Design of a multi-formalism application and distribution in a data-flow context: an example *

Loïc Besnard **, Patricia Bournai ***, Thierry Gautier †, Nicolas Halbwachs ‡, Simin Nadjm-Tehrani §, and Annie Ressouche ¶

Abstract. This paper describes a multi-formalism experiment design in the domain of real-time control systems. It uses several synchronous languages on a case study which is a realistic industrial example. Co-simulation is provided through the use of a common format, and automatic distributed code generation is experimented in the context of the graphical environment of the data-flow language SIGNAL.

1 Introduction

In industrial projects, multi-formalism development is very often an issue, especially when several teams, with possibly different cultures, collaborate, but also when different parts of the application require different formalisms, which can be better adapted for some of the encountered problems.

In the field of embedded control systems, in particular safety critical systems, another very important issue is to significantly reduce the risk of design errors (and also to shorten overall design times). This can be achieved through the use of the maximum degree of automation, especially with respect to verification, and also sequential or distributed code generation, entirely replacing the manual coding phase still employed in current industrial design flows [3]. A requirement for that is, at the front-end level, the use of specification tools that are based on a formal semantical model.

The *synchronous languages* [4, 13] have been precisely introduced for that purpose. All of them rely on a precise semantics, which allows not only automatic code generation through formal transformations, but also verification of

* This work is supported by the Esprit LTR project SYRF (EP 22703). Part of the results have been obtained from the Esprit R&D project SACRES (EP 20897).

** IRISA / CNRS, Campus de Beaulieu – 35042 Rennes Cedex – FRANCE – e-mail: Loic.Besnard@irisa.fr

*** IRISA / INRIA, Campus de Beaulieu – 35042 Rennes Cedex – FRANCE – e-mail: Patricia.Bournai@irisa.fr

† IRISA / INRIA, Campus de Beaulieu – 35042 Rennes Cedex – FRANCE – e-mail: Thierry.Gautier@irisa.fr

‡ VERIMAG / CNRS, Centre Equation, 2, avenue de Vignate – 38610 Gières Cedex – FRANCE – e-mail: Nicolas.Halbwachs@imag.fr

§ Dept. of Computer & Information Science, Linköping University – S-581 83 Linköping – SWEDEN – e-mail: simin@ida.liu.se

¶ INRIA, 2004, route des Lucioles – 06902 Sophia Antipolis Cedex – FRANCE – e-mail: Annie.Ressouche@sophia.inria.fr

properties on the programs (by model checking, for instance). Some of them are based on a declarative, data-flow style: this is the case for LUSTRE [14] and SIGNAL [17], that mainly differ in the more intricate notion of *clock* in the latter, which aims at a more direct handling of control-dominated applications. Other have an imperative style, like ESTEREL [8], but also the graphical formalisms of *mode automata* [18] or Statecharts [15].

In addition, joint efforts have been made to propose a common format, relying itself on a formal semantics, to represent programs expressed in a synchronous language. Then this family of synchronous languages becomes naturally a candidate to develop multi-formalism designs, including formal verification and automatic code generation.

In this paper, we present such an experiment of multi-formalism design, on a relatively small case study which is part of the control of a climatic chamber. In section 2, we first present the common format of synchronous languages, which is called DC+. Then, the case study and its programming using LUSTRE, ESTEREL and SIGNAL are described in section 3. Finally, the application of a distribution methodology provided by the SIGNAL graphical environment is shown in section 4 for that case study.

2 Multi-formalism through a common semantic and syntactic format: DC-DC+

When multi-formalism designs are considered, using together for example both state-based and data-flow specification styles, a common representation is in some way mandatory. One such common representation is the DC+ format. This format implements the paradigm of *synchronous programming* in its full generality [21]. The *Declarative Code* DC+ is a high-level format dedicated to both the representation of declarative or data-flow synchronous programs, and to the equational representation of imperative programs. It is a parallel, structured format, where programs are considered as a network of operators.

Although very close in its syntax to the *synchronized data-flow model* advocated by the SIGNAL language, it constitutes a model for the semantic integration of SIGNAL, LUSTRE, ESTEREL and Statecharts specifications.

The semantical basis of the DC+ format is that of *Symbolic Transition Systems* presented in [20]. This model includes in particular scheduling specifications, which are used to represent causality relations, schedulings, and communications [7].

2.1 A brief overview of DC+

DC+ allows to handle behavioural as well as structural or mixed descriptions.

The basic object in DC+ is the *flow* (also called *signal* in some contexts). A flow is a sequence of values synchronized with a clock: it is a typed object which holds a value at each instant of its clock. A program receives input flows and computes output flows, possibly using local flows which are not visible from the environment.

Flows can be related via definitions and dependencies.

Flow definitions. A DC+ equation `equ: x y at: w` defines the flow `x` to be equal to the expression `y`, when the boolean activation condition `w` is *true*. Such an equation relates both the values and the clocks of the flows `x`, `y` and `w`: in particular, `x`, `y` and `w` must have the same clock (when `w` is *false*, `x` keeps its preceding value, at the instants at which it is defined).

A special kind of definition, called a *memorization*, defines the value of its left-hand flow at the *next* instant of its clock: `memo: x y at: w`.

Flow expressions are built from basic terms and operators. A number of operators are available via predefined functions which are standard operators on predefined types, or *polychronous* operators to relate flows with different clocks.

A system of flow definitions can also be seen as a network of operators, or as a generalized circuit, the "wires" of which can carry values of arbitrary types.

Dependencies. Basically, the flow definitions must be evaluated according to their dependency order. These dependencies can be conditional.

A flow `x` depends on a flow `y` "at" a boolean condition `w` (noted $y \stackrel{w}{\to} x$) if, at each instant for which `w` is present and *true*, the event setting a value to `x` cannot precede the event setting a value to `y`.

The dependencies can be those induced by the definitions, or explicitly added ones.

Assertions. Through the *assertions*, DC+ offers a way of expressing properties stating that a boolean flow is invariantly *true*. This allows to express either known properties of the environment — for optimization purposes —, or desired properties of the program — to be proved by verification tools, or dynamically checked by the target code.

Nodes, clocks. DC+ allows complex systems of flow definitions to be structured into *nodes*. A node is an encapsulated system of definitions, assertions and dependencies, relating input flows and output flows of the node, possibly using its own local flows. In a node, the flows can be associated with their clocks through a *table of clocks*. Clocks are represented by boolean flows which are *true* if and only if the clock is present, otherwise they can be *false* or absent (a particular case is the representation of a clock by a *pure* flow, which is always *true* when it is present). A node can be instantiated (in another node) by a special kind of definition (node call), by providing it with actual input parameters. The definitions of a given node can be composed of flow equations, or of flow equations together with node calls, or exclusively of node calls (so giving a *structural* description of the node).

For a DC+ code to be used as input of a code generator, every output or local flow of a node has to be defined. However, when a DC+ code is used, for instance, to express and check properties, some of them can be undefined, or partially defined. To this purpose, two *tables of definitions* are distinguished in a node:

– the *definition table* contains explicit definitions;
– the definitions of the *assertion definition table* are considered as an implicit equational system defining a relation (with possible non determinism) on the concerned flows.

