A design method for implementing specifications including control in distributed embedded systems

Nicolas Pernet, Yves Sorel

INRIA Rocquencourt,
Le Chesnay, FRANCE
Nicolas.Pernet@inira.fr, Yves.Sorel@inria.fr

Abstract

Because a real-time system combines control and data processing designers specify it using different languages. Such systems are often distributed and the problem is to obtain a distributed implementation from these distinct specifications. Indeed, the method based on separated code generation and manual distribution leads to inconsistent implementations. We propose to unify all these specifications into a unique one. The resulting specification is a conditioned data flow graph which exhibits the potential parallelism necessary to an efficient use of distributed resources. Finally, we use the SynDEx software in order to automatically produce a distributed and consistent implementation from the resulting specification.

1. Introduction

We focus on distributed real-time embedded systems. We call system, the implementation of functionalities, or algorithms onto physical architecture. Such systems are, first of all, reactive [1], that is to say, they interact with their environment by getting input signals, computing several operations and producing output signals. Real-time systems are reactive systems for which timing constraints are imposed between two consecutive input events (period) or between an input event and the corresponding output event produced by the system, in reaction to this input event (latency). In embedded systems we are interested in, the physical architecture is often distributed and the functionalities consist in control and data processing. The functionalities of the system are mainly described with graphical specification languages.

The following section presents the characteristics of control and data processing algorithms and explains how these characteristics lead to the use of different specification languages. We discuss the usual design method used to specify and implement distributed embedded systems in order to emphasize its limits and drawbacks. In order to cope with distributed implementation, a *conditioned data flow* model which enables control specification is proposed in the third section. It is a hierarchical version of the model proposed by Buck [2] that we flatten to produce a new data flow graph well suited for distribution. The principles of the flattening transformation are described in the fourth section. The fifth section proposes a design method which covers the whole development cycle from specification to implementation using the system level CAD software SynDEx.

2. Control and data processing systems

2.1. Control and data processing aspects

Functionalities of real-time embedded systems usually combine control aspects and data processing aspects. Data processing consists in consuming input data, computing them, and producing output data. Control consists in sequencing data processing, possibly by choosing according to the result of a test, one data processing to execute among several exclusive ones. When such test refers to a previously produced data, this data must be stored in order to represent the state of the system, typically leading to FSM (Finite State Machine).

2.2. Specification languages

There are two types of models for graph-based (i.e. using graphs) languages used to specify embedded systems including control and data processing: control flow and data flow models. A model may be common to several languages whose mainly the syntax differs.

In a control flow graph, representing a FSM, each vertex is a state and each edge is a transition triggered by a condition used to switch from one state to another one. Data processing may be triggered either by a state or by a transition.

In a data flow graph each vertex is an operation representing a data processing and each edge is a data dependence between two operations. In order to specify control in data flow, Dennis [3] defines two special vertices called *switch* and *merge*. These vertices allow to specify conditional operations, and data flow languages including control are based on this elementary mechanism. Nevertheless, these vertices are not adapted to distributed implementation due to implicit data dependences.

In control flow graph the set of edges defines a total order onto states and consequently also onto data processings. Moreover, data are stored in global variables which may be modified by different states or transitions, leading to potential inconsistency problems in the case of distributed implementation.

On the contrary, in data flow graphs, the set of edges defines a partial order onto operations, that we call "potential parallelism". Indeed, two operations without data dependence may be executed at the same time on different processors if the physical architecture allows it. Furthermore, all data dependences are explicit even the ones used by control.

