Reactivity, concurrency, data-flow and hierarchical preemption for behavioural animation

Stéphane Donikian, Eric Rutten

Reactivity, concurrency, data-flow and hierarchical preemption for behavioural animation

Stéphane Donikian and Éric Rutten

IRISA

F-35042 RENNES, France

donikian@irisa.fr, rutten@irisa.fr

Abstract

Behavioural models offer the ability to simulate autonomous entities like organisms and living beings. Such entities are able to perceive their environment, to communicate with other creatures and to execute some decided actions either on themselves or on their environment. Building such systems requires the design of a reactive system treating flows of data to and from its environment, in a complex way needing modularity, concurrency and hierarchy, and involving task control and preemption. Accordingly, in this paper we address the adequateness to the decisional part of the behavioural model of the following programming paradigms: reactivity, concurrency, data-flow and hierarchical preemption.

The reactive languages provide users with complete design environments (including graphical tools for designing, simulating, implementing and formally verifying) for such systems. The specification of concurrent behaviours is naturally supported in the synchronous languages, and some of them are more suited for control-intensive applications (sequencing and preempting tasks), while others address computation-intensive applications (data-flow). SIGNAL GTi is an extension of the language SIGNAL where data-flow processes can be composed into nested preemptive tasks.

An application in the simulation of a transportation system shows how these programming paradigms can be of use, and how SIGNAL GTi can support their implementation.

Keywords: Simulation, Behavioural Animation, Reactive Systems, Synchronous Languages, Concurrency, Data-flow, Hierarchical Preemption.

1 Introduction

Context. Behavioural animation consists in a high level motion control of dynamic objects, which offers the ability to simulate autonomous entities like organisms and living beings. Such entities are able to perceive their environment, to communicate with other creatures and to execute some actions either on themselves or on their environment. This requires to make a deliberative choice of the behaviour, and for that we have to design a *reactive system* continuously in communication with the environment (*data-flow*). The behaviour of a creature, even for the simplest, is composed of different activity lines (*modularity*) which can be completely distinct but also *concurrent*. *Hierarchy* and *preemption* are some other important notions for such a system, because they enable them in a modular and re-usable way, defining behaviours from sub-behaviours, sequenced, interrupted or suspended by preemption.

Reactive languages (and particularly synchronous ones) handle these programming paradigms in different forms. Additionally, they open up perspectives concerning the formal verification of safety-critical behaviours, the distributed compilation of applications, ... However they do so by concentrating each on one or two paradigms, not integrating them all. The tasking extension of SIGNAL called SIGNAL GTi integrates the concurrency and data-flow reactivity of SIGNAL with notions of sequencing and hierarchical preemption of tasks. Hence, it is a fitting candidate for the programming of the class of applications evoked above.

Behavioural animation. The objective of animation is the calculation of an image sequence corresponding to discrete time states of an evolving system. Animation consists at first in expressing relationships linking successive states (specification phase) and then making an evaluation of them (execution phase). Motion control models are the heart of any animation/simulation system that determines the friendliness of the user interface, the class of motions and deformations produced, and the application fields. Motion control models can be classified into three general families : descriptive, generative and behavioural models [15]. Descriptive models are used to reproduce an effect without any knowledge about its cause. This kind of models include key frame animation techniques and procedural methods. Unlike preceding models, generative models offer a causal description of objects movement (describe the cause which produces the effects), for instance, their mechanics. In this case, the user control consists in applying torques and forces on the physical model. Thus, it is not easy to determine causes which can impose some effects onto the mechanical structure to produce a desired motion. Two kinds of tools have been designed for the motion control problem: loosely and tightly coupled control. The loosely coupled control method consists in automatically computing the mechanical system inputs from the last value of the state vector and from the user specification of the desired behaviour, while in the other method, the motion control is achieved by determining constraint equations and by inserting directly these equations into the mechanical system motion equations. Motion control tools provide the user with a set of elementary actions, but it is difficult to control simultaneously a large number of dynamic entities. The solution consists in adding a higher level which controls the set of elementary actions. This requires to make a deliberative choice of the object behaviour, and is done by the third model named *behavioural*. The goal of the behavioural model is to simulate autonomous entities like organisms and living beings. A behavioural entity possesses the following capabilities: perception of its environment, decision, action and communication. As mentionned above, paradigms needed for programming a *realistic* behavioural model are reactivity, data-flow, modularity, concurrency and hierarchical preemption.