Pragmas. DC+ offers the notion of *pragma:* a pragma is an "executable comment", used to represent any sort of specific information which is not directly representable in the syntax. This information is considered as "documentation" by the parser. However, specific tools are able to handle some of these specific informations (which can have a particular syntax). Pragmas can be associated with any object.

2.2 Modularity in DC+

External and imported objects; packages. Some objects are defined within the format, while others are "external", i.e., possibly defined in another language, and used in a DC+ program. Types, constants, functions, procedures and nodes can be external. For example, an external node does not contain a definition table; however, it may contain an assertion definition table and a table of dependencies, allowing to specify properties (clock relations, dependency relations, etc.) on input and output flows of the node.

Besides this possibility, a DC+ program can consist of several *packages,* and a package may explicitly *import* objects from other packages.

2.3 The DC+ architecture

Different levels of DC+, or sub-formats, have been identified (see figure 1). The objective of the characterization of these different levels, and of the inter-level (or inter-format) transformations, is to adapt a given representation to the functions that can be applied to it.

DC+ itself is the most general level.

In the bDC+ (for "boolean DC+") sub-format, all the flows have an explicit clock, which is represented by a *boolean* flow. The clocks are organized in a hierarchy for which there exists a master clock, *tick* (*tick* is the single clock represented by a boolean flow which is always *true* when it is present). The bDC+ sub-format is the right entry point for tools based on the clock hierarchy, such as for example, code generators.

The STS (for "Symbolic Transition Systems") sub-format of bDC+, in which the hierarchy of clocks is flat (the present/absent status of any flow is defined at any instant by some Boolean which is present at *tick*), is used as an input for verification tools.

The DC sub-format is the single-clocked sub-format of DC+: all the flows are always present, no clock is used. The control is totally encoded via Booleans and dependencies.

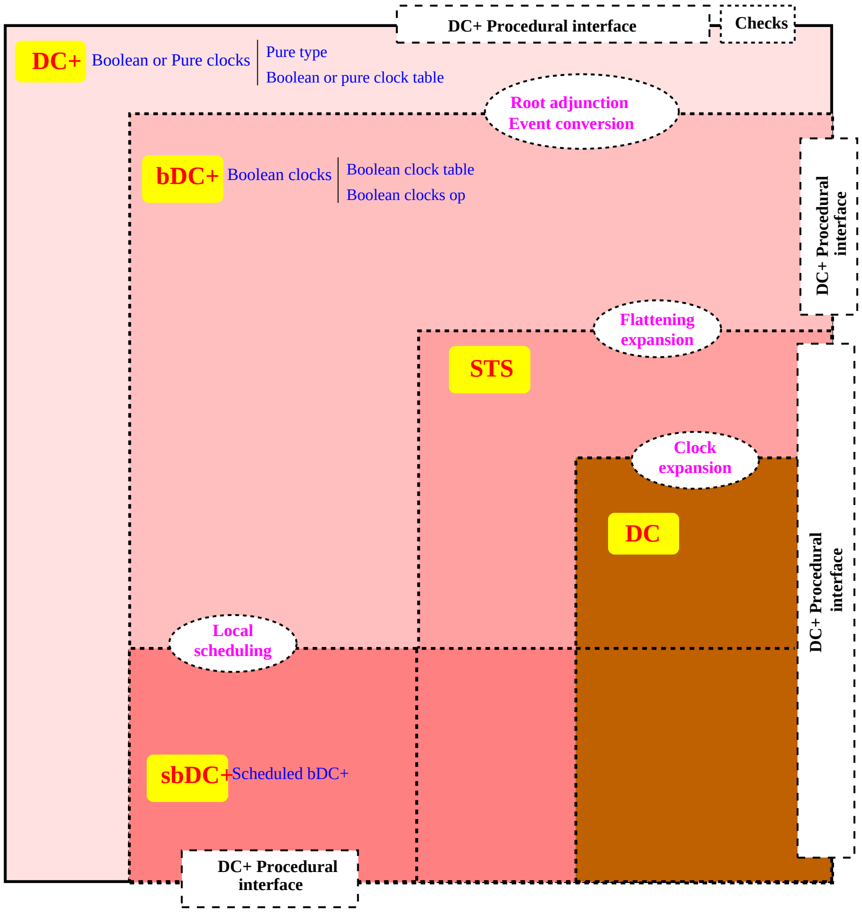

Fig. 1. *The DC+ architecture*

Inter-format transformations, DC+ → bDC+, bDC+ → STS, STS → DC, are defined between the different levels of DC+ [21].

2.4 Translations of languages to DC+

A translator from SIGNAL to DC+ is available in the SIGNAL compiler. Besides that, the SIGNAL compiler has been restructured so that the internal graph representation is now the same one for SIGNAL programs and for DC+ ones (however, windows and arrays are not yet implemented in this DC+ version). This gives access to the same functionalities for DC+ and for SIGNAL programs: clock calculus, graph calculus, inter-format transformations, interface calculus, code generation, access to proof systems, etc.

A first version of the LUSTRE-V5 front-end, called LUS2DC, is available. It performs static verifications (types, clocks, unique definition, absence of dependence loops) and translates a LUSTRE program into DC (this first version does not deal with arrays).

The ESTEREL translation to DC goes first through an intermediate proprietary format called SC. ESTEREL, which is a control-dominated imperative language, has an interpretation in *boolean equation system* for its control part. In the SC format, there are specific instructions so that other data operations are triggered exactly when the control demands. So, the ESTEREL translation to DC via SC requires a SCDC translator, which reintroduces data variables and values in the equation system.

There exists also translations to DC or DC+ for other formalisms: mode-automata for example have a translation to DC, and in the SACRES project, a translation from Statecharts to DC+ has been defined [1, 2]. In another context, the DC+ representation of languages of the IEC 1131 standard concerning industrial automatisms is under study.

3 A case study: climatic chamber

The case study we consider consists of a climatic chamber with a control system which regulates and monitors the flow and the temperature of air which is circulating in the chamber. This is a demo system which exhibits some of the problems typically appearing in system development for aircraft air control systems e.g. JAS Gripen.

