

Mode-Automata Based Methodology for Scade

Ouassila Labbani, Jean-Luc Dekeyser, and Pierre Boulet

Laboratoire d'Informatique Fondamentale de Lille,
Université des Sciences et Technologies de Lille, Bâtiment M3,
Cité Scientifique, 59655 Villeneuve d'Ascq Cedex, France
{labbani, dekeyser, boulet}@lifl.fr

Abstract. In this paper, we present a new design methodology for synchronous reactive systems, based on a clear separation between control and data flow parts. This methodology allows to facilitate the specification of different kinds of systems and to have a better readability. It also permits to separate the study of the different parts by using the most appropriate existing tools for each of them.

Following this idea, we are particularly interested in the notion of running modes and in the Scade tool. Scade is a graphical development environment coupling data processing and state machines (modeled by the synchronous languages Lustre and Esterel). It can be used to specify, simulate, verify and generate C code. However, this tool does not follow any design methodology, which often makes difficult the understanding and the re-use of existing applications. We will show that it is also difficult to separate control and data flow parts using Scade. Regulation systems are better specified using mode-automata which allow adding an automaton structure to data flow specifications written in Lustre. When we observe the mode-structure of the mode-automaton, we clearly see where the modes differ and the conditions for changing modes. This makes it possible to better understand the behavior of the system.

In this work, we try to combine the advantages of Scade and running modes, in order to develop a new design methodology which facilitates the study of several systems by respecting the separation between control and data flows. This schema is illustrated through the Climate case study suggested by Esterel Technologies¹, in order to exhibit the benefits of our approach compared to the one advocated in Scade.

1 Introduction

Development of complex and critical reactive systems requires reliable and efficient tools and methods. Some failures and crashes of these systems can lead to data or time losses, incidents that can potentially be catastrophic. For this reason, these systems are often submitted to severe requirements of good functioning, aiming the *zero error* quality. Their reliability becomes at the same time

¹ www.esterel-technologies.com

a more and more important stake and a problem which gets harder and harder to solve.

To address those needs, several studies have been launched in the reactive system domain. We often speak about approaches and tools for modeling, simulating, and checking of these systems. These tools are based on different models, depending on their basic hypotheses (synchronous or asynchronous, control or data flow, ...) and use formal techniques having a well defined syntax accompanied by a rigorous semantics based on mathematical models.

In this paper, we study synchronous reactive systems and we propose a new approach for modeling these systems. Our study is inspired by the principles used in Scade (Lustre + Esterel) and mode-automata. It is based on the precise and clear separation between control and data flow parts, that allow us, on the one hand, to avoid the use of conditional structure of Lustre and to have a best readability, and on the other hand, to facilitate the separated study of the different parts by using the most appropriate tools for each part.

2 Context

2.1 Reactive Systems and Synchronous Approach

Reactive Systems are computer systems that react continuously to their environment, by producing results at each invocation [1]. These results depend on data provided by the environment during the invocation, and on the internal state of the system. This class of systems contrasts with *transformational systems* and *interactive systems*. Transformational systems are classical programs whose inputs are available at the beginning of their execution, and which deliver their outputs when terminating, as compilers for instance. Interactive systems are programs which react continuously to their environment, but at their own speed, as operating systems for instance.

D. Harel and A. Pnueli [1] have given to reactive systems the image of a black box that react to its environment at a speed determined by the latter (figure 1).

Specification of software or hardware reactive systems behavior is complex. It can lead to difficult and important errors. Indeed, such systems are not only described by transformational relationships, specifying outputs from inputs, but also by the links between outputs and inputs via their possible combinations in one step [3]. Modeling reactive systems is therefore a difficult activity.

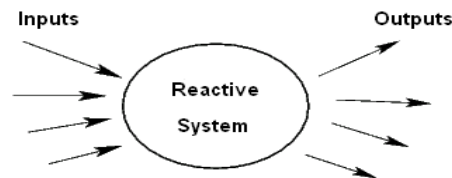


Fig. 1. Reactive System

In the beginning of the 80's, the family of synchronous languages and formalisms has been a very important contribution to the reactive system area [2]. Synchronous languages have been introduced to make programming reactive systems easier [4]. They are based on the *synchrony hypothesis* that does not take reaction time in consideration. Each activity can then be dated on the discrete time scale. This hypothesis considers that the computer is infinitely fast and each reaction is instantaneous and atomic.

Synchronous languages are devoted to the design, programming and validation of reactive systems. They have a formal semantics and can be efficiently compiled into C code, for instance. These languages can be classified into two main families: *declarative languages* and *imperative languages*.

Declarative or data flow languages like Lustre ([5] and [6]) or Signal ([7] and [10]) are used when the behavior of the system to be described has some regularity like in signal-processing. Their main task consists in consuming data, performing calculations and producing results.

Imperative or control flow languages like Esterel ([9], [8] and [16]) or Argos [11] are more appropriate for programming systems with discrete changes and whose control is dominant: for instance coffee machines. Their purpose is to manage the processing of data by imposing an execution order to operations and by choosing one operation among several exclusive.

