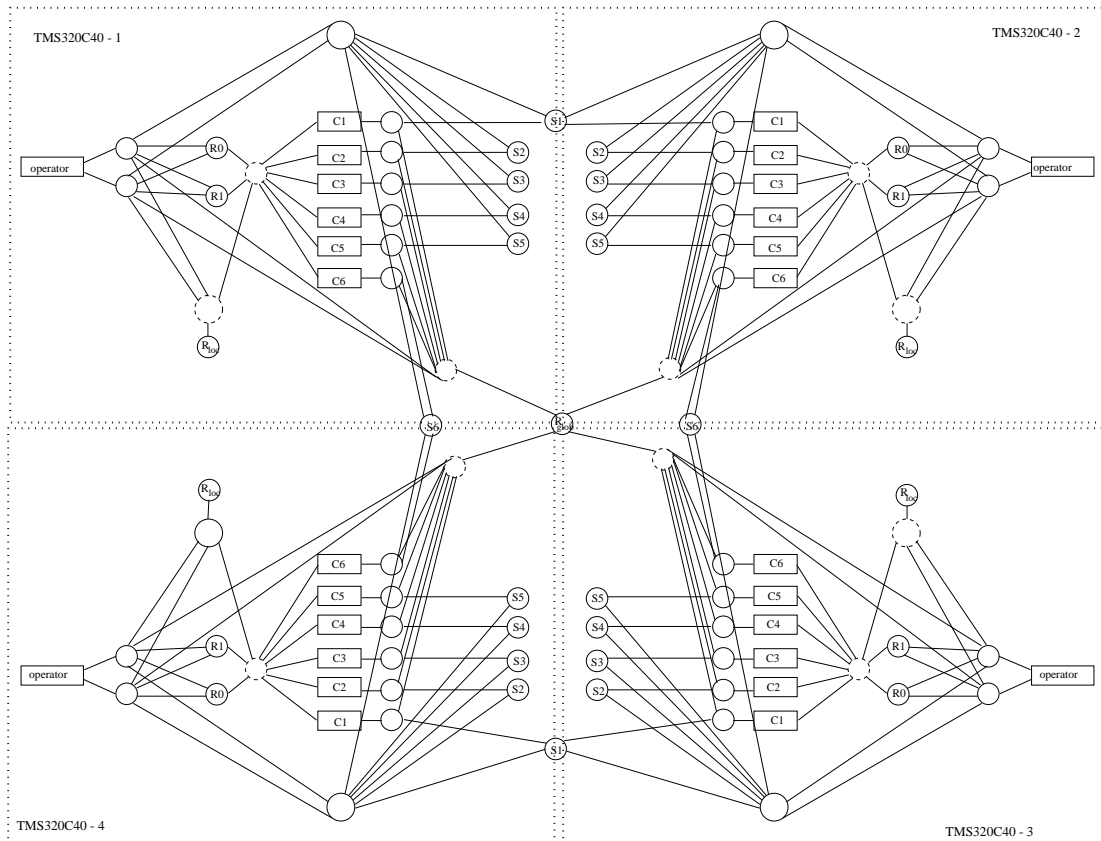


Figure 5: TMS320C40 quadri-processor architecture graph

of data dependences by the name of the route it has been distributed onto. Finally, for each data dependence which connects two elements of operations partition (inter-partition edge), *communication operations* must be created and inserted. There are as much communication operations as there are of communicators in the route the data dependence has been distributed onto. If the route does not contain any communicator, like in a direct communication resource using a shared RAM, it is not necessary to create a communication operation. Indeed, in this case the operator performs the data transfer. Although, the drawback is that no parallelism (decoupling) is possible between computations and communications, since the operator is also required to perform data communications. Actually, each communication operation is composed of two vertices. In the case of a SAM it corresponds to a *send* vertex and a *receive* vertex. The *send* is executed by the communicator which sends the data to the SAM, and the *receive* is executed by the communicator

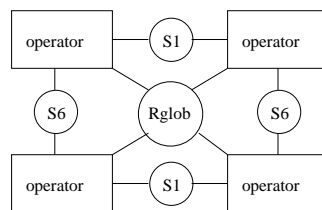


Figure 6: Simplified TMS320C40 quadri-processor architecture graph

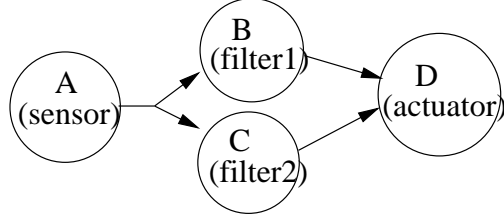


Figure 7: Basic example of an algorithm graph

which receives the data from the SAM. Similarly, in the case of a RAM it corresponds to a *write* vertex and a *read* vertex.

The scheduling consists in transforming the partial order of the corresponding subgraph of operations assigned to an operator, in a total order. This “linearization of the partial order” is necessary because the operator is a sequential machine which executes sequentially the operations. Similarly, it also consists in transforming the partial order of the corresponding subgraph of communications operations assigned to a communicator, in a total order. Actually, both amount to add edges, called *precedence dependences*, to the initial algorithm graph.

Finally, memory allocation is also necessary in order to take into account, on the one hand the program memories used to store each operation, and on the other hand the buffers necessary to transfer data from an operation to another operation distributed onto the same operator. Then *alloc* vertices must also be added for each operation and for each edge in order to be distributed onto the program and the data RAM connected to the operator.

The distribution, the scheduling and the memory allocation lead to the *implementation graph*.

Figure 9 shows a simple implementation example of the algorithm graph presented in figure 7 onto the architecture presented in figure 8-b. Such an implementation graph is automatically 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 (*send_{BD}*, *receive_{BD}* and *send_{CD}*, *receive_{CD}*) must be inserted and associated to *Com1* and *Com2* in order to realize data transfers on the shared SAM *S* (9-a). Allocation vertices (*alloc_{AB}*, *alloc_{BD}*, *alloc_{AC}* ...) have also been added in order to model all required memory allocations (9-b). Since operations *B* and *C*, which are not dependent, are distributed onto the same operator, an order of execution must be chosen between them. Notice that in this example, in order to simplify the figures, we do not take advantage of the potential parallelism between operation *B* and *C*. They should have been executed in parallel if distributed onto different operators. Thus, we add a precedence dependence edge between *B* and *C*, and a precedence dependence edge between *send_{BD}* and *send_{CD}* because they are scheduled on the same communicator *Com1*, and symmetrically a precedence dependence edge between *receive_{BD}* and *receive_{CD}* executed on *Com2*. This corresponds to the bold arrows of figure 9-c.

Hence to summarize, the set of all the possible implementations of a given algorithm onto a given architecture may be mathematically formalized in intention, as the composition of three binary relations: namely the *routing*, the *distribution*, and the *scheduling*. Each relation is a mapping between two pairs of graphs (algorithm graph, architecture graph), from the set $G_{al} \times G_{ar}$ on the set $G_{al} \times G_{ar}$. It also may be seen as an external compositional law, where an architecture graph operates on an algorithm graph in order to give as a result a new algorithm graph, which is the initial algorithm graph distributed and scheduled according to the architecture graph. In this case this is a mapping from the set $G_{al} \times G_{ar}$ on the set G_{al} .