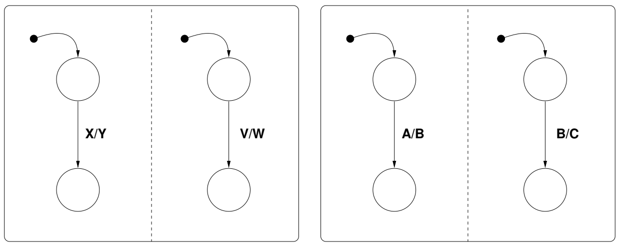

Figure 1. Two examples of parallelism specification in control flow graphs

Notice that some control flow languages allow to compose in parallel several control flow graphs. Since the different graphs may share global variables, this parallelism does not necessarily leads to potential parallelism. For example, in figure 1, the parallelism specified by the dashed line in the left hand side specification leads to potential parallelism because no variable is shared by both graphs. The label "X/Y" on edges means "if X then emit Y". On the contrary, in the right hand side specification, the dashed line does not lead to potential parallelism since it exists, between both graphs, an implicit data dependence (for B) which prevents to exploit the specified parallelism. Indeed B is written in one graph and read in the other one.

Consequently, when distributed implementation is intended a data flow graph is suited for the two following reasons. On the one hand the partial order describes potential parallelism which could be exploited by the distributed implementation, and on the other hand all data dependences are explicit which strongly reduces data inconsistency problems onto distributed implementation.

A last important difference concerns the executable code. A data flow graph describes the way to compute a set of reactions (output) from a set of stimuli (input). When an executable code corresponding to a data flow graph is executed, the whole set of inputs is needed and the whole set of outputs is produced. On the contrary, a control flow graph describes the set of outputs which has to be produced depending on, firstly, the current state and secondly, the set of inputs. When an executable code corresponding to a control flow graph is executed, the set of necessary input depends on the current state as the set of output does.

2.3. Usual design method for distributed embedded systems

Since embedded systems include both control and data processing, designers use different languages to specify their functionalities [4]. This is the case in industry when Stateflow (control flow) is used with Simulink[1] (data flow). After verification and simulation of the different specifications, the problem of distributed implementation remains. Indeed, the majority of these languages only proposes mono-processor code generation. Thus, the designer has to transform each specification in a sequential mono-processor code and allocate each code to a processor. Since data may be shared by different codes, he has to distribute and schedule inter-processor communications too. These human decisions often lead to an inconsistent implementation.

Moreover, the distribution of sensors and actuators is usually imposed onto distributed architecture, i.e. they are associated to a specific processor. Consequently, the designer has to carefully choose the processor executing each code since the number of necessary communications may vary and the corresponding overhead too. When an executable code resulting of a control flow graph is allocated onto a processor, some of its inputs and outputs correspond to sensors and actuators distributed onto the other processors. The designer has to schedule and distribute the corresponding communications. But as we saw previously the set of necessary inputs depends on the current state. Since the other processors do not know the current state, the only solution is to forward every input to the processor even if this input is useless. Similarly, a processor which executes an output operation cannot know if it will receive data or not. The solution is to produce the whole set of outputs at each reaction of the system. Consequently, an executable code which centralizes control leads to useless communications which reduce the effectiveness of the distributed implementation.

In order to improve the implementation of specifications including control in distributed embedded systems, let us define the principles of a better suited design method. Such design method must allow the use of different specification languages in order to specify the different aspects of a system. These different specifications must be translated into a unique data flow graph including control, which is well suited for distribution. Such design method must also be able to automatically schedule and distribute the unique data flow graph, and then automatically generate executable codes in order to reduce human decisions between specification and distributed implementation.

[1] http://www.mathworks.com

3. Conditioned data flow model

3.1. Reactive data flow

We previously presented the typical data flow model [3]. Before introducing control, we first need to extend this model to reactive systems. Due to the reactive properties of distributed embedded systems, the data flow graph must be infinitely repeated, each repetition corresponding to a logical instant of the synchronous languages [5] used to specify real-time systems with a denotational semantic. Every vertex is an operation which has to be executed before the beginning of the next repetition of the graph. Each operation has input and/or output ports and each edge, also called dependence, connects one output port to one or several input ports. Each operation cannot be executed until all its inputs are available. A specific vertex called *delay* is necessary if an operation need data produced during a precedent repetition of the graph.