However, rarely these systems have an exclusively regular or discrete behavior. The most realistic and used embedded systems combine control and data processing. Such global systems may be totally specified with imperative languages, but data dependences between operations can not be clearly specified and furthermore problems may occur due to shared variables. Similarly, they may be totally specified with declarative languages, but the control is hidden in data dependences making it difficult to specify tests and branchings necessary for verification or optimization purposes. For these reasons, we need efficient tools and methods taking in consideration this kind of systems.

Several approaches have been proposed in this domain. We can find the *multi-languages* approach which combines imperative and declarative languages, like using Lustre and Argos [18]. It is based on a linking mechanism and allows the re-use of existing code. However, when using several languages it is very difficult to ensure that the set of corresponding generated codes will satisfy the global specification. Another design method consists in using a *transformational* approach which allows the use of both types of languages for specification but, before code generation, the imperative specifications must be translated into declarative specifications, or vice-versa, allowing to generate a unique code instead of multiple ones. N. Pernet and Y. Sorel give in [19] an example of this approach which translates SyncCharts, a control flow language, into SynDEx, a data flow language which allows automatic distributed code generation. However, definition of transformation rules remains a difficult task and can induce several errors.

The transformational approach is efficient for describing reactive systems combining control and data processing. However, there are systems whose be-

havior is mainly regular but can switch instantaneously from a behavior to another. They are the systems with running modes. The most adapted method to describe this kind of system consists in using a *multi-styles* approach which makes it possible to describe with only one language the various behaviors of the system. The mode-automata represent a significant contribution in this field. Their goal consists in adding an automaton structure to the Lustre programs.

In our work we choose to study the transformational approach using Scade, where Esterel code is transformed into Lustre, and the concept of mode-automata allowing the description of different running modes of the system. The goal of our work consists in proposing a mixed approach which can facilitate the specification of a variety of synchronous reactive systems.

2.2 Scade

Scade (Safety Critical Application Development Environment) [13] is a graphical development environment commercialized by Esterel Technologies. The Scade environment was defined to help and assist the development of critical embedded systems. This environment is composed of several tools such as a graphical editor, a simulator, a model checker and a code generator that automatically translates graphical specifications into C code.

The Scade language is a graphical data flow specification language that can be translated into Lustre. Scade is built on formal foundations. It is deterministic and provides efficient solutions for the development of reactive systems. Thus, Scade enables the saving of a significant amount of verification efforts, essentially because it supports a *correct by construction* process [14] and automated production of the life cycle elements. It has been used in important European avionic projects (Airbus A340-600, A380, Eurocopter) and is also becoming a de-facto standard in this field.

Scade uses two specification formalisms: *block diagrams* for continuous control and *state machines* for discrete control [12]. It adds a rigorous view of these formalisms which includes a precise definition of concurrency and a proof that all Scade programs behave deterministically.

By *continuous control* we mean sampling sensors at regular time intervals, performing signal-processing computations on their values, and outputting values often using complex mathematical formulas. Data is continuously subject to the same transformation. In Scade, continuous control is graphically specified using block diagrams. Scade blocks are fully hierarchical: blocks at a description level can themselves be composed of smaller blocks interconnected by local flows.

By *discrete control* we mean changing the behavior according to external events originating either from discrete sensors and user inputs or from internal program events. Discrete control is generally represented by state machines. A richer concept of hierarchical state machines has been introduced in Scade to avoid the state explosion problems. The Esterel Technologies hierarchical state machines are called *Safe State Machines* (SSMs). These evolved from the Esterel programming language and the SyncCharts state machine [15].

Large applications contain cooperating continuous and discrete control parts. To make the specification of such systems easier, Scade makes it possible to seamlessly couple both data flow and state machine styles. Most often, one includes SSMs into block-diagram design to compute and propagate functioning modes. Then, the discrete signals to which a SSM reacts and which it sends back are simply transformed into boolean data flows in the block diagram.

Scade does not give any design methodology. It does not impose a well defined technique or rules to follow for the construction of the system, which gives more freedom to users. However, users can specify their system in a not very organized way which makes it difficult to understand and to re-use existing specifications. Thus, the application of formal verification techniques on such models is very difficult and even impossible. Errors are more and more serious and the resulting system will be unstable. It is also difficult to specify mainly regular systems which change instantaneously their behavior with Scade. These systems are more easily specified using mode-automata. In [20], F. Maraninchi and Y. Rémond show through a production-cell case study that real industrial applications can be better specified by using a mode-structure if their behavior is mainly regular. For these reasons, it becomes necessary to introduce a design methodology and the concept of running modes in Scade to facilitate the specification, the verification and the re-use of various applications.

2.3 Mode-Automata

Informal Presentation. One way of facing the complexity of a system is to decompose it into several "independent" tasks. Of course the tasks are never completely independent, but it should be possible to find a decomposition in which the tasks are not too strongly connected with each other. Different formalisms are used in the reliability engineering framework in order to design these models of systems under study: *Boolean formalisms* like block diagrams, and *states/transitions formalisms* like Petri nets.