Given an algorithm and an architecture graphs, there is a finite number of possible distributions and schedulings [22]. Indeed, it is possible to perform different partitions with the same number of elements

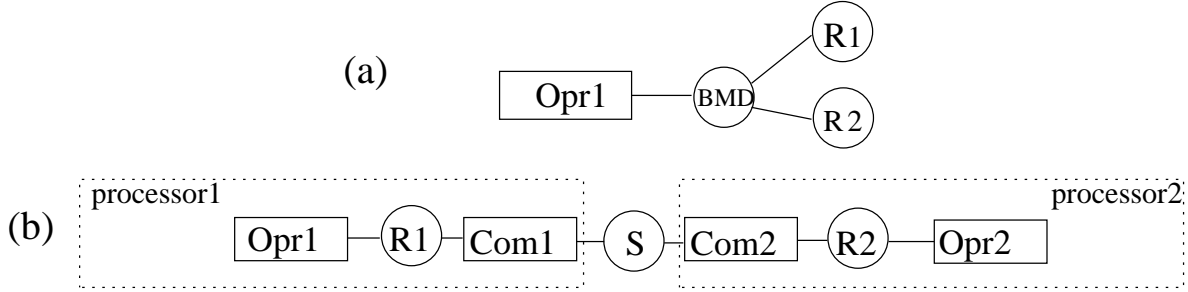


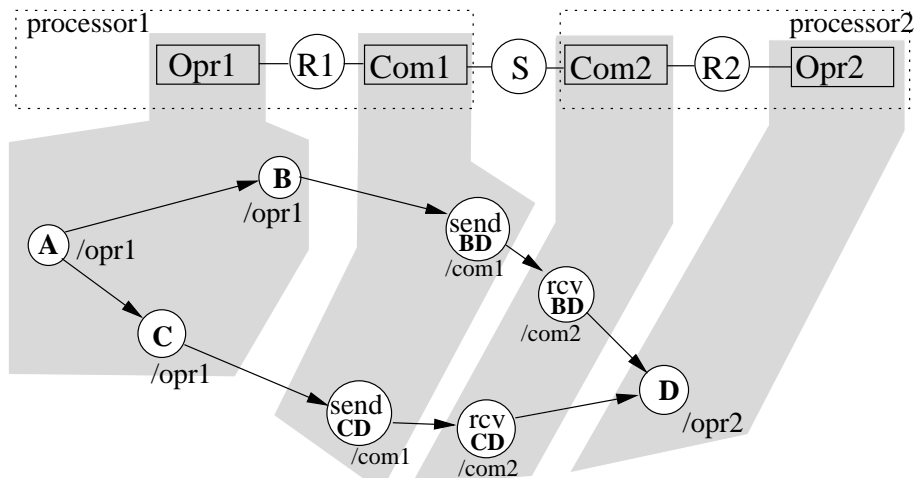
Figure 8: Basic examples of architecture graphs

(namely the number of operators), and for each subgraph assigned to an operator, it is possible to perform different linearizations, and identically for the communication operations relative to the routes and the communicators. But this leads to a very high number of possible combinations. However, it is necessary to eliminate all the schedulings which do not preserve the logical properties, remember in terms of ordering, shown with the synchronous languages as mentioned before. This amounts to preserve the transitive closure [22] of the partial order associated to the initial algorithm graph when the relation “scheduling” is applied. Moreover, the partial order of the resulting algorithm graph, which corresponds to a reduction of the potential parallelism, must be compatible with the partial order of the initial algorithm graph. Note that there is no such problem when the relation “distribution” is applied.

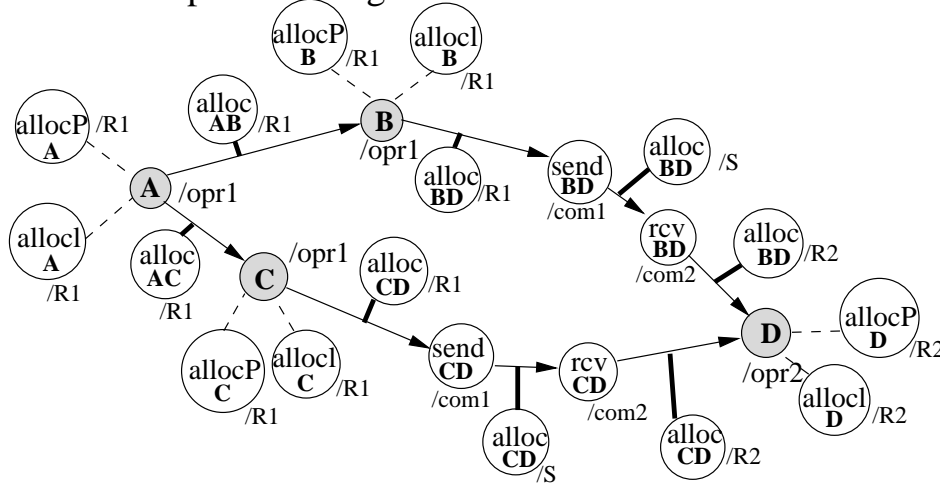
Our implementation model, called *Macro-RTL*, may be seen as an extension of the typical implementation model called RTL (Register Transfer Level) [23]. An operation of the algorithm graph corresponds to a *macro-instruction* (a sequence of instruction instead of one instruction) or a combinatorial circuit. A data dependence corresponds to a *macro-register* (several memory cells). The consumption and the production of data by an operation corresponds to a data transfer between registers through a combinatorial circuit. This model encapsulates details relative to the instructions set, the micro-programs, the pipe-line, and the cache. In that way it filters these characteristics too difficult to take into account during the optimizations. This model has a reduced complexity well adapted to the rapid optimization algorithms we aim at, however giving accurate optimization results.

3.3.2 Impact of the granularity and potential parallelism

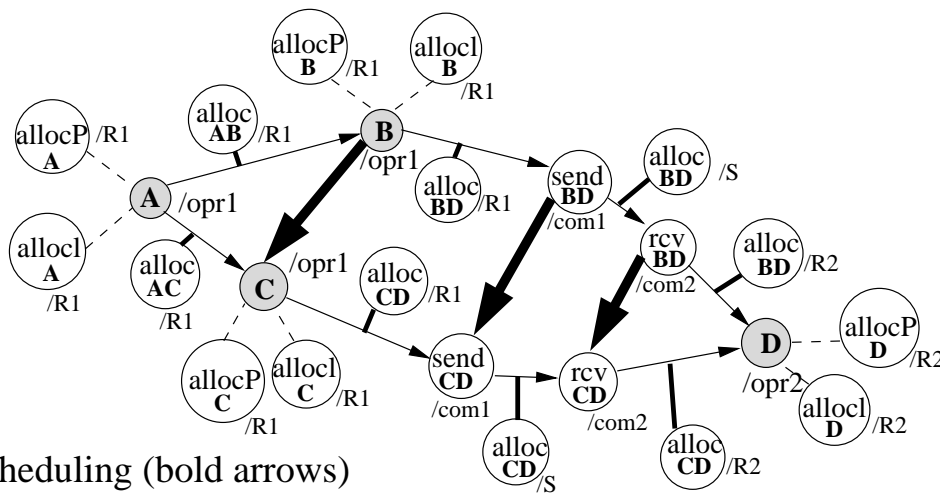
A given algorithm offers a granularity relative to the number of operations (vertices) and data dependences (edges) it is composed of, and a level of potential parallelism relative to the lack of data dependences relative to all its possible data dependences (pairs of vertices). It is obvious that these two parameters are inter-dependent. This issue has consequences in terms of possible implementations. If the number of operations and data dependences is not sufficient enough relatively to the number of hardware resources (computation and communication sequencers), it is not possible to balance correctly the load on each resource. Similarly, if the level of potential parallelism is low, there is not enough degree of freedom when reducing the potential parallelism in order to match the physical parallelism of the architecture. On the contrary, if the number of operations and data dependences is too high, then each operation and data dependence encapsulates only few details, because in this case it has generally a low complexity, leading to a less efficient filtering of the characteristics difficult to take into account. On the other hand, the high level of potential parallelism leads to a huge number of possibilities when reducing the potential parallelism in order to match the physical parallelism. The impact of the granularity and potential parallelism is discussed in detail in [24] for an example of image processing. It is shown that an a priori choice of the granularity and potential parallelism may be modified if the real-time constraints are not satisfied. In this case, some operations must be decomposed in



a) Distribution: partitioning & communication vertices



b) Distribution: insertion of allocation vertices



c) Scheduling (bold arrows)

Figure 9: Example of implementation graph

several operations, possibly with potential parallelism.

