From CRSM to a Tasking Design

Jean-Jacques Schwarz[1], Katarina Jelemenska[2], Zhongwei Huang[3], Régis Aubry[1], Jean-Philippe Babau[1]

[1] Laboratoire L3i, B502, INSA de Lyon, Villeurbanne, France
[2] Dept. Computer Science & Engineering, Slovak University of Technology, Bratislava, Slovakia
[3] Dept of computer science, Harbin Institute of Technology, Heilongjiang, P.R. China

This paper introduces elements to facilitate crossing of the gap between analysis and design in the case of real-time applications relying on multitasking operating system. The chosen specification method is based on the use of Shaw's CRSM (Communicating Real-Time States Machines) and our purpose is to put the basis of a method for an easier translation of a CRSM-based modelling of a system into a real-time multitasking execution model. In order to do this, we present guidelines for translating the basic constructs of a CRSM model (communicating machines, channels, transitions) into programs involving the usual objects and primitives found in off-the shelf real-time multitasking operating systems (tasks or threads, message passing, event signalling...). The guidelines are illustrated with the classical specification example of the Martian Lander. The aim is to overcome the gap between a specification made with the CRSM and a multitasking execution model: this will then enable good possibilities for verification. The specification can be executed and the design can be verified for correctness (liveliness, safety). Eventually, a comparison between the behaviour of the specified model and that of the target program can be made.

1. Introduction

This paper introduces elements to facilitate the crossing of the gap between analysis and design in the case of real-time applications relying on multitasking operating systems.

The transition between the specification and design software lifecycle phases can generally be seen from two points of view. The first approach takes into account the fact that specification and design are two well-differentiated phases, both being devoted to specific aims. It therefore seems natural to use for each phase a personalised notation. Therefore it is clear that using the most appropriate modelling tool for each step is of great advantage. On the other hand, it is obviously necessary, while making the transition between the two phases, to operate a mapping from the first notation to the second one, facing all the implied and known drawbacks. A good specification notation is aimed at stating and analysing a problem clearly (unambiguously); while a good design tool is aimed, given a hardware and software target architecture, at allowing the designer to harness carrying out a (one) solution. If we assume that a well-analysed problem is a half-solved one, the remaining part to solve, the last but not the least, has yet to be built.

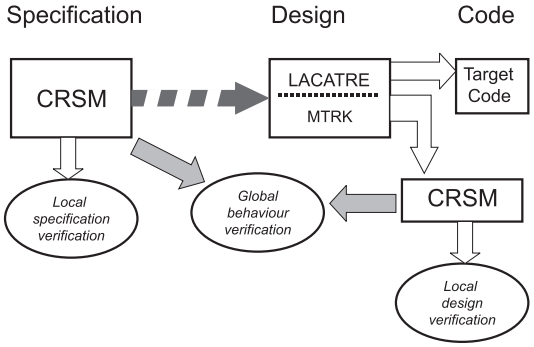


Fig. 1. Translation Process.

The second approach consists of adopting a single notation thus avoiding any risky mapping or transformation. But in this case, the main disadvantage is that the solution addresses then a specific architecture (ie SPRING, [5, 6]) and/or does not allow a complete control of the archi-

A previous short version of the paper appeared in *Proceedings of the IEEE International Symposium on Industrial Electronics*, Bled, Slovenia, July 12–16, 1999.

tectural choices in the case of standard operating systems (i.e. overhead generated by a virtual machine).

As far as concerned, this paper is dedicated to the first approach. The purpose is to facilitate the transition between the specification and design steps for the numerous industry designers that are familiar with state-machines family notations at the specification level and who are used to implement their applications, using off-the-shelf multitasking operating systems.

In order to highlight the technical features of our proposal we will rely, having practical considerations, on Shaw's CRSM at the analysis level and, at the design level, on the LACATRE modelling formalism that allows to abstract a given operating system. Beyond these particular notations, the proposed ideas can probably be helpful to a much larger community of industry real-time designers.

Starting with a specification based on CRSM (Communicating Real-Time States Machines) [4], our purpose is to propose guidelines for an easier translation towards a real-time multitasking execution model represented in the LACATRE formalism [3]. The final aim is to overcome the gap between a specification made with the CRSM and a multitasking execution model: this will enable good possibilities for verification. The specification can be executed, the design can be verified for correctness (liveliness, safety) and eventually a comparison between the behaviour of the specified model and that of the target program can be made (Figure 1).

The paper is structured as following. Section II recalls the main features of the CRSM. Section III describes the LACATRE formalism as an abstraction level to standard COTS real-time kernels, and highlights the main semantic differences between CRSM and LACATRE and section IV presents some of our translation rules. In section V, some of the guidelines are illustrated with an example.

2. CRSM as a Specification Language

CRSM (Communicating Real-Time State Machines) represents a complete and executable notation for specifying the requirements of real-time systems including the monitored and controlled physical environment. They are essentially state machines that communicate synchronously in a manner much like the input-output in Hoare's CSP (Communicating Sequential Processes). In addition, there is a set of facilities for describing timing properties and accessing real time. This language allows not only for simple and natural requirements specification, but also some algorithms are available for executing or simulating CRSM, as well as some techniques for reasoning about the specification. The representation was developed with the aim to satisfy, as completely as possible, the needs of real-time applications.

It allows to express concurrency, communications and synchronisation of various real-time processes, as well as to express time explicitly.

When specifying a system using CRSM, any system model has at least two machines, one to model the environment and the other to specify the required computer system behaviour. The machines execute concurrently and independently of one another in case there is no active communication among them. Any pair of state machines can communicate together, sending messages over channels that are unidirectional, uniquely identified, and there is a message type associated with each channel.

In case of communication the two machines are synchronised because the communication only takes place when both machines are ready for the communication. That means that the first machine that is ready to communicate must wait until the second one is also prepared to communicate.

There are no global variables within the model except for global real time, so each of the state machines can only operate on its local variables.

