Scenario and Property Checking of Real-Time Systems Using a Synchronous Approach

C. André, M-A. Peraldi-Frati, J-P. Rigault
Laboratoire Informatique Signaux et Systèmes (I3S)
Université de Nice Sophia Antipolis
CNRS UMR 6070
06903 Sophia Antipolis, France

May 22, 2001

Abstract

This paper addresses the design of control-dominated systems using a synchronous approach and the UML. The work aims at formally checking the design: scenarios/controller consistency, and safety properties. For this, a strengthening of UML behavioral models is necessary: SyncCharts are used instead of Statecharts, and Sequence Diagrams are modified by adding synchronously sound constructs akin to Message Sequence Charts. The formal foundations of the approach and the associated tools are briefly presented.

Publication: This paper was presented at the Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001), Magdeburg (Germany), 2-4 May 2001. In the proceedings of the conference, pp 438–444.

1 Introduction

We are interested in the design of control-dominated systems as used in real-time applications. Our approach relies on synchronous programming [6] and object-oriented modeling (UML).

In the UML, Sequence Diagrams and Statecharts [10] are generally used for expressing dynamic behavior of objects and classes. *Sequence Diagrams* express scenarios, which are rather informal and constitute partial examples of system usage. Moreover they often "*leave required properties about the intended system implicit*" [13]. *Statecharts* are a state-based representation of class and object behaviors. Although they rely on formal semantics, the evaluation of their semantics is complex and may induce undesirable non-deterministic input/output behaviors. Note that the current *UML* definition does not provide any form of semantic relationship between these two types of diagrams.

Introduced by the telecom community, *Message Sequence Charts* (MSC) [12] are a popular substitute for Sequence Diagrams. Their introduction in the UML has even been considered. They offer high level constructs like modularity, concurrency, iteration, ... There are many reports in the literature about analysis of MSC (*e.g.,* [1]) and associated tools (*e.g.,* uBET from Lucent). However, there exist several semantic interpretations of MSC and each of the above mentioned works relies on a particular one. Choosing "*a simple, yet expressive formal framework*", Krüger et al [11] proposed an automated translation from MCS to Statecharts. This is a formal attempt to bridge the gap between the scenario-based and the state-based models.

The work presented in this paper adopts a similar type of translation between scenarios and state-based models. It differs by choosing paragdims, hypotheses, and techniques that make it possible to check formal properties of the design.

Our underlying paradigm is the *synchronous programming paradigm* [6] which is based on a clear and deterministic semantics. The word "synchronous" may be misleading. It is in no way related to such concepts as "synchronous exception" or "synchronous rendez-vous".... It refers to strongly controlled executions of software in a way similar to synchronous circuits. The synchronous graphical model *SyncCharts*[1] [2] is substituted for the UML Statecharts in order to express the state model. Concerning the expression of scenarios, we introduce synchronous-oriented enrichments to the Sequence Diagram model. These extensions, relying on a formal synchronous semantics, are suitable to express typical control situations and are liable to automated processing. Thus, model checkers like XEVE [7] can be used to establish properties such as whether a scenario is executable or whether it is never possible.

[1]"SyncCharts" is the name of the model. A syncChart is an instance of this model.

Substituting SyncCharts for Statecharts may seem only a matter of personal convenience. Already published papers (*e.g.*, [4]) justify this choice. In the present paper we replace sequence diagrams with *Synchronous Interface Behavior* (SIB), which is a synchronous variant of MSC. Such a proposal is surprising because MSCs address distributed systems which are basically asynchronous. Indeed, we keep the MSC structuring power but we change its semantics for a synchronous one that is more suitable for tightly coupled agents. This is one of the objectives of this paper to point out the benefit of this approach in the field of modeling and validation of control-dominated systems subject to real-time constraints.