3.2. Hierarchical model

Our conditioned data flow model is a hierarchical version of the Buck's Integer-controlled Data Flow (IDF)[2]. Hierarchy in data flow means an operation may be described by a subgraph of operations. In order to specify control in data flow graphs, we introduce a new type of vertex called *conditioning operation*. Such operation is described by several subgraphs introducing a first level of hierarchy. It consumes a *conditioning data* carried by a *conditioning dependence*. Depending on the value of this data, only one of the subgraphs is executed exclusively to the other ones. Each operation of the subgraphs of a conditioning operation is called a *conditioned operation*. The *condition* of a conditioned operation is the boolean corresponding to the test "is the conditioning data equal to the value specified for the subgraph?". A conditioned operation is executed when its condition is true. Again, thanks to hierarchy a conditioned operation may be in turn a conditioning operation. Figure 2 shows an example of conditioned data flow graph. It consists in five non-conditioning operations A, B, D, E and F, and one conditioning operation C. The conditioning data of C is the output of the operation A. If $output(A) = 1$, the execution of C is equivalent to the execution of the conditioned operation C_{11}. Otherwise, if $output(A) = 2$ the execution of C is equivalent to the execution of the conditioned operation C_{21} followed by the execution of the conditioned operation C_{22}.

3.3. Conditioned operations

We call *mutually exclusive* operations two conditioned operations whose conditions cannot be true at the same logical instant (infinite repetition). This means that only one of both operations will actually be executed at the corresponding physical instant. This property is interesting from the point of view of distributed scheduling because two operations in mutual exclusion can be scheduled on the same processor at the same time. In the example of figure 2, C_{11} and C_{21} are mutually exclusive.

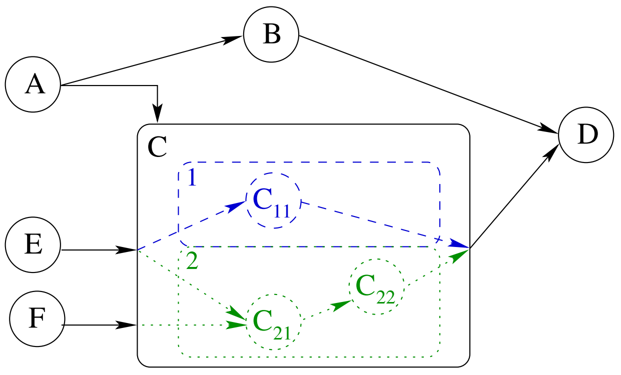

Figure 2. Conditioned data flow graph

Due to the data flow model a data consumed by an operation has to be produced, otherwise the consumer cannot begin its execution. Consequently, a conditioned operation cannot be the only producer of a data consumed by a non-conditioned operation. Indeed, when the condition of the producer is false the consumer is blocked waiting for this data. To solve this problem each output of a conditioning operation must be produced by exactly one operation of each exclusive subgraph. In the example of figure 2, the two exclusive conditioned operations C_{11} and C_{22} are the two producers of the data consumed by D.

Notice that, the reverse is not true. A non-conditioned operation may produce a data consumed by only one conditioned operation. Indeed, the producer is executed but the data is not consumed when the conditioned operation is not executed. Consequently, the input of a conditioning operation may be consumed, or not, by operations of each subgraph. In the example of figure 2, the data produced by the operation E is consumed by two mutually exclusive conditioned operations, C_{11} and C_{21}, whereas the data produced by F is consumed by only one conditioned operation, C_{21}.

3.4. Conditioned dependences

A dependence which connects at least one conditioned operation is called a *conditioned dependence*. The condition of a conditioned dependence is the condition of the consumer. For example, the condition of the dependence $A \rightarrow B$ is the condition of B. There is three types of conditioned dependences:

- an *incoming dependence* of a conditioning operation connects an external operation of the conditioning operation to one operation of a subgraph of the conditioning operation. In figure 2, the dependence $E \rightarrow C_{11}$ is an incoming dependence, $F \rightarrow C_{21}$ too.
- an *internal dependence* of a conditioning operation connects two operations of a subgraph of the conditioning operation. In figure 2, the dependence $C_{21} \rightarrow C_{22}$ is an internal dependence.
- an *outgoing dependence* of a conditioning operation

connects one operation of a subgraph of the conditioning operation to an external operation of the conditioning operation. In figure 2 on the page before, the dependence $C_{22} \rightarrow D$ is an outgoing dependence.