3.1 Description

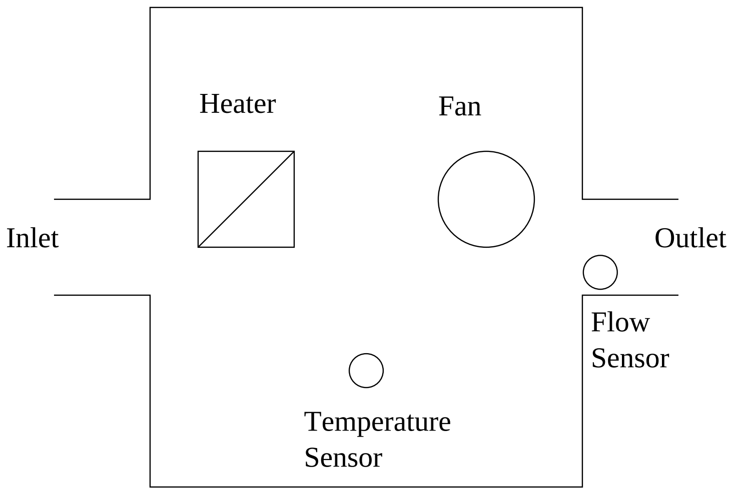

Fig. 2. *The hardware components of the climatic chamber system*

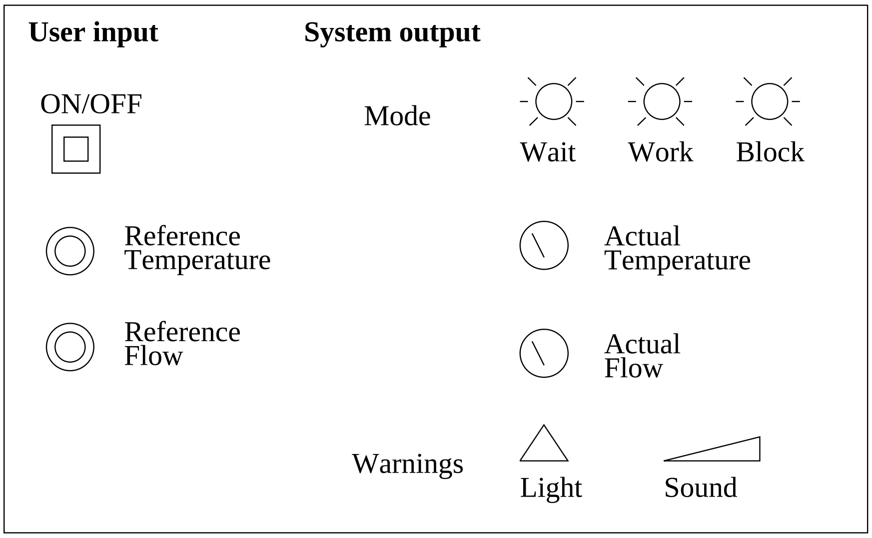

Fig. 3. *The external interface of the climatic chamber system*

Figures 2 and 3 present the component model and physical interfaces between the software controller and the chamber hardware.

The chamber is to be ventilated through the inlet and outlet and has a given volume. It has two sensors for measuring the internal air temperature and the air flow. The external interface primarily consists of an on-off button, two analog knobs for setting the values for required temperature and flow (reference values), as well as warning signals in terms of a light and a sound. It also includes lights for showing some of its internal modes of operation.

The controller has three modes while it is on. It has an initialising "wait" mode in which the heater and the fan are used to bring the chamber temperature and flow within a given scope. It also has an "active" mode in which more accurate regulation is achieved. This mode in itself consists of two modes, the "solution" mode in which the actual temperature and flow values are brought to levels close to the reference values, and the "work" mode in which the actual values are maintained in the required region (within Δ of the reference values). The shut-down mode, denoted as "block" mode, is devoted to abnormal situations. It is brought about when the earlier sound and light warnings have not led to changes in the reference values by the operator, or when the actual values fall outside the allowed scope despite manual intervention (for example due to unforeseen changes in unmodelled inputs, e.g. the incoming air temperature).

The overall requirement of the system is to keep the chamber's temperature and flow within given values, and to warn the operators if this goal is not achievable due to problems encountered.

The functional specification can be described as follows.

temperature control

- At start up, a requested temperature *optreq* is selected and the heater is turned on. It is possible to choose *optreq* in the interval $[treqmin, treqmax]$, where *treqmin* is greater than the (lowest) outside air temperature *envtemp*. Any other chosen values outside the above interval are automatically adjusted up (down) to the lower (upper) limit respectively.
- Let $T_{min} = optreq - 2\Delta_{temp}$ and $T_{max} = optreq + 2\Delta_{temp}$. Then the system is considered to be in the **wait** mode while the actual temperature (*chamtact*) is not within those limits.
- The system will be in the **sol** mode (read solution mode) from the time the temperature hits the region $T_{min} \leq chamtact \leq T_{max}$ until it leaves the mode. The system makes a transition to the **work** mode when $\mid chamtact - optreq \mid \leq \Delta_{temp}$ is fullfilled. The time taken for the system to enter and leave the **sol** mode is expected to be within a given fixed bound called *solution_time*.

ventilation control

- The ventilation or the mass flow rate is measured by the time it takes to exchange one volume of air.
- At start up, a requested flow time *optvent* is selected and the fan is turned on. This chosen time for the rate of air change should be in the interval $[tventmin, tventmax]$. Other chosen values are automatically adjusted upwards (downwards) to these limits respectively.
- Ventilation is regulated so that the air in the chamber is changed at least once every *optvent*.
- The observed rate of change *flowtime* is based on a measure of the air flow *chamflow* delivered by the sensor.

monitoring

- The continuously measured values of *chamtact* and *chamflow* are to be displayed on the control panel.
- Three lamps indicate being in the system modes **wait**, **work** and **block** respectively.
- A warning by light and sound shall be activated whenever $\mid chamtact - optreq \mid > \Delta_{temp}$ after being in the **sol** mode for a duration of *solution_time*.
- The warning light is activated if $flowtime > X\ optvent$ for some fixed ratio X.
- When in **work** mode, if $\mid chamtact - optreq \mid > 2\Delta_{temp}$, or the derivative of the temperature, based on *chamtact* exceeds a maximum value *tgrad*, then the mode shall change to the **block** mode.

- When in **block** mode, the heater is immediately turned off, the warning light and the light indicating the **block** mode are activated, and the fan is active for another *tblock* seconds before it is turned off. It shall not be possible to influence the system by changing *optreq* or *optvent*. The system must be turned off before restart.
- If *optreq* is changed such that $optreq > chamtact + \Delta_{temp}$ the system makes a transition to the **wait** mode, and the heater is turned on.
- If *optreq* is changed such that $optreq < chamtact - \Delta_{temp}$, the system is changed to **wait** mode, the heater is turned off and the fan will continue to work.

goals of verification

The functional description above has a prescriptive nature. It describes how a controller should be implemented, giving some details about what should happen in each mode. However, both to help programming and to focus the formal verification work, we had to deduce the overall goals of the control system: those requirements which are to be enforced by the suggested design.

We have identified the following global requirements.

- Keeping the reference values constant,
  - the **work** light will be lit within a time bound from the start of the system, and
  - the system will be stable in the **work** mode.
- Chamber temperature never exceeds a given (hazardous) limit T_H.
- Whenever the reference values are (re)set, the system will (re)stabilize within a time bound or warnings are issued.

Note that these are not properties of the controller on its own. Rather they arise as an interaction with the physical environment not modelled here. However, to prove these it is necessary to prove certain subsidiary requirements on the controller, for example:

- the system is deterministic (can be at most in one mode at a time for any combination of variable values),
- $treqmax + 2\Delta < T_H$,
- when the system is in **wait**, **sol**, or **work** mode, $chambtact < T_H$.

