A Design Framework for Real-Time Reactive Applications

Marie-Agnès Péraldi, Jean-Dominique Decotignie

Swiss Federal Institute of Technology, Lausanne
Industrial Computer Engineering Laboratory
EPFL-DI-LIT
CH-1015 Lausanne, Switzerland

Abstract - **This paper deals with the design of reliable control systems using a modular and systematic approach. We introduce the Hierarchical Reactive Synchronous Boxes model (HRSB) as executable specifications for reliable control. This framework allows the user to focus on the expression of the *control*, the *fault diagnosis*, *fault recovery* of a given system using a *formal language* (synchronous language). The coordination between modules is given by the behavioral semantics of the model. This model inherits its semantics from those formal semantics of synchronous languages.**

I. INTRODUCTION

We are interested in the design of real-time reactive applications. One characteristic of these applications is their high level of complexity. Quoting Halang and Stoyenko [5] the main requirements are

- reliability of the control,
- safe evolutions,
- deterministic behavior,
- timeliness.

These systems are present in different application domains such as *flexible manufacturing systems* (FMS), *robotics* and *Computerized Numerical Control* (CNC) [4]. Designers in such domains tackle the problem in different ways. In FMS and in robotics, systems are decomposed into layers [13][3][10]:

- coordination,
- supervision,
- local control.

Real-time behaviors appear at the local control level but this control is made of simple actuators and sensors with reflex behaviors. Real-time constraints are managed at higher levels.

We tackle the problem of expressing real-time constraints in a different way. We consider that *reliability* and *safety* problems are relevant to the local control level. Hence, we centralized the treatment associated with these behaviors at this level.

We propose a model called Hierarchical Reactive Synchronous Boxes HRSB [8], that handles the normal control, a set of classical errors in FMS and provides mechanisms to recover from these errors.

Using this framework, the designer has to express in independent modules the *Control*, the *Fault Detection*, the *Fault Diagnosis* and the *Fault recovery*. As these modules are highly reactive, we have chosen a synchronous language (ESTEREL) as programming language[1]. Another advantage of this choice is that ESTEREL is a formal language.

The cooperation between modules is given by the operational semantic of our model. We use ESTEREL for expressing this cooperation. The originality of this approach is many-folds

- The HRSB model provides a framework for expressing reactive control in a modular way,
- It provides executable specifications as the HRSB model is writen in the synchronous language ESTEREL,
- It inherits from ESTEREL its formal semantic,
- It allows formal verifications on the code generated.

The rest of the paper is organized as follows: In a first part, we present the HRSB model from a functional point of view. In the second part we bring out the concept of the synchronous approach and a synchronous language like ESTEREL and we justify the choice of a synchronous behavior for our model. In the third part we show how a HRSB can be translated into an ESTEREL program. In the last part, we give some conclusions and perspectives we foresee for this work.

II. THE HRSB MODEL

The HRSB model provides a framework for expressing the *systematic* design of control programs including

[1]Note that recent works [1] provide a systematic translation from GRAFCET into ESTEREL. So GRAFCET is an other possible source language for programming modules

Figure 1. The HRSB Model

failure detection, *diagnosis* of errors and *recovery* from these errors. Designers must operate at the module level by expressing these different behaviors. Figure 1. represents the decomposition of the HRSB model into five modules. Each module is a block diagram. Connection between blocks is performed by the exchange of signals: The decomposition is