Each machine has a finite number of states with one start state and zero or more halt states. The state transitions are described as guarded commands with optional guard (Boolean expression over local variables) and a command that can be either an input, output, or internal command.

Internal command can specify either a sequential program or a physical activity. Its execution time is marked off by a pair of time values, *Tmin* and *Tmax*, indicating that the duration of the command is somewhere in the interval $< Tmin,\ Tmax >$. These bounds can be represented as a requirement or as a given behaviour.

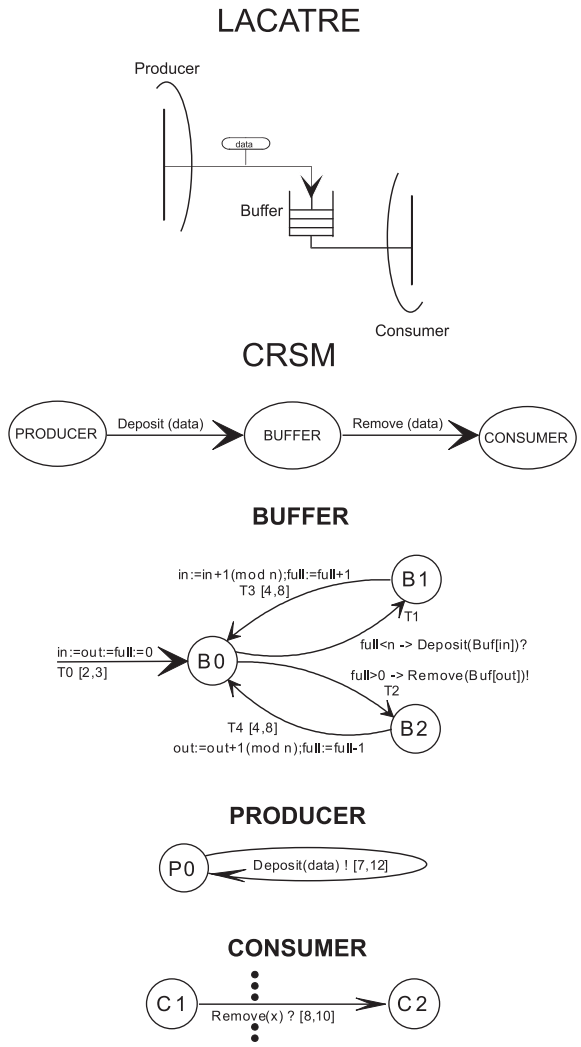


Fig. 2. CRSM Modelling of a LACATRE Mailbox.

The state transition with an internal command can be executed when its "from" state is entered and its guard is evaluated to *true*.

Several commands may be eligible for execution at the same time; selection is then made non-deterministically.

Input and output are modelled directly after CSP: the communication is synchronous, 1-way, and has a unique sender and receiver. What is more, the communication can only occur when two machines are sending/expecting the message of the same type, over the same channel, and can perform their IO commands at the same time.

Using the synchronous type of communication over channels for real-time specifications has several advantages compared to other alternatives such as an asynchronous method with or without broadcast. Namely, very simple assignment semantics of the IO, no messages are "lost", there is no need to assume unbounded buffer capacities on communication channels, and finally, it allows to introduce the real time in a natural and easy manner.

IO times are also represented by pairs of time values, in this case denoting the earliest and latest times that the IO can occur after entering a given state. So the time values represent the minimal and maximal delay of an IO from the time when the machine has entered the "from" state of the transition. So if the time intervals of sending and receiving machine do not overlap, the communication cannot occur at all and neither of the transitions can be executed. IO is instantaneous and any preparation, processing, or communication time that should be accounted for can be expressed either in time values or using an internal action immediately following the IO.

In spite of the instantaneous nature of commands there is always a minimal nonzero amount of time δ that the machine will spend to do an IO or an internal command in case the specified time values are zero.

There is a real-time clock machine RTC_M associated with each machine M. This machine communicates with machine M over channel RT_M and is always ready to send the current value of real time, represented by global variable *rt*, that is globally available to all clock machines.

An IO command $RT_M(x)?[y]$ executed by machine M will generate a timeout at relative time y and set x to the real time *rt* at the timeout. So x can be used as a time stamp for time-out event. By omitting y the approximate time of

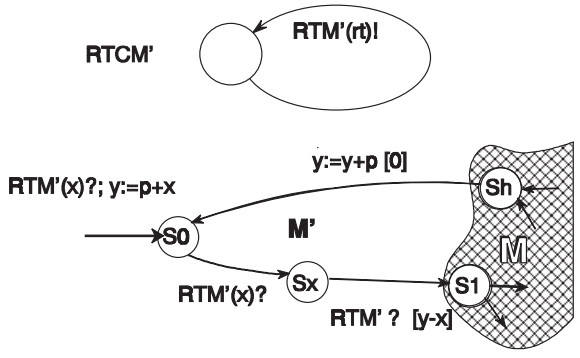


Fig. 3. CRSM Modelling of a Periodic Process.

entering the "from" state of the transition will be assigned to x. Omission of x will generate a pure time-out.

Thus CRSM provide mechanisms for accessing absolute and relative real time and for describing timeouts, the main functions available on almost all computer systems. This method of time specification is quite simple and the description of periodic (Fig. 3) or sporadic process control, the typical parts of real-time application, can be done without any difficulties. If necessary, it is also possible to define a discrete time clock 0.

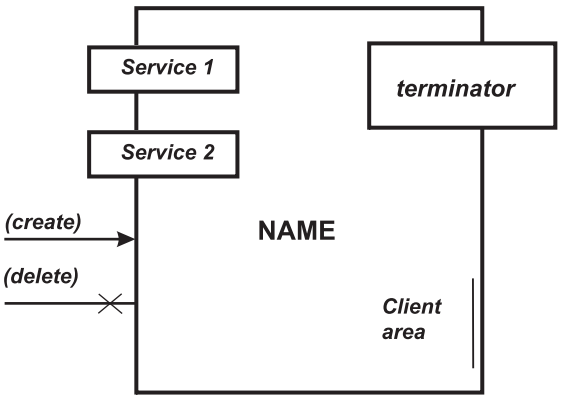


Fig. 4. Applicative Object.