4. Flattening transformation

4.1. Limitation of the classical data flow model

We previously claimed that existing data flow models are not adapted for distribution. In order to explain why, let us consider the example of figure 3. It shows an IDF graph which introduces two vertices called *case* and *endcase*. These two vertices are an integer version of the *switch* and *merge* vertices proposed by Dennis [3]. A *case* vertex consumes an input data from its input i and a control data from its input c. A *case* vertex may have n outputs where each of the first $n-1$ outputs is labeled by an integer, and the n^{th} one is labeled by Def. The input data is copied on the output labeled by the value of the control data. If there is no corresponding label the data is copied on the output labeled *Def*.

In the same way, an *endcase* vertex consumes a control data, and the input data labeled by the value of the control data. If there is no corresponding label the consumed input data is the one labeled *Def*. The consumed input data is copied on the output o. In the given example, the data produced by the vertex A is consumed by C if the value of the data produced by B is equal to 1, consumed by D if the value of the data produced by B is equal to 4, and consumed by E otherwise. The vertex F consumes the data produced by C if the value of the data produced by B is equal to 1, the data produced by D if the value of the data produced by B is equal to 4, or the data produced by E otherwise. Notice that C is executed exclusively of D and E.

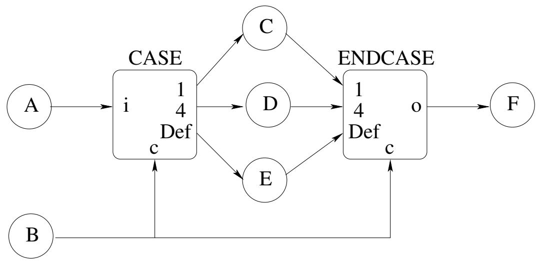

Figure 3. Example of IDF graph

Let us consider a distribution of the data flow graph of figure 3 such that vertices C and F are executed on processor P_1 and the rest of the graph is executed on P_2. According to the IDF graph, C distributed on P_1 seems to only need the data produced by the *case* vertex. Actually, the communication from *case* to C depends on the value of the B output, i.e. the communication takes place only if the data produced by B is equal to 1. Consequently, P_1 cannot decide whether C will be executed or not without knowing the value of B. That means a data dependence between B and C is missing. In the following, we describe how the flattening transformation adds systematically the data dependences necessary for distributed implementation.

4.2. Flattening transformation

We need to transform the graph to solve the problem arose by the distribution of conditioned data flow graphs. Indeed, we have to take into account that a processor which has to decide whether to execute a conditioned operation or not, needs to know the corresponding conditioning data. Moreover, a dependence implies a communication if it connects two operations distributed onto different processors. Obviously, a conditioned dependence implies a conditioned communication. In the case of a conditioned communication both consumer and producer need to know if the communication takes place or not. That is the reason why the flattening transformation adds vertices and dependences to the initial graph.

First, each conditioning operation is replaced by its subgraphs of conditioned operations and the flattening transformation adds for each conditioned operation a conditioning dependence corresponding to the conditioning data. Thanks to these conditioning dependences a processor, which has to decide whether to execute a conditioned operation or not, is forced to know the corresponding conditioning data.

With regard to the incoming conditioned dependence, when producer and consumer are distributed onto different processors, we have to force the producer to know the conditioning data of the conditioned dependence. Indeed, if this dependence implies a communication the producer would not send data without testing if the communication has to be done. The flattening transformation solves this problem by inserting a *CondI* operation on incoming conditioned dependences. Such operation has two inputs, one carrying the conditioning data, and another carrying the data previously carried by the incoming conditioned dependence. In addition, it has an output carrying the data previously carried by the incoming conditioned dependence. Because a *CondI* operation is not conditioned, its incoming dependences are not conditioned, contrary to its outgoing dependence which is conditioned like its consumer. Because of the incoming dependence carrying the conditioning data, the processor executing the *CondI* operation knows the condition of the communication, and consequently knows whether the communication has to take place or not. Finally, for one conditioning operation the flattening transformation adds as many *CondI* operations as there are of operations which have an incoming conditioned dependence toward this conditioning operation. An example is shown on figure 4 on the following page.