3.4 Optimized implementation: adequation

3.4.1 Principles

An adequation consists in searching, among all the possible implementations of the algorithm onto the architecture, for the one which corresponds to an optimized implementation relatively to the real-time and embedding constraints. The optimization problems considered in this paper, that is, the minimization of the latency and/or the cadence when the architecture is fixed, and the minimization of the architecture resources, are NP-hard problems [25].

Because it is impossible to obtain an exact solution in a reasonable time relatively to the human life, we use heuristics which are rapid enough and give a solution as close as possible to the exact (optimal) solution. In the case of complex applications involving control, signal and image processing, “rapid prototyping” is necessary. Such heuristics are well suited in order to rapidly test several variants of an implementation relatively to the cost and the availability of the hardware components, and also relatively to the addition of new functionalities. It is the reason why we first use “deterministic greedy” heuristics, i.e. no random choice and no back-tracking, and especially its “list scheduling” version because they rapidly give a result with a good precision [26]. A solution obtained with this kind of heuristics may be improved by back-tracking [27], such that the choices are modified locally or globally when elaborating a partial solution, according to the so-called “neighborhood” techniques. However, this kind of heuristics is dramatically slower. Finally, in order to improve again the quality of the solution, it may be in turn used as an initial solution for stochastic heuristics, i.e. where random choices allow to go from one solution to another one. Actually these heuristics are very slow, but are more precise mainly because they avoid local minima that the deterministic ones do not avoid. Because we are dealing with heuristics, it is also interesting to exploit the user’s skills, who is able to restrict the search space by imposing distribution or scheduling constraints in order to avoid “bad tracks” leading to local minima.

The heuristics optimizing the latency are based on the “critical paths” of the algorithm graph labeled by the durations, not only of the operations relatively to the possible operators but also of the data transfers relatively to the communication resources, whereas the heuristics optimizing the cadence are based on its “critical loops”, i.e. the cycles in the algorithm graph containing delays, leading to pipe-lining and re-timing by just moving these delays. To optimize simultaneously and independently both latency and cadence is a very difficult problem. It is the reason why usually, one is fixed while the other is intended to be optimized.

3.4.2 Example of adequation heuristics

In this section we present an efficient deterministic greedy list heuristics optimizing one latency which takes accurately into account inter-processor communications which are often neglected [28]. Its efficiency has been compared in [27] relatively to heuristics of the same family. We assume that there is only one input at the beginning and one output at the end of the algorithm graph (it is easy to transform the graph if this is not the case), and that the cadence is equal to the latency. For the sake of simplicity, we do not take into consideration the conditioning and the memory capacity. Moreover, subgraph repetitions are assumed to be entirely defactorized. The reader interested in these issues may consult [16], [19]. We have works in progress to take into account several constraints e.g. several latencies and cadences [7].

Here are the principles of the heuristics which tends to construct a global optimum from several local optima while the distribution and the scheduling are performed simultaneously. It iterates on the set O_s of *schedulable* operations in the algorithm graph. An operation not yet scheduled becomes schedulable when all its predecessors, excluding the delays, have already been scheduled. Initially O_s is composed of all the operations which are either input or which have only delays as predecessors. During an iteration i of the

heuristics, one of the schedulable operations o and an operator p onto which this operation will be scheduled, are chosen using a cost function detailed in the next section. p belongs to the set of operators P of the architecture graph. Consequently, some successors of o become in turn schedulable. The iteration is repeated until $O_s = \emptyset$ and then, each operation has been distributed and scheduled onto an operator after a complete exploration of the initial partial order associated to the algorithm graph, thus leading to a new compatible partial order. When an operation is scheduled onto an operator after an operation which is not its predecessor, this amounts to modify the initial partial order by adding a new edge which is only an execution precedence (no data is transferred since there were no edge between both operations). If an operation o is scheduled onto an operator p and if its predecessor o' has been scheduled onto a different operator p' , it is necessary to choose a route r , joining p and p' (a path in the architecture graph), and to create and insert between o and o' as much communication operations as there are of communication resources composing r , and to schedule each communication operation onto the corresponding communicator of the communication resource m . When a communication operation is scheduled onto a communication resource, this also amounts to modify the initial partial order by adding a new edge which is only an execution precedence (no data is transferred between both communication operations). It is worth noting that on each operator and on each communicator of a communication resource we obtain a total order which is compatible by construction (it is ensured that an operation is scheduled only after its predecessor) with the initial partial order. However, we obtain globally a partial order, the operations on different operators as well as the communications operations on different communicators may be executed in parallel, which is also compatible with the initial partial order. This principle guarantees that the complete execution of the operations and of the communications will never cause a deadlock.

Cost function

The goal of the heuristics consists in minimizing the latency, and here the latency is equal to the critical path of the algorithm graph labeled by the durations of the operations and of the data transfers because we only have one input at the beginning and one output at the end of the graph. The cost function is defined in terms of the start and end dates of each operation. We denote by $\Delta(o, p)$ the execution duration of the operation o belonging to the algorithm graph Gal , and by p the operator belonging to the architecture graph Gar which executes o . $\Gamma(o)$ is the set of the successors of o and $\Gamma^-(o)$ the set of its predecessors. For each schedulable operation and for each operator where it is possible to distribute and schedule this operation, R the partial critical path, $S(o)$ and $E(o)$ (resp. $S^-(o)$ et $E^-(o)$) the earliest start and end dates of execution from the beginning of the graph (resp. from the end of the graph), and $F(o)$ the schedule flexibility are processed:

$$\begin{aligned}
S(o) &= \max_{x \in \Gamma^-(o)} E(x) \quad (\text{or } 0 \text{ if } \Gamma^-(o) = \emptyset) \\
E(o) &= S(o) + \Delta(o, p) \\
E^-(o) &= \max_{x \in \Gamma(o)} S^-(x) \quad (\text{or } 0 \text{ if } \Gamma(o) = \emptyset) \\
S^-(o) &= E^-(o) + \Delta(o) \\
R &= \max_{o \in Gal} E(o) = \max_{o \in Gal} S^-(o) \\
F(o) &= R - E(o) - E^-(o)
\end{aligned}$$

Note the symmetry in the formulas, the dates are processed relatively to opposite directions and origins: $\min_{o \in Gal} S(o) = 0 = \min_{o \in Gal} E^-(o)$. Note also that, often in the literature [29] $S = ASAP$ and $R - S^- = ALAP$. The schedule flexibility $F(o)$ represents the freedom degree of an operation, i.e. a time interval inside which o may be executed without increasing the critical path.