3. Multitasking Design

We are focusing our work on designs relying on off-the-shelf Real-Time Multitasking (or Multithreading) Kernel MTRK (or Operating System) used in industry. Even if these kernels have similar functional purposes, they may differ in the basic tasking objects proposed to the programmer. That is the reason why we have chosen the LACATRE language and tool for the execution model of an application.

The LACATRE graphical language was initially designed for educational purposes: the same tool is used, on the one hand, to describe and model the objects being manipulated by real-time kernels and, on the other hand, as a design tool for real-time applications. Within the software lifecycle, LACATRE covers both the preliminary design and detailed design steps. It gives access to the various approach axes required for the design of a real- time application:

* The abstraction-level axis settles the link between the low-level basic objects (the kernel objects: tasks, semaphores, messages, mailboxes...) and the objects that are defined during the specification step of an application.
* The behavioural axis allows modelling of the dynamic behaviour of an application, from the multitasking and communication/synchronisation points of view.
* The transformational axis defines how and where the application data processing is done.
* The application phase axis deals with the various running modes of an application: the initialisation, the normal running (kernel), the exception handling and the shutdown.

For the purpose of this paper we have focused only on the dynamic behaviour of an application.

A dual textual language, which can be considered as an intermediate language, is associated with the graphical language. Starting from the textual form of his programme, the designer can use different tools, such as the code generator for a given target (for example, a C-C++ programme with system calls for a given real-time executive). In addition, it is obvious that a program has to be validated (verified) before its layout. These verifications are done using both the principles of hierarchical relative correctness [7] and the modelling of each LACATRE atomic object by means of communicating real-time state machines [1].

During the preliminary design step, LACATRE allows the designer to structure his solution by splitting it up into various actors represented by high-level objects called "applicative objects". Each applicative object proposes to the other actors a set of specific functions (services) which have therefore to be defined by the designer.

As a client does not have to know how a service is processed, the realisation of this service can be hidden. Nevertheless, it may require services offered by other applicative objects: hence, an applicative object is often both a server and a client. In addition, an applicative object may have to interact with the outside real world to be monitored or controlled: these interactions are represented by means of "terminators" as in SART.

The applicative objects can be handled by way of state primitives. These state primitives (*create, delete, suspend, resume, connect, disconnect*...) are common to the whole set of LACATRE objects.

During the detailed design step, the description of the behaviour of an applicative object, that means the processing of the state and service primitives, is obtained thanks to combining basic LACATRE objects, called "atomic objects". These objects can be considered as the final leaves in a complete hierarchical decomposition of an applicative object. These low-level objects and their associated primitives are images of the objects and system calls offered by the existing real-time kernels. These atomic objects are not only used to implement services, but also to specify the access protocols to these services.

They can be classified into two classes:

- The server objects: they are concerned with data flows and synchronisation. Their behaviour is user-configurable, but not programmable. The current version of LACATRE proposes the following objects: message, mailbox, resource, semaphore and event (Fig. 5a.).

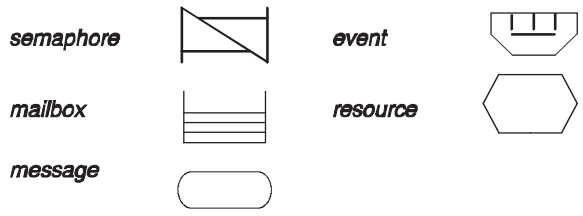


Fig. 5a. Low Level Server Objects.

- The client objects: the processing (control flow) associated to such an object is specified by the programmer. LACATRE currently proposes two main execution mechanisms: the task and the exception handling (including interrupts) (Fig. 5b).

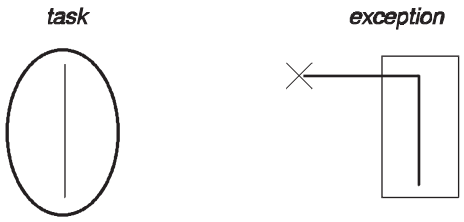


Fig. 5b. Low Level Client Objects.

Server objects, providing clients objects with communicating-synchronising services, are the usual semaphore (and derivatives), message and mailbox (queues, message passing protocols), event (broadcast synchronisation) and resource (read/write embedded in access-release operation for mutual exclusion functioning).

Actions (primitives) are associated to these objects (create, delete, suspend, resume, P, V, send_message, acces-read,...) and show on the graphical design the links between clients and servers.

Task and Interrupt objects are said to be clients because they have been programmed to use communication objects (servers). Tasks (threads) are under the control of the operating system kernel (scheduler). Exception (Interrupt) objects correspond to the code (interrupt handler) activated by the physical environment (sensors, actuators, devices, ...).

Task parameters are introduced to take the task characteristics into account. A task owns an identifier and a Task Time Requirement Descriptor (offset). The TTRD is a septuple, which describes the nature and the time constraints of the task:

$$\mathrm{TTRD_i} = \left\{ \mathrm{N_i},\ \mathrm{S_i},\ \mathrm{P_i},\ \mathrm{r_i},\ \mathrm{c_i},\ \mathrm{d_i},\ \mathrm{T_i} \right\},$$

$\mathrm{Task_i}$ descriptor,

with:

$\mathrm{N_i}$: task type: periodic or sporadic,
$\mathrm{S_i}$: execution site,
$\mathrm{P_i}$: task priority (statically or dynamically assigned),
$\mathrm{r_i}$: task starting time,
$\mathrm{c_i}$: task computing time,
$\mathrm{d_i}$: task deadline
$\mathrm{T_i}$: task period.