With regard to the internal conditioned dependence there is no problem thanks to the previously added conditioning dependence (when the conditioning operation was replaced by its subgraphs).

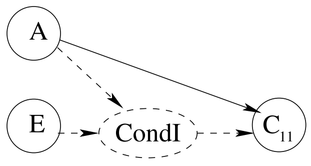

Figure 4. Transformation of the incoming dependence $E \rightarrow C_{11}$ of figure 2 on page 3

With regard to the outgoing conditioned dependences there is a problem due to multiple producers of a same data. We saw that for each subgraph of a conditioning operation each output is produced by only one conditioned operation. Since there are several subgraphs, there are several mutually exclusive producers of a same output data. That is the reason why a new operation is necessary in order to choose the producer according to the conditioning data, and to forward the corresponding data toward the consumer of the outgoing conditioned dependence. In order to solve this problem, the flattening transformation adds a *CondO* operation for each output of the conditioning operation. An example is shown in figure 5. A *CondO*

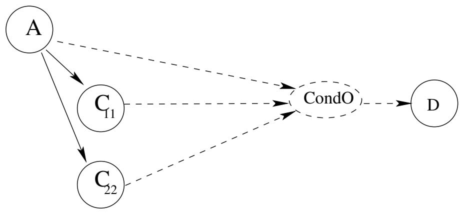

Figure 5. Transformation of the outgoing dependences $C_{11} \rightarrow D$ and $C_{22} \rightarrow D$ of figure 2 on page 3

operation has one output connected to the consumer of the corresponding outgoing conditioned dependence. This operation has one input carrying the conditioning data of the conditioning operation, and has as many additional inputs that there are of mutually exclusive producers of the corresponding data. According to the value of the conditioning data a *CondO* operation consumes the data corresponding to the executed conditioned operation, and forwards this data to its output. Because of the incoming dependence carrying the conditioning data, a processor which executes a *CondO* knows which conditioned operation was previously executed, and consequently from which processor the data is going to be send.

Figure 6 shows the flattened conditioned data flow graph corresponding to the graph of figure 2 on page 3.

5. Design method for distributed embedded systems

In the following section we present the SynDEx software for scheduling and distributing hierarchical conditioned data flow graphs by using the flattening transformation.

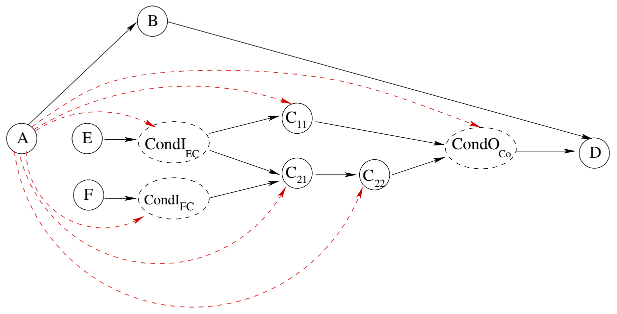

Figure 6. Flatten conditioned data flow graph corresponding to the graph of figure 2 on page 3

5.1. SynDEx

SynDEx[2] is a system level CAD software based on the Algorithm Architecture Adequation (AAA) methodology [6] for rapid prototyping and optimizing the implementation of distributed real-time embedded applications onto "multicomponent" architectures, i.e. made of programmable components (processor) and non programmable components (FPGA, ASIC) possibly of different types connected by communication media possibly of different types. The goal of the AAA methodology is to find the best matching between an algorithm specifying the functionalities, the system has to perform, and a multicomponent architecture, while satisfying real-time and embedding constraints. Adequation means an efficient matching obtained manually or automatically with optimization heuristics by exploring possible implementations. The AAA methodology is based on graphs models to exhibit both the potential parallelism of the algorithm and the available parallelism of the multicomponent. In SynDEx the graph model used to specify the functionalities (algorithm) is the conditioned data flow graph model previously presented. The implementation consists in distributing and scheduling the algorithm graph on the multicomponent graph. This is formalized in terms of graphs transformations. Heuristics taking into account execution time durations of computations and inter-component communications, are used to optimize real-time implementation. In SynDEx the first step of the adequation consists in the flattening transformation previously presented, the second step consists in a heuristic which tends to minimize the end-to-end execution time of the algorithm [7]. Finally, from the result of the adequation, SynDEx automatically generates distributed executives supporting the application [8].