The paper is organized as follows. In section 2, we briefly describe the Synchronous approach. Section 3 introduces the *Synchronous Interface Behavior* model, its (graphical) syntax, and gives the flavor of its mathematical semantics. The use of SIB in scenario and property checking constitutes the next section. Realistic examples of SIB modeling are presented in section 5. As a matter of conclusion, we sum up the main features of the new model as well as our research in progress.

2 The Synchronous Approach

Synchronous languages have been introduced to address the issue of reactive programming. The synchronous approach adopts an *abstract* and *ideal view* of a system. Interactions take place at discrete instants. *Simultaneity* of occurrences is an unambiguous concept. The synchronous paradigm relies on two main hypotheses: Communications are supported by signals which are *instantaneously broadcast*; Reactions are generated *without a delay* and in *perfect synchrony* with the stimuli that triggered them. A noteworthy specificity of synchronous programming is the use of a multi-form logical time. Any event, and especially repetitive ones, may be considered as defining a time unit. Thanks to these strong hypotheses, synchronous languages have been given a clear and strict mathematical semantics so that the correctness of the design can be formally established.

Esterel is an imperative textual synchronous language and SyncCharts is a graphical form of Esterel. SyncCharts are clearly inspired by Statecharts but differ in several aspects. The SyncCharts semantics is fully synchronous and perfectly fits Esterel's semantics. The semantics of SyncCharts, relying on the synchronous hypotheses, is simpler than the Statecharts one (micro-step semantics). Moreover, SyncCharts offer a richer expression of pre-emption. SyncCharts are now fully integrated in the Esterel commercial platform[2]. As a consequence, SyncCharts have

[2]"Esterel Studio" marketed by Simulog.

direct access to the whole programming platform developed for Esterel: compilers, simulators (XES), model-checkers (XEVE), and circuit optimizers that rely on SIS and TIGeR (an efficient BDD-based tool).

3 The Synchronous Interface Behavior Model

Usually, Sequence Diagrams show the sequence of messages between objects. We have enriched the semantics of sequence at the object interface (corresponding to the events on the vertical line associated with an object). The model we propose, called *Synchronous Interface Behavior* (SIB), is a trade-off between complexity, expressiveness, and rigor. Basically, a sib [3] represents a *sequence of expected event occurrences* (signals in terms of synchronous modeling) as seen at a given controller-object interface. In what follows, signals and events are synonyms.

The SIB model has a textual and a graphical syntax. Both syntaxes can even be mixed, which is very useful in some applications. In this paper, we focus on the graphical notation.

Since a sib describes a partial observation of an object behavior, a set of "*observed events*" must be given. To capture time-related constraints or properties, some events are chosen as "*time-bases*" and their occurrences denote time passing. An evolution either can match a sib, or can fail because an observed event has occurred when not expected. In the former case the bottom of the sib is reached, the sib is said to be "*accepted*"; in the latter case, the sib is said to be "*not applicable*".

3.1 Basic Constructs

The flow of control is top-down in a sib. A vertical red line (black on the pictures) represents this flow. *Expected events* are denoted by solid red dots on this line. Fig. 1 shows that a signal can be received (input) or sent (output). The distance between two consecutive expects is meaningless, only the ordering is relevant, in accordance with the logical time used by synchronous models.

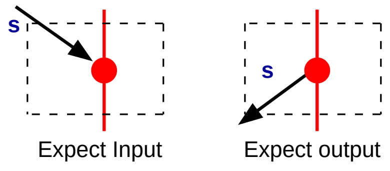

Figure 1: Expect.

[3]SIB is the model; a sib is an instance of this model.

SIB is a block-stuctured model. Expects are the bricks. The main construct is the sequence. In the figures below, sequences are drawn as green (grey on the pictures) vertical rectangles. A sequence is an ordered set of actions. **expect** is the simplest action. Any block described below is also an action.

3.2 Temporal behaviors

In real-time applications, event occurrences are temporally constrained. Binding logical time to real time can be done by relevant events (*e.g.*, a 1 kHz physical clock that generates signal "ms"). Multi-form (discrete) timing constraints are expressed by two special constructs: **within** (Fig. 2a) and **before** (Fig. 2b).