The parameters are not necessarily set according to the task time constraint knowledge that the developer has at a given date. For example, for the standard real-time executives previously considered, only the priorities will be taken into account. For a sporadic task, the TTRD parameters have a different meaning, some of them being non-significant and therefore not set. For example, the $\mathrm{T_i}$ parameter will possibly represent the associated pseudo- period, which means

the minimum time interval between two successive computations of this task, if sporadic tasks are taken into account as periodic tasks [7].

4. From CRSM to Design: Rules

In a CRSM specification any system model has at least two machines, one to model the environment and the other one to specify the required computer system behaviour. This is a somewhat unusual approach and the question naturally arises, whether to incorporate the given behaviour of an environment, described in CRSM representation, into the execution model or not.

In LACATRE, the main object that is able to express some activity is the *task* (*thread*), so quite naturally we would assume that each state machine in CRSM representation should be represented with it. But there are other objects available. For example, a display or a reading device can be represented using a *resource* object and a switch can be represented by an *interrupt*.

The problem is how to distinguish between given and required behaviour when they are mixed together and represented uniformly in CRSM model. So, further on we will assume that the two associated types of state machines are clearly identified within a CRSM representation (for instance shaded objects for ESMs-Environment State Machine, non-shaded objects for SSMs- System State Machines).

The other thing where CRSM and LACATRE differ substantially are directly supported means of communication. In CRSM, the communication is synchronous channel-based: the *channel* provides a direct communication mechanism between two state machines, and the communication can only occur when the two machines are sending/expecting the message in the same time interval.

On the other hand, LACATRE provides several communication entities (*semaphore*, *mailbox*, *interrupt* and *event*) but none of them requires the two communicating tasks to wait for each other. Neither is there an action that would enable the tasks to communicate directly just like any two state machines can by means of channel in CRSM representation.

Based on these differences, the translation of the following constructs in CRSM has to be considered.

A. Environment State Machine Translation

The actual representation of ESM in LACATRE will be decided based on its behaviour and communication with other state machines.

- If there is only one IO command and zero or more internal commands in ESM, the internal commands will be ignored and the ESM will be represented as one LACATRE object (resource and/or interrupt) according to the IO command.
- If there are several IO commands and zero or more internal commands in ESM, its behaviour will be simulated using LACATRE *task* object. However, to distinguish this task from other tasks, representing SSMs, all the communications between this ESM and a SSM will be represented using either *resource* or *interrupt* object, depending the IO command.

B. System State Machine Translation

In general, each of the state machines modelling System State Machines (SSM) will be described in LACATRE by means of one task object, using the state machine's name as task identifier. However, in some cases several new tasks can be created temporarily by this main task in case some parallel actions are modelled within the SSM.

C. Execution time of an internal command

The most natural way of expressing execution time of an internal command in LACATRE is to associate each internal command (with specified time values) with a procedure call specifying minimal and maximal times of execution (timed procedure).

Since in CRSM a transition with an internal command is ready at the same time as its "from" state is entered [5] both time values will be relative to the time point when the procedure was called. We suppose that an alarm will be set to

the value of deadline simultaneously with the procedure call in order to generate an interrupt (or an error message) in case the deadline was not met. Each time the return from the procedure is executed, the minimal execution time is checked as well producing some warning (error) message in case the execution was too fast.

D. CRSM's Input/Output Commands

Two main issues concerning communication will be discussed in this section:

- Transformation of single communication CRSM channel into one of several different communication objects available in LACATRE, and
- The difference between synchronous and asynchronous communication.

There are several solutions to these problems. Any indirect means of communication, supported in LACATRE can be represented in CRSM. However, if we required CRSM specification to use only indirect means of communications, it would result in substantial loss of readability of the design, since these indirect means would introduce new machines into it.

The simplest solution would be to extend LACATRE to include channels among its basic communication entities which would, however, only postpone the problem until later in case channels are not supported by target real- time executive. That is why we decided to find a suitable transformation of channel-based communications into objects and actions available in LACATRE.

The translation of CRSM channels will depend on the type of communicating state machines and on the input./output command.

1) Channels between two SSM or two ESM

SSMs in CRSM are represented by tasks and therefore channels could be easily represented by inter-tasks communication means like semaphores and mailboxes.

In case some time values are specified with input and/or output command in CRSM, there is always a possibility that the communication will never take place, since the two machines are required to be at the point of communication at strictly defined time interval. So, if one of them misses the time interval the message will not be sent at all. Obviously in this case it will be necessary to make the two communicating tasks wait for each other, otherwise the required behaviour will not be attained.

A time interval specified with input command can be represented in LACATRE using a waiting primitive with flow control on the mailbox (semaphore) representing the channel. This means that the reading task awaits the message

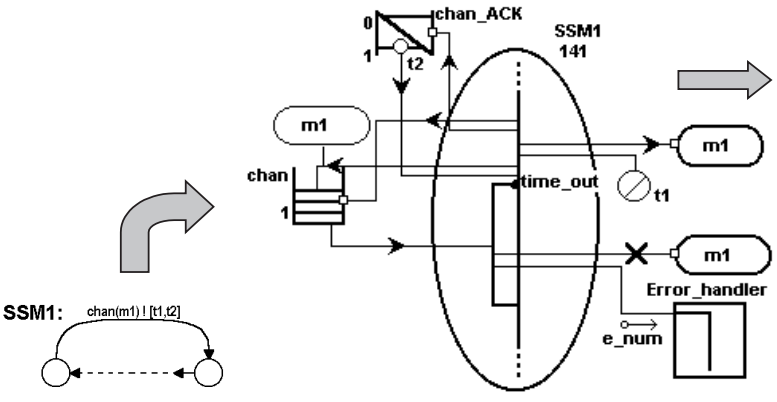


Fig. 6a. Graphical Modelling SSM1.

```
CREATE_MAILBOX (chan);
CREATE_SEMAPHORE (chan_ACK);
...
CREATE_MESSAGE (m1);
DELAY(t1);
SEND_to_MBX(chan, m1);
WAIT_on_SEM (chan_ACK, ,t2);
IF time_out THEN    /* end of
                overlapping period*/
      WAIT_on_MBX (chan); //
                /* get back the message */
      DELETE_MESSAGE(m1);
                /*(no communication)*/
      Error_handler(e_num);
ELSE
                /* next commands */
END_IF;
```

Fig. 6b. Textual Description.