In order to obtain, using SynDEx, a distributed implementation of different specifications including control, we propose two automatic translations producing a unique

[2]http://www.syndex.org

conditioned data flow graph. Then we show how to combine them, using SynDEx, in order to obtain their distributed implementation.

5.2. SyncCharts/SynDEx translation

SyncCharts [9] is a control flow language, close to the Statechart language [10], but totally deterministic. It is based on the Esterel synchronous language and provides formal verification. This language allows hierarchical specification, i.e. a state may be described by a subgraph of states, and parallel composition, i.e. several graphs may be executed in parallel. Figure 7 shows a SyncCharts example.

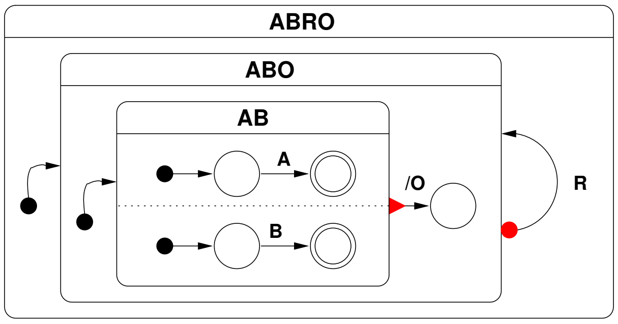

Figure 7. A SyncCharts example: ABRO

ample. The ABRO graph has a state ABO which is described by a graph in which one state is described by the parallel composition of two graphs. Its behavior is the following. The system waits for one occurrence of A and one occurrence of B. When the two occurrences happened the output O is produced. Each occurrence of R reset the system.

The SynDEx conditioned data flow graph produced by an automatic translation of the SyncCharts example appears in the upper part of figure 9 on the next page. Four delay operations correspond to the four states of the four graphs SyncCharts (ABRO, ABO, A and B). Four conditioning operations describe for each graph, using nested conditioning operations, the transitions allowing to leave the different states. Finally, on the right one vertex produces the output O. This translation was previously described in [11].

5.3. Scicos/SynDEx translation

Scicos[3] [12] is a toolbox of Scilab. It allows to specify a system with a graph which mixes control and data flow, and provides the corresponding simulation. This language is an alternative to Simulink.

Figure 8 shows a Scicos graph with activated operations. An input port used to activate an operation is located at the top of each operation. Some control operations, such as *If Then Else*, produce boolean data used to activate other operations. The translation of this example into a SynDEx graph appears in the lower part of figure 9 on the next page.