3.2 Multi-formalism programming

The application is split in three sub-parts, plus a *main* calling them:

1. One component computes the current mode of the system according to various temporal conditions. Each time a mode is reached, a lamp (*Panwait*, *Panoper* or *Panblock*) is highlighted on the control panel. There is also a *Panaudio* warning to be emitted when the temperature overpasses its allowed boundaries. This component (*modeselect*) will be detailed below.

2. A second component regulates the actual temperature of the chamber (*Conheat* signal) inside the prescribed interval.
3. A third component similarly regulates the ventilation of the chamber as required from the sensors of the airflow (*Confan* signal).

The main module will be presented below too.

The three parts above can be viewed as finite state machines and their behavourial steps form the control part of the application.

Since multi-formalism design was the main purpose of the study, it has been decided to use the three *synchronous languages* ESTEREL, LUSTRE and SIGNAL, to describe the application. The control part has been encoded using ESTEREL, while the data-flow computation of boundary conditions from sensor values, which is purely combinational, was encoded in LUSTRE. In addition, the *main* module was described using the SIGNAL graphical environment to take advantage of this environment for code distribution.

Note that other design choices could have been made. In particular, although they are often naively associated, an imperative style is not always the best way to describe control parts. Especially when it is combined with the notion of *clock,* the data-flow style is a valuable alternative.

To illustrate the combination of formalisms, we consider the part 1. above, the purpose of which is to compute the current mode of the system. The desired mode automaton is shown in figure 4. The following data are needed to update flow and temperature: constants *deltatemp* (temperature variation allowed), *solution_time* (maximal time to stay in **sol** mode), and *tgrad* (limit for temperature derivative); and variable parameters *chamtact* (actual chamber temperature), *pantreq* (requested chamber temperature—computed from operator request and temperature upper and lower bounds), and *con_onoff* (on-off switch). The conditions useful to determine the next mode use the following boolean conditions:

- $In2DeltaTemp = |chamtact - pantreq| \leq 2 \star deltatemp$
- $OutDeltaTemp = |chamtact - pantreq| > deltatemp$
- $TempLess5Tgrad = true$ while the two last values of *chamtact* differ from less than $5 \star tgrad$
- $newpanreq = true$ when a new *pantreq* occurs
- $ten\_solution\_time\_con\_onoff = true$ each $10 \star solution\_time$ of *con_onoff*

The LUSTRE and ESTEREL code corresponding to this part are given respectively in annex 6.1 and 6.2. The ESTEREL *module,* which describes the finite state machine, is called as an *imported node* from the LUSTRE *node.*

The *main* module of the application is described as a SIGNAL program which mainly computes *pantreq* and *pantvent* (respectively the requested chamber temperature—computed from operator request and temperature upper and lower bounds—and the requested chamber ventilation—also computed from operator request and upper and lower bounds), and calls the three sub-parts described above. Each sub-part is defined just like the *modeselect* component, as a LUSTRE node calling itself an ESTEREL module. In the SIGNAL program, they are represented as *external processes.* The SIGNAL program is given in annex 6.3. A graphical view of this program is shown in figure 5.

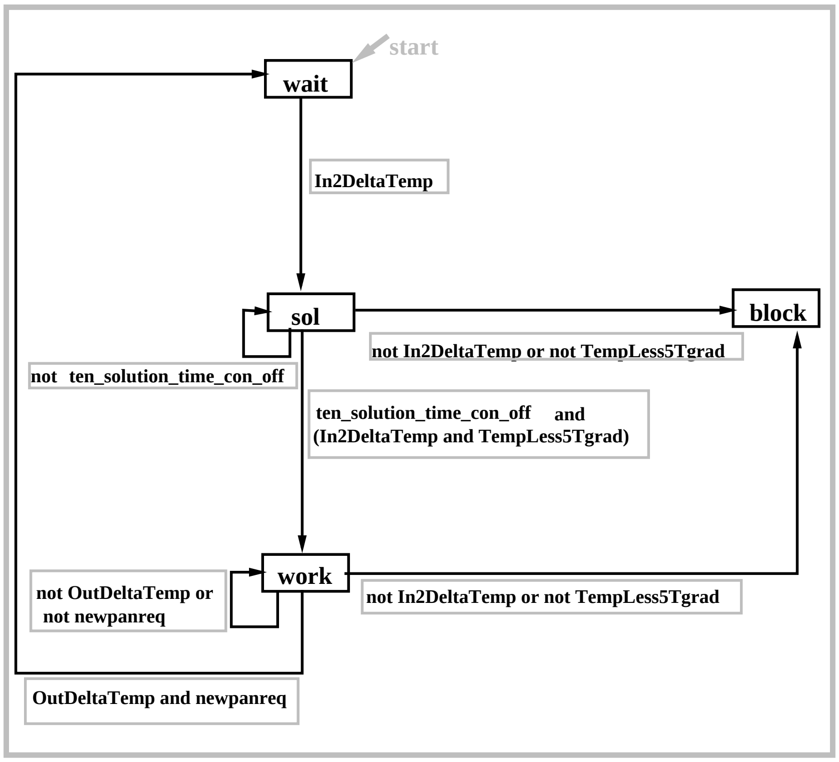

Fig. 4. *Climatic chamber modes computation*

3.3 Compilation and simulation

The integration of the different parts of the application is obtained through the common DC+ format. First, each one of the Lustre nodes and Esterel modules is translated in a DC node, thanks to the corresponding translators. The Signal part could be translated as well in DC+, but this is in fact not strictly necessary since there exists a common Signal/DC+ compiler, which allows to manipulate both formalisms.

Then, the DC codes from Lustre and Esterel corresponding to one of the three sub-parts presented above are linked together in order to obtain one single DC code for each sub-part. Each one of the three resulting DC codes is then separately compiled with the DC+ compiler, with two results:

1. an abstraction of the corresponding DC node giving the existing clock relations and dependencies between the inputs and outputs of the node (in the case of a DC node, clock relations are simply equality, but the same algorithms apply to more general DC+ nodes);
2. a C code corresponding to the DC node.