Mode automata have been proposed in [17]. They introduce, in the domain-specific data-flow language Lustre for reactive systems, a new construct devoted to the expression of *running modes*. It corresponds to the fact that several definitions (equations) may exist for the same output, that should be used at distinct periods of time.

A mode automaton is an input/output automaton. It has a finite number of states, that are called *modes*. At each moment, it is in one (and only one) mode. It may change its mode when an event occurs. In each mode, a transfer function determines the values of output flows from the values of input flows. Mode automaton can be combined in order to design hierarchical models. They generalize both bounded Petri nets and block diagrams.

Figure 2 represents a simple example of mode-automaton. It has two states, and equations attached to them. The transitions are labeled by conditions on X. The important point is that X and its memory are *global* to all states. The only thing that changes when the automaton changes states is the transition function; the memory is preserved.

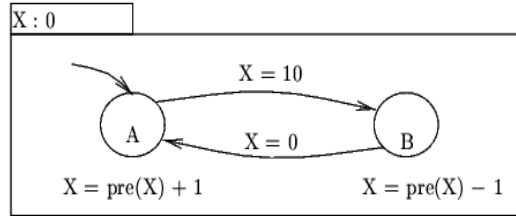


Fig. 2. Mode-automaton: simple example

Formal Definition. A mode-automaton is a tuple $(Q, q_0, V_i, V_o, I, f, T)$ where:

- Q is the set of states of the automaton part;
- $q_0 \in Q$ is the initial state;
- V_i and V_o are the sets of input and output variables, respectively. Input and output variables form disjoint sets (i.e. $V_i \cap V_o = \emptyset$);
- $I : V_o \rightarrow D$ is a function defining the default value of output variables;
- $T \in Q \times C(V) \times Q$ is the set of transitions, labeled by conditions on the variables of $V = V_i \cup V_o$
- $f : V_o \rightarrow (Q \rightarrow EqR(V))$ is a function used to define the labeling of states by total functions from V_o to the set $EqR(v_i \cup V_o)$ of expressions that constitute the right parts of the equations.

$EqR(V)$ has the following syntax: $e ::= c|x|op(e, \dots, e)|pre(y)$ where c stands for constants, x stands for a name in $V_i \cup V_o$, y stands for a name in V_o and op stands for all combinational operators. The condition in $C(V)$ are Boolean expressions of the same form, but without pre operators.

3 Case Study: Climate

3.1 Climate Description

In this section, we present our approach through a case study. We chose to study the *Climate* example that contains both pure control logic and data handling. In this example, we consider the simple case where the system responds to only four inputs of Boolean type. These inputs correspond to the buttons Climate, Left, Right and Ok (figure 3.a). As output, we can have: the climate mode, the temperature, the level of ventilation and the ventilation mode (figure 3.b). The types of the output values are as follows:

- **ClimateMode**: enum {Auto, Manual} initially Auto;
- **Temperature**: integer in [17, 27] initially 19;
- **VentilationLevel**: integer in [0, 100] initially 0;
- **VentilationMode**: enum{CAR, FACE, FEET, DEFROST, CIRCULATION} initially CAR.

Initially, the climate is in automatic (Auto) mode. The switch to Manual mode will be after the Adjust state that allows to confirm a choice using the OK button (figure 4).

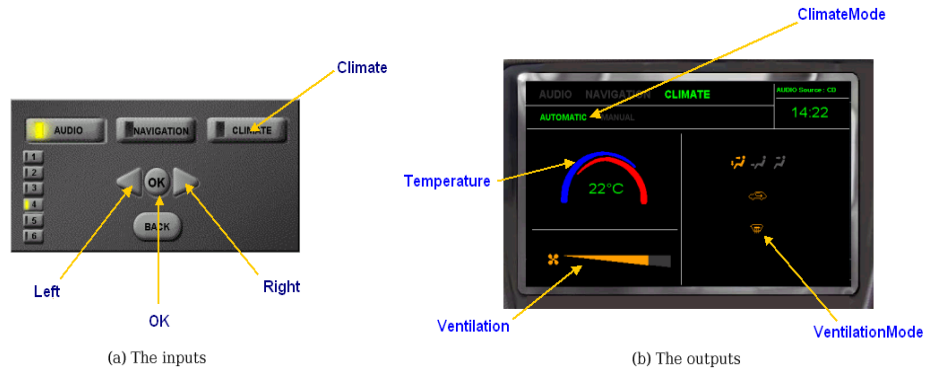


Fig. 3. Climate: inputs and outputs

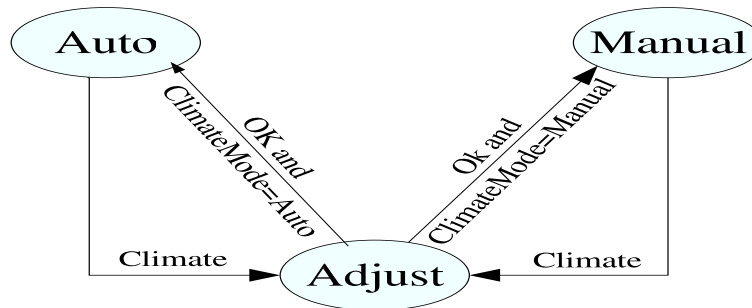


Fig. 4. The different states of Climate

The Auto State

- Set the temperature:
 - Left button decrease the temperature by 1 down to 17;
 - Right button increase the temperature by 1 up to 27.
- Climate button goes to state Adjust.