- *Command filter* : receives commands from the processes and gives priority to them in case of conflict, for example. The *Command filter* can be used to detect inconsistent commands provided by the upper level.
- *Control module*: The *control module* expresses the normal control of the process i.e. the different commands to be executed to perform an action. This can be, for example, the management of a motor. This module reacts to commands provided by the *command filter* and waits for the acknowledgments. The interface of the control module is very simple. The module receives inputs (`ControlF_k`). In return it indicates if the command initiated completed or not. The control can be sought multiple times. This is the case when the command doesn't complete and the necessity of retrieving this command has been decided.
- *Detection module*: The *detection module* is in charge of detecting the completion of a command and also three kinds of errors:
  - *timeout errors*: when a command is requested (`ControlF_k`) the execution and the completion of this command is monitored by a timer that produces a timeout error signal (`TO_Error`) when the deadline expires. The command is monitored in a temporal manner.
  - *sensor errors*. Sensors can be used for control purposes but also for error detection. Sensor errors are detected by polling their values. Specific signals are emitted in case of error detection `Error_k_li` .
  - *signal errors*. The HRSB may have different levels of hierarchy. The execution of a command at level `k` may imply the completion of a command at level `k` $+$ `1`. The detection module reports the non-completion of subcommand `Error_k` $+$ `1(integer)`. The value conveyed by this signal is an indicator of the cause of failure.
- *Diagnosis* module: The *diagnosis module* reasons based on the different signals sent by the detection module. According to the seriousness of the error detected, different strategies can be suggested to the decision module: `TryAgain`, `TrySomethingElse`, `or Abort`. The internal behavior of a module is essentially control-oriented. Only diagnosis modules have a transformational part. A procedure of recovery can be triggered in the diagnosis module.
- *Decision* module: As its name indicates, the *decision module* makes a choice between the different possibilities proposed by the diagnosis module. This module contains a set of recovery procedures that correspond to the possible known errors. If failures are unrecoverable, it reports the decision to higher levels. In all cases, the decision module transmits a failure report to this level.

Each module has a template to fill in. This approach is particularly convenient for designers not familiar with synchronous languages. A module can be seen as a black box that produces a reaction according to given stimuli. For a given application, the expression of control is inevitable. The designer must express this control with respect to some elementary rules such as the set of inputs and outputs, and the acknowledgment signals associated with each command.

However, depending on the reliability level, the control must be endowed with mechanisms for the detection of errors. In that case, the designer can fill the other modules (completely or partially).

Each module is characterized by its interface (signals) and its internal behavior. Signals provide the communication interface whereas internal behavior is relative to control. There exist three levels of communication.

- Communication between a HRSB and the upper levels of supervision and coordination. This communication is called **external communication**.
- Communication between the different modules inside a HRSB. This is what we call **internal communication**.
- The third level of communication concerns **local communication**. Local signals are used to provide synchronization inside the different modules.

These three levels of communication are modeled in ESTEREL by different type of signals (internal or external). Signals use broadcast as a diffusion mechanism, i.e., all the modules see the same things at the same time. Interactions between modules and information transfers are given by the operational semantics of the HRSB. This semantics is formalized in ESTEREL, i.e., an ESTEREL program is associated with a HRSB and expresses the synchronization skeleton of the modules. A characteristic of this skeleton is that it contains a high-level of parallelism and different levels of preemption. For these reasons, this skeleton is automatically generated.

Details about this translation are presented in section IV.

III. THE SYNCHRONOUS APPROACH FOR EXPRESSING REACTIVITY

Synchronous Approach

The synchronous approach introduced in the early 80's is a noteworthy advance in the design of real-time reactive systems. Synchronous formalisms can be classified according to their input format:

- *visual* formalisms such as GRAFCET or STATECHARTS,
- *textual* formalisms (ESTEREL, LUSTRE, SIGNAL).

A second distinction can be done based upon programming style that can be *imperative* or *declarative*.

Imperative languages such as ESTEREL rely on the notion of states. *Declarative* languages such as LUSTRE and SIGNAL lead to functional descriptions. The expressiveness of both languages is the same. However, according to the type of applications, one approach can be more suitable than another.

We have chosen an imperative style for the description of the HRSB model: the ESTEREL language. Imperative styles are clearly well-adapted to problems where the control is prevalent (e.g. systems where sporadic events trigger changes in functioning modes.)