Synchronous reactive languages, SIGNAL **and** SIGNAL*GTi*. *Reactive systems* are characterized by the fact that they are in constant interaction with their environment: their pace is determined by discrete events to which they react. The *synchronous languages* are derived from theoretical and applied studies on discrete event systems with real time aspects, and on specification methodologies and programming environments for their development [7, 14]. Synchrony consists in that the languages have a notion of simultaneous events, and that synchronous communications allow processes to synchronize their transitions on shared instants. This model of time is the basis of clear semantics, a model of concurrency easier to handle than the asynchronous one, and efficient analysis tools. These languages evolved into a technology supported by commercial products that are used in industrial applications. Their aim is to support the design of safety critical applications, especially those involving signal processing and process control. The synchronous approach guarantees the determinism of the specified systems, and supports techniques for formal verification (like the detection of causality cycles and logical incoherences). A family of languages is based on this approach [14], featuring ESTEREL, LUSTRE, SIGNAL, ARGOS and also STATECHARTS.

Among them, SIGNAL [18] is a data-flow language, with a declarative style: processes are systems of equations on values and their occurrence instants. The compiler transforms the specification into an optimized executable code (in C, FORTRAN or ADA) computing and outputting the solutions to this system of equations at each reaction. Automated formal verification of dynamic properties of programs is based on polynomial dynamic systems. Experimentations of the specification, simulation and verification methods have concerned the controller for a robotic production cell, an active robot vision system [20], and a control system for a power transformer station [21]. A recent extension to SIGNAL provides constructs for the specification of *nested preemptive tasks* on intervals of time [28, 29]. The model of time in SIGNAL is based on instants, and actions of the language are executed within the reaction. It is not particularly suited to the specification of systems commuting between activities, involving the abortion or suspension of tasks with a duration. In particular, there are no constructs enabling directly the specification of complex behaviours with preemption on hierarchies of tasks and sub-tasks. This is why notions of task and of associated time interval were

introduced in the language. This extension is implemented in Signal GTi[1][30], a pre-processor to Signal, fully compatible with the tools in the environment, and hence benefitting from their functionalities such as proof, code generation in various languages, ...

Organization of the paper. The nature of behavioural animation is presented in the following section. The section 3 presents an application which requires *Realistic* Behaviours. In the section 4, a set of characteristics needed for the specification of a virtual driver are listed and a Modular and Hierarchical Behavioural Model integrating these characteristics is presented. Section 5 gives a presentation of the synchronous reactive data-flow language Signal, and of its extension with hierarchical preemptive tasks Signal GTi. Next, section 6 discusses the adequacy of the paradigms proposed by Signal GTi for the description of our behavioural model, with the example of a driving simulation system.

2   Behavioural Animation

There are two kinds of behavioural models: internal transformation models (plant growth, facial animation, ...) [9, 24] and external animation models (animals, humans; alone or in group) [5, 6]. In this paper, we are not interested by internal models and in the following, we will only discuss external ones, which are based on two kinds of relationships between the object and its environment: perception and action (cf figure 1). In this case, the temporal dependency relation of the state of an entity $E(t_i)$ at the time t_i is globally expressed by a function $f(E(t_{i-l}), \ldots, E(t_{i-1}), P(t'_{j-k}, \ldots, t'_j), C(t_i), t_i)$, where $C(t_i)$ is the set of messages received by the entity at time t_i and P represents the environment perception function during the temporal interval $[t'_{j-k}, \ldots, t'_j]$, with $t'_j \leq t_i$.
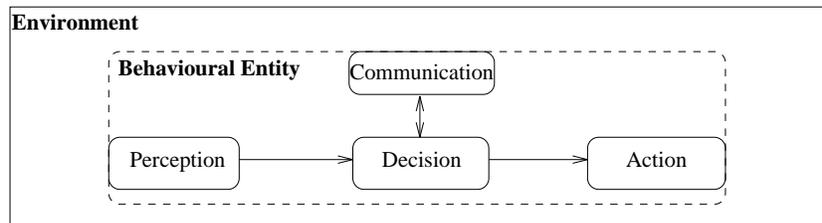

Figure 1: A behavioural agent immersed in its environment.