The Adjust State

- Navigate with Left/Right buttons through the ventilation mode and climate mode in the following order: CAR, FACE, FEET, DEFROST, CIRCULATION, Auto, Manual.
- OK button select the activated state and leave the Adjust mode. It goes to the Auto state if `ClimateMode` is Auto, and to the Manual state otherwise.

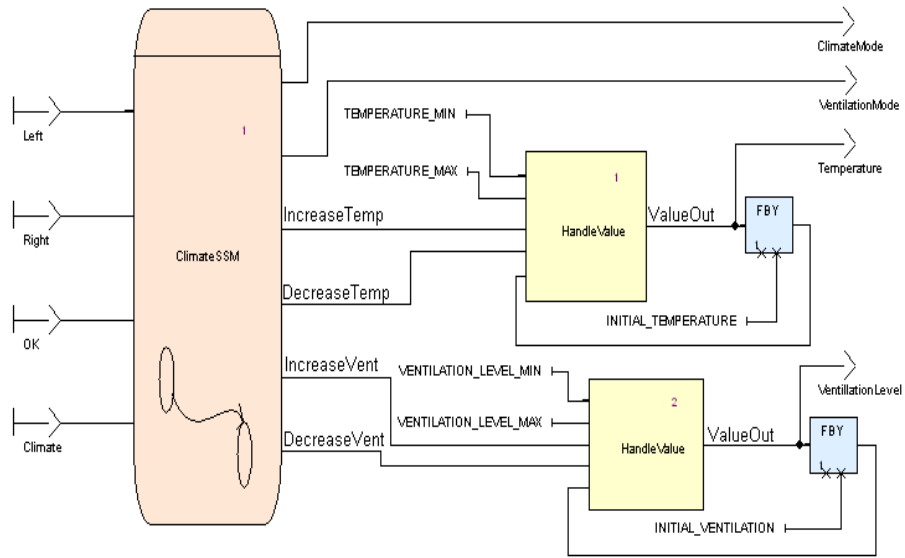


Fig. 5. Climate in Scade

The Manual State

- Set the ventilation level:
 - Left button decrease the ventilation level by 1 down to 0;
 - Right button increase the ventilation level by 1 up to 100.
- Climate button goes to state Adjust.

The specification of the Climate system contains control and calculation. The goal of our work consists in having a clear design of this specification, in which we separate control and data parts.

3.2 Conception of Climate System with Scade

The solution proposed by Esterel Technologies using Scade for the conception of Climate system is represented by figure 5.

This system possesses four inputs relative to buttons: Left, Right, Ok and Climate. As output, it provides four results: ClimateMode, VentilationMode, Temperature and VentilationLevel. Input values pass through a control part represented by the SSM ClimateSSM of figure 6. This SSM gives ventilation and climate modes as result. It also allows to activate the calculation part HandleValue by two different signals: Incr and Decr which correspond to the increase and decrease of the temperature or ventilation level. The activation of HandleValue depends on input values (buttons pushed) and the present state of the system. Each state in ClimateSSM represents a macro-state which specify the behavior of the global state.

The operator FBY (followed by) which appears on figure 5 is a predefined temporal operator in Scade. It makes it possible to preserve the value of a

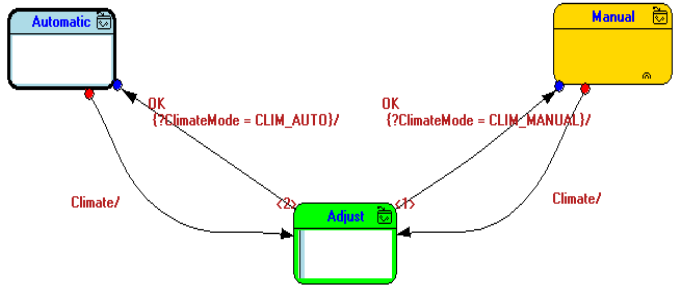


Fig. 6. ClimateSSM

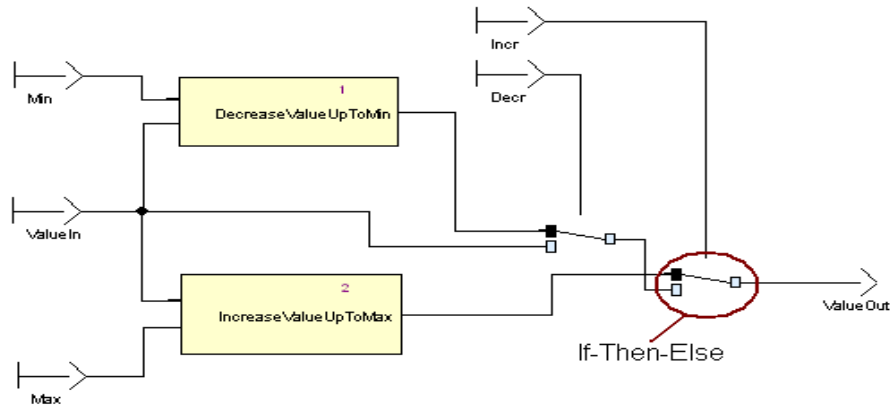


Fig. 7. HandleValue in Scade

given expression on several cycles. In Scade, $FBY(E, n, Init)$ is equivalent to $Init \rightarrow pre(Init \rightarrow pre(\dots \rightarrow pre(E)))$ in Lustre, where E is an expression which defines the sequence (e_1, e_2, \dots, e_n) and n is a static expression which value is strictly positive.