Hypotheses of the Synchronous Approach

The synchronous approach has chosen to abstract the details of the implementation. Ideal real-time primitives are assumed, and the physical characteristics of the environment are represented by logical signals.

The main characteristic of the *synchronous approach* is that reactions do not overlap. There is a clear notion of an instant. Reaction has a zero duration. This is the *strong synchrony* hypothesis.

A second characteristic of synchronous programs is that communications are performed by *instantaneous broadcasting*, so that, any components can perceive the same information at the same time.

The expression *perfect synchrony* qualifies models that assume both the first and the second hypotheses. The perfect synchrony allows drastic simplifications in modeling and analysis.

The ESTEREL *language*

Synchronous languages, and more particularly ESTEREL, are based on a strong assumption (synchrony hypothesis) that allows mathematical semantics making formal proofs easier. As a result, programming languages lend themselves to *simple and safe programming*. The ESTEREL compiler generates fully deterministic code. Determinism is a valuable contribution to the predictability requirements of real-time systems. Moreover, compilers of synchronous languages can produce very efficient code (finite automaton).

In this paper, we consider the ESTEREL language which is well-suited to event-driven system programming (mostly control, little data processing). Here, we experiment the use of ESTEREL for the generic programming of control, monitoring and error recovery. We start the presentation of the language by the example of a watchdog.

```
module TODetection:
input Top, BoT, EoT;
output TOError, Done;
constant deadline: integer;
        await immediate BoT;
        do
                await deadline Top;
                emit TOError
        watching  EoT
        timeout
                emit Done
        end %watching
. % end module
```

The basic entity in ESTEREL is the module. The interface of a module is given by its input and output signals. As we can see in this example, synchronous languages do not consider physical time. The signal `Top` is the logical representation of physical time whereas other signals are relative to pure events (`BoT`, `EoT`), but any repetitive input signals can be taken as a generalized time basis so that the time is said to be multiform.

The constant `deadline` gives the value of the deadline. On the occurrence of the `BoT` signal (Begin of Task) the control pass to a `do ...watching` structure that monitors in parallel the completion of a task (`EoT`) and the duration of this task that does not exceed `deadline` occurrences of the signal Top. If the task meets its deadline, the signal `Done` is emitted, if not, a `TOError` is sent. In case of simultaneous occurrences, of `Top` and `EoT`, priority is given to the signal `done` .

The `do ...watching` structure performs a *strong preemption* (prevent a block from being executed). An other structure `trap ...handle ...end trap` allows the expression of a *weak preemption* (concurrent statements terminate what they have to do during the current instant and then abort). Preemption mechanisms are useful for expressing changes in the execution mode of a process.

Other primitives are available in ESTEREL like a parallel operator that allows a restricted form of concurrency (fork-join like structures).

IV. SYSTEMATIC TRANSLATION OF HRSB INTO ESTEREL

The notion of HRSB is a generalization of the watchdog module, i.e., it consists of the systematic design of a control that provides different execution modes.

Coordination structure of a HRSB

The HRSB is a composition of different modules with different rules of preemption and communication. The different modules are embedded in a synchronization skeleton. The code of this skeleton is the following:

```
module BHRS:
input Control_k, Top, Ack_k+1,
      Nack_k+1(integer);
output  Ack_k, Nack_k(integer);
task recover_task()();
sensor SError_k_l1, SError_k_l2:
       boolean;
constant error_k, error_k+1, nopower:
         integer;
signal Control_k_Done, ControlF_k,
      ControlF_k_Done, Error(integer)
in
 trap WORKING  in
  run Filtrage;  exit WORKING;
  ||
 signal TO_Error, Error_k_l1, ...  in
  trap DONE  in
   trap ABORT  in
   loop
    trap TRYAGAIN, TRYSELSE  in
     run Control; emit Success
     ||
     run Detection
     ||
     run Diagnosis
     ||
     await
      case Tryagain  do exit TRYAGAIN
      case TrySElse  do exit TRYSELSE
     end await
    handle TRYAGAIN  do run MTryAgain
    handle TRYSELSE  do run MTrySElse
    end trap %Tryagain, Tryselse
   end loop
   ||
   await
    case Success  do exit DONE
    case Abort  do exit ABORT(?Abort)
   end await
  handle ABORT  do
   emit Control_Abort_k+1;
   await Ack_k+1;
   emit Error(??ABORT)
  end trap %Abort
 handle DONE  do
    emit Done
   end trap %Done
   end signal
   end trap %Working
  end signal
  end module
```