Different approaches have been studied for the decision part of these models [10]:

Sensor-effector Approach The behaviour of objects is defined by sensors, effectors and a neural network which connects them. The way an object behaves in the environment depends on how the environment is sensed, and how this information is passed through the neural network to the effectors that produce the motion of the object. The same neural network can produce different kinds of motion depending on the parameterization of nodes and on the weight of connections. [33, 35]

Behaviour Rule Approach Like the previous approach, it takes sensed information as its input and motor controls as its output, but the behaviour of the object is controlled by a set of behaviour rules. The possible behaviours can be represented by a decision tree in which each branch represents one alternative behaviour [32, 16, 25, 31, 34]. This method offers a higher level of description than the preceding one, but the difficulty results from the choice of the rating strategy.

Predefined Environment Approach This approach is based on the fact that the environment is predefined and static: thus, all possible paths from the initial position to the goal position can be explored. It is easy to determine an optimal solution inside the set of enumerated solutions, according to behaviour

---

[1] which stands for *Gestion de Tâches et d'intervalles*, the french for: tasks and intervals management.

criteria, but on the other hand a little change in the environment implies a complete recomputation of the motion [26].

Finite Automaton Approach In this approach, the behaviour of an object is either controlled by one finite automaton or by combining elementary behaviours and designing a supervisor for the resulting composite behaviour. The use of a single automaton is not convenient: firstly making conceptually simple changes in behaviours requires widespread modification of the finite state machine, secondly it is difficult to express concurrent constraints on control processes [8]. Modularity, hierarchy and concurrency can be obtained by using Hierarchical Concurrent State Machines (HCSM) [1]. We have also described the behaviour of a car-driver by using a hierarchical modular system, in which the higher level (supervisor) is principally composed by a hierarchical parallel automata [11, 3].

Most of these systems have been designed for some particular examples, in which modularity and concurrency are not necessary. This is due to the fact that entities possess only one activity line and because possible interactions between an object and its environment are very simple: sensors and actuators are reduced to minimal capabilities which, most of the time, permit only to avoid obstacles in a 2D or 3D world. Another point which is generally not treated is the notion of time. In a system mixing different objects described by different kinds of models (descriptive, generative and behavioural), it is necessary to take into account the explicit management of time, either during the specification phase (memorization, prediction, action duration, etc.) or during the execution phase (synchronization of objects with different internal times).

3 An application which requires *Realistic* Behaviour: Driving Simulation for the Praxitele Project

3.1 The Praxitele Project

The Praxitele project combines the efforts of two large government research institutes, one in transportation technologies (INRETS), the other in computer science and automation (INRIA), in cooperation with large industrial companies (RENAULT, EDF, CGEA). This project designs a novel transportation system based on a fleet of small electric public cars under supervision from a central computer [23]. These public cars are driven by their users but their operation can be automated in specific instances. The system proposed should bring a solution to the congestion and pollution in most cities through the entire world.

The concept of a public transport system based on a fleet of small electric vehicles has already been the subject of experiments several times but with poor results. The failure of these experiments can be traced to one main factor : poor availability of the vehicles when a customer needs one. To solve this main problem, Praxitele project develops and implements automated cooperative driving of a platoon of vehicles, only the first car is driven by a human operator [22]. This function is essential to move easily the empty vehicles from one location to another.

The realization of such a project requires experiments of the behaviour of autonomous vehicles in an urban environment. Because of the danger of this kind of experiments in a real site, it is necessary to design a virtual urban environment in which simulations can be done. Our simulation platform permits to simulate a platoon of vehicles evolving in a virtual urban environment and so to test control algorithms of the automated cars.

3.2 A Simulation Platform

With the intention of making the three kinds of motion control models work together, we have been interested by their integration into a single system and we have proposed the architecture of a simulation platform [11]. The simulation platform is composed of a set of *agents/actors* whose synchronization and communication are managed by a real-time kernel. The main part of this kernel is the general controller. Communication between the agents is both synchronous and asynchronous. The synchronous part is data-flow based where

each agent has its own frequency and is managed by the general controller. So, the data-flow communication channels include all the mechanisms to adapt to the local frequency of the sender and receiver agents (over-sampling, sub-sampling, interpolation, extrapolation, etc...). The asynchronous part is based on two mechanisms :

- event based communication between agents and the general controller.