In the `Climate` example, `FBY` allows to keep the preceding value of `Temperature` or `VentilationLevel` which will be transmitted to `HandleValue` operator. Initially, `FBY` transmits the initial value of the temperature (`INITIAL_TEMPERATURE`) or that of the ventilation level (`INITIAL_VENTILATION`).

By descending to a lower level of the hierarchy, the conception model that corresponds to the operator `HandleValue` is indicated by figure 7. `HandleValue` allows to increase or decrease a given value depending on the values of signals `Incr` and `Decr`².

In Scade, the correspondence between various levels of the hierarchy does not use a naming mechanism but rather the link between inputs and outputs. For example, inputs values: `TEMPERATURE_MIN`, `IncreaseTemp_MAX`, `TEMPERATURE`

² `Incr` and `Decr` can not be activated at the same time.

and `DecreaseTemp` of the `HandleValue` operator which appear on figure 5 correspond respectively to the values of `Min`, `Max`, `Inc` and `Dec` of the `HandleValue` operator appearing on figure 7.

In this model, we notice that the calculation part `HandleValue` contains a mixture of calculation (`DecreaseValueUpToMin` and `IncreaseValueUpToMax`) and control (`If_Then_Else`). This mixture can make difficult the comprehension of the system, as well as the use of already existing tools, dedicated exclusively to processing the calculation part or the control part. Thus, as shown in the figures 7, `HandleValue` is composed of two calculation parts: `DecreaseValueUpToMin` and `IncreaseValueUpToMax`. Independently of the values of `Inc` and `Dec`, the two parts are activated and the output value will be chosen depending on signal's values of `Inc` and `Dec`. This corresponds to the strict and compound nature of the conditional structure `If_Then_Else` in Lustre. In this case, the two branches of the conditional structure are always evaluated which can introduce side-effect problems.

The goal of our work consists in proposing a conceptual model that allows to have a clear separation between control and data parts. This will allow us, on the one hand, to avoid the use of the Lustre conditional structure and to have a best readability, and on the other hand, to facilitate the separated study of the different parts by using the most appropriate tools for each category, notably concerning the application of the different formal verification techniques.

4 Control/Data Flow Separation Using Scade

First, we have tried to apply the concept of separated Control/Data Flow by using Scade. To make this, we have studied the Climate example by separating control and data parts. The diagram corresponding to our approach is shown on figure 8.

In this example, we have divided the problem into three sub-problems that correspond to the different states of the system: Auto, Adjust and Manual. The activation of each state is made by the SSM `ControlClimate` depending on the input values of `Ok` and `Climate`.

In this approach, we can clearly distinguish inputs and outputs of the system, control parts, and data parts. Contrary to what its name indicates, the data part does not only designate an exclusive data processing. It can also contain a SSM followed by a data part, or only the control part. The lowest level in the hierarchy represents an homogeneous part that can exclusively contain the control or the elementary calculation.

The application of this approach in Scade raises some issues. For example, the value of `ClimateMode` can be modified by two different states: Auto and Adjust. However, in Scade it is impossible to link the same output to two different operators. In Scade, each data must have a unique definition at a given time, which makes the connection of the same output to several different operators impossible. This requires the introduction of the `If_Then_Else` operators, which complicate the model and break the control/data flow separation concept. To fill

