# Optimized Implementation of Distributed Real-Time Embedded Systems Mixing Control and Data Processing

Nicolas Pernet
OSTRE Team
INRIA Rocquencourt
78153 Le Chesnay Cedex, France
nicolas.pernet@inria.fr

Yves Sorel
OSTRE Team
INRIA Rocquencourt
78153 Le Chesnay Cedex, France
yves.sorel@inria.fr

**Abstract**

Most distributed real-time embedded systems are specified combining state diagram and data flow languages. This leads to several real-time codes which together do not necessarily satisfy the global specification, and consequently increases the development cycle because more tests are needed. Then, we propose, in order to obtain a single real-time code, to translate state diagram specification into data flow specification because it best exhibits the potential parallelism necessary for efficient distributed implementation. We choose to translate SyncCharts, a state diagram language which is deterministic and then well adapted to real-time, into SynDEx, a data flow language which allows automatic distributed code generation. This approach optimizes the distributed real-time code, and reduces the development cycle of complex distributed real-time embedded systems.

**Keywords**: Distributed Embedded Real-Time, Control Data Processing.

## 1 Introduction

We focus on distributed real-time embedded systems. Such systems are, first of all, reactive, 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 the embedded systems we are interested in, the physical architecture is often distributed. The behavior of the system is mainly described as a functional specification. In this article we discuss classical languages and their specific features used to provide such specifications and conclude that in the general case, it is necessary to mix some of them. Then, we propose a translation between these languages in order to obtain a single and optimized specification allowing code generation for efficient distributed implementation.

## 2 Specification of distributed embedded systems mixing control and data processing

### 2.1 Basics of control and data processing

Data processing, that is to say, performing operations on data, consists in consuming input data, computing them and producing output data. Control consists in managing data processing, by imposing the execution order (sequencing) of the operations and by choosing (test and branching) among several exclusive operations one of them. The execution order imposed by the control implies a global knowledge of the system (state), that is to say, the operations already executed and the next operation to execute.

### 2.2 Control oriented specification

To start with an example, when specifying the management of an elevator, control is predominant. Each operation, that is to say, data processing the system has to perform, takes place when transiting from one state to another one, and depends on the event which triggers this transition. This leads to a sequence of operations. Here, data processing is a direct consequence of control. For instance, the operation *open-the-door* can be executed only if the event *floor* occurs and the system transits from the state *cabin-move* to the state *cabin-stop*. Statechart [7] is an example of graphical state diagram languages for specifying such systems. Notice that data and states are global variables shared by all the operations specifying the system.

### 2.3 Data oriented specification

Let us consider another example, when specifying signal filtering, data processing is predominant. Each operation consumes data from and produces data to other operations. Here, the control is a direct consequence of data dependences which sequence operations. For instance, the operation *add* consumes data produced by another operation *mul*, and is therefore executed after *mul*. Signal [4], Simulink2 [1] are examples of data flow (graphical or not) languages [5] for specifying such systems. In order to express the other control aspects, state and branching, solutions are different depending on the data flow language. To provide control branching, in the Dataflow Procedure Language of Dennis [5], specific vertices and edges are used with boolean data, whereas in Simulink2 a block (operation) may consume a specific boolean data from an input called *trigger* in order to control its execution. Concerning state, in Signal and AAA/SynDEx [6], a vertex called *delay* is used. Notice that each edge represents only a data transfer, consequently there is no global variable shared by all the operations, and control is handled through data.

### 2.4 Control and data processing combination

Most realistic embedded real-time systems combine control and data processing. In this case the global system is usually composed of a high level control oriented sub-system which executes different data processing subsystem for each state (mode) transition. However, data processing sub-system may in turn include control. Such global systems may be totally specified with state diagram languages, but data dependences between operations cannot be clearly specified and furthermore problems may occur due to shared variables. Similarly, they may be totally specified with data flow languages, but the control is hidden in data dependences making it difficult to specify imbricated tests and branchings. Moreover, states imply delay vertices and tests imply boolean data dependences, that increases the complexity of the specification. Finally, as control is hidden, it is more difficult to make analyses for verification or optimization purposes. Consequently, each sub-system of a system is usually specified with the best suited language, either state diagram or data flow. Some languages allow verification and automatic code generation which satisfies the specification. When using several languages it is very difficult and even impossible to ensure that the set of the corresponding generated codes will satisfy the specification. Furthermore, since specifications of sub-system have to be modified until the global behavior of the system becomes satisfactory, the development cycle is longer than when only one language is used. To conclude, the solution is to use both types of languages for the specification but, before code generation, translate the state diagram specifications into data flow specifications and combine all the data flow specifications, or vice-versa, and finally generate a unique code instead of multiple ones.