Basically, to be *accepted*, a sequence of actions must be completely matched within a temporal window for the **within** construct, and before a deadline for the **before** construct. Under the hypotheses of a synchronous approach which is fully deterministic, a strict application of these constraints would be too restrictive for high-level specifications. So, we have weakened the rules. First, we introduce a slack in the deadline occurrence: the deadline can occur at random within a given interval. Second, we permit departure from the rule of full matching of the sequence of actions. The trailing part of the sequence can be optional, that is if the deadline occurs while in the optional part, the sib is still considered to match, so far. Graphically, the optional part of the sequence is denoted by a dashed control flow line. Fig.2 shows the graphical notation for the within and before constructs. Below are examples with their (informal) semantics.

- "**do** p **within** 1..4 ms" says that the sequence of actions p must take place between the first and the fourth future occurrence of ms. Note that replacing ms with meter is semantically perfectly correct and allows generalized forms of time constraints: "**do** stop_the_car **within** 20 .. 30 meter".

- "**do** p **before** 2..4 ms" expresses another behavior: the sequence of actions p must terminate before a delay of 2, 3 or 4 occurrences of ms has elapsed. The deadline is clearly denoted by an (implicit) **expect**, at the exit of the before block. Taking a lower bound different from the upper bound of the delay is a way to introduce a restricted form of non determinism. At each instant between the lower and the upper bound, the deadline can or cannot occur, arbitrarily. In Fig. 2c, p is the sequence A;B followed by the optional sequence C;B. Sequences A;B, A;B;C, A;B;C;B are definitely accepted if they occur before the second future occurrence of ms. A then B occurring later than the fourth future occurrence of ms is definitely *not applicable*. A then B occurring between the second and the fourth future occurrence of ms, may or may not be accepted.

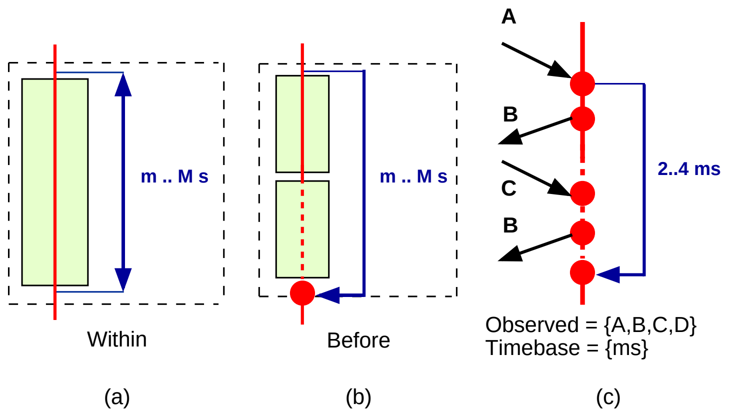

Figure 2: Temporal constraints.

3.3 Synchronous specific constructs

Synchronous modeling induces subtle issues, generally irrelevant to traditional approaches. Since instants are discrete, one can expect a stricly future occurrence of an event, or consider a possible present occurrence (*immediate* variant of **expect**). Also, since simultaneous occurrences are possible, one can expect a conjunction of event occurrences (*Conjunctive* expect). Our notation captures these nuances (see Fig. 3).

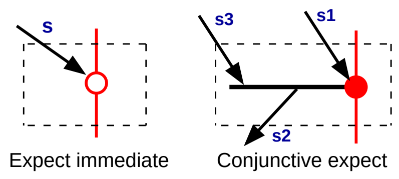

Figure 3: Variants of Expect.

3.4 Advanced constructs

A strict sequential representation of concurrent evolutions needs interleaving of events and induces “parasitic” ordering. A parallel construct is better at expressing partial ordering. Fig. 4 shows the **parallel** construct made of at least two sequences. This is a restricted form of concurrency (fork-join).