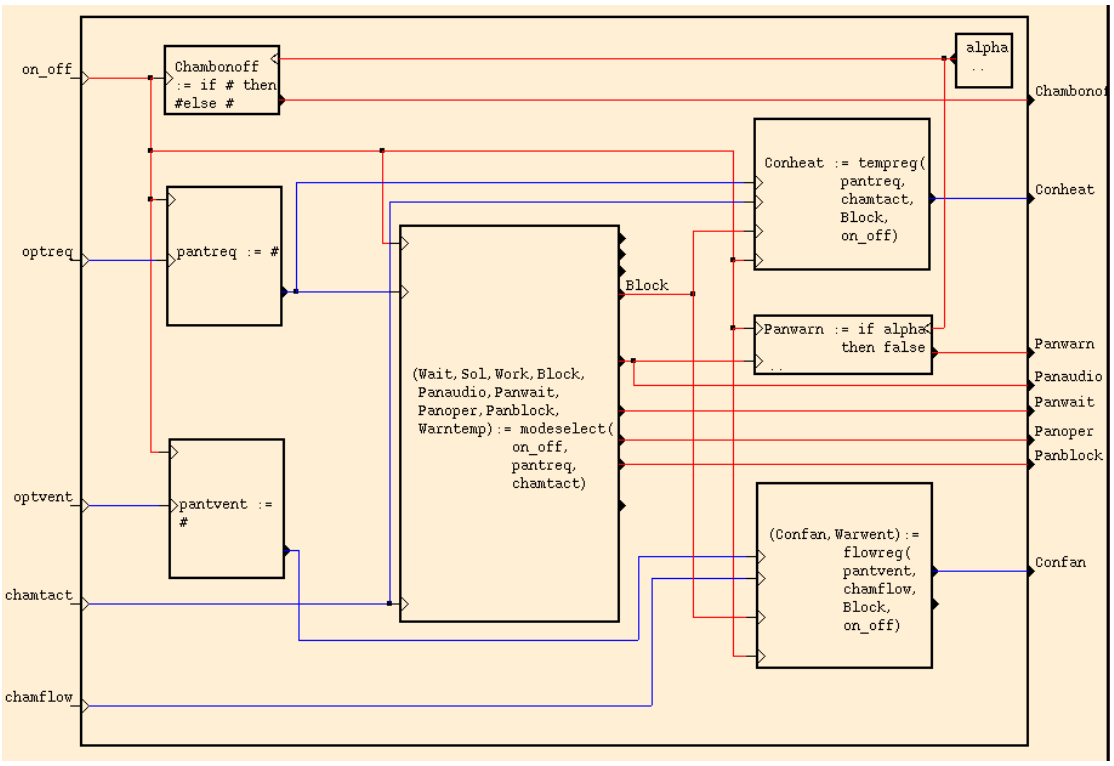

Fig. 5. SIGNAL *graphical view of the main program*

An example of clock and dependency relations calculated in this way is given in annex 6.4 for the *modeselect* component (note that some internal flows, used for example as boolean conditions at which the dependencies are valid, can be made visible at the interface of the abstracted node).

The abstractions of the separately compiled modules are included in the main SIGNAL program so as to get a "complete" view of the application. This program can then be compiled, cycles can be detected, C code can be generated.

Then, the C codes corresponding to the separately compiled modules are linked together and an executable is obtained. An example of simulation is given in figure 6 (the simulation environment has been described in SIGNAL).

4 Applying the distribution methodology provided by the SIGNAL/DC+ environment

Although relatively small, the climatic chamber case study was used to illustrate different steps of the general distribution methodology provided by the SIGNAL/DC+ graphical environment.

The purpose is to obtain automatic distributed code generation from:

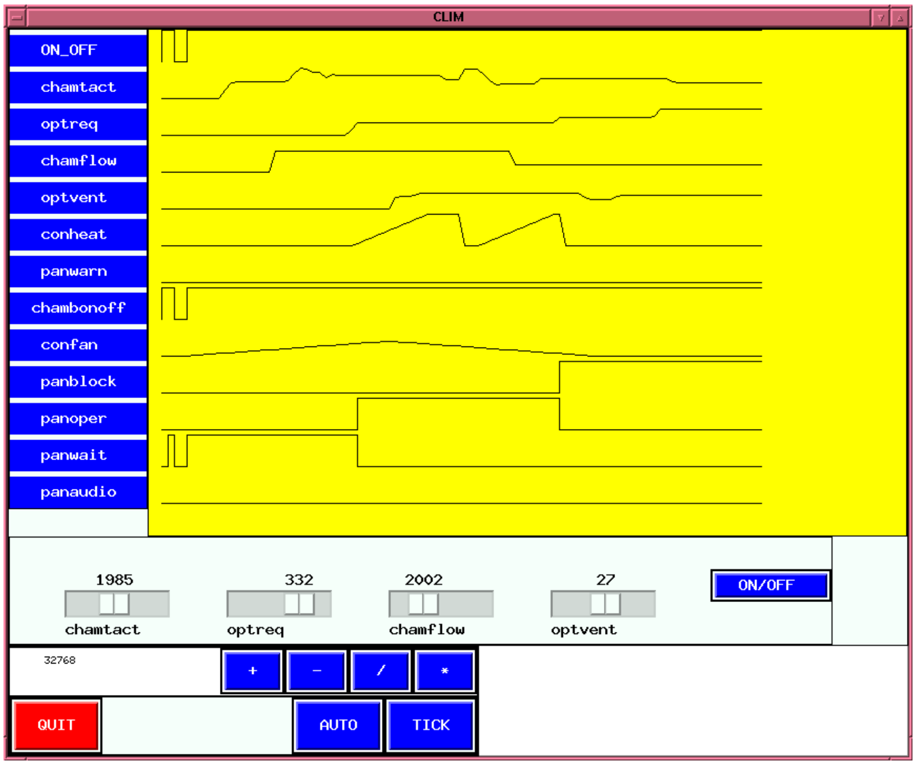

Fig. 6. *Simulation panel*

- the software architecture of the application,
- a representation of the distributed target architecture,
- a manual mapping of the software modules onto the hardware components.

It is not our purpose to detail here the formal principles of the method: this has been done elsewhere [3, 5, 7, 12]. Instead, we want to illustrate the method on our multi-formalism example.

4.1 Virtual mapping

Here, the software architecture of the application is that of figure 5, where external processes have been replaced by their abstraction. Then we consider a target architecture roughly composed of, say, two processors, represented by two boxes in the graphical editor. Thanks to the graphical environment, each one of the software components is mapped on a processor by simple click-and-point. Note that the idempotence property of the composition allows to duplicate some of the components.

The required connections are automatically established. This is shown in figure 7, where the two main boxes represent the two processors, on which software components have been mapped.