### 2.5 Distributed implementation

As we said in introduction, we focus on distributed systems. Thus, we aim at performing an efficient implementation of the specification onto a distributed architecture. This architecture presents physical parallelism, that is to say, the possibility to execute concurrently several operations of the specification on different operators (hardware components) of the architecture. This is the reason why we need languages able to specify *potential parallelism*, in other words the possibility for two operations to be concurrently executed only if the architecture has the corresponding ressources (hardware components). We should emphasize that, in the case of state diagram languages, potential parallelism must be explicitly specified through a specific parallel constructor. In the case of data flow languages potential parallelism is more implicit. If two operations are not related by a data dependence, they can be executed in parallel since there is no execution order imposed by any data dependence. However, because the potential parallelism is deduced from data dependences, the user must be aware that he is specifying potential parallelism. Another important issue concerns the data management. In data flow languages, each data dependence is clearly specified and then may be easily translated into a data communication in the case where two operations in data dependence are distributed on two different processors. In state diagram languages, data is global variables possibly shared by different operations, the management of which, raises well-known difficulties in the distributed case. Consequently, data communications between operations are difficult to identify from the specification. For those reasons, data flow languages are preferred when the target is a distributed implementation.

## 3 Implementation with AAA/SynDEx

### 3.1 Principles of AAA

AAA stands for Algorithm Architecture Adequation. The goal of the AAA methodology consists in finding the best matching between an algorithm and an architecture, while satifying constraints. Adequation means an efficient matching. An algorithm describes the functionnalities the system has to perform. It is a set of operations partially ordered by data dependences, which is infinitely repeated according to the reactive nature of the system. The AAA methodology is based on graphs models to exhibit both the potential parallelism of the algorithm and the physical parallelism of the multiprocessor architecture. The implementation consists in distributing and scheduling the algorithm graph onto the architecture graph while satisfying real-time constraints. This is formalized in terms of graphs transformations. Heuristics taking into account execution time durations of computations and inter-processor communications, are used to optimize performances of real-time systems.

### 3.2 Implementation with SynDEx

In SynDEx the user has to specify two graphs, the algorithm and the architecture. The algorithm graph is a data flow graph which is infinitly repeated. All the operations of the data flow graph must be executed before the beginning of the next infinite repetition. Each vertex (operation) has input and/or output ports and each edge (data dependence) connects an output port to an input port. Every operation is executed when all its input data is available. A specific vertex called *delay* is used when an operation needs a data produced during a previous infinite repetition. A vertex may be specified in turn as a sub-graph allowing hierarchical specification. It is possible to condition the execution of alternative data flow graphs according to a *conditioning dependence* similar to test and branching. The architecture graph is a directed graph where the vertices are operators (microcontroler, DSP, FPGA) or media (Ethernet, CAN, RS232) and edges are connections between them. After specifying these two graphs, the user has to provide the duration of each operation (resp. each data dependence) onto each operator (resp. each medium). Moreover, the user can specify distribution constraints (e.g. a specific operation onto a specific operator). Finally, an adequation may be performed and its result visualized as a timing diagram, which is also a graph, simulating the real-time execution. This result is an off-line distribution and real-time scheduling, which is exploited to automatically generate distributed real-time executives. These dedicated executives are deadlock free with very low overhead since they do not use any RTOS support.

### 3.3 Control in SynDEx

As explained before, the way to specify test and branching is different from one data flow language to another. In SynDEx, an operation may have a conditioning dependence such that for each value it may take, a different sub-graph is specified. For each infinite repetition of the data flow graph, the conditioning dependence value is tested and the corresponding sub-graph is executed. Thus, conditioning in SynDEx, that is to say, the use of conditioning dependences, corresponds to the control structure *if (...) then* {...} in C language, or *switch case* if there are more than one sub-graph. Conditioning allows alternative behaviors for the algorithm. Delays are used to maintain the state of the algorithm from one infinite repetition of the data flow graph to another one.

The figure 1 shows a simple control oriented algorithm specified with SynDEx. The flat automaton specifying the control oriented algorithm is at the top left hand corner of the figure 1. It has two states *On* and *Off*, and two transitions respectively triggered by events *Go* and *Stop*. The corresponding SynDEx algorithm is shown at the top middle part of the figure 1. *Ev_Go* and *Ev_stop* vertices are two sensors which acquire the signals (events) *Go* and *Stop*. *Automaton* vertex has a conditioning dependence, the input port *oldstate* of which, is colored in gray. Its two alternative sub-graphs are shown at the bottom of the figure 1. According to the value of the input signals and the current state stored in a delay vertex, these subgraph use a same conditionned vertex from which the alternative subgraphs are shown at the right side of the figure 1. As shown in the previous example, control-oriented algorithms are specified with SynDEx using imbricated conditionings. However, algorithm with a large amount of control are rapidly tedious to specify by hand, due to these numerous imbrications. Fortunately, it is possible to automatically translate a specification, made with a state diagram language, towards a SynDEx algorithm graph, and then after specifying the architecture graph, to obtain an efficient distributed implementation.