The **upto** construct, in Fig. 4, expresses alternative: One out of several sequences is taken. The taken sequence is the one whose guarding event occurs first. If several guarding events occur at the same instant, which is perfectly possible in a synchronous model, the left-most sequence for which the guard is satisfied, is taken. Thus, we have a deterministic choice. This construct has been called upto because, before the occurrence of the guard, the sib has been awaiting in an optional sequence, and stays there up to the occurrence of a guarding event.

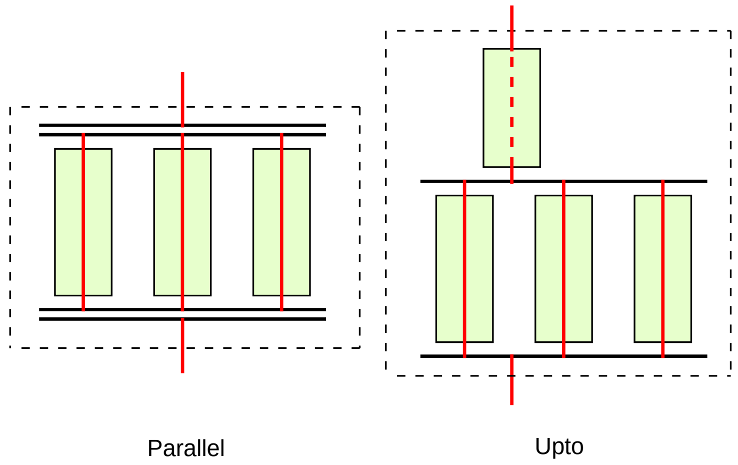

Figure 4: Advanced SIB (1).

Like with MCS, subsequences can be iterated. The **repeat** construct (Fig.5) allows folding of sequences. This is only "syntactic sugar" that denotes the unfolding of the loop.

A last construct is also very useful, especially in real-time systems and in protocols. We call it the **watchdog** construct (Fig.5). This block is abandoned on the occurrence of a deadline event, the solid red (black) dot on the exit of the block. As suggested by the picture, the watchdog construct is akin to the before block. The difference is that the former uses a disarming of the deadline: If the sequence in the watchdog block terminates before the deadline, then, the deadline is "re-armed". For flexibility, the number of occurrences may vary within an interval, and the sequence may have one optional part. An example of watchdog will be presented in section 5, Fig.6.

3.5 Mathematical semantics of the SIB

Let $\mathcal{I}$ be the set of signals seen by the application (*i.e.*, the controller). $\mathcal{I} \supseteq \mathsf{Observed} \cup \mathsf{Timebase}$. The set of output signals is $\mathcal{O} = \{\mathsf{Active}, \mathsf{Accepted}, \mathsf{Not\_Applicable}\}$.

At the j^{th} instant, let I_j be the current input event: $I_j \subset \mathcal{I}$ such that

$$\forall S \in \mathcal{I}, S \in I_j \iff S \text{ is present at instant } j$$

For a given sequence of input events $I_1 \; ; I_2 \; ; \cdots$, the behavior of a sib "p" is defined by a sequence of *reactions*:

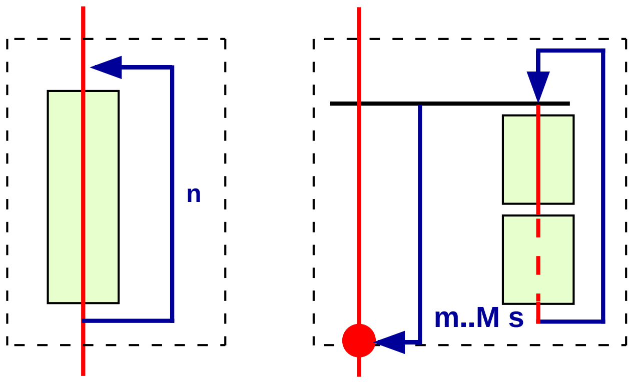

Figure 5: Advanced SIB (2).