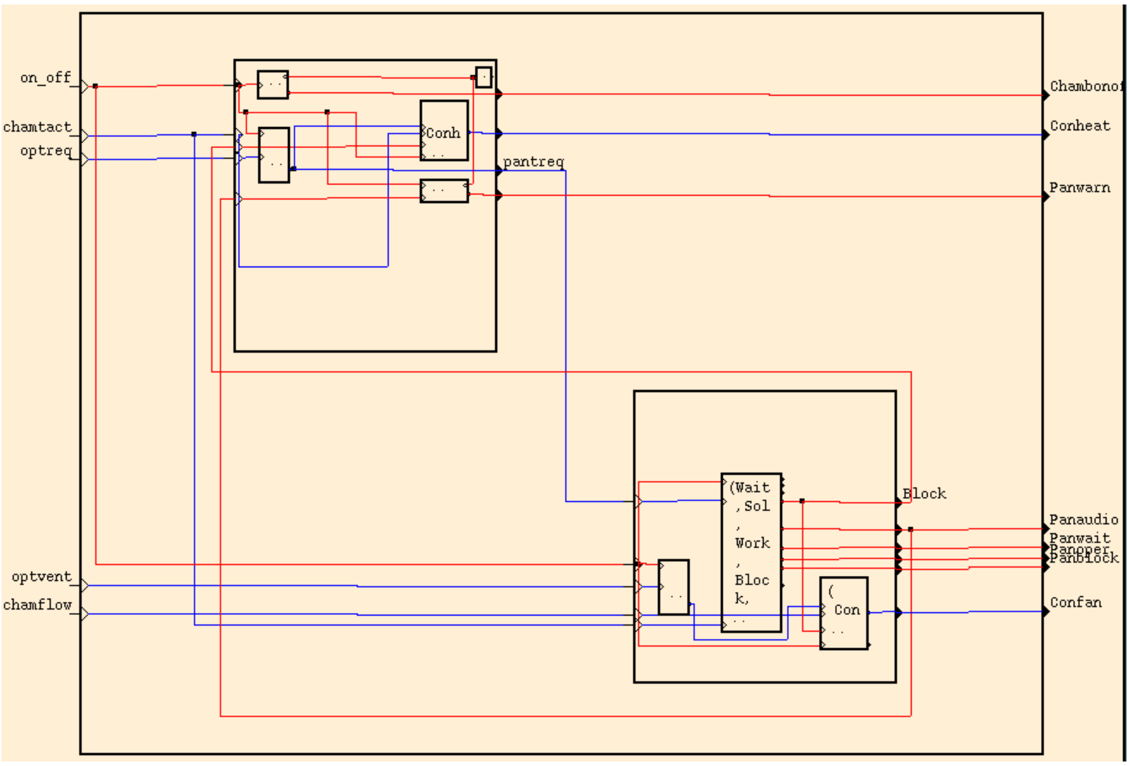

Fig. 7. *The result of the mapping*

4.2 Traceable compilation

After the virtual mapping, the next step is a global compilation that will preserve the new structure of the application, so that each *sub-graph* (corresponding here to one processor) will be executed on one location. In the general case, it will be necessary to add some boolean clock definitions to be communicated between the different locations to ensure that the semantics of synchronous communication will be preserved even though an asynchronous communication medium is used. This refers to properties of *endocrony* and *isochrony* [7]. In particular, each processor will have a local tree of boolean clocks (it can be represented by a bDC+ *endochronous* program).

This is obtained automatically through the global compiling by using some heuristics (to complete the clock hierarchy and to transmit the Booleans used for communication).

The result of this global compiling is made available to the user, by replacing the contents of each box corresponding to one location by the compilation result viewed as a new SIGNAL or DC+ program. In each box, the equations representing clock relations are separated from the definitions of signals; these definitions themselves are partitioned between boolean ones and non boolean ones; invariant equations are separated from the state evolution ones, etc.

4.3 Generating schedulings for separate modules

The actual implementation of a module has to conform to the dependencies resulting from the causality analysis. In general, further sequentialization has still to be performed to generate code. Of course, this additional sequentialization can be the source of potential, otherwise unjustified, deadlock when the considered module is reused in the form of object code in some environment. Thus, the purpose is to be able to structure the code into pieces of sequential code and a scheduler, aiming at guaranteeing separate compilation and reuse. In addition, the cost of dynamic execution leads to reduce as much as possible dynamism. To do that, the nodes of a sub-graph will be gathered in such a way that they can be considered as atomic. In this case, the scheduler only has to manage sets of nodes instead of nodes themselves.

The first level of atomicity which is automatically provided is obtained by building the classes of nodes transitively depending on the same subsets of input flows. For the sake of scheduling, such a class can be seen as a procedure call: it can be executed as an atomic action depending upon its set of inputs. The abstraction of a class is thus a *black box* abstraction. Then to ensure a correct read-write sequencing of the classes of a module from the point of view of the environment of that module, the internal communications have also to be abstracted as dependency relations. We obtain what we call a *grey box abstraction* of a module (in the sense that we know more than just its interface: we have to know the internal classes—black box abstractions—and their dependencies).

On the example, we have to calculate the grey box abstractions of the two modules corresponding to the two processors.

4.4 Adding communications

Communication features, such as a shared memory, a fifo, etc., can be described as some process abstraction. In figure 8, communications have been added between the two processors. Other ones should be added also between the processors and their environment. From this complete representation of the application, including its virtual distribution on a targeted architecture, it is possible to make a global compilation to be able to simulate the application on the considered architecture. It is also possible to study the cost of the implementation, to evaluate time consumption, etc. [6, 16].

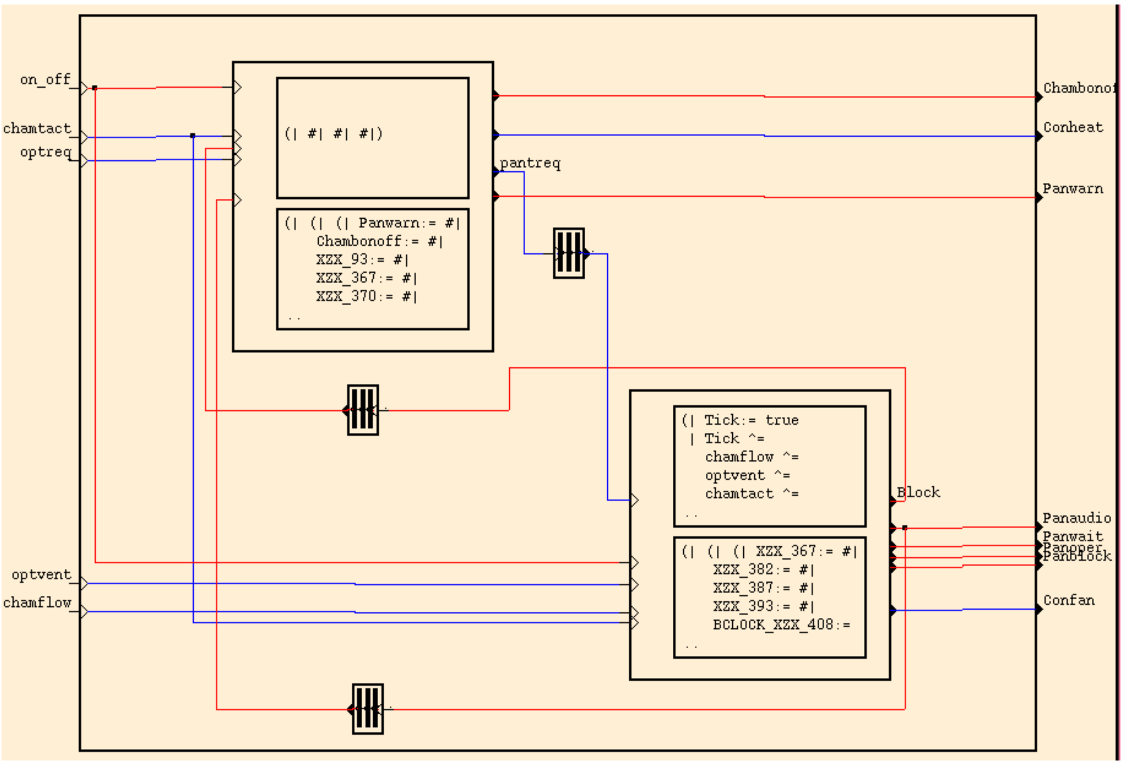

Fig. 8. *Adding communications*

On the other hand, each module corresponding to one processor can be compiled separately for embedded code generation. In this case, only the "send or receive" part of the communication channels is considered, and communications are generated by making calls to the corresponding OS primitives, provided by the user (for example).