- global data-structure updating and accessing mechanisms.

Different experimentations have been realized for the car-driving application:

- A mono-user modular version in which synchronization and control are specified in the synchronous reactive language SIGNAL.

- Two multi-processes versions by using PVM (Parallel Virtual Machine) [13] and Chorus (micro-kernel technology) [27].

A first version of the simulation platform is currently in developpment. PVM is used to define the real-time kernel which is in charge of communication and synchronization between different processes, while SIGNAL is used to assume internal management of each process.

3.3   Simulation of an Urban Environment

An urban environment is composed of many dynamic entities evolving in a static scene. These dynamic entities have to be both autonomous and controllable and also realistic in term of behaviour. It is necessary to combine the three motion control models to describe dynamic entities of the environment. For example, to describe traffic lights it is not necessary to use a generative model when a descriptive model (finite state automata) is sufficient. On the other hand, for a realistic car driving, we need both generative and behavioural models (the first one to simulate the dynamic of the vehicle and the second one to simulate the driver).

The static scene.   As we want to control entities evolving, we need to link dynamic entities with the static scene in which they are moving. This link requires a semantic knowledge on the scene. If we want to simulate as completely as possible the life of a city, we need a lot of semantic informations. Informations required for the simulation are:

- geometric (geometry of the town, roadsigns),

- topologic (the road network, a grid of visibility),

- semantic

    - road informations: roadsigns, color of traffic lights, qualitative aspect of the road, ...
    - city informations: name of streets, quarters, particular buildings, squares, ...

To describe such a scene, an urban modeling system is currently in development. This system is an extension of the Scriptography (Declarative Design System) [12], in which geometric, topologic and semantic informations are mixed.

Dynamic objects.   To take into account natural phenomena, the first work is to choose a physical model to represent the object. A vehicle is an articulated rigid object structure if we do not consider the deformations of the tires and the component flexibility [4]. From a high level description of articulated rigid body systems, a simulation blackbox is generated whose inputs are two torques (motor and guidance) and outputs are position and orientation parameters. We have now to determine how to control this physical model that is to say, depending on the actual state of the entity what kind of torque must we apply to it to obtain the desired motion ? In the case of an automatic motion control, this question can be decomposed in two parts:
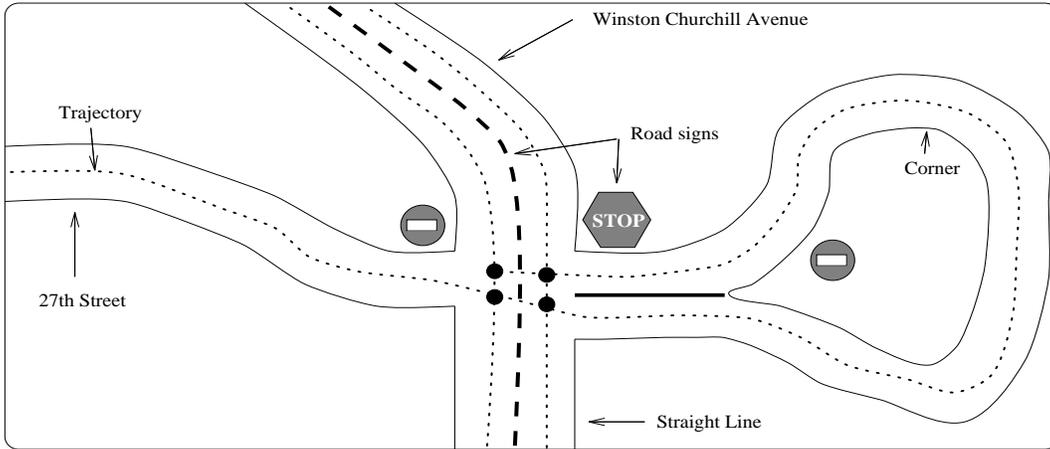
Figure 2: Geometric, topologic and semantic informations.

1. how to control the physical model?

2. what is the desired motion?

The answer to the first question consists in using motion control algorithms [19] well known in the automatic and robotic communities, which can permit to build a library of elementary actions. The behavioural model tries to answer to the second question by defining actions and reactions of an entity [32, 11, 17].