Input Output interface

The input and output signals represent the interface of the HRSB. The *control flow* is represented by the `Control_k` signal, but in case of real application this signal will be replaced by the physical signal of the application. `Top` is a clock signal, the signals `Ac_k+1` and `Nack_k+1` represent the monitoring flow that go up to the higher level. The sensors allow a periodic acquisition of the state of the physical process under control.

Synchronization skeleton

The trap mechanisms allow the management of exceptions. It is a way to reroute the execution. The HRSB module includes different levels of traps overlapped that correspond to the actual state of the process.

- the `WORKING` trap corresponds to the normal control. All the modules are activate in parallel (`run` *module_name*). We exit from this trap when commands are executed.
- the `DONE` and `ABORT` traps from which we exit when the execution of a command is done or impossible.
- the `TRYAGAIN` and `TRYSELSE` traps. These traps correspond to the different policies for the recovery from errors. Inside this trap the control executes two different recovery treatments.

The way in which the traps have been overlapped has a great importance. There are different levels of depth. We have to notice that when the control exits from an external trap the execution inside the internal one is stopped.

We can see here the importance of giving to the designer one such generic scheduler that executes these different mechanisms. In fact, the ESTEREL program that has this behavior is complex to design for a non specialist of synchronous languages. In addition the skeleton proposed is error prone and the logical behavior has been checked.

The different modules

The different modules of the model have been included inside the different traps. The execution of a module starts when the control pass inside a trap and it completes at the exit of the trap. Decisions for the exit of the trap are taken inside the trap according to the signal emitted. The structure of a HRSB is generic, i.e., the structure is the same. The composition of the module can vary according to the application. This decomposition is very useful for the designer because it allows an expression of the control in different independent modules without taking care of the interactions between these modules. Designers can fill partially or omit to fill some modules. For example, if designers want to express an "unsafe" control, they are free to avoid the expression of the body of the detection, diagnosis and decision modules.

Here we give an example of the *diagnosis* module :

```
module Diagnosis:
input TO_Error, Error_k_l1,
 Error_k_l2, Error_k+1(integer);
output Success, Abort(integer),
 TryAgain, TrySElse;
constant er_k, er_k+1: integer;
task recover_task()();
await
 case TO_Error  do emit TryAgain
 case Error_k_l1  do emit TrySElse
 case Error_k_l2  do emit Abort(er_k)
 case Error_k+1  do
  if ?Error_k+1 = nopower  then
   exec recover_task()();
   emit TryAgain
  else
   emit Abort(er_k+1)
  end if
end await
end module
```

The diagnosis module uses the information sent back by the detection module. We have isolated three kinds of possible diagnoses with different levels of severity.

- the `TryAgain` diagnosis that is due to the detection of a timeout error or when an attempt to repair has been done on a defective component. This repairing can be done by an external operator. This diagnosis puts the control in a state that allows an other emission of the command.
- the `TrySElse` diagnosis. An error of level one has been detected. The control triggers a module that has in charge of the execution of a special treatment. This treatment must be define by the designer.
- the `Abort` diagnosis with the indication of the type of error. It is the worst case, i.e., the local level cannot take decisions and defers the error to a higher level . Such diagnosis is done when an error of level two or the attempt to repair the defective components has failed.