$$\begin{aligned} p &= p_0 \xmapsto[I_1]{O_1} p_1 \xmapsto[I_2]{O_2} \cdots \xmapsto[I_n]{O_n} p_n \xmapsto[I_{n+1}]{\emptyset} 0 \xmapsto[I_{n+2}]{\emptyset} 0 \cdots \\ & \text{for some } n \in \mathbb{N} \cup \{\omega\}, \text{and} \\ & O \in \{\{\mathsf{Accepted}\}, \{\mathsf{Not\_Applicable}\}, \{\mathsf{Active}\}\}. \end{aligned}$$

If n is finite, the execution of p is said to terminate at instant n.

A reaction is computed by induction on the structure of the term. For this, we use an auxiliary relation (structural transition) defined by conditional rewriting rules. A structural transition is denoted by:

$$p \xrightarrow[E]{A,k,b} p'$$

where p is a term of the algebra, E the signal environment (the set of present signals), A the set of signals *accepted* by p under E. k is either an integer or ω, called the termination code, b is a Boolean that indicates whether the transition has been taken in an optional process, or not. p' is the residue of p after the rewriting. $k = \omega$ means that p has gone through a deadline.

$$p_j \xmapsto[I_j]{O_j} p_{j+1} \quad \text{iff there exists a rewriting} \quad p_j \xrightarrow[I_j]{A,k,b} p_{j+1}$$

O_j and the continuation depends on A and k. Let $I'_j = I_j \cap \mathsf{Observed}$ be the set of present signals at instant j restricted to the set of observed signals.

$$\begin{aligned}
O_j &= \{\mathsf{Not\_Applicable}\} \text{ and the execution terminates} \\
&\quad \text{if } (I'_j \setminus A \neq \emptyset) \vee (k = \omega) \\
O_j &= \{\mathsf{Accepted}\} \text{ and the execution terminates} \\
&\quad \text{if } (I'_j \setminus A = \emptyset) \wedge (k = 0) \\
O_j &= \{\mathsf{Active}\} \text{ and the execution pauses till the next} \\
&\quad \text{instant otherwise.}
\end{aligned}$$

Each primitive construct in SIB, is associated with an operation of the process algebra; then the semantics of the operation is given by conditional rewriting rules. This technique is often used in synchronous programming (see for example the semantics of the (pure) Esterel language [5] and the semantics of SyncCharts [2]). About thirty rules define the semantics of SIB, details are available in a research report [3].

4 Using SIB in Scenario and Property

A sib may be used for a better understanding of the behavior of the object, or for formally checking a property of the object. In both cases, the idea is to consider the sib as a specification of an *observer*. In synchronous programming, an observer [9] is a synchronous module, run in parallel (synchronous composition) with the controller (the synchronous program to check). The sib, or more accurately the associated module, observes the inputs and outputs of the controller. As soon as an unexpected event occurs, the sib module terminates and emits Not_Applicable. If the reactions of the controller match the whole scenario, then the sib module emits *Accepted*.

The interpretation of Accepted and Not_Applicable depends on the type of property to check, either in an existential or a universal form.

4.1 Existantial form

The simplest form is the *applicability* of a scenario: does the given sib match the input-output trace of a possible execution of the controller? To answer this question it is sufficient to show that Accepted is *possibly* emitted. Whenever Not_Applicable is emitted, the considered input sequence must be given up, and another one is tried. This is just to see if the controller we have designed can do what has been specified with the given sib.

4.2 Universal form

More interesting is checking *safety properties*. A safety property claims that "bad news" will never occur. A classical way to establish a safety property is to verify that no sequence leading to a violation of the property is applicable. So, it is sufficient to elaborate a sib that expresses the violation of the property. In this case, the Accepted signal indicates the violation. For all evolutions the sib must never match. The module associated with the sib is run in an infinite loop, so that each termination of the sib makes it restart again.