4   A Virtual Driver

4.1   A Modular and Hierarchical Behavioural Model

We have chosen to define a modular and hierarchical behavioural model, as shown in the figure 3, to specify the behaviour of a car driver. This model is decomposed in a set of specialized agents who use themselves some experts to reconstruct the new state of the world, before proposing their diagnosis to the decisional agent. Therefore, the supervisor decides to activate some of the specialized agents and this decision depends on its own state and on sensor data. The *road signs* module is, at the moment, in charge of determining the value of three parameters: speed limitation (real value), overtaking (YES | NO) and crossroads priority (right of way | priority to the right way | stop | traffic lights). The *itinerary* module is in charge of determining the new direction at each crossroads, so this module is only activated when the vehicle is near one of them. The *obstacle detection* module has to determine if there are some possible intersections between the vehicle desired trajectory and predicted trajectories of other vehicles, and if so to propose a new trajectory. Actions managed by the *state feedback control* module consist in determining the guidance torque (follow a desired trajectory) and the motor torque (accelerate, brake, stop, etc). In order to simplify calculation, the human vision is not completely simulated but is replaced by a global knowledge on the scene geometry and on the location of objects, then by using visual sensors we obtain qualitative informations about objects in the vision cone.

The work of the decisional model is to compare and mix different possible behaviours according to the desired behaviour and then to make a decision of the adopted behaviour. To deal with complex and concurrent behaviours, we have proposed to use hierarchical parallel transition systems to describe the decisional model of the driver. The use of hierarchical parallel transition systems allows us to take into account concurrency and abstraction in the description of the behaviour, like Kearney et al. with their hierarchical concurrent state machines [17, 1]. Because of the integration of our behavioural model in a
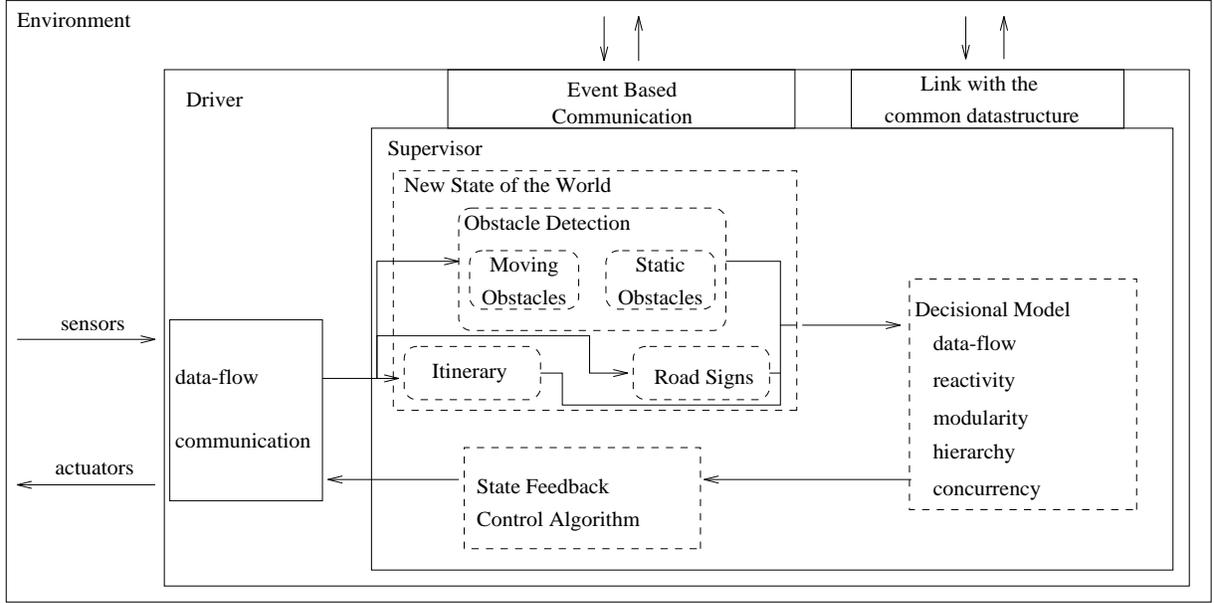
Figure 3: Modular and hierarchical structure of the driver.

simulation platform, we have also the ability to deal with real time during the specification and the execution phases.