V. HRSB AS EXECUTABLE SPECIFICATIONS

The compilation of a HRSB generates a finite state automaton in many general purpose languages such as C, C++ or ADA. Therefore, direct implementation on a target machine is possible. Compilers of synchronous languages are reliable and they produce efficient code, e.g., an automaton semantically equivalent to the program. At this point additional verifications are necessary in order to assure that the implementation and the specification are consistent. The designers have to express the *logical correctness* of the program generated. This can be done using the simulation environment of the ESTEREL language. *Safety properties* (nothing undesirable will never occur) must be verified also. The dependability of the software generated can be checked using the automaton prover.

VI. CONCLUSION AND FUTURE OBJECTIVES

In this paper, we have presented a systematic way of programming reliable local control in FMS. The originality of this approach is to provide an executable model that allows a modular description of an application with a clear separation between the normal control and fault detection and recovery. In [11] authors suggest similar solutions in the context of faulty links in networks. The reactive behaviors and synchronization skeleton are expressed in the synchronous ESTEREL language. The model is based on formal behavioral semantics inherited from the semantics of ESTEREL. As the ESTEREL compiler generates deterministic automaton, logical properties of the application can be checked.

Future extensions of this model are envisaged. We will continue our research efforts in two directions. The first direction is the application domain. In fact, HRSB can be used as a basis for building reliable programs in other domains such as robotics, and local area networks. The most basic issue faced in such scientific pursuit is to select for each domain a pattern of fault occurrences [7] [9] [12]. The second direction concerns the *reusability* of such basics components. The introduction of Object-Oriented methods is one way to cope with *reusability* and *genericity* in the design of control applications [6]. As a consequence, HRSB can be seen as generic *reactive components software.*

VII. REFERENCES

[1] C. André and D. Gaffé. Coopération grafcet/estérel. In *Colloque AGI'94*, pages 221–224, Poitiers, Juin 1994. Association AGI.

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

[3] J.P. Bourey, E. Castelain, J.C. Gentina, and M. Kapusta. CASPAIM: A computer aided design of the control system of FMS. pages 131–135. IMACS Annuals on Computing and Applied Mathematics, 1989.

[4] J.-D. Decotignie. Fieldbus and cnc architecture: towards a cnc software factory. 3rd IEEE Symposium

on Emerging Technologies and Factory Automation (ETFA), Tokyo (Japan), Novenber 1994.

[5] W.A. Halang and A.D. Stoyenko. *Constructing Predictable Real-Time Systems.* Kluwer Academic Publishers, Amsterdam, 1991.

[6] P.A.V. Hall. *Software reuse and reverse engineering in practice.* Chapman] Hall, 1992.

[7] G. Le Lann. Panel discussion on limits in dependability. In *IEEE CS 23th Int. Symp. on Fault-Tolerant Computing*, pages 608–615, June 1993.

[8] M-A. Peraldi. *Conception et Réalisation de Systèmes Temps-Réel par une Approche Synchrone.* PhD thesis, Doctorat de l'Université de Nice-Sophia Antipolis, Juillet 1993.

[9] D. Powell. Failure mode assumptions and assumption coverage. In *IEEE CS 22nd Int. Symp. on Fault-tolerant Computing*, pages 386–395, July 1993.

[10] D. Simon, B. Espiau, E. Castillo, and K. Kapellos. Computer-aided design of a generic robot controller handling reactivity and real-time control issues. Technical Report 1801, INRIA, November 1992.

[11] C. Wang and M. Schwartz. Identification of faulty links in dynamic-routed networks. *IEEE Journal on selected area in communications*, 11(9):1449–1460, December 1993.

[12] R.C. Waters and Y.M. Tan. Toward a design apprentice: Supporting reuse and evolution in software design. *ACM SIGSOFT,Software Engineering Notes*, 16(2):33–44, April 1991.

[13] J. Wikander and M. Torngren. Decentralized control systems for modular machines. pages 1490–1494. IECON'94, Bologna (Italy), September 1994.