Bounded responsiveness (*i.e.,* an event B must occur in response to an event A, before the occurrence of an event C) is an instance of safety property often required in real-time applications. The **within** construct we have proposed is especially suited for expressing bounded responsiveness.

However, even if your design passes successfully all your simulation tests, you are not sure that a safety property holds. You need an exhaustive simulation of the controller behavior. Symbolic executions of the model can solve this problem. XEVE, the symbolic model checker, part of the Esterel distribution, does this job very well. A limitation is that signals must not convey values. This is the case for modules associated with a sib: SIB uses only pure signals (associated with event occurrences) and counters. When a safety property is violated, XEVE generates a counter-example input sequence. This sequence can be played back with XES in order to understand the flaw.

4.3 Higher description level

Since SIB has been given a semantics compatible with SyncCharts semantics, modules associated with sibs can be composed as SyncCharts macro-states. A high-level SIB (HLSIB) is an arbitrary complex composition of sibs using iteration, parallel composition and various preemptions. HLSIBs are at least as powerful as advanced MSCs and they are compilable into equivalent Esterel programs. The study of HLSIB is definitely beyond the scope of this paper.

5 Application of SIB

In order to illustrate the use of SIB in modeling and validation, we consider the design of a controller for a "premium car seat". It contains 6 motors, 28 sensors, driven and controlled by software. This challenge was proposed by Daimler-Chrysler. The main constraint is that "*For the modeling contest, the seat control software must be modelled and implemented in an object-oriented manner*". For a full description of the application, interested readers should refer to the web page (`http://www.automotive-uml.de`). We extract some typical scenarios and properties for illustrative purpose.

5.1 The calibration function : sequence, parallel and watchdog constructs

The controller of the seat must start periodically a calibration of the position of all the adjustment axes. The specification of the calibration function is defined as follows: "*The seat is calibrated in that all adjustment axes are moved to their stops. A stop is identified when no more ticks are supplied by the Hall sensor although the motor is supplied with voltage. Once the stop is reached, the motors remain activated for 250 ms so that the control unit can identify the end position.*"

Modeling this specification using a sib is almost straightforward. Let Observed = {Cal, Tick, LAon, LAoff, . . .} and Timebase = {ms}, where LAon, Laoff, . . . are commands sent to motor LA (Longitudinal Adjustment) . . . . For simplicity we only represent one branch of the parallel construct and we give the generic names Xon and Xoff to represent the motor commands.

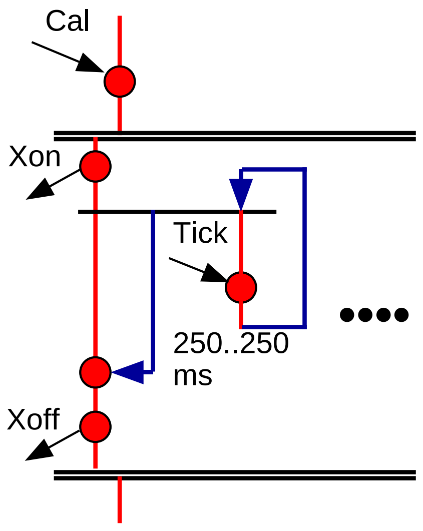

Figure 6: Sib of the calibration function.

In Fig. 6 we can see the sequence, the parallel and the watchdog constructs. Cal is the Calibration event. Since the calibration operates on all adjustment axes, as soon as the Cal event occurs, the six motors are started concurrently.

On the watchdog construct two signals appear:

- Tick, which is produced by the Hall sensors and indicates the adjustment evolution.

- ms, which is the time base for the watchdog

The watchdog is re-armed every Tick, and the stop position is reached when the 250 ms timer expires without receiving a Tick.

