From specification to optimized implementation of distributed embedded systems mixing control and data processing

Pernet Nicolas, Yves Sorel

INRIA Rocquencourt,
BP 105 - 78153 Le Chesnay Cedex, France
Tel: 33(0)1 39 63 55 80, Fax: 33(0)1 39 63 57 86
nicolas.pernet@inria.fr, yves.sorel@inria.fr

Abstract

The specification of control and data processing for distributed real-time embedded systems usually combines state diagram and data flow languages. In order to reduce the development cycle, an automatic unique code generation is needed but unfortunately it is not possible with multi-languages specifications. Thus, multi-languages specifications must be merged into only one language specification. Moreover, data flow languages best exhibit the potential parallelism necessary for efficient distributed implementation. This article presents a method based on a translation from state diagram to data flow in order to reduce the development cycle and optimize the distributed implementation of systems mixing control and 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 where a delay is imposed between an input event arriving into the system and the output event produced by the system, in reaction to this input event. Since delays are imposed, the behavior of such systems must be deterministic. In the embedded systems we deal with, 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 on 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'. Sequential Function Chart [1] or Statechart [2] are examples of graphical state diagrams languages for specifying such systems. Notice that data and states are global variables shared by all the operations.

2.3 Data oriented specification

Considering 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 [3], Simulink2 [4] 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 languages. 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 sub-system 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. Most languages allow verification and automatic code generation which satisfies the specification. When dealing with several languages it is impossible to ensure that the set of the corresponding generated codes will satisfy the specification. Furthermore, the development cycle is longer than when only one language is used. As conclusion, 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 fusion all the data flow specifications, or vice-versa, and finally generate a unique code.

### 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 resources (components). We should emphasize that, in the case of
state diagram languages, potential parallelism is explicitly specified through a specific parallel constructor.
For example a pipe character must be used to compose in parallel several state diagrams. In the case of
data flow languages potential parallelism is more implicit. If two operations are not related by a data de-
pendence, they can be executed in parallel since there is no execution order imposed by any data dependence.
Yet, 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 state diagram languages,
data is global variables possibly shared by different operations, the management of which, raises well-
known difficulties in the distributed case. The data transfer between two operations is implicit through their
common variable. Consequently, data communications between operations are difficult to identify from
the specification. This is the reason why 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 is to find the
best matching between an algorithm and an architecture, while satisfying constraints. Adequation means an
efficient matching; note that it is different from the word ”adequacy” which implies a sufficient matching.
An algorithm is the result of the transformation of an application specification, which may be more or
less formalized, into a software specification adapted to its digital processing by a computer or a specific
integrated circuit. More precisely, as defined by Turing [7] an algorithm is a finite sequence of operations
(total order) that must be processed in a finite time and with a finite hardware support. Here, we need 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 react to the physical environment must be finite because real-time constraints must be satisfied. Consequently, instead of a total order (sequence of operations) we use a partial order which describes a potential parallelism. The AAA methodology is based on graphs models to exhibit both the potential parallelism of the algorithm and the available parallelism of the multicomponent architecture. The implementation consists in distributing and scheduling the algorithm graph on 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-component communications, are used to optimize real-time performances and resources allocation of embedded real-time applications.

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. According to the reactive features of the system we are dealing with, this data flow graph is infinitely 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. The algorithm must have input (sensor) and output (actuator) vertices. 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 (general purpose processor, microcontroller, DSP, ASIC, FPGA) or media (Ethernet bus, CAN bus, RS232 link) and edges are connections between them. A medium cannot be connected to another medium and an operator to another operator. 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 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. The figure 1 shows the GUI of SynDEx.

Figure 1: From left to right graphs of Algorithm, Architecture and Adequation with AAA/SynDEx

### 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.

The figure 2 on the next page 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 2 on the following page. It possesses two states `On` and `Off`, and two transitions. If the current state is `Off` and the event `Go` occurs, then `On` becomes the current state. If the current state is `On` and the event `Stop` occurs, then `Off` becomes the current state. The corresponding SynDEx algorithm is shown at the top middle part of the figure 2 on the next page:

- `Ev_Go` and `Ev_Stop` vertices are two sensors which acquire the signals (events) `Go` and `Stop`,
Figure 2: Specification of a flat automaton

- \textit{StateOff} and \textit{StateOn} vertices are two constants used in order to provide the two possible values of the state (1 for \textit{Off} and 2 for \textit{On}),

- \textit{Mem.\hspace{1pt}OnOff} vertex is a delay memorizing the state,

- \textit{Automaton} vertex has a conditioning dependence, the input port of which, is colored in gray.

The two alternative sub-graphs are shown at the bottom of the figure 2. Then, according to the value of the data on the input port named \textit{oldstate} (which is the current state of the previous repetition) the behavior of the operation \textit{Automaton} is different. In the case where the value of \textit{oldstate} is 1 (at the bottom left hand corner of the figure 2), \textit{Off} is the current state and the algorithm has to evaluate the input \textit{Go}. A conditioned operation named \textit{OffToOn} is then used in order to update the state of the system. In the case where the value of \textit{oldstate} is 2, \textit{On} is the current state, the operation named \textit{OnToOff} is used to evaluate \textit{Stop} and update the state of the system. \textit{OnToOff} and \textit{OffToOn} are the same operations, their corresponding sub-graphs are shown at the right side of the figure 2. The state changes when the value of the conditioning dependence is 1, otherwise the state remains unchanged.
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, thanks to conditioning and delays, 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 perform efficient distributed implementation.