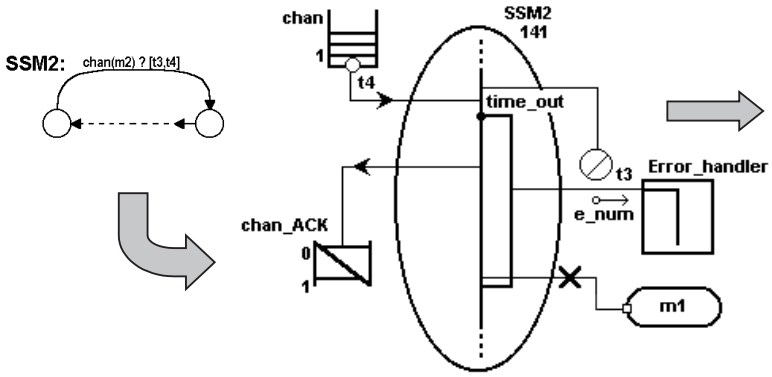


Fig. 7a. Graphical Modelling SSM2.

```
DELAY(t3);
WAIT_on_MBX(chan, t4);
IF time_out THEN
        Error_handler(e_num);
ELSE
    SEND_to_SEM (chan_ACK);
        /* next commands */
        DELETE_MESSAGE(m1);
END_IF;
```

Fig. 7b. Textual Description.

only for a specified amount of time. If a time-out occurs while awaiting a message the two tasks do not meet within the time interval and an exception handler must manage this fault.

When a time interval is specified with output command the message should not be sent in case the reading task is not ready to read the message. Similar behaviour can be reached when sending task deposits a message to mailbox or semaphore and then withdraws it again after a specified time in case it has not yet been read by reading task. This can be represented in LACATRE quite easily without introducing any new object or primitive, provided that a semaphore can be accessed by several tasks. However, a simple CRSM channel may be represented by two or more objects: two semaphores or mailboxes, message and a semaphore. The following example illustrates the situation.

Suppose there is an output command $chan(m1)![t_1, t_2]$ in SSM_1 (Figure 6a.) and an input command $chan(m2)?[t_3, t_4]$ in SSM_2 (Figure 7a). The first time value t_1 represents some internal activity, needed to prepare the communication. This internal activity has not yet been specified and therefore will be mapped to the LACATRE of $DELAY(t_1)$ system call. The second time value t_2 represents time-out of output operation, so the task SSM_1 should wait until the message has been read by task SSM_2 or time t_2 has expired. The output command will then be translated as shown in figures 7a and 7b. The corresponding reading task SSM2 waits for the time t_3 first and then awaits a message. If a time-out occurs while awaiting the message, the two tasks do not meet and an exception handler will manage the situation. Otherwise the task sends an acknowledgement. Figure 6b. illustrates the dual textual form of sending task SSM1.

2) Channels between SSM and ESM

A channel between SSM and ESM will be represented either by *resource* or *interrupt* object, depending on the message and the direction. Translation rules are based on the assumption that no time restrictions can be specified on the side of ESM. Any output to ESM and non-empty input from ESM in SSM with time restrictions will be translated into timed procedure call. The *access* or *access and release* action system calls (depending on the resource configuration) will then be included in the procedure itself. An empty input from ESM command in SSM with time restrictions is mapped to a timed *wait for interrupt* system call.

E. State Transitions

State transitions represent the execution flow of state machine. In CRSM they are described as guarded commands with an optional guard and a command that can be either an input-output, or an internal command. From now on, we will use the expression "guarded transition" to denote a transition labelled with complete command (explicit guard part) and the expression "unguarded transition" otherwise.

Task is a basic LACATRE programmable object with a progress bar representing its execution path. Several algorithmic forms can be placed onto this bar to change the execution flow: *forever, if condition, while condition, repeat-until condition, switch variable, procedure call*. Some of these algorithmic forms include a *condition* and are therefore used to translate a guarded transition. The decision, which of these algorithmic forms is to be used to translate a particular guarded transition, is

not easy to make, since a thorough inspection of state machine's structure will be necessary for the decision. To simplify this job, some restrictions have to be defined and given.

3) Restrictions

For the purpose of specifying the restrictions and the rules for translation of state transitions some terms have to be defined first.

Definition 4.1. For each state S of state machine M each state transition that ends up in state S is called *incoming transition* and similarly, each state transition that leads out of the state S is called *outgoing transition*.

Definition 4.2. Any sequence of states and state transitions of state machine M is called *path* iff it starts in a state S that is reachable from start state of state machine M.

Definition 4.3. A path starting in state S_i and ending in state S_j is called *full path* iff state S_j is either halt state or is included in a path from start state to state S_i.

Definition 4.4. Each incoming transition of state S that is on a path from start state to state S is called *initialising transition*.

Definition 4.5. A state S_i of a state machine M is called *quasi-start state* iff none of the paths starting in the state S_i includes a state S_j that is on a path from start state to state S_i.

Definition 4.6. A set of all paths starting in state S_i with the same state transition t and ending in the same state S_i is called *standalone loop* iff none of the paths includes a state S_j that is on a path from start state to state S_i.

Definition 4.7. A set of all paths (at least two) starting in state S_i and ending in state S_j is called *parallel construction* iff each of the paths has the same set of commands describing the state transitions on the path.