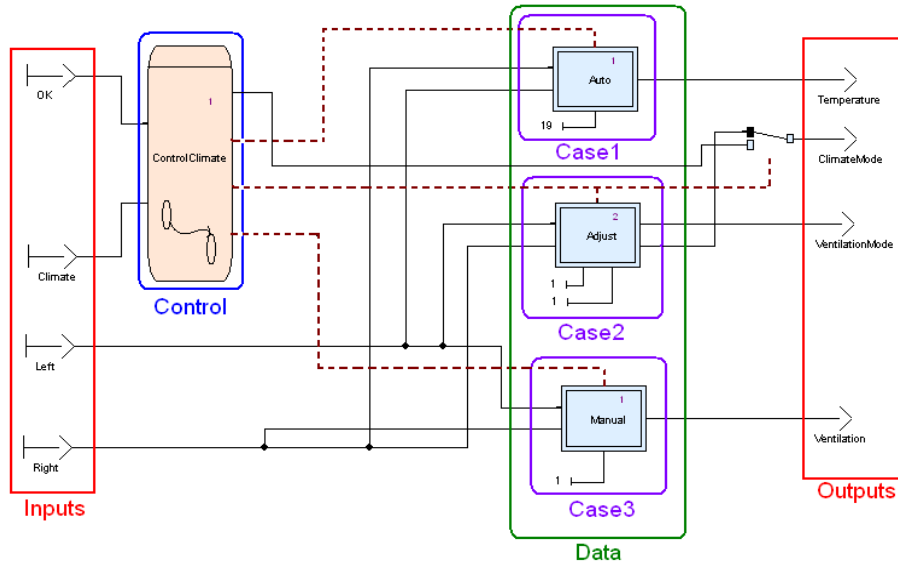
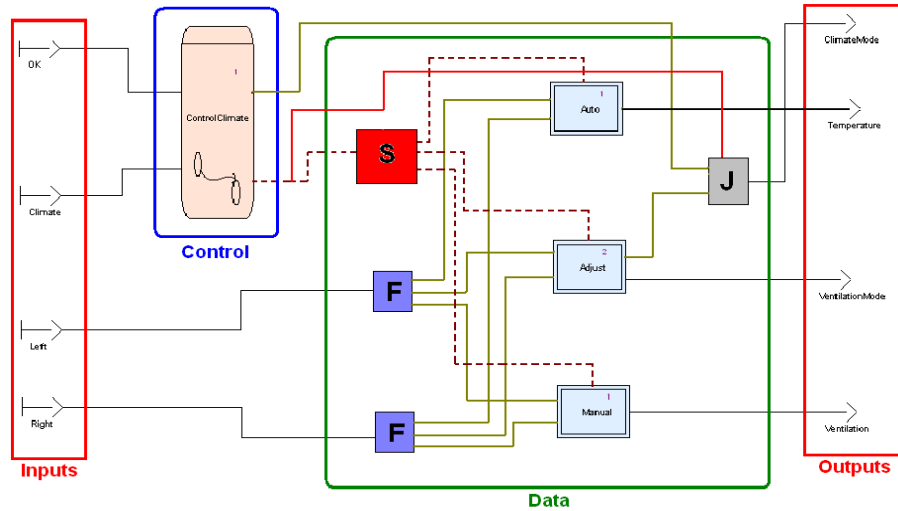


Fig. 8. Climate: trying the separation control/data flow with Scade



S: Selector, F: Fork, J: Join

Fig. 9. Control/data flow separation model using Scade and Fork/Join operators

this gap, we have proposed to add special operators that play the role of *Fork* and *Join* which allow the division of data between several operators. We have also added a *selector* operator that receives as input a value provided by a SSM, according to which it can choose the state to activate (figure 9).

The function of the *Fork* operator consists in diffusing the input value on all its output points, while the role of the *Join* operator consists in giving an

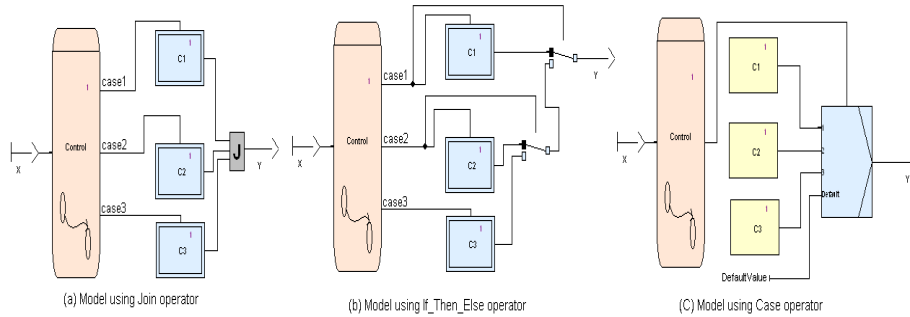


Fig. 10. Example of the Join operator and its equivalent in Scade

output value among those received as inputs and according to the value provided by the SSM.

Selector and Fork operators represent only an optimization of notations used in Scade because, in this tool, it is possible to connect the same value to several operator’s inputs. However, the Join operator replace the conditional structures *If_Then_Else* and *Switch_Case* used in Scade. In this context, one Join operator with n inputs can be used to replace a structure of $n - 1$ *If_Then_Else* operators or one *Switch_Case* operator with n inputs.

In the case of the *If_Then_Else* operators, it is obvious that the complexity of the model increases according to the number of inputs which makes difficult the comprehension of the model. Thus, if we use the *Switch_Case* operator, calculation blocks are not conditioned and then all inputs must be computed before the operator chooses the selected one. This behavior leads to difficult problems and can be very expensive regarding time and memory. Moreover, the default value used in *Switch_Case* operator does not have any interest because we suppose that one and only one component must be activated at a given time³. For these reasons, we prefer introducing a Join operator which allows an implicate use of conditional structures and facilitates the comprehension of the model. Figure 10 gives an example of Join operator and its equivalent in Scade.

The model suggested in figure 9 makes it possible to have a better design methodology based on the separation between control and calculation parts. This representation gives a possible solution to complete the Scade model of figure 8 and facilitates the use of separation control/data flow model with Scade.