When the heuristics considers an operation o , all its predecessors have already been distributed and scheduled, but no successor has already been scheduled. Then, when E^- and S^- are processed Δ must be

defined independently of all operators. The execution duration of an operation o which is not yet scheduled is defined as the arithmetic mean of all its possible execution durations on the set $K(o)$ of the operators able to execute o :

$$K(o) = \{p \in Gar | \Delta(o, p) \text{ is defined}\}$$

$$\Delta(o) = \frac{1}{\text{Card}(K(o))} \sum_{p \in K(o)} \Delta(o, p)$$

When an operation o is scheduled onto an operator p , it is scheduled after all the operations previously scheduled onto p , its execution duration $\Delta(o)$ becomes $\Delta(o, p)$, and for each predecessor o' of o scheduled onto an operator $p' \neq p$, communication operations must be created and inserted between o and o' . These communication operations must also be distributed and scheduled. Consequently, $S(o)$ as well as the critical path R will have greater or equal values (when no communication is necessary) but never smaller, becoming $S'(o)$ and R' in order to indicate that communications may have been taken into account.

The cost function $\sigma(o, p)$ called the *schedule pressure* is the difference between the *schedule penalty* $P(o) = R' - R$ (critical path increase) and the schedule flexibility $F(o)$ before the critical path increases:

$$\sigma(o, p) = P(o) - F(o) = S'(o) + \Delta(o, p) + E^-(o) - R$$

$\sigma(o, p)$ is an improved version of the commonly used cost function $F(o)$ which is extended by $P(o)$. Indeed, when $S'(o)$ (taking into account possible communications) increases, o becoming more and more critical $F(o)$ decreases until being null and then remains null. $P(o)$ which until now was null begins to increase. Finally $\sigma(o, p)$ which is the composition of $F(o)$ and $P(o)$, is a function which increases continuously. Note that $F(o)$ depends on R' (taking into account possible communications) which may be different at each iteration of the heuristics, whereas $\sigma(o, p)$ does not depend on R which remains the same whatever the iteration is. Then, it is not necessary to process the value of R at each iteration.

Choice of the best operator

The *best operator* $p_m(o)$ for a schedulable operation o is either the operator on which, the user has constrained the operation to be executed (distribution constraints), or the operator which gives the smallest schedule pressure, i.e. the greater schedule flexibility and the smaller schedule penalty (increase of critical path). If several operators lead to the same results one of them is randomly chosen:

$$\exists p_m(o) \quad | \quad \sigma(o, p_m(o)) = \min_{p \in Gar} \sigma(o, p)$$

On the other hand, the schedulable operation which is the most urgent to schedule onto its best operator, is the operation with the greatest schedule pressure, unless its start date is greater than the date of another schedulable operation which can be executed before. It is the reason why practically it is necessary to restrict O_s to:

$$O'_s = \{o' \in O_s \quad | \quad S_m(o') < \min_{o \in O_s} E_m(o)\}$$

where $S_m(o)$ and $E_m(o)$ are the respective values of $S(o)$ and $E(o)$ for o scheduled onto its best operator $p_m(o)$.

Finally, the operation chosen at each iteration is o_m such as:

$$\exists o_m \quad | \quad \sigma(o_m, p_m(o_m)) = \max_{o \in O'_s} \sigma(o, p_m(o))$$

If several operations lead to the same results one of them is randomly chosen. The chosen operator is $p_m(o_m)$ the best operator on which o_m is scheduled.

Creation, distribution and scheduling of communications

When an operation o is scheduled onto an operator p , each data dependence between this operation and one of its predecessors $o' \in \Gamma^-(o)$ already scheduled onto an operator $p' \neq p$, is an *inter-operator data dependence*. For each of these dependences, it is necessary to choose a route which will support the transfer of the data produced by o' in the local memory of p' to the local memory of p in order to be consumed by o . The transfer is achieved through the different communication resources composing the route.

The number of possible routes (path in the architecture graph) may be large for complex architectures, and consequently the choice of the best route may take a very long time because it is necessary to compare, for a data dependence to transfer, all the possible cases of distribution and scheduling in all the communicators of the route. In order to reduce this time the heuristics performs an incremental choice using *routing tables*. This method is described in details in [28]. Briefly, for each operator the shortest routes (minimum number of communication resources) between this operator and all the other operators, is determined. These shortest routes and the first communication resource of this route are memorized in its routing table. If there are several possible routes between two operators with the same minimum number of communication resources, these routes, called *parallel routes*, are memorized. Then, each time the heuristics has to evaluate the cost of an inter-operator data dependence, for each operator of the route (p', p) it chooses in the routing table among the possible first communication resource m of the shortest routes from this intermediate operator and p , the one which first completes the communication operation c . This communication operation may be either another communication operation which has been previously scheduled onto m because it has to transfer the same data (this is the case of a data diffusion using the same first part of the route), or c is a new communication operation that must be created and inserted in the algorithm graph between the previous communication operation in the route (or o' at the beginning of the route) and o . c must be scheduled onto m by adding a precedence dependence between the last communication operation scheduled onto m before c , and c . The heuristics proceeds like this until p is reached. This approach allows to take into account on the one hand parallel routes in order to balance the load of the communication resource, and on the other hand the possibilities to reuse already routed communications avoiding to needlessly duplicate communications.

3.4.3 Resources minimization

In the previous section we assumed that the architecture graph is given and then, all the resources like operators, communicators, etc, are also given, as well as the way they are interconnected. In this case we aim at exploiting the architecture resources as good as possible. However, in some cases it is possible to decrease the number of resources while satisfying the real-time constraints. Therefore, an iterative process may be set up, where the user tries to decrease the number of resources and verifies if the real-time constraints are still met.

3.5 Executives generation

Here, we only present the main principles of the executives generation for multicomponent architecture. The code executable in real-time, is the result of an ultimate graph transformation of the implementation graph obtained after the optimized distribution and scheduling described previously. The graph transformation and the obtained code are detailed in [19, 30].

As soon as a distribution and a scheduling have been chosen, that is to say determined “off-line”, it is possible to automatically generate dedicated real-time distributed executives. They are mainly static with a dynamic part only for taking into account conditionings which depends of test values only known when the application is executed in real-time. From the same informations, it is also possible to configure the fixed priorities of a set of tasks scheduled by a standard RTOS, like: Vrtx, Irmx, Virtuoso, Lynx, Osek, RT-Linux, etc... However, this approach although it should allow the use of COTS (Component Of The Shelf) which

may reduce costs, will obviously decrease the performances by increasing the overhead of the executives. Note that in both cases we use an “off-line” approach, well suited to real-time applications which need to be deterministic.

The dedicated executives are mainly based on the one hand on the sequencing, possibly conditioned, of the algorithm operations distributed on a particular processor, and on the other hand on an inter-component communications system without any deadlock by construction, ensuring a global synchronization between all the operations running on different processors. This is the reason why we chose the name SynDEX, acronym for Synchronized Distributed Executives, for the system level CAD software presented in section 4 which implements the AAA methodology. Deadlocks due to data dependence cycles are detected during the algorithm graph specification and during the graph transformations, taking into account the architecture graph, and leading to the implementation graph.

The algorithm graph specified by the user, possibly through a high level language (perhaps performing verifications), is transformed during the optimized distribution and scheduling, avoiding cycles since its partial order is reinforced without introducing any cycle. Similarly, the implementation graph is also transformed in order to produce the executives, by adding to the implementation graph new vertices and their corresponding edges. In order to satisfy the real-time characteristics of the algorithm, each executive includes an infinite repetition due to the reactive nature of the applications, and synchronizations which ensure that the data communications will be executed, without any deadlock, according to the scheduling chosen by the optimization heuristics. This preserves the logical properties shown with the high level specification languages when some are used. The synchronization operations guarantee execution precedence between computation operations and communication operations belonging to different sequences, sharing data in mutual exclusive access. Each synchronization operation uses a semaphore automatically generated. In [19] it is shown with Petri nets that these semaphores allow the executives to verify the partial order of the initial algorithm graph.