In CRSM, any Boolean expression over local variables can be used as a guard. But it would be very difficult and also time consuming, to consider all the possible combinations of transitions and their guards. Therefore some restrictions concerning allowed combinations of transitions and their guards are defined in this section. Based on the previous restrictions, following combinations of outgoing transitions from any state S are legal:

- one unguarded outgoing transition described with any command,
- exactly two guarded outgoing transitions described with any command,
- any number (implementation dependent) of unguarded outgoing transitions belonging to the same parallel construction,
- any number of unguarded outgoing transitions described with input commands; only one of the commands can represent input from real-time clock machine,
- two unguarded outgoing transitions one of them described with an input from real-time clock machine command and the other one with common input/output command.

When a state machine reaches a halt state, that means that its execution is finished. Since real-time applications have usually repetitive nature the halt states will most often represent erroneous situations. That is why they will be represented in LACATRE by procedure called *Error_handler(e_num)*, where parameter *e_num* represents a particular situation number.

4) Transformation rules

The following rules are supposed to be used during transformation of one state machine (certain ESMs and SSMs) of CRSM into LACATRE task in order to decide which of the algorithmic forms (control flow) available in LACATRE to use in a particular situation.

- If state S has only one outgoing transition and no incoming transition (except for initialising transition), analysis of state S is finished.
- If state S is a start state of state machine M and has only unguarded outgoing transition(s) and at least one incoming transition (except for initialising transition) *FOREVER* algorithmic form is placed onto the progress bar (see for instance state S1 in figure 10).
- If state S is neither start state nor quasi-start state of state machine and has only unguarded

outgoing transition(s) and at least one incoming transition (except for initialising transition) *REPEAT . . . UNTIL* algorithmic form is placed onto the progress bar (see for instance state S1 in figure 10).

- If state *S* has two guarded outgoing transitions with complementary guards $cond_S$ and $\neg cond_S$ and each of them starts one standalone loop, *FOREVER* algorithmic form is placed into LACATRE representation, followed by algorithmic forms *WHILE* $cond_S$ *. . . END_WHILE* and *WHILE* $cond_S$ *. . . END_WHILE*. Analysis of state *S* is finished. The standalone loops will be translated inside the respective WHILE algorithmic forms.
- If there is a guarded outgoing transition from state *S* with guard $cond_S$ that starts a standalone loop, the algorithmic form *WHILE* $cond_S$ *. . . END_WHILE* is placed into LACATRE representation. Analysis of state *S* is finished. The standalone loop will be translated inside the *WHILE* algorithmic form and all full paths starting with the other guarded transition will directly follow the *WHILE* algorithmic form.
- If neither of the guarded outgoing transitions from state *S* starts a standalone loop, the algorithmic form *IF* $cond_S$ *THEN . . . ELSE . . . END_IF* is chosen.

5) LACATRE extensions

Each state machine in CRSM representation, although sequential in its nature, allows to express a non-deterministic behaviour of an environment or a part of system. For example, if several outgoing transitions in state *S* are ready to be executed at the same time, the selection will be made non-deterministically. However, even if only one of them were ready, there is still some kind of non-determinism, because at the time of entering Let's take several outgoing transitions described with input commands first. In this case the decision depends on which of the state machines, sending the messages, will be ready to communicate as the first one. That means, the channels should be sampled in a loop until one of them has been activated by a sending task. In OCCAM programming language, which is also (like CRSM notation) based on Hoare's CSP, such a behaviour can be described simply, using one ALT construction. Since we would like to keep LACATRE design, especially its graphical form, as simple as possible, it will probably be interesting to introduce two new algorithmic (*ALT* and *PAR*) into the graphical language. A parallel construction in a state machine (as by definition 7 in section 4. E) describes several activities (e.g. internal commands) that can be executed in any order, therefore, are independent of each other. That means, that they can, but do not have to, be executed in parallel. The only way, how to describe this behaviour in LACATRE is to execute them in parallel, that means as separate tasks. Of course this will result in some additional overheads (e.g. creation and deletion of new objects). To keep the LACATRE graphical representation legible the additional commands will be included only in textual form. Graphical form will display only original commands (those from CRSM translated into LACATRE) using the newly introduced *PAR* algorithmic form.

5. Illustrating Example

The chosen example is the Martian Lander system taken from [2]. The landing system oper-

Primitive Actions:

Action	Time in ms	Description
RACC	10	Starts IO device to read acceleration; sets DONE state predicate to FALSE
STMR	10	Starts hardware watchdog timer
IACC	10	Input measured acceleration
ADJM	20	Adjust motor thrust
TDP	10	Transmit info to display panel
IEM	10	Initiate emergency mode if DONE is FALSE by setting ELSM state predicate to TRUE
ETC	10	Other housekeeping function
IVEL	20	Input measured velocity
IALT	20	Input measured altitude
CKDT	10	Check input data for consistency
RRM	10	Retro-rocket module

Composite Actions:

Action	Description	Primitive Actions
N1	Phase 1 of Normal Operation of Landing System	RACC ; STMR
N2	Phase 2 of Normal Operation of Landing System	IACC ; (ADJM‖TDP)
TIH	Timer Interrupt Handler	IEM ; ETC
E	Emergency Operation of Landing System	(IVEL‖IALT) ; CKDT; RRM

State Predicates:

Predicate	Description
ELSM	Emergency landing system mode is on; initially FALSE
DONE	I/O status flag denotes no I/O in progress; initially TRUE

External Events:

Event	Description
START	System start-up
IOINT	I/O device signals completion; sets DONE to TRUE
TMRINT	Time Interrupt, occurs at least 100 ms after start of STMR

Timing Constraints:

a) While ELSM is off, execute N1 : period = 200 ms, deadline = 40 ms

b) When external event IOINT occurs, execute N2 with deadline = 60 ms, separation = 100 ms

c) When external event TMRINT occurs, execute TIH with deadline = 60 ms, separation = 100 ms

d) While ELSM is on, execute E : period = 200 ms, deadline = 100 ms

Fig. 8. Event-Action Specification.

ates in one of two modes: normal or emergency. Emergency mode is activated when some timing constraints are not met.