5 Conclusion

Although relatively small, the experiment we have presented can be considered in several aspects as representative of multi-formalism designs using semantically well-founded languages, such as synchronous languages. Co-simulation of these languages has been experimented through the use of a common format and tools developed around this format. Moreover, a distribution methodology and tools for that are provided in the framework of the graphical SIGNAL environment. The approach can be compared with the implementation of synchronous programs in POLIS [9], and with other distributed implementations [10, 11].

Other experiments, not described here, have been applied on the same case study: verification of some safety properties, for example (see section 3.1). On the other hand, a mathematical model of the physical environment has been given, so that we can handle some *hybrid system,* composed of a combination of discrete and continuous specifications [19].

Finally, the same sort of experiments have been made, using LUSTRE and mode automata on one hand, and SIGNAL and Statecharts on the other hand.

References

1. J.-R. Beauvais, T. Gautier, P. Le Guernic, R. Houdebine, E. Rutten, "A translation of STATECHARTS into SIGNAL", in *Proceedings of the International Conference on Application of Concurrency to System Design (CSD'98)*, IEEE Publ., Aizu-Wakamatsu, Japan, March 23–26, 1998, 52–62.
2. J.-R. Beauvais, R. Houdebine, P. Le Guernic, E. Rutten, T. Gautier, *A Translation of Statecharts and Activitycharts into Signal Equations,* Inria Research Report RR-3397, April 1998, `http://www.inria.fr/RRRT/RR-3397.html`.
3. A. Benveniste (& contributors), "Safety Critical Embedded Systems Design: the SACRES approach", in *Formal Techniques in Real-Time and Fault Tolerant systems, FTRTFT'98 school,* Lyngby, Denmark, September 1998, `http://www.irisa.fr/ep-atr/Publis/Abstract/benveniste98c.htm`.
4. A. Benveniste, G. Berry, "Real-Time systems design and programming", *Another look at real-time programming*, special section of *Proceedings of the IEEE,* vol. 79, n^o 9, September 1991, 1270–1282.
5. A. Benveniste, T. Gautier, P. Le Guernic, E. Rutten, "Distributed code generation of dataflow synchronous programs: the SACRES approach", in *ISLIP 98, Proceedings of the Eleventh Annual International Symposium on Languages for Intensional Programming,* B. Wadge, editor, Sun Microsystems, Palo Alto, California, USA, May 7–9, 1998, 19–47, `http://www.irisa.fr/ep-atr/Publis/Abstract/benveniste98e.htm`.
6. A. Benveniste, C. Jard, S. Gaubert, "Algebraic techniques for timed systems", in *CONCUR'98 Concurrency Theory, 9th International Conference,* Nice, France, September 1998, D. Sangiorgi, R. de Simone, editors, Lecture Notes in Computer Science 1466, Springer.
7. A. Benveniste, P. Le Guernic, P. Aubry, "Compositionality in Dataflow Synchronous Languages: Specification & Code Generation", in *Proceedings of 1997 Malente Workshop on Compositionality,* W.P. de Roever, H. Langmaack, editors, Lecture Notes in Computer Science, Springer, see also a revised version co-authored with B. Caillaud, Feb. 1999, to appear in *Information & Computation*, `ftp://ftp.irisa.fr/local/signal/publis/articles/IC-submitted:sem_distr.ps.gz`.
8. G. Berry, "The Foundations of Esterel", in *Proof, Language and Interaction: Essays in Honour of Robin Milner,* G. Plotkin, C. Stirling, M. Tofte, editors, MIT Press, 1998, `ftp://ftp-sop.inria.fr/meije/esterel/papers/foundations.ps`.
9. G. Berry, E. Sentovich, "An Implementation of Constructive Synchronous Programs in POLIS", November 1998, `ftp://ftp-sop.inria.fr/meije/esterel/papers/esterel-polis.ps`.
10. B. Caillaud, P. Caspi, A. Girault, C. Jard, "Distributing Automata for Asynchronous Networks of Processors", in *European Journal of Automation (RAIRO-APII-JESA),* 31(3):503–524, 1997.

11. P. Caspi, A. Girault, D. Pilaud, "Distributing reactive systems", in *Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94,* Las Vegas, USA, October 1994, ISCA.
12. T. Gautier, P. Le Guernic, "Code generation in the SACRES project", in *Towards System Safety, Proceedings of the Seventh Safety-critical Systems Symposium,* Huntingdon, UK, 1999, F. Redmill, T. Anderson, editors, Springer, 127–149, `http://www.irisa.fr/ep-atr/Publis/Abstract/gautier99a.htm`.
13. N. Halbwachs, *Synchronous programming of reactive systems,* Kluwer Academic Pub., 1993.
14. N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, "The synchronous dataflow programming language Lustre", *Proceedings of the IEEE,* vol. 79, n^o 9, September 1991, 1305–1320.
15. D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming,* 8:231–274, 1987.
16. A.A. Kountouris, P. Le Guernic, "Profiling of SIGNAL Programs and its application in the timing evaluation of design implementations", in *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems,* HP Labs, Bristol, UK, IEE, February 1996, `http://www.irisa.fr/ep-atr/Publis/Abstract/kountouris96a.htm`.
17. P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, "Programming real-time applications with SIGNAL", *Another look at real-time programming*, special section of *Proceedings of the IEEE,* vol. 79, n^o 9, September 1991, 1321–1336.
18. F. Maraninchi, Y. Rémond, "Mode-automata: About modes and states for reactive systems", in *European Symposium On Programming,* Lisbon, Portugal, March 1998, Lecture Notes in Computer Science 1381, Springer.
19. S. Nadjm-Tehrani, "Integration of Analog and Discrete Synchronous Design", in *Proceedings of the second international workshop on Hybrid Systems: Computation and Control,* Nijmegen, Netherlands, March 1999, Lecture Notes in Computer Science 1569, Springer.
20. A. Pnueli, N. Shankar, E. Singerman, "Fair Synchronous Transition Systems and their Liveness Proofs", in *Proceedings of Formal Techniques in Real-Time and Fault Tolerant systems, FTRTFT'98,* Lecture Notes in Computer Science, Springer Verlag, September 1998.
21. SACRES consortium, *The Declarative Code DC+, Version 1.4,* Esprit project EP 20897: Sacres, November 1997, `http://www.irisa.fr/ep-atr/Publis/Abstract/sacres-dc.html`.