5 Using Mode-Automata with Scade

As indicated in section 2.3, the mode-automata makes it possible to divide the specification of the system into several running modes. The switch between the modes is made accordingly to the activation conditions which appear on the tran-

³ This concept enable us to avoid the introduction of the default value relating to the *conduct* operators in Scade.

sitions. We notice that our approach of control/data flow separation presented in section 4 is similar to that of the mode-automata. The idea consists in introducing the concept of running modes into Scade models to facilitate the specification of the mainly regular systems and to give a more readable design methodology.

It is also easy to generate the Scade model relating to a given mode-automaton. The basic idea consists in representing each operating mode in the mode-automaton by a calculation part which will be controlled by a SSM equivalent to the studied automaton. This procedure can be summarized as follow:

1. Extracting inputs and outputs of the system.
2. Building the SSM equivalent to the automaton structure.
3. Modeling each operating part of mode-automaton (Lustre equations) by a calculation block in Scade.
4. Connecting the SSM and calculation parts using Selector, Fork and Join operators.
5. adding the Delay operators if necessary.

Figure 11 gives an example of a mode-automaton and its equivalent in control/data flow separation model with Scade. In this example, the modes *A* and *B* are respectively replaced by the components *AC* and *BC*. The switch between the various modes is done via the SSM *Control* which, according to the value of *X* and the state of the system, makes it possible to choose the component to be activated. In this context, the Lustre equations of the mode-automatons are replaced by calculation components in our design model, while the structure of

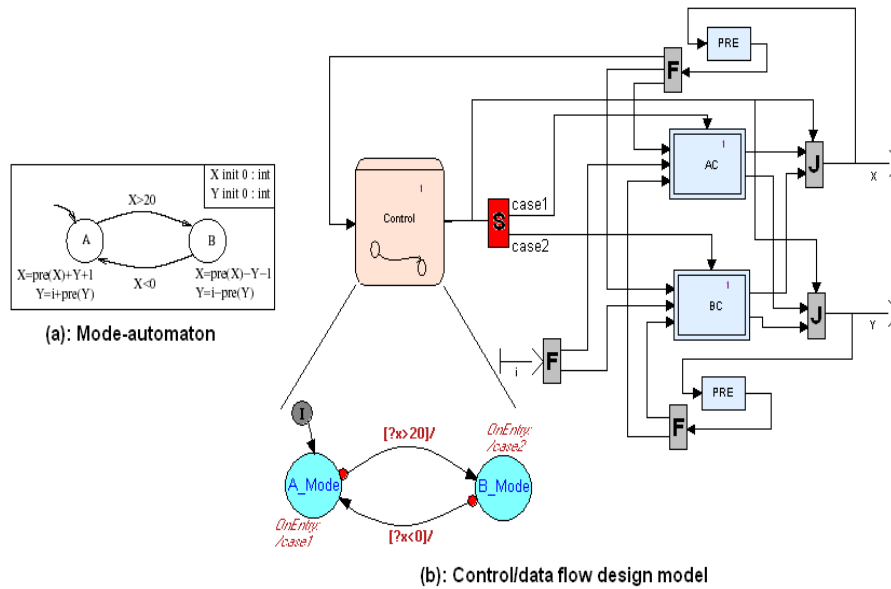


Fig. 11. Mode-automaton and its equivalent in control/data flow model

the automaton is replaced by a SSM responsible for the activation of the various parts of calculation.

6 New Formalism for Scade

In this section, we propose a new formalism for Scade based on the running modes concept. This formalism allows to have a clearer and easy to re-use model. For that, we introduce the concept of components with *same interface* to facilitate the introduction and deletion of components. In this context, the operators or states of execution in a given level of hierarchy must have the same inputs and outputs. Thus, if an operator does not modify an output value, its role only consists in giving its preceding value. A global view of the model that we wish to have for the Climate example is represented by figure 12.

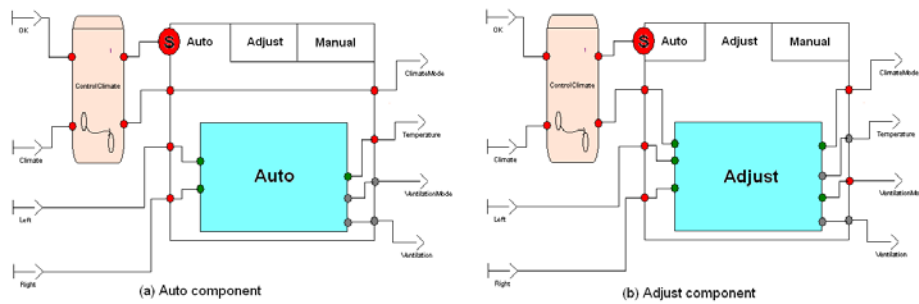


Fig. 12. Control/Data Flow Separation: use of the components with single assignment

In this model, the various situations of the system are represented in tabs. The activation of each case is done by the selector according to the value provided by the control part (SSM). In other words, the part controlled by a SSM can be seen as a black box with a set of inputs and outputs. In this box and according to the value provided by the selector we can connect various components having a single assignment and the same type and number of inputs and outputs.