5.2 The memory function: sequence, within and upto constructs

The position of the seat can be stored in an EEPROM and restored using a control panel. This control panel contains four buttons: M, M1, M2, E. M is the button that starts the memory phase, M1 and M2 characterize the storage location, and E is used to restore a position. All of them have two states: pressed or released. The sib in Fig. 7a describes the following behavioral requirements: *"If the position of the seat is to be stored, first the memory button has to be pressed and then, within two seconds, the desired storage button . . . "*

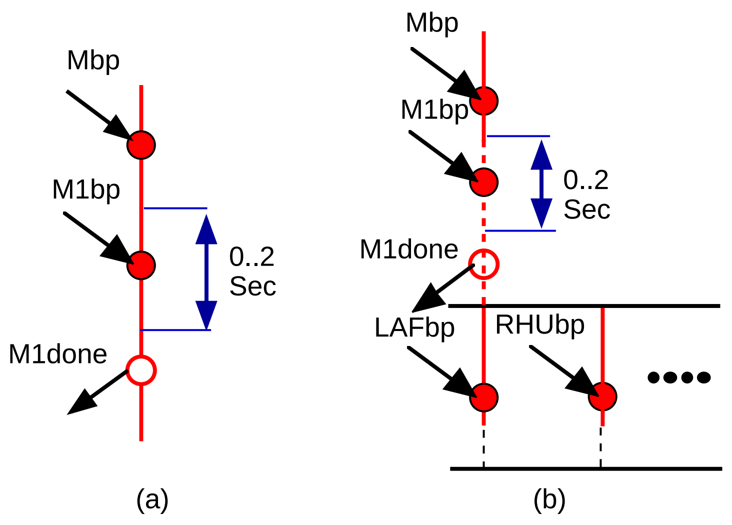

Figure 7: Sib of the memory function.

For simplicity, we consider only the storage associated with memory 1. Observed = {Mbp, M1bp, M1done} where bp stands for button pressed and M1done is the signal emitted when the memory process completes successfully. This signal is not part of the specification. Timebase = {Sec}. The sib in Fig. 7a can be interpreted as follows: as soon as Mbp is received, if M1bp occurs within 0 and 2 seconds, then the storage process completes successfully and M1done is emitted. The immediate expect allows the controller to emit the M1done signal, instantaneously or later.

For the same function an additional specification says that: *"The storage process is stopped when another button is pressed."*

This sentence raises the classical problem of real-time systems which is *preemption*. The SIB construct that expresses preemption is the **upto** construct. Possible events that may preempt the memory updating are those which activate the adjustment motors: Longitudinal Adjustment Forward (LAFbp), Rear Height Up (RHUbp) .... Note that in Fig.7b, **Expect** M1bp is now in an optional part (dashed line) and can be preempted without making the sib not applicable.

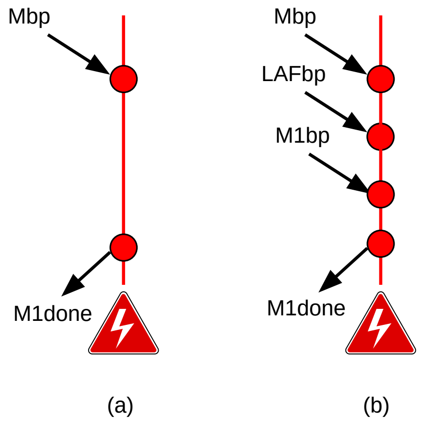

Figure 8: Safety properties.

The following safety property P1: "*Whenever Memory 1 is updated, a pre selection by M1 has occurred since the last occurrence of the activation of the memory function by M*" can be restated as "*Whenever M1done is emitted, M1bp has occurred since the last occurrence of Mbp*". The sib in Fig. 8a, where Observed = {Mbp,M1bp,M1done}, expresses the negation of this property. Notice the absence of M1bp in the sib, but its presence in the set of observed signals. In order to check P1, it is sufficient to show that this sib is never accepted. Given a controller and this sib, XEVE can easily do that.