6 Annex

6.1 LUSTRE *modeselect* node

```
---------------- imported node declaration
node compute_mode
   (
   _con_onoff__bool : bool;
   _ten_solution_time_con_onoff__bool : bool;
   _In2deltaTemp__bool : bool;
   _OutdeltaTemp__bool : bool;
   _TempdLess5Tgrad__bool : bool;
```

```
   _newpantreq__bool : bool
   )
returns
   (
   _Wait__bool : bool;
   _Sol__bool : bool;
   _Work__bool : bool;
   _Block__bool : bool;
   _Panaudio__bool : bool;
   _Panwait__bool : bool;
   _Panoper__bool : bool;
   _Panblock__bool : bool;
   _Warntemp__bool : bool
   ); imported;
---------------------------------------------
const
  deltatemp = 50 ;
  tgrad = 50 ;         -- gradient in 1/10oC per 1/2 sec
  solution_time = 600; -- time in sec


node modeselect (con_onoff: bool; pantreq: int; chamtact: int)
returns (Wait, Sol, Work, Block,
         Panaudio, Panwait, Panoper, Panblock, Warntemp: bool);

     var temp0, temp1, temp2, temp3, temp4, temp5: int;
         newpantreq : bool;
         In2deltaTemp, OutdeltaTemp, TempLess5Tgrad: bool;
let
     newpantreq = false -> if (pre(pantreq) = pantreq) then false
                           else true;
    temp0 = chamtact->pre(temp1);
    temp1 = chamtact->pre(temp2);
    temp2 = chamtact->pre(temp3);
    temp3 = chamtact->pre(temp4);
    temp4 = chamtact->pre(temp5);
    temp5 = chamtact;

     TempLess5Tgrad = true-> if (con_onoff) then
                                if (temp5-temp0 < 5*tgrad) then true
                                else false
                             else pre(TempLess5Tgrad);
     In2deltaTemp = true ->
                    if (con_onoff) then
                       if ((pantreq-chamtact <= 2*deltatemp) and
                           (chamtact-pantreq <= 2*deltatemp)) then true
                       else false
                    else pre(In2deltaTemp);

     OutdeltaTemp = false ->
```

```
                    if (con_onoff) then
                       if ((pantreq-chamtact > deltatemp) or
                           (pantreq-chamtact < deltatemp)) then true
                       else false
                    else pre(OutdeltaTemp);

     (Wait,Sol,Work,Block,Panaudio,Panwait,Panoper,Panblock,Warntemp) =
        compute_mode(con_onoff, timer(con_onoff),
                     In2deltaTemp, OutdeltaTemp,
                     TempLess5Tgrad, newpantreq);

tel -- modeselect
```

6.2 Esterel *compute_mode* module

```
module compute_mode:

input con_onoff, ten_solution_time_con_onoff;

output Wait, Sol, Work, Block;  %modes
output Panaudio, Panwait, Panoper, Panblock, Warntemp; %as panelout

input  In2deltaTemp, OutdeltaTemp, TempdLess5Tgrad, newpantreq;

suspend
[
   trap block in
      await tick;
      loop                     %wait state
         weak abort
            [sustain Wait || sustain Panwait]
         when immediate In2deltaTemp;
         weak abort            %sol state
            weak abort
               [
                  weak abort
                     await tick;
                     [
                        sustain Sol
                      ||
                        await immediate [not (In2deltaTemp)]
                           do emit Warntemp end
                     ]
                     when ten_solution_time_con_onoff
                     do await tick;   %work state
                     [
                        sustain Work
                      ||
                        await immediate OutdeltaTemp
                           do emit Panaudio;emit Warntemp end
```

```
                ]
              end weak abort
             ||
              await tick;sustain Panoper
            ]
         when [not (In2deltaTemp and TempdLess5Tgrad)]
         do exit block
         end weak abort
        when [OutdeltaTemp and newpantreq]
        do  await tick
        end weak abort
     end loop
   handle block do
      await tick; %block state
      [sustain Block || sustain Panblock || sustain Warntemp]
   end trap
]
when immediate [not con_onoff]
end module
```

6.3 Signal main process

```
process clim_cham_sys =
     ( ? boolean on_off;
         integer optreq;
         integer optvent;
         integer chamtact;
         integer chamflow;
       ! boolean Chambonoff;
         integer Conheat;
         boolean Panwarn;
         boolean Panaudio;
         boolean Panwait;
         boolean Panoper;
         boolean Panblock;
         integer Confan;
     )
   (| Chambonoff := if alpha then false else on_off
    | pantreq := if on_off then if (optreq>=treqmin) and (optreq<=treqmax)
                 then optreq*10 else (if optreq<treqmin then treqmin*10
                 else (treqmax*10))else (pantreq$ init 200)
    | pantvent := if on_off then if (optvent>=tventmin) and (optvent<=
                  tventmax) then optvent*60 else (if optvent<tventmin then
                  tventmin*60 else (tventmax*60))else (pantvent$ init 3000)
    | (Wait,Sol,Work,Block,Panaudio,Panwait,Panoper,Panblock,Warntemp)
         := modeselect(on_off,pantreq,chamtact)
    | Conheat := tempreg(pantreq,chamtact,Block,on_off)
    | Panwarn := if alpha then false else (if on_off then Panaudio else
                 Panwarn$)
    | (Confan,Warwent) := flowreg(pantvent,chamflow,Block,on_off)
```

```
  | alpha := (not (^alpha))$ init true
  |)
 where
 integer pantvent, pantreq;
 boolean alpha, Warwent, Warntemp, Block, Work, Sol, Wait;
 constant integer treqmax := 300, treqmin := 100, tventmax := 30, tventmin
                 := 5;
 process modeselect = ...
 process tempreg = ...
 process flowreg = ...
 end
;
```

6.4 Clock and dependency relations for the *modeselect* component (SIGNAL syntax)

```
process modeselect =
     ( ? boolean con_onoff;
         integer pantreq;
         integer chamtact;
       ! boolean Wait;
         boolean Sol;
         boolean Work;
         boolean Block;
         boolean Panaudio;
         boolean Panwait;
         boolean Panoper;
         boolean Panblock;
         boolean Warntemp;
         boolean XZX_1104;
     )
     spec (| (| con_onoff --> Wait
              | con_onoff --> Sol
              | con_onoff --> Work
              | con_onoff --> Block
              | con_onoff --> Panaudio
              | {pantreq --> Panaudio}when XZX_1104
              | {chamtact --> Panaudio}when XZX_1104
              | con_onoff --> Panwait
              | con_onoff --> Panoper
              | con_onoff --> Panblock
              | con_onoff --> Warntemp
              | {pantreq --> Warntemp}when XZX_1104
              | {chamtact --> Warntemp}when XZX_1104
              |)
           | (| con_onoff ^= pantreq ^= chamtact ^= Wait ^= Sol ^= Work ^=
                Block ^= Panaudio ^= Panwait ^= Panoper ^= Panblock ^=
                Warntemp ^= XZX_1104 |)
           |)
;
```