In the Climate example, the controllable part is made of three tabs corresponding to the different states of the system: Auto, Adjust and Manual. These three components have the same number and type of inputs and outputs. Their role consists in providing output values according to input ones. This representation makes it possible to give a more readable model and facilitates the update and re-use of various existing components. It is also important to note that our model supports a hierarchical construction in all its design levels. This concept is similar to that used in Scade.

7 Conclusion and Future Work

In this paper, we have introduced a new formalism to specify complex synchronous reactive systems. The goal of our works consists in having a clear model separating control and data parts, which enables us to have a more readable and reusable specification and a better use of the various existing tools.

First, we have studied the possibility of separation between control and data parts using Scade. This study has shown that it is very difficult and even impossible to have a strict control/data flow separation with Scade, because each variable can only have one definition at the same time and it is then impossible to share the same variable between several operators. Thus, the ternary and strict nature of the conditional structure *If_Then_Else* in Lustre can induce several side-effect problems. We have also shown that the principle of the model that we wish to have is very similar to that of running modes. For this reason we have studied the mode-automata and the possibility of their integration in our design model.

Based on these results, we have proposed a design model mixing mode-automata and Scade. This model gives a good control/data flow separation model by allowing the use of running modes when the system changes its behavior. Its principle consists in adding some concepts in Scade allowing to take into account this kind of behavior.

A strict separation between control and data parts is interesting for the modeling of some systems where the distinction between the various running modes is obvious. However, there are several systems which are mainly regular and where the control part is not too present. In this case, the separation of the system in several parts controllable by a SSM becomes very complex, it can introduce problems of redundancy and unverifiable errors. In future work, to face this problem, we will propose to use the concept of running mode locally for a sub-part of the system which contains the control. We also wish to give an internal format and to provide transformation rules making possible the switch between our model and the internal format used in Scade. This would enable the use of different services existing in Scade, in particular for formal verification and code generation.

References

1. D. Harel and A. Pnueli: On the development of reactive systems. *Logics and Models of Concurrent Systems (NATO ASI Series)*. **13** (1985) 477–498
2. N. Halbwachs: *Synchronous programming of reactive systems*. Kluwer Academic Pub. (1993)
3. L. Zaffalon and P. Breguet: *Conception de Systèmes Réactifs*. *Revue Scientifique de l'EIVD*. (2001)
4. G. Berry and A. Benveniste: The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*. September. **79** (1991) 1270–1282

5. P. Caspi, D. Pilaud, N. Halbwachs and J. A. Plaice: Lustre, a declarative language for real time programming. Proceedings ACM Conference on Principles of Programming Languages (1987)
6. N. Halbwachs and P. Caspi and P. Raymond and D. Pilaud: The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE. September. **79** (1991) 1305–1320
7. Albert Benveniste, Patricia Bournai, Thierry Gautier and Paul Le Guernic: SIGNAL: a Data Flow Oriented Language for Signal Processing. INRIA, centre de Rennes IRISA. March. (1985)
8. Gerard Berry and Georges Gonthier: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Science of Computer Programming. **19** (1992) 87–152
9. Frédéric Boussinot and Robert De Simone: The Esterel Language. Another Look at Real-Time Programming. Proceedings of the IEEE. September. **79** (1991) 1293–1304
10. P. Le Guernic and T. Gautier and M. Le Borgne and C. Le Maire: Programming Real-Time applications with SIGNAL. Another Look at Real-Time Programming. Proceedings of the IEEE. September. **79** (1991) 1321–1336
11. F. Maraninchi and Y. Rémond: Argos: an Automaton-Based Synchronous Language. Computer Languages. Elsevier. **27** (2001) 61–92
12. Esterel Technologies: Efficient Development of Airborn Software with SCADÉ *SuiteTM*. (2003). <http://www.esterel-technologies.com/v3/?id=41490>
13. Esterel Technologies: SCADÉ Language Reference Manual. (2004)
14. Peter Amey: Correctness by Construction: better can also be cheaper. Journal of Defense Software Engineering. March. (2002)
15. Charle Andrés: Representation and Analysis of Reactive Behaviors: A Synchronous Approach. Computational Engineering in Systems Applications (CESA). IEEE-SMC. July. (1996) 19–29
16. Gérard Berry: The Foundations of Esterel. Proofs, Languages, and Interaction, Essays in Honour of Robin Milner. MIT Press. (2000)
17. F. Maraninchi and Y. Rémond: Mode-automata: About modes and states for reactive systems. European Symposium On Programming. LNCS 1381. March. (1998)
18. M. Jourdan and F. Lagnier and F. Maraninchi and P. Raymond: A multiparadigm language for reactive systems. IEEE International Conference on Computer Languages (ICCL). Toulouse, France. (1994)
19. Nicolas Pernet and Yves Sorel: Optimized Implementation of Distributed Real-Time Embedded Systems Mixing Control and Data Processing. International Conference: Computer Applications in Industry and Engineering. Las Vegas, USA. November. (2003)
20. Florence Maraninchi and Yann Rémond: Applying Formal Methods to Industrial Cases: The Language Approach (The Production-Cell and Mode-Automata). Proc. 5th International Workshop on Formal Methods for Industrial Critical Systems. Berlin. April. (2000)