P2 is another safety property that shows that memory 1 cannot be updated if an adjustment, say the "Longitudinal Adjustment Forward", occurs between the activation of the memory function and the selection of memory 1. In this case, take the sib in Fig. 8b, where Observed = {Mbp,M1bp,M1done,LAFbp}.

6 Conclusion

Our main objective is the design of safe controllers for critical reactive systems. The UML is now a standard methodology for the design of complex systems. In order to use it in reactive and real-time system design, well-founded models are necessary for expressing the dymanic behavior of classes and objects whereas ambiguous models are disqualified. Real-time UML, such as proposed in Douglass's book [8], makes use of enriched Sequential Diagrams with stereotypes taylored to real-time applications. We propose to go further and adopt two models that rely on a strict semantics: *Synchronous Interface Behavior* (SIB) for scenarios and SyncCharts for based-state model. In this paper we have addressed only the SIB model.

Indeed, *Sequential Diagrams* (SDs) are often used to express expected behaviors of the controller. Since Sequential Diagrams are not given a clear semantics, there is no way to formally validate the behavior they specify. To overcome this problem, we have introduced SIB as a substitute for SD. The semantics of SIB is mathematically defined in terms of a synchronous process algebra. SIB can be composed with SyncCharts, a synchronous state-based graphical model. A behavior expressed with SIB can be compiled into a semantically equivalent Esterel program. This program is then liable to validation either by interactive simulation (with XES) or by formal property checking (with XEVE).

Our first experience and the examples of SIB given in this paper address a real application and show the concisness and power of expression of the model. Yet we still have to assess its scalability and its user-friendliness:

- *Scalability*: The current translation from SIB to Esterel programs is a structural translation, with little optimization. The size of the generated code might be excessive for real-world systems. A second limitation is the size of the (symbolic) reachability set, which is used in property validations. The quantified timing constraints (within, before, and watchdog constructs) might cause rapid expansion of the state space.

- *User-Interface*: SIB and its semantics are strongly influenced by the synchronous approach. Some concepts may seem strange to users not familiar with the synchronous paradigm. A collection of typical examples should be available to convince potential users.

We believe that a better collaboration between the object paradigm (through UML) and the synchronous paradigm may facilitate the specification and design of embedded real-time controllers and the formal verification and validation of some of their properties.

References

[1] R. Alur, G. J. Holzmann, and D. Peled. An analyzer for message sequence charts. *TSoftware Concepts and Tools*, 17(2):70–77, 1996.

[2] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC.

[3] C. André. Synchronous Interface Behavior: Syntax and Semantics. Technical Report RR 00–11, I3S, Sophia-Antipolis, France, December 2000.

[4] C. André and M.-A. Peraldi-Frati. Behavioral Specification of a Circuit Using Syncharts: a Case Study. In *Euromicro 2000, Digital System Design*, pages 91–98, Maastricht (NL), September 2000. IEEE.

[5] G. Berry. Preemption in concurrent systems. *Proc FSTTCS, Lecture notes in Computer Science*, 761:72–93, 1992.

[6] G. Berry. The foundations of Esterel. In C. S. G. Plotkin and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[7] A. Bouali. XEVE: An esterel verification environment. volume 1427, Vancouver (BC, Canada), 1998. Int'l Conf. on Computer-Aided Verification (CAV'98), LNCS. also available as a technical report INRIA RT-214, 1997.

[8] B. P. Douglass. *Doing Hard Time*. Object technology series. Addison-Wesley, Reading, Massachusetts, 1999.

[9] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Amsterdam, 1993.

[10] D. Harel. STATECHARTS: A visual formalism for complex systems. *Science of computer programming*, 8:231–274, 1987.

[11] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From mscs to statecharts. In F. J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[12] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on message sequence charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, December 1996.

[13] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Tr. on Software Engineering*, 24(12):1089–1114, December 1998.