[3]http://www.scicos.org

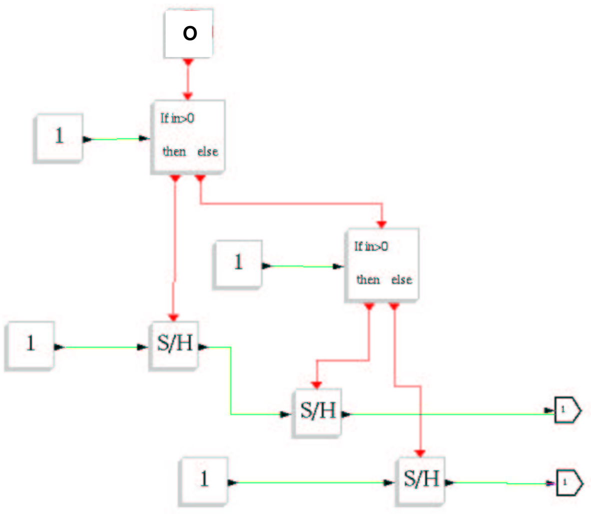

Figure 8. An example of Scicos graph

5.4. Distributed implementation by combining translations

In order to distribute and schedule a set of specifications made with different languages we use the two translations to convert each specification into a unique conditioned data flow graph. If the specifications share data, corresponding dependences must be added in the resulting graph. For example if the input O of the previous Scicos graph is the output O of the SyncCharts graph a dependence between both corresponding vertices must be added. Notice that the combination of Scicos with SyncCharts is a free software alternative to the use of Stateflow with Simulink. The translation into a unique SynDEx graph offers the possibility to generate automatically a distributed executable code, on contrary to the Mathworks tools which only allow mono-processor code generation. Moreover, this distributed code is consistent thanks to correct scheduling of communications (no deadlock by construction). Figure 9 on the following page shows the SynDEx graph obtained from translation of the SyncCharts and the Scicos graphs. After connecting the output O of the subgraph resulting of the SyncCharts graph to the input O of the Scicos graph, the designer may use SynDEx to specify an architecture (distributed or not). Figure 10 on the next page shows an architecture graph where three workstations are connected by a medium of communication.

Then, he has to characterize the duration of each operation (resp. each dependence) on each processor (resp. each communication medium), and may specify distribution constraints imposed by the physical architecture, i.e. an input operation has to be executed on a specific processor because the corresponding sensor is physically connected to this processor. Finally, the designer can compute different adequations (distribution and scheduling) onto different architectures (implementation exploration) and choose to generate distributed executable codes resulting

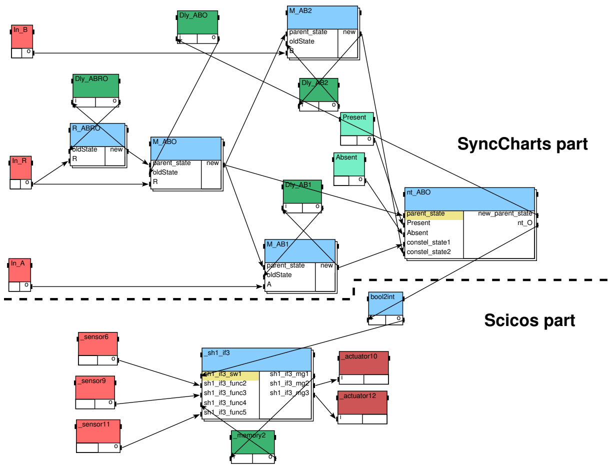

Figure 9. The Scicos and SyncCharts specifications translated into a unique SynDEx graph

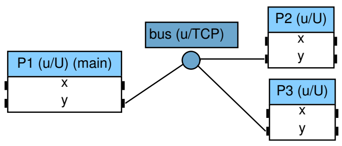

Figure 10. Architecture graph

from an adequation.

Figure 11 shows the result of the adequation between the algorithm of figure 9 and the architecture of figure 10. Thanks to both previously presented translations SynDEx can take advantage of an increased potential parallelism taking into account control aspects. Each column gives the schedule, from top to bottom, for a processor ($P1$, $P2$, $P3$) or a medium (bus). On a processor, thanks to mutually exclusive operations, several operations may be scheduled at the same time (at the same level in the column) as in the schedule of $P2$ and $P3$. Indeed, from the heuristics point of view the schedule of a processor or a medium consists in several mutually exclusive schedules which appear side to side in the same column. However, the generated code will take into account the value of the conditioning data in order to execute the corresponding exclusive operation.

Contrary to the usual design method the designer does not have to distribute and schedule communications because translation, adequation, and code generation are automated. This decreases the number of tests and increases the design consistency. Moreover, by converting control flow in data flow more potential parallelism is specified allowing a better (finer) exploration of the possible implementations.