4.2 Hierarchical parallel transition systems for decision

The Decisional model consists in a reactive system, which is composed of a hierarchy of state machines (possible behaviours). Each state machine of the system can be viewed as a blackbox with an In/Out data-flow and a set of control parameters. It can be defined by the following tuple $< S, \Gamma, IS, OS, CP, LV, IF >$ in which:

S : is a set of sub-state machines,

Γ : is the activity function,

IS : is a set of Input Signals,

OS : is a set of Output Signals,

CP : is a set of Control Parameters,

LV : is a set of Local Variables,

IF : is the integration function.

State Machine. Each state machine of the system is either an atomic state machine ($S = \emptyset$), or a composite state machine. An activity parameter is associated to each state machine which corresponds to the current status of this machine. This parameter is accessible by using the function *status(state,instant)*, which has five possible values:

$$(\forall s \in S)(\forall k \in \mathcal{N}), status(s, time(k)) \in \{started, active, suspended, terminated, idle\}$$
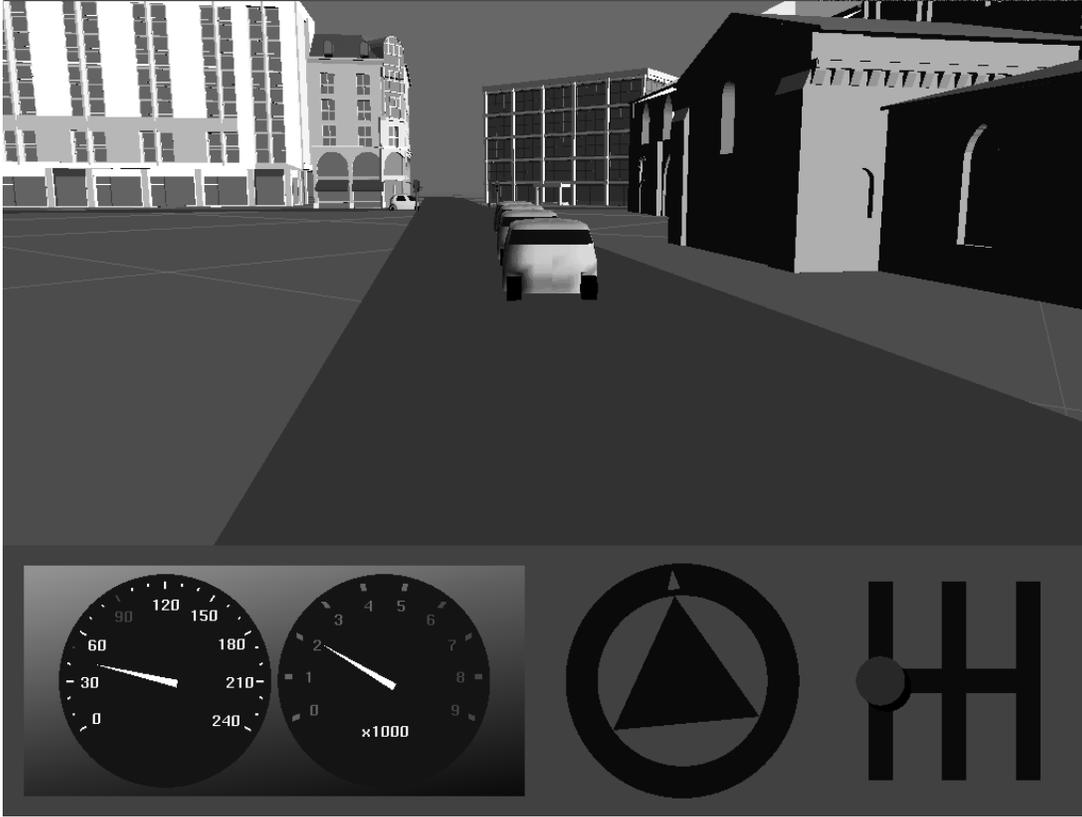
Figure 4: One example of obtained simulation: the traffic light being red on his way, the user decides to stop the first car of the platoon, then the three others decelerate also, while an automatically driven car is passing through the crossroad.