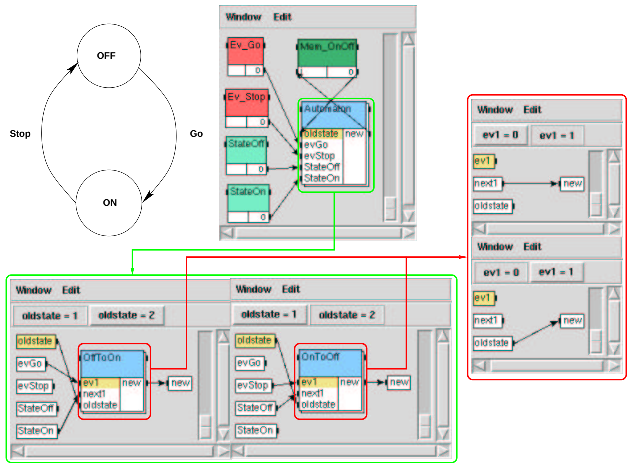

Figure 1: Specification of a flat automaton

## 4 A translation of state diagram language to data flow language for optimized distributed implementation

### 4.1 Overview

Although there exists some work on translation from state diagram language to data flow language, none is able to provide an efficient distributed implementation. For instance, Stateflow, a state diagram language, may be used with Simulink. The Stateflow state diagram is compiled into a C program and included as a vertex in the Simulink specification. This solution is not acceptable since the imported state diagram is a black box, that is to say, an atomic operation, without any potential parallelism. A translation from a specification made with the ARGOS language into boolean equation of the DC data flow language is described in [8]. Similarly, a translation of Statecharts and Activity-Charts into SIGNAL equations is described in [3]. However, both translations were proposed with different purpose than our, as they are optimized in order to perform formal verifications on DC or SIGNAL programs and centralized rather than distributed implementation.

## 4.2 Choice of a state diagram language

We had to choose among the state diagram languages one that would be translated in SynDEx. The most popular one is Statechart [7] but its semantics may lead to non-deterministic behavior that is not compliant with the deterministic real-time scheduling of SynDEx. Therefore, we chose SyncCharts [2] which is is better suited to real-time systems, because this language is deterministic and provides, in addition to the Statechart features, some specific edges which allow different kind of abortion. In [9], many Statechart semantics variants are compared and their lacks are discussed. In this article, table 3 summarizes this comparison. Unfortunately, as SyncCharts is too recent, it does not appear in it. In order to enable a comparison between SyncCharts and the other variants, we provide the missing collumn in the table 1.

| | |
|---|---|
| graphical/textual | textual |
| negated trigger event | - |
| timeout event | - |
| timed transition | - |
| disjunction of trigger events | - |
| trigger condition | + |
| state reference | - |
| assignement to variable | + |
| inter-level transition | - |
| history-mechanism | - |
| operationnal/denotational | denotational |
| compositional | + |
| synchrony hypothesis | + |
| deterministic | + |
| interleaving/true concurrence | true concurrence |
| discrete/continuous time | discrete |
| globally consistent | + |
| causal | + |
| instantaneous state | + |
| finite transition number | + |
| priorities | + |
| non-preemptive intererrupt | + |
| preemptive interrupt | + |
| distinction internal/external events | + |
| local event | + |
| discrete/continuous event | discrete |

Table 1: Characteristics of SyncCharts according to distinctives features corresponding to the table 3 of Von der Beck's article

## 4.3 Principles of SyncCharts

SyncCharts includes the main features of Statechart. First, the states of an automaton can be refined by another automaton (hierarchy). Second, several automata may be composed in parallel. Third, local variables may be specified in order to synchronize several parallel automata. In SyncCharts, an operation may be specified to be executed during a specific transition or when a specific state is the current state. Moreover, a trigger signal, that is to say, the boolean signal which is tested before performing the transition, is specified for each transition. To avoid non-deterministic behavior possible with Statechart, a priority must be specified when several transitions have the same source state. Moreover, SyncCharts provides four type of edge. First of all, the *abortion edge* may be *weak* or *strong* (with a circle at its source). When the trigger signal of abortion edge is true, the current state changes and if the old state was refined, its refinement is evaluated in the case of weak abortion or ignored in the case of strong one. Then, *normal termination edge* (with a triangle at its source) may be used if the source state is a macrostate (is refined) with several automata (constellations) in parallel. In each constellation, a final state is specified by a double border. When the source of a normal termination edge is the current state and all the final states of the parallel constellations are reached, this edge is crossed instantaneously. Finally, when the trigger signal of a *suspension edge* (with a circle at its destination) is present, no transition is evaluated inside the constellations which refine the arrival macrostate.