While in normal landing mode, the pilot can control acceleration, velocity, and position by adjusting the downward thrust generated by the rocket motor, thus bringing the space vehicle to a safe landing. This task is continually performed in two phases. In phase 1, an I/O device is started to read the acceleration set by the pilot and a hardware timer is started which generates a timer interrupt (i.e., external event TMRINT) after 100ms. In phase 2, when the I/O operation is completed (i.e., external event IOINT), the value of acceleration is read and the motor thrust is adjusted appropriately. However, if for some reason the I/O operation is not done within 100 ms, the timer interrupt initiates the emergency landing mode.

While in emergency landing mode, the altitude and velocity is periodically sampled and retrorocket is automatically fired to bring the vehicle to a safe landing.

Event-action specification of Martian Lander is given in figure 8. Based on this specification we described the Martian Lander using CRSM

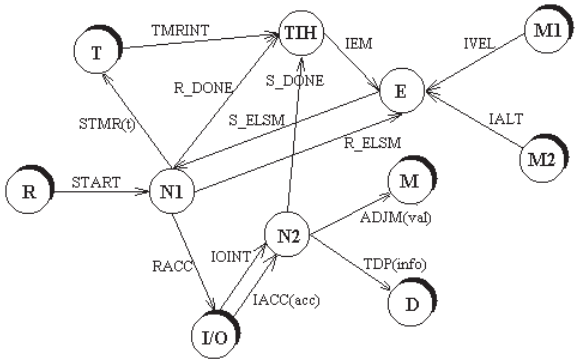


Fig. 9. Global CRSM Modelling.

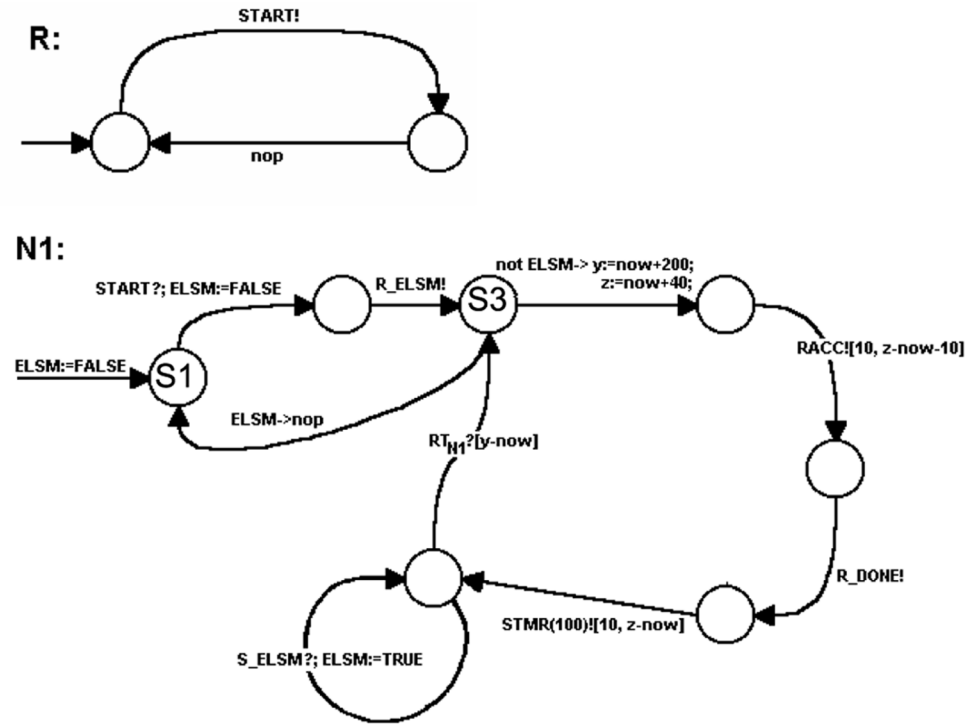
notation which was then translated (manually) into LACATRE.

In CRSM representation each composite action is described by one SSM. Primitive actions are represented, according to the context, either as messages sent down the channels or as internal activities. In the first case some ESMs are added to send or receive the messages. External events are also represented as messages and ESMs that generate these messages. State predicates represent local variables in some SSMs and their values are set according to the communication between these SSMs. Timing constraints are modelled in the same way as it was described in [4]. The global view of the system is given in Figure 9.

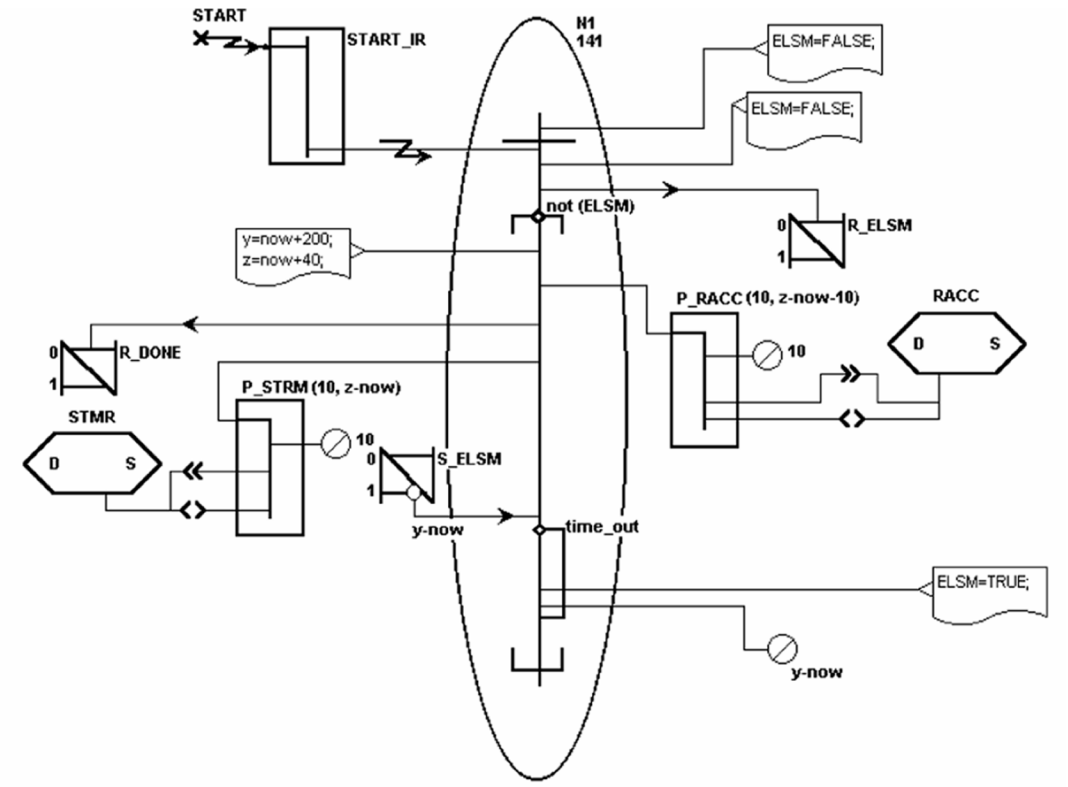
Figure 10 is extracted from the state machines of this system and their manual translation into LACATRE. The final target code is then straightforward to obtain. The figure shows the result of the translation applied to one phase of Normal Operation of Landing System (example of SSM).