There are as many generated executives as there are of processors. Each executive file is a *macro-code* which is independent of the processor type. It is composed of a list of macros which will be translated by a macro-processor, for example the Gnu tool “m4”, using the appropriate definitions of macros, into a source program (C, assembler, etc...). Then, each of these source programs will be compiled and linked, and finally loaded and executed on the target processor in order to run in real-time. The definition macros which are dependent of the processor, are of two types. The first one is an extensible set of *application definition macros* describing the operations behavior, e.g. an addition or a filter. The second one is a fixed set of *system definition macros* describing the application support: loading and initialization of program memory, management of data memories, sequencing (conditional and unconditional branchings respectively for conditionings, and finite and infinite loops), inter-processor data communications (send, receive, write, read), synchronization inside a processor between a sequence of computations and one or several sequences of communications, synchronization between sequences of communications belonging to different processors, and finally chronometric recording for operations and data transfers characterization. This latter set of definition macros is called the *executive kernel* and one is needed by processor.

The process of the executives generation is perfectly systematic. It automates the work performed by hand by a system programmer, leading to a very low overhead even though they are automatically generated.

The executives generation is performed following four steps [19]: (1) transformation of the optimized implementation graph into an execution graph, (2) transformation of the execution graph into as many macro-code as there are of processors, (3) transformation of each macro-code into a source file, (4) compilation, download and execution of each source file.

3.5.1 From implementation graph to execution graph

This transformation consists in adding new types of vertex: *Loop*, *EndLoop* and *pre-full/suc-full*, *pre-empty/suc-empty* vertices. This is done following two steps:

1. since the considered applications are reactive (i.e. they are in constant interaction with the environment that they control) the sequence of operations distributed onto each operator must be infinitely repeated. For each sub-graph of the algorithm distributed onto an operator a *Loop* vertex is added and connected before the first operation of the sequence, and a *EndLoop* vertex is added and connected after the last operation (Cf. figure 12),
2. when two operations distributed onto two different operators are data dependent, a communication must be performed between these operators. The operator which executes the producing operation must cooperate with a communicator in order to send (resp. write) the data to a SAM (resp. a RAM), symmetrically another communicator must cooperate with the operator which executes the consuming operation in order to receive (resp. read) the data from the SAM (resp. the RAM). When considering one infinite repetition, for each pair operator-communicator, operator and communicator must be synchronized because both these sequencers share the data to send. This synchronization is necessary in order to carry out the inter-partition edge, represented by a bold arrow on figure 10. It is implemented on *processor1* by replacing this edge with 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 on the same partition as the producing (resp. consuming) operation of the initial inter-partition edge. *Pre-full* and *suc-full* vertices are operations able to read-modify-write a binary semaphore allocated into the memory shared by the two sequencer partitions. If *suc-full* (which precedes the send operation) is executed before the connected *pre-full* (which follows the operation B) then the *suc-full* waits for the end of the *pre-full* execution which signals that the buffer containing the value produced by the operation B is full. This mechanism ensures a correct execution order between the execution of the operation B and the operation *send BD* which sends the value produced by B to the operation D executed on another *processor2*. When considering two consecutive infinite repetitions, it is also necessary to avoid that a producing operation overwrites the data which has not yet been sent. For this purpose a pair of *suc-empty*, *pre-empty* vertices is inserted. *Pre-empty* is inserted after the consuming operation *send BD* while *suc-empty* is inserted before the producing operation B. *Pre-empty* signals that the sent of the data produced by B during the previous repetition was terminated.

Symmetrically, on *processor2* which receives and consumes with operation D the result produced by operation B executed on *processor1*, the synchronization represented by a bold arrow on figure 11 is implemented by replacing this edge with 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 11). Similarly, when considering two consecutive infinite repetitions, a *pre-empty* is inserted after the consuming operation D while *suc-empty* is inserted before the producing operation *rcv BD*.

Notice that when a SAM is used to transfer the data between the communicators, no other synchronizations is necessary since this type of memory ensures an hardware write-read synchronization. In the case of a RAM, a synchronization, similar to the one between operator and communicator, must be added.

Figure 12 depicts a complete example of the execution graph obtained after the transformation of the implementation graph given in figure 9. *Loop/EndLoop* vertices have been added on *Opr1,Com1,Com2* and *Opr2* operations. In order to simplify the graph, allocation vertices are not represented.

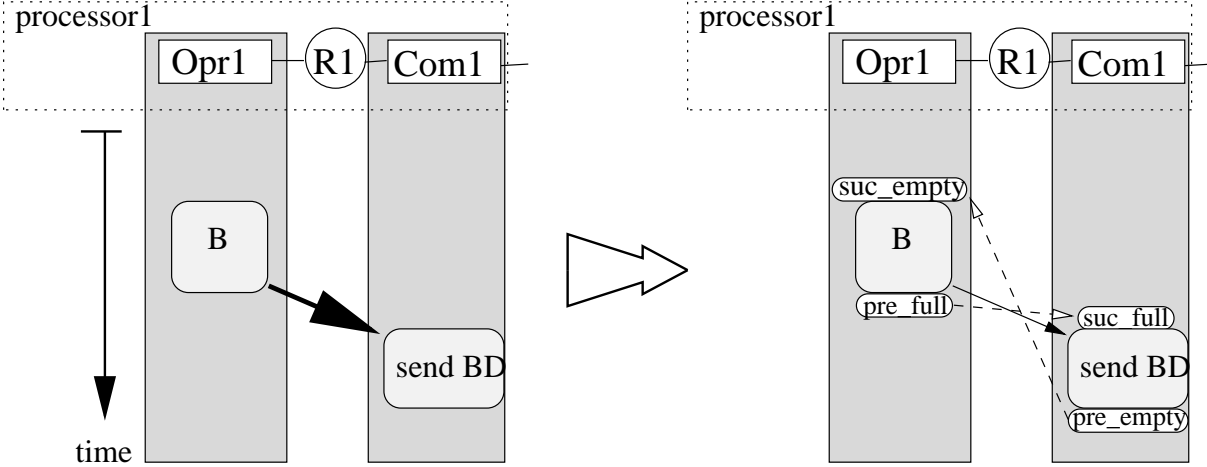


Figure 10: Principle of synchronization send

Synchronization 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, storing the data, 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. Moreover, they guarantee that no data is lost. Therefore, the implementation optimization, even if it may be biased by inaccurate architecture characteristics, is safe in the sense that it cannot induce, unlike human programmers, runtime synchronization errors (such as deadlocks, or lost data). Indeed, such synchronizations are usually hand-written inside the application code such that deadlocks may occur if the designer misses one of them or does not write them in the the correct order. Finally, since synchronization 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 trouble due to run-time overhead (as consensus waiting problem) induced by synchronization. The run-time overhead induced by the synchronizations is completely mastered and its cost can be taken precisely into account by the optimization heuristics. The proposed technique allows big savings thanks to a minimization of the coding process which actually is reduced to the one of the application operations. In addition, it leads to a minimum debugging time.