## 4.4 SyncCharts/SynDEx translation: the ABRO example

We use a simple SyncCharts example called ABRO in order to illustrate the principles of the translation. The ABRO automaton is shown in the figure 2. Its expected

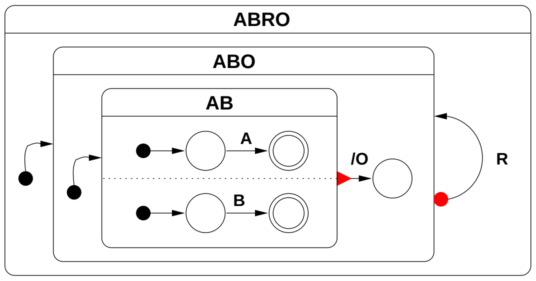

Figure 2: A SyncCharts example: ABRO

behavior is the following. The system waits for the occurrences of signals A and B and emits O as soon as both signal A and B have occurred. Each occurrence of R reinitializes the system. The hierarchy of the ABRO automaton is the following. ABRO has a state which is refined by ABO, which in turn has a state refined by AB, which is composed by automata A and B in parallel. The figure 3 on the next page shows the corresponding SynDEx algorithm, resulting from the automatic translation of the ABRO SyncCharts. Each constellation of the SyncCharts specification

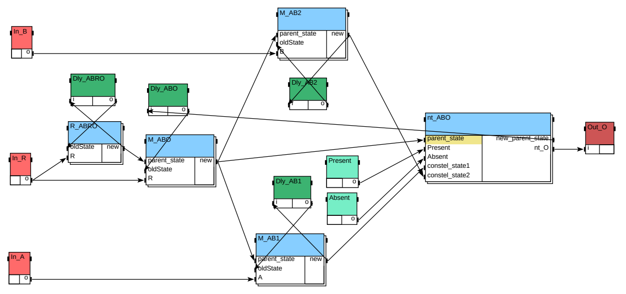

Figure 3: ABRO obtained by the translation

is translated into a vertex using imbricated conditionings and a delay to memorize the current state. After specifying the architecture, an efficient implementation of the ABRO SyncCharts may be performed by SynDEx.

### 4.5 Benefits of the translation

In order to present the advantages of our approach, we have to compare it with the usual approaches mixing state diagram and data flow. In the usual approach, since state diagram languages usually generate only mono-processor code, the control part of the algorithm is entirely executed on one processor (not distributed), although the architecture is actually distributed. Yet, on such architectures, sensors and actuators, providing the inputs and outputs of the control part, are constrained to be allocated onto specific processors, in order to minimize electric wiring. Then data communications are necessary between these specific processors and the processor where the control is executed. These data communications may induce an important overhead. Our translation enables to exhibit potential parallelism and to distribute efficiently the control through the processors where the sensors and actuators are allocated. Thus, data communications are minimized and consequently the total time spent to execute the algorithm, is reduced.

## 5 Conclusion and future research

Real-time distributed system, we focus on, are usually specified by combinating state diagram and data flow languages. However, in this case, this is very difficult to obtain an implementation satisfactory the set of specifications. In order to automatically obtain an efficient distributed implementation, we propose to translate the state diagram specification in data flow specification, which best exhibit potential parallelism. In this article, we propose a translation from the state diagram language SyncCharts, to the data flow language SynDEx. To specify a system, SyncCharts is used for control and SynDEx for data processing, then SyncCharts is translated to SynDEx. Then, an architecture and real-time or distributed constraints may be specified in order to perform an adequation which provides an efficient distributed implementation. This translation technics could be use also for Statechart or Stateflow state diagram languages. Currently, control is made periodic due to off-line scheduling of SynDEx, i.e. inputs of the control part are acquired periodically. However, control is naturally aperiodic. Then, we plan to support aperiodic event by mixing on-line and off-line scheduling in SynDEx.

## References

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

[2] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *CESA'96 IEEE-SMC*, Lille, France, July 1996.

[3] J. R. Beauvais, R. Houdebine, P. Le Guernic, E. Rutten, and I. Gautier. A translation of statecharts and activitycharts into signal equations. Technical Report Research Report No. 3397, INRIA, July 1999.

[4] A. Benveniste, M. Le Borgne, and P. Le Guernic. Signal as a model for real-time and hybrid systems. In *ESOP '92. 4th European Symposium on Programming*, Berlin, 1992.

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

[6] 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.

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

[8] F. Maraninchi and N. Halbwachs. Compiling argos into boolean equations. In *Lecture Notes in Computer Science*, volume 1135, pages 72–90, September 1996.

[9] M. von der Beeck. A comparison of statecharts variants. In *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, volume LNCS 863, pages 128–148. Springer-Verlag, 1994.