6. Conclusion

The aim of this paper is to propose some rules for the translation from a CRSM specification of a real-time system to a multitasking execution model based on LACATRE. It has been shown that, by introducing some restrictions in the use of CRSM and probably some extensions in LACATRE, the translation is possible leading thus to a possible comparison of the specification behaviour and that of the execution model. The translation from LACATRE to CRSM, which is a much easier issue, is done as shown in [1].



CRMS model



Multitasking execution model

Fig. 10. Phase 1 of Normal Operation of Landing System (SSM).

Currently, the translation is hand-made by using the rules, some of which have been described in this paper. Work under process tries to show, thanks to more complex real time applications, that these rules and restrictions cover most of the current situations. The next step consists of building, with the help of a graphical environment, basic CRSM state "forms" and of associating them with basic LACATRE "patterns" in order to assemble them in a semi-automated assisted way.

References

[1] Z. HUANG, A. LEGAIT, M. MARANZANA, E. NIEL, J. J. SCHWARZ, J. SKUBICH, *About Techniques for the Verification of the Behaviour of Real-Time Multitasking Components*, IFAC'99 World Congress, Beijing 1999.

[2] F. JAHANIAN AND A. MOK, Safety Analysis of Timing Properties, in *Real-Time Systems, IEEE T.S.E.*, vol. SE–12, No. 9, pp. 890–904, Sept. 1986.

[3] J. J. SCHWARZ AND J. J. SKUBICH, Graphical Programming for Real Time Systems, *Control Eng. Practice*, vol. 1, No. 1, pp. 43–49, 1993.

[4] A. SHAW, Communicating Real-Time State Machines, *IEEE Trans. on Software Eng.*, vol. 18, pp. 805–816, Sept. 1992.

[5] J. A. STANKOVIC, K. RAMAMRITHAM, The Spring Kernel: A New Paradigm for Real-Time Systems, *IEEE Software* **8**(3): 62–72 (1991).

[6] J. A. STANKOVIC, K. RAMAMRITHAM, D. NIEHAUS, M. HUMPHREY, G. WALLACE, The Spring System: Integrated Support for Complex Real-Time Systems, *Real-Time Systems* **16**(2–3): 223–251 (1999).

[7] T. SZMUC, P. SZWED, J. J. SCHWARZ, AND J. J. SKUBICH, *Hierarchical Correctness Verification in Multiphase Real-Time Software Design*, IFAC WRTP, 1994.

[8] J. XU AND D. L. PARNAS, On satisfying Timing Constraints in Hard Real-Time Systems, *IEEE TSE.*, Vol. 19 n. 1, pp. 70–84.

Received: April, 2000
Revised: June, 2000
Accepted: July, 2000

Contact address:

Jean-Jacques Schwarz
Laboratoire L3i, B502
INSA de Lyon
20 Av A. Einstein, 69621
Villeurbanne, FRANCE
e-mail: jjs@iuta.univ-lyon1.fr

Katarina Jelemenska
Dept. Computer Science & Engineering
Slovak University of Technology
Ilkovicova 3, 812 19 Bratislava
SLOVAKIA
e-mail: jelemenska@dcs.elf.stuba.sk

Zhongwei Huang
Dept of computer science
Post Box 318
Harbin Institute of Technology
15001 Harbin, Heilongjiang
P.R. CHINA

JEAN-JACQUES SCHWARZ is a professor in the Department of Computer Science at the Technological Institute of the University of Lyon. He was head of the Industrial Computing Research Laboratory (L3i) at INSA de Lyon. His reserch interests primarly concern graphical programming and validation of complex multitasking real-time systems.

KATARÍNA JELEMENSKÁ was born in 1962 in Slovak Republic. She received her Msc. and PhD. from Slovak University of Technology Bratislava. She is an assistant professor at the Department of Computer Science and Engineering of the same university. Her scientific interests include fault-tolerance and reliability, real-time systems, means of hardware (and software) specification.

ZHONGWEI HUANG is an assistant professor in the Department of Computer Science at the Harbing Institute of Technology (PRC). His reserch interests include distributed real-time operating systems and structural correctness verifying of real-time sofware.

RÉGIS AUBRY (graduate of the Computer Sciences Department at INSA) is a senior lecturer of Computer Science at Institut National des Sciences Appliquées (INSA) de Lyon. His research interests include software engineering, complex system , software quality, dependability. He is responsible for software engineering, quality assurance and dependability courses.

JEAN-PHILIPPE BABAU is an assistant professor in the department of computer science at the INSA (engineer school) of Lyon. He received his PhD in computer science from the University of Poitiers in 1996. His research interests include the design, object-oriented techniques, and the analysis of temporal behavior for complex real-time systems.