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 satisfying since the imported state diagram is a black box, that is to say an atomic operation, where no potential parallelism appears. 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 ActivityCharts into SIGNAL equations is described in [9]. Yet both translations were proposed with different purpose than our, as they are optimized in order to perform formal verifications on DC or SIGNAL programs.

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 [2] but its semantics may lead to non-deterministic behavior that is not compliant with the deterministic (off-line) scheduling of SynDEx. Therefore, we chose SyncCharts [10], because this language is deterministic and provides, in addition to the Statechart features, some specific edges well adapted to the embedded real-time systems. In [11], many Statechart semantics variants are compared and their lacks are discussed. The table 3 of the cited article 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 column in the table 1 on the next page.
4.3 Presentation of SyncCharts

SyncCharts includes the main features of Statechart:

- hierarchy: the states of an automaton can be refined by another automaton,
- parallelism: several automata may be composed in parallel,
- synchronization: local variable may be specified in order to synchronize two parallel automata.

Figure 3: Semantic of SyncCharts
There are two different approaches, Mealy and Moore, to specify operations on state diagram languages. In Mealy state diagrams, operations are performed on transitions whereas, in Moore state diagrams, operations are performed on states. As SyncCharts state diagram language mixes Mealy and Moore approaches, 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. The figure 3 on the preceding page describes the semantics of SyncCharts. The new features compared to Statechart are the four different edges of SyncCharts:

- weak abortion edge: when the trigger signal is true, the current state changes and if the old state was refined, its refinement is evaluated;

- strong abortion edge (with a circle at its source): when the trigger signal is present, the current state changes and if the source state was refined, its refinement is not evaluated. Note that if the source state is not refined, this edge is equivalent to a weak abortion one;

- normal termination edge (with a triangle at its source): this edge is 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. This is the reason why no trigger signal is needed for this type of edge;

- suspension edge (with a circle at its destination): when the trigger signal is present, the evolution of the destination macrostate is frozen, that is to say no transition is evaluated inside the constellations which refine this macrostate and the current states of these constellations remain unchanged.

### 4.4 Translation SyncCharts/SynDEx: the ABRO example

We use a simple example called ABRO in order to illustrate the principles of the translation. The automaton is shown in the figure 4 on the next page. The expected behavior is as follows:

- the system waits for the occurrences of signals A and B. We have two constellations waiting in parallel;
Figure 4: A SyncCharts example: ABRO

- as soon as both signal A and B have occurred, signal O is emitted, by using a normal termination edge;

- waiting can be aborted by an occurrence of R (strong abortion);

- each occurrence of R re-initializes 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 5 on the following page shows the corresponding SynDEx algorithm, resulting from the automatic translation of the ABRO SyncCharts. In an automaton, for a given current state only the events corresponding to its outcoming transitions need to be known. Therefore, in our translation, the sub-graph corresponding to this state only requires the data corresponding to these events (and not to all possible events). Moreover, SyncCharts state diagrams which mix Mealy and Moore approaches are first automatically translated in SyncCharts state diagram using only Moore approach and then translated to SynDEx. Indeed, the knowledge of the curent state is sufficient to compute the adequate operation in Moore state diagrams contrary to Mealy ones. Each constellation of the SyncCharts specification is translated into a vertex using imbricated conditionings and a delay to memorize the current state. In the figure 5 on the next page, the conditionned vertices R_ABRO,
Figure 5: ABRO obtained by the translation

M_ABO, M_AB1, M_AB2 and the delays Dly_ABRO, Dly_ABO, Dly_AB1 and Dly_AB2 are respectively the translations of the constellations ABRO, ABO, A and B of the SyncCharts specification. The sensor vertices In_A, In_B and In_R acquire the trigger signals used in the SyncCharts specification. Similarly, the actuator vertex Out_O emits the signal corresponding to the normal termination edge of the specification.

After specifying the distributed architecture, an efficient implementation of the ABRO SyncCharts specification may be performed by SynDEx adequation.

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. Since state diagram languages allow only mono-processor code generation, the control part of an algorithm is centralized when usual approaches are employed on distributed architecture. Yet, on such architectures, sensor and actuator, corresponding to the input and output of the control, are allocated to specific operators in order to minimize electric wires, for instance in cars. This implies data communication to (input) and from (output) the operator where the control is centralized. These data communications induce an important overhead in addition to time spent to evaluate control.

Our translation enables to exhibit potential parallelism and to distribute efficiently the control through the operators, which is not possible with usual approaches. Thus, data communications are minimized and the critical path, that is to say the time spent to perform the algorithm, is reduced.
5 Conclusion and future research

We explained that real-time distributed system we focus on may be specified through the combination of state diagrams and data flow languages. Nevertheless, by specifying such systems with several languages, we cannot obtain an implementation satisfying the set of the specifications. In order to automatically obtain an efficient implementation, the set of the sub-systems specified with state diagram languages should be translated in data flow languages, which best exhibit potential parallelism. In this article, we presented a translation from a state diagram language, SyncCharts, to a data flow language, the SynDEx algorithm graph. To specify a system, we can use SyncCharts for control and SynDEx for data processing, then translate the SyncCharts specifications to SynDEx graphs and finally obtain a global SynDEx algorithm. Then, architecture and real-time or distributed constraints may be added. Finally, we perform an adequation which provides an efficient implementation. Even if we chose to translate SyncCharts state diagrams, we can obviously consider that a translation from Statechart or Stateflow is feasible too.

In our future works we will try to include aperiodic operations in SynDEx. Indeed, on the contrary to data processing which is often periodic, control is aperiodic and can not always be reduced to a periodic operation by using polling methods. Consequently, we have to introduce on-line scheduling in the off-line scheduling of SynDEx.

References