The activity function. Activity of a state evolves during the simulation, and this is determined by an activity function Γ. This function possesses four parameters and returns the new status.

$$(\forall s \in S)(\forall k \in \mathcal{N}^*), status(s, time(k)) = \Gamma(status(s, time(k-1)), IS, CP, LV)$$

This function permits to represent some transitions between different sub-state machines, but more than one sub-state can be active at each instant (concurrency). For example, in the figure 5, *Driving* and *Road Shape* sub-state machines work in parallel. This function handles also hierarchical preemption, by the fact that one argument of the function is a set of Control Parameters (CP) which permits to deal with internal events as "substate i is *terminated*" or external events as "*an ambulance is coming*".

Input / Output parameters. Input and output parameters correspond to continuous signals of standard type (integer, real, boolean, ...). The value of an Output signal is undetermined when the state machine is *idle* or *suspended*. For the driver example, some of the input signals of the *Automatic Driving* State Machine and of its sub-state machines are shown on the figure 6. In this example, there are two output signals: *Guidance action* and *Motor action*; the first one determines what lane has to be followed and actions are for example *turn left* or *filter to the right*, while the second one determines what kind of action must be applied to the vehicle motor and can be for example *brake* or *cover(distance, delay)*.
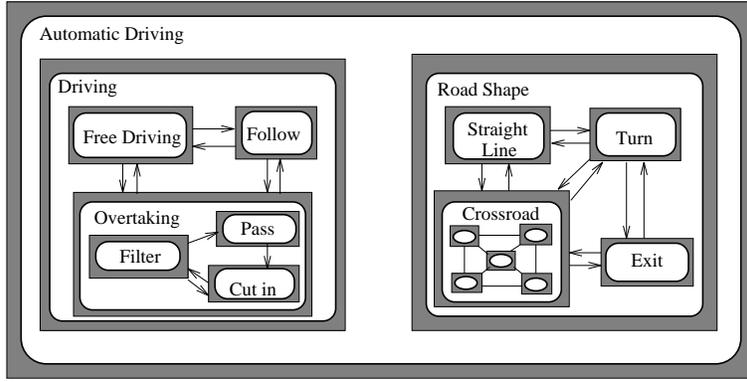
Figure 5: State transition aspect of the state machine.

Local variables. Local variables are some variables of standard type, whose value is computed by using the Ω function. Local variables can either retain their values between activations or be reinitialized on each reactivation (*started* status).

$$(\forall v \in V)(\forall k \in \mathcal{N}^*), value(v, time(k)) = \Omega(value(v, time(k-1)), status(s, time(k-1)), IS, CP)$$

Control parameters. Control parameters permit to modulate the behaviour of an entity, depending on external or internal decision. The type of a control parameter is either boolean or interval (*value* $\in$ $[Vmin, Vmax]$). For example, a sub process can inform its parent process that its status become *terminated*, while a process can notify another subprocess that it has to be *started*.

The integration function. The integration function has to manage the coherence of the actions proposed by the different sub-processes, and make a synthesis of them. This is in fact a function which takes as inputs the outputs of all sub-processes and delivers the value of the process outputs. In the case of concurrent behaviours proposed by different sub-state machines, this function has to make a choice and to deliver a unified behaviour as output of the state machine (cf figure 6). Let E be $< S, \Gamma, IS, OS, CP, LV, F >$.
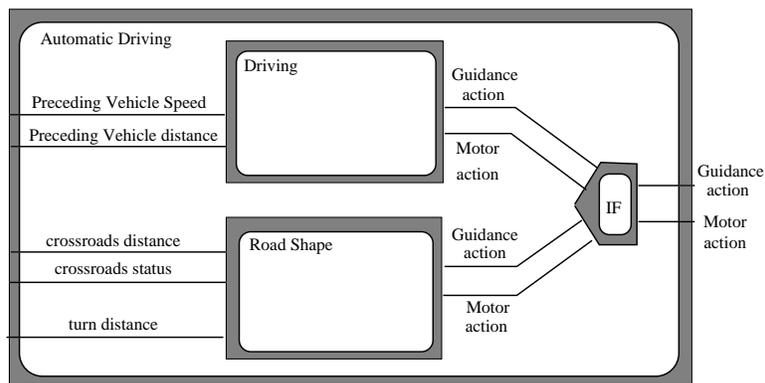
$$OS = IF(output(S), LV, CP)$$

Figure 6: Data-flow aspect of the state machine.