3.5.2 From execution graph to macro-code

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

The macro-code structure for an operator *opr* is sequentially composed of:

- macros allocating semaphores and buffers for each allocation vertex allocated to each RAM connected to *opr* we generate an `alloc_(name)` macro. `name` is generated from the operation name producing the data,
- as many communication sequences 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 `ComThread_`, `EndComThread_` 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

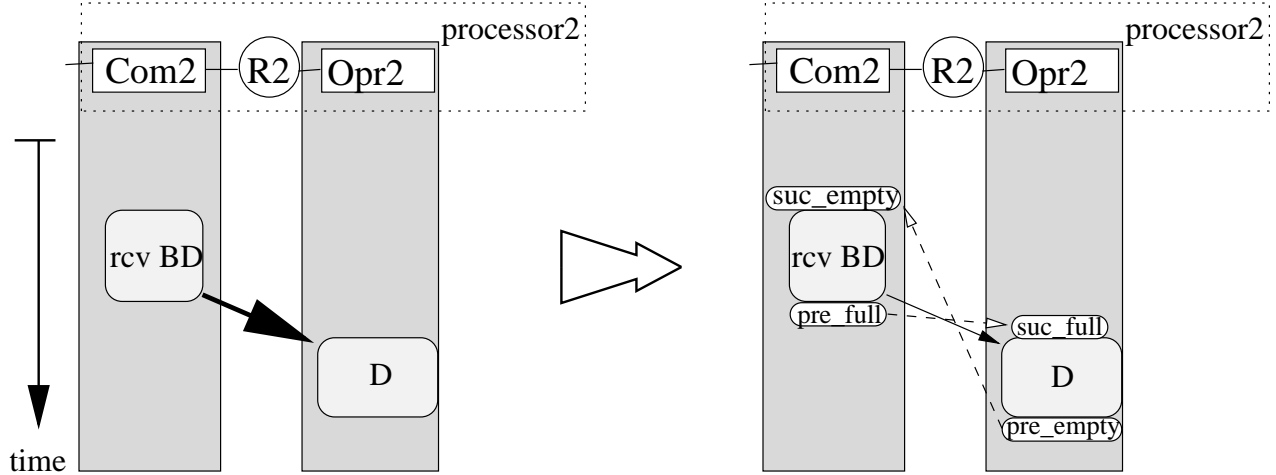


Figure 11: Principle of synchronization rcv

we generate a corresponding macro (`send_`, `receive_`, `read_`, `write_`, `pre_full`, `suc_full`, `pre_empty`, `suc_empty`). In order to distinguish a pair (`pre_`, `suc_`) synchronizing a sequence of communications with a sequence of computations from a pair (`pre_`, `suc_`) synchronizing a sequence of computations with a sequence of communications, we use a pair (`pre0_`, `suc0_`) for synchronizing a sequence of communications with a sequence of computations, and a pair (`pre1_`, `suc1_`) for synchronizing a sequence of computations with a sequence of communications. The arguments of these macros are computed from the edges connected to their corresponding vertices.

- a unique computation sequence. This sequence is generated between a pair of `Main_`, `EndMain_` macros. Such a sequence is also built by exploration of the sequence of totally ordered vertices allocated to the operator partition. A `spawn_thread_(com1)` macro has in charge to run the communication thread `com1`. This thread is executed under DMA interrupt (end of transfers interrupt) of the main thread.

In order to generate an executable code whose partial order is consistent with the implementation graph, it is important to remember that the translation/print process follows exactly the order given to the vertices distributed onto this operator.

In order to measure the real-time performances of an application carried out with the AAA methodology, it is possible to generate executives with *chronometric operations* automatically inserted before each computation and each data communication. The real-time performances measure are performed in two steps: first on each processor the real-time start and end dates are measured and memorized using the real-time clock of the processor, second at the end of the application all the memorized values are transferred to one of the processors with mass storage capabilities. These measures may be compared to those computed by the heuristics in order to determine the optimized distribution and scheduling. The difference between the real-time measures and the computed measures, is representative of the difference between the models used in AAA and the reality. Moreover, these measures allow to determine the execution duration of the operations and data transfers, necessary to perform the architecture characterization as described in section 3.2.2.

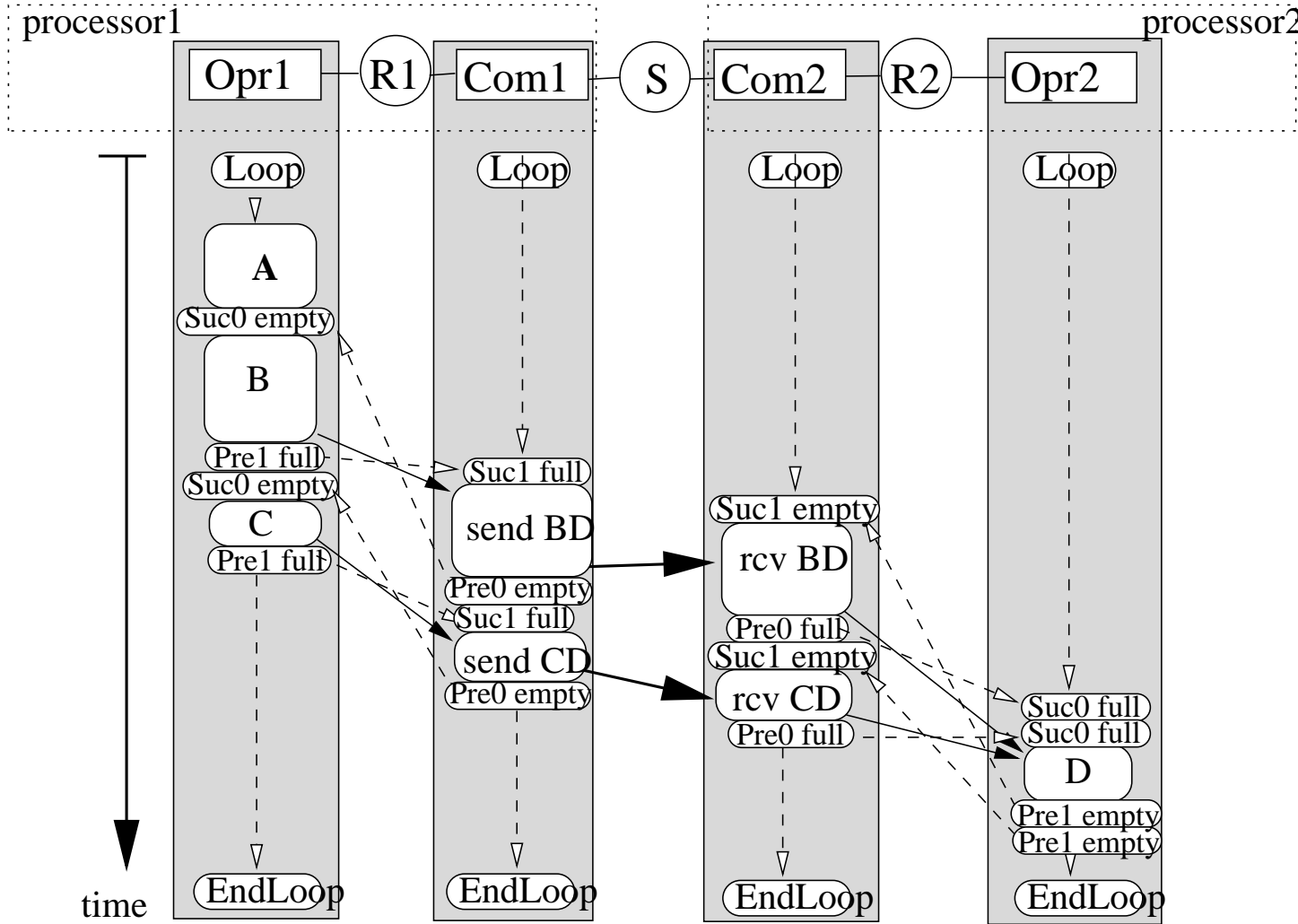


Figure 12: Execution graph after transformation of implementation graph of figure 9

3.5.3 Macro-code to source files

Each macro-code is translated by a macro-processor into a source code depending on the language chosen for the target operator. We use the free software Gnu-m4 macro-processor (<http://www.gnu.org/software/m4>).