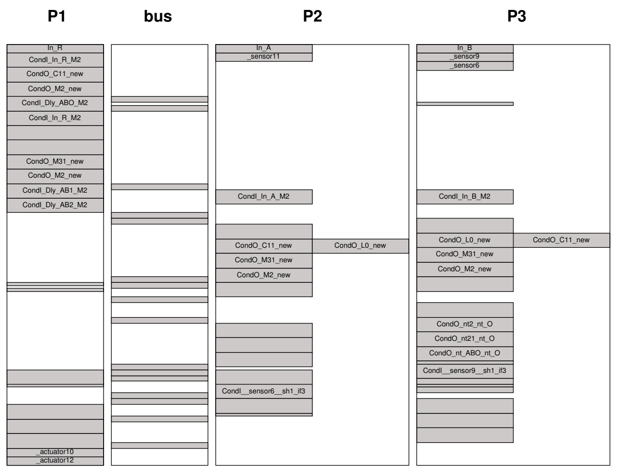

Figure 11. Result of an Adequation

6. Conclusion

We aim at distributing and scheduling in embedded systems specifications including control and data processing. The main problem consists in achieving a correct implementation from the combination of these specifications. Indeed, in the usual method the designers make mainly empirical decisions in order to distribute and schedule the different codes obtained from these different specifications.

We proposed a design method in order to distribute and schedule these specifications, ensuring the implementation consistency. It is based on a conditioned data flow model which enables control specification, and may be flattened in order to produce an new graph better suited for distribution. Each specification including control is automatically translated into a conditioned data flow graph. We presented two examples of such translation and how they may be combined in a unique conditioned data flow graph. Then, we showed how SynDEx can provide a simple and efficient way for distributing and scheduling as well as for automatically generating executable code.

This design method has multiple advantages. First of all, the designer continues to use the same specification languages as with the usual design method. For example, we showed that the combination of SyncCharts and Scicos may be a free software alternative to Stateflow/Simulink. Secondly, the conditioned data flow graph obtained from the translation of the combined specifications exhibits a potential parallelism which is of finer granularity that the potential parallelism obtained with the usual design method. Thus, the physical parallelism of the architecture may be better exploited by the heuristics of SynDEx. Finally, because the designer's empirical decisions are minimized, the consistency of the implementation is increased. This method is presently experimented by the car manufacturer PSA for implementing, onto a distributed architecture using the CAN bus, the coupled specifications of an ESP and a power-assisted steering.

References

[1] D. Harel and P. Amir. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*. Springer Verlag, New York, 1985.

[2] J.T. Buck and E.A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Proc. in the IEEE ICASSP*, Minneapolis, 1993.

[3] J.B. Dennis. First version of a dataflow procedure language. In *Lecture Notes in Computer Sci.*, volume 19, pages 362–376. Springer-Verlag, 1974.

[4] X. Liu, J. Liu, J. Eker, and E. A. Lee. Heterogeneous modeling and design of control systems. In *Software-Enabled Control: Information Technology for Dynamical Systems*. Wiley-IEEE Press, Tariq Samad and Gary Balas edition, April 2003.

[5] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[6] Y. Sorel. Massively parallel systems with real time constraints, the algorithm architecture adequation methodology. In *Proceedings of Conf. on Massively Parallel Computing Systems*, Ischia, Italy, May 1994.

[7] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *CODES'99 7th International Workshop on Hardware/Software Co-Design*, Rome, May 1999.

[8] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Formal Methods and Models for Codesign Conference (MEMOCODE'2003)*, pages 123–133, Mont Saint-Michel, France, June 2003.

[9] C. André. Computing syncharts reactions. In *Synchronous Languages Applications and Programming*, Porto, Portugal, July 2003.

[10] D. Harel. Statecharts: a visual formalism for complex systems. In *Science of Computer Programming*, volume 8, pages 231–274, 1987.

[11] N. Pernet and Y. Sorel. Optimized implementation of distributed real-time embedded systems mixing control and data processing. In *Proceedings of the ISCA 16th International Conference: Computer Applications in Industry and Engineering (CAINE-2003)*, Las Vegas, Nv, US, November 2003.

[12] R. Nikoukhah and S. Steer. Scicos-a dynamic system builder and simulator. In *Proceedings of the 1996 IEEE International Symposium on Computer-Aided Control System Design*, September 1996.