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 in two sets corresponding to two kinds of libraries. The first one is an extensible set of *application macros*, which support the algorithm operations. The second, constituting an executive kernel, 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 characterize algorithm operations and to profile the application).

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.

3.5.4 Example of macro-code

Figure 13 is an example of code generation obtained by transformation of the execution graph given in figure 12. This example will focus on processor p1 (the code of processor p2 given in figure 14 is generated symmetrically):

- generation of a `semaphores_` macro (lines 5-10) which allocates all the necessary semaphores, one pair `_full` `_empty` for each communication. The semaphores are managed by `pre_` and `suc_` synchronization operations;
- generation of `alloc_` macros (lines 11-14) for each allocation vertex associated to RAM *R1* of figure 9 (the reader must remind that for readability allocation vertex are not drawn on figure 12);
- the unique communicator sequence is generated between a pair of `thread_(SAM,x,p1,p2)` and `endthread` macros (lines 15 to 28), where *SAM* is the type of the communication, *x* is the name of the communication gate, *p1*, *p2* are the communicating processors. Each communication vertex scheduled on the communicator *com1* is translated into a `send_` (lines 21 and 24) or a `recv_` macro. Each synchronization vertex is translated into the corresponding macros `pre_(empty/full)` and `suc_(empty/full)` in order to synchronize the communicator sequence with the operator sequence, and vice-versa the operator sequence with the communicator sequence. The pair (`pre1_(full)`, `suc1_(full)`) (lines 36 and 20) synchronizes the operator sequence with the communicator sequence in the same repetition, whereas the pair (`pre0_(empty)`, `suc0_(empty)`) (lines 22 and 34) synchronizes the communicator sequence of the current repetition with the operator sequence of the previous repetition to guarantee that the `send` (line 21) in the previous repetition is terminated;
- the unique operator *opr1* sequences its operations between a pair of `main_` and `endmain_` macros (lines 29 to 43). Each operation vertex is translated into a macro with the same name *A*, *B*, *C* (lines 33, 35 and 38) taking the allocation vertex names as argument.

Then, these files are translated into the language of the target processor by the Gnu-m4 macro-processor using a processor specific library containing the macro-definitions of each system macro, and the macro-definitions of each application operation.

3.5.5 Example of macro-definition

Below is an example of macro-definition used by the macro-processor Gnu-m4 to translate SynDEx macro-instruction `alloc_` into C code.

Consider the macro-instruction:

```
alloc_(int,x,3)
```

To produce C code the definition of this macro is performed in two steps:

- First step: (in `syndex.m4x`, standard library)

```
def('alloc_',
    define('$2_type_', $1)dnl
    define('$2_size_', ifelse($3,,1,$3))dnl
    ifdef('$1_alloc_', '$1_alloc_', 'basicAlloc_')($2)')
```

- Second step: (in `U.m4x` library, C-Unix specific library)

```
define('basicAlloc_', '_'($'1_type_'_'1[$'1_size_];)
```

Consequently, the result given by Gnu-m4 for the above macro-instruction is:

```
int x[3];
```

Similarly, the translation of a `send_` macro may be `DMA_config_write_(alloc_BD, size_of(alloc_BD), com2)` if `com2` is the addresses of a RAM writable by a DMA channel of the processor. The implementation of synchronization macros is generally coded in assembly language, since performance and context switching minimization between the communication sequences and the computation sequence are required.

4 SynDEX: system level CAD software

SynDEX is a system level CAD software, i.e. the tool associated with the AAA methodology, for rapid prototyping and optimization of distributed real-time embedded applications. It may be freely downloaded at: www-rocq.inria.fr/syndex or syndex.org, and offers through a GUI the following functionalities:

- algorithm specification of the functionalities with a factorized conditioned data flow graph or interface with some high level specification languages,
- architecture specification of the multicomponent with a directed graph,
- heuristics execution for the optimized distribution and scheduling of the algorithm onto the architecture,
- visualization of the heuristics results, as a timing diagram corresponding to a simulation of the real-time execution,
- generation of the distributed real-time executives, mainly static and without any deadlock. They are built with a minimum overhead, from executive kernels, presently available for the following processors: ADSP216X(Sharc), TMS320C4X, TMS320C6X, i80C196, MC68332, MPC555, i80X86, and workstations under UNIX and LINUX. Executive kernels are easily ported on other processors from the existing ones.

The way to practically use the GUI of SynDEX is described in its User Manual and examples are given in its Tutorial.

5 Conclusion

We presented a formal methodology based on graphs, in order to optimize the implementation of distributed real-time embedded applications. SynDEx is a system level CAD software based on this methodology. When it is associated with a domain oriented language, the compiler of which provides the algorithm specification and allows monoprocessor simulation, if this language allows to verify logical properties the AAA methodology guarantees that these properties are maintained through all the steps of the implementation. Moreover, the resulting distributed real-time embedded application will behave like its monoprocessor simulation while verifying, in addition, real-time and embedding constraints. This approach providing a seamless software environment from the specification to the distributed real-time embedded executable code, leads to a high level of dependability which may even increase when fault tolerance is also specified in the same environment. Moreover, mainly because real-time tests are reduced and because code is automatically generated, the development cycle duration of distributed real-time embedded applications is also drastically reduced.

References

- [1] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.
- [2] E.M. Clarke, E.A. Emerson, and A.P. Sistla. *Automatic verification of finite-state concurrent systems using temporal logic specifications*. ACM TOPLAS, 8(2), 1986.
- [3] R.E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transaction on Computers*, C-35(8):677-691, August 1936.
- [4] David Harel and Amir Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*. Springer Verlag, New York, 1985.
- [5] A.M. Turing. On computable numbers, with an application to the entscheidungs problem. In *Proc. London Math. Soc.*, 1936.
- [6] Grady Brooch, Ivar Jacobson, James Rumbaugh, and Jim Rumbaugh. *The Unified Modeling Language User Guide (The Addison-Wesley Object Technology Series)*. Addison-Wesley, 1998.
- [7] R. Kocik. *Sur l'optimisation des systèmes distribués temps réel embarqués : application au prototypage rapide d'un véhicule électrique semi-autonome*. PhD thesis, Université de Rouen, Spécialité informatique industrielle, 22/03/2000.
- [8] A. Girault, H. Kalla, and Y. Sorel. A scheduling heuristics for distributed real-time embedded systems tolerant to processor and communication media failures. *International Journal of Production Research*, 42(14):2877–2898, July 2004.
- [9] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In *Proceedings of International Conference on Dependable Systems and Networks, DSN'03*, San Francisco, California, USA, June 2003.
- [10] R. Balakrishnan and K. Ranganathan. *A Textbook of Graph Theory*. Springer, 2000.
- [11] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [12] J.B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11), 1980.
- [13] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1), 1986.
- [14] I. Rival. The role of graphs in the theory of ordered sets and its applications. In *Nato Advanced Study Institute on Graphs and Order*, 1984.
- [15] Y. Sorel. Massively parallel systems with real time constraints, the algorithm architecture adequation methodology. In *Proceedings of Conference on Massively Parallel Computing Systems, MPCS'94*, Ischia, Italy, May 1994.
- [16] A. Vicard. *Formalisation et optimisation des systèmes informatiques distribués temps réel embarqués*. PhD thesis, Université de Paris Nord, Spécialité informatique, 5/07/1999.
- [17] A. Dias, C. Lavarenne, M. Akil, and Y. Sorel. Optimized implementation of real-time image processing algorithms on field programmable gate arrays. In *Proceedings of Fourth International Conference on Signal Processing, ICSP'98*, Beijing, China, October 1998.

- [18] A.Y. Zomaya. *Parallel and distributed computing handbook*. McGraw-Hill, 1996.
- [19] T. Grandpierre. *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés*. PhD thesis, Université de Paris Sud, Spécialité électronique, 30/11/2000.
- [20] F. Gecseg. *Products of automata*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1986.
- [21] Texas Instruments. Tms320c4x user's guide.
- [22] Oystein Ore. *Theory of Graphs*. AMS, 1962.
- [23] C.A. Mead and L.A. Conway. *Introduction to VLSI systems*. Addison-Wesley, 1980.
- [24] Y. Sorel. Real-time embedded image processing applications using the algorithm architecture adequation methodology. In *Proceedings of IEEE International Conference on Image Processing, ICIP'96*, Lausanne, Switzerland, September 1996.
- [25] Michael R. Garey and David S. Johnson. *Computers and intractability : a guide to the theory of NP-completeness*. W.H. Freeman, 1979.
- [26] Zhen Liu and Christophe Corroyer. Effectiveness of heuristics and simulated annealing for the scheduling of concurrent task. an empirical comparison. In *PARLE'93, 5th international PARLE conference, June 14-17*, pages 452–463, Munich, Germany, November 1993.
- [27] A. Vicard and Y. Sorel. Formalization and static optimization for parallel implementations. In *Proceedings of Workshop on Distributed and Parallel Systems, DAPSYS'98*, Budapest, Hungary, September 1998.
- [28] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.
- [29] Y.K. Kwok and I Ahmad. Dynamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems*, 7(5):506–521, Munich, Germany, May 1996.
- [30] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEM-OCODE'03*, Mont Saint-Michel, France, June 2003.

```

01: include(syndex.m4x)dnl ; Include generic kernel
02: dnl
03: processor_(proc,p1,al, ; START FILE p1.m4
04: SynDEx-7.0.2 (C) INRIA 2001-2009, 2009-10-07 10:33:55)
05: semaphores_( ; Semaphores declarations
06: Semaphore_Thread_x,
07: _al_C_CD_p1_x_empty,
08: _al_C_CD_p1_x_full,
09: _al_B_BD_p1_x_empty,
10: _al_B_BD_p1_x_full)
11: alloc_(int,_al_A_AB,1) ; Buff declarations
12: alloc_(int,_al_A_AC,1)
13: alloc_(int,_al_B_BD,1)
14: alloc_(int,_al_C_CD,1)
15: thread_(SAM,x,p1,p2) ; START SEQ COMMUNICATIONS
16: loadDnto_(,p2)
17: Pre0(_al_B_BD_p1_x_empty,,_al_B_BD,empty)
18: Pre0(_al_C_CD_p1_x_empty,,_al_C_CD,empty)
19: loop_
20: Suc1(_al_B_BD_p1_x_full,,_al_B_BD,full) ; Wait for buff BD full
21: send(_al_B_BD,proc,p1,p2) ; Send buff BD p1 -> p2
22: Pre0(_al_B_BD_p1_x_empty,,_al_B_BD,empty) ; Signal buff BD empty
; in current repetition
23: Suc1(_al_C_CD_p1_x_full,,_al_C_CD,full) ; Idem as send buff BD
24: send(_al_C_CD,proc,p1,p2)
25: Pre0(_al_C_CD_p1_x_empty,,_al_C_CD,empty)
26: endloop_
27: saveFrom_(,p2)
28: endthread_ ; END SEQ COMMUNICATIONS
29: main_ ; START SEQ COMPUTATIONS
30: spawn_thread_(x) ; Launch comm thread
31: A(_al_A_AB,_al_A_AC)
32: loop_
33: A(_al_A_AB,_al_A_AC) ; Compute A (sensor)
; write result in buff
; AB and AC
34: Suc0(_al_B_BD_p1_x_empty,x,_al_B_BD,empty) ; Wait for buff BD empty
; in previous repetition
35: B(_al_A_AB,_al_B_BD) ; Compute B read in buff
; AB write in buff BD
36: Pre1(_al_B_BD_p1_x_full,x,_al_B_BD,full) ; Signal buff BD full
; allowing send BD
37: Suc0(_al_C_CD_p1_x_empty,x,_al_C_CD,empty) ; Idem as compute B
38: C(_al_A_AC,_al_C_CD)
39: Pre1(_al_C_CD_p1_x_full,x,_al_C_CD,full)
40: endloop_
41: A(_al_A_AB,_al_A_AC)
42: wait_endthread_(Semaphore_Thread_x) ; Wait end comm threads
43: endmain_ ; END SEQ COMPUTATIONS
44: endprocessor_ ; END FILE p1.m4

```

Figure 13: Macro-code corresponding to the algorithm graph of figure 9 for processor p1

```

include(syndex.m4x)dnl
dnl
processor_(proc,p2,figure9,
SynDEx-7.0.2 (C) INRIA 2001-2009, 2009-10-07 10:33:55)

semaphores_(
  Semaphore_Thread_x,
  _al_C_CD_p2_x_empty,
  _al_C_CD_p2_x_full,
  _al_B_BD_p2_x_empty,
  _al_B_BD_p2_x_full)

alloc_(int,_al_B_BD,1)
alloc_(int,_al_C_CD,1)

thread_(SAM,x,p1,p2)
  loadFrom_(p1)
  loop_
    Suc1_( _al_B_BD_p2_x_empty, , _al_B_BD,empty)
    recv_( _al_B_BD,proc,p1,p2)
    Pre0_( _al_B_BD_p2_x_full, , _al_B_BD,full)
    Suc1_( _al_C_CD_p2_x_empty, , _al_C_CD,empty)
    recv_( _al_C_CD,proc,p1,p2)
    Pre0_( _al_C_CD_p2_x_full, , _al_C_CD,full)
  endloop_
  saveUpto_(p1)
endthread_

main_
  spawn_thread_(x)
  D(_al_B_BD,_al_C_CD)
  Pre1_( _al_B_BD_p2_x_empty,x, _al_B_BD,empty)
  Pre1_( _al_C_CD_p2_x_empty,x, _al_C_CD,empty)
  loop_
    Suc0_( _al_B_BD_p2_x_full,x, _al_B_BD,full)
    Suc0_( _al_C_CD_p2_x_full,x, _al_C_CD,full)
    D(_al_B_BD,_al_C_CD)
    Pre1_( _al_B_BD_p2_x_empty,x, _al_B_BD,empty)
    Pre1_( _al_C_CD_p2_x_empty,x, _al_C_CD,empty)
  endloop_
  D(_al_B_BD,_al_C_CD)
  wait_endthread_(Semaphore_Thread_x)
endmain_

endprocessor_

```

Figure 14: Macro-code corresponding to the algorithm graph of figure 9 for processor p2