4.3　Conclusion

Characteristics needed for the specification and for a high level programming of the behavioural model are:

- reactivity, which encompasses sporadic or asynchronous events, exceptions, ...

- modularity in the behaviour description, which allows for parallelism and concurrency of sub-behaviours,

- data-flow, for the specification of the communication between different modules,

- hierarchical structuring of the behaviour, which means the possibility of preempting sub-behaviours on transitions in the meta-behaviour, as a kind of exception or interruption. It means also that sub-behaviours can notify the meta-behaviour of their activity.

This combination of data-flow and event-driven features is not present in many languages; MANIFOLD is one of them, a language for the coordination of processes through a data-flow network, which can change state as a result of the occurrence of an event. Its behaviour is asynchronous, and can be defined in terms of transition systems [2]. In the synchronous approach to reactive systems, there exists a combination of the languages LUSTRE into ARGOLUS, where a state in the hierarchical automata language ARGOS can be refined into a data-flow process in LUSTRE, and in a LUSTRE data-flow network, a node can be refined into an ARGOS automaton. In our approach, we chose the data-flow synchronous language SIGNAL, and its extension with hierarchical preemption tasks: SIGNAL GTi, both presented in the next section. In section 4, we will illustrate that this language can answer most of the requirements of a behavioural model combining the preceding characteristics.

5　Nested preemption of reactive data-flow tasks: SIGNAL GTi

5.1　The synchronous reactive data-flow language SIGNAL

As said in Section 1, SIGNAL is a synchronous real-time language, data flow oriented (i.e., declarative) and built around a minimal kernel of operators [18]. It manipulates signals, which are unbounded series of typed values (**integer**, **logical**, ...). They have an associated clock determining the instants where values are present. For instance, a signal **X** denotes the sequence $(\mathbf{x}_t)_{t \in \mathrm{IN}}$ of data indexed by time-index t. Signals of a special kind called **event** are characterized only by their clock i.e., their presence (they are given the boolean value **true** at each occurrence). Given a signal **X**, its clock is **CX** obtained by **CX := event X**, giving the event present simultaneously with **X**. The constructs of the language can be used to specify, in an equational style, relations or constraints between signals i.e., between their values and between their clocks. Systems of equations on signals are built using composition.

The kernel of SIGNAL comprises the following *primitive processes* and their composition:

Functions are defined on the types of the language (e.g., boolean negation of a signal **E: not E**). The signal (Y_t), defined by an instantaneous function f: $\forall t, Y_t = f(X_{1_t}, X_{2_t}, ..., X_{n_t})$ is specified in SIGNAL by: $\boxed{\texttt{Y := f\{ X1, X2, ... , Xn\}}}$. The signals **Y, X1, ..., Xn** are constrained to have the same clock.

Delay gives the past value of a signal X (i.e. at instant $(t-1)$ of its clock): $ZX_t = X_{t-1}$, with initial value V_0 (i.e., when $t = 1$: $ZX_1 = X_0 = V_0$). It is specified by: $\boxed{\texttt{ZX := X\$1 init V0}}$. **X** and **ZX** have the same clock.

Selection of a signal **X** according to a boolean condition **C** is: $\boxed{\texttt{Y := X when C}}$. Signal **Y** is present if and only if **X** and **C** are present at the same time , and **C** has the value **true**. When **Y** is present, its value is that of **X**. In other terms, the clock of **Y** is the intersection of that of **X** and the subset of that of **C** when **C** has the value **true**.

Deterministic merge defines the prioritary mixing of two signals of the same type. It is written as follows: `Z := X default Y`. The value of `Z` is the value of `X` when it is present, or else that of `Y` if it is present and `X` is not (i.e., priority is given to `X`). The clock of `Z` is the union of that of `X` and that of `Y`.

Parallel composition of processes is made by the associative and commutative operator "|", denoting the union of the underlying systems of equations. Systems communicate and interact through signals defined in one system and used in others. For these signals, composition preserves constraints from all systems, especially temporal ones. This means that they are present if the equations systems allow it. In SIGNAL, for processes P_1 and P_2, composition is written (with parentheses): `(| P1 | P2 |)`.

Furthermore, it is possible to confine signals locally to a process. This is done with the restriction operator "/": restricting signal `X` to process P is written `P / X`.

The following table illustrates each of the primitives with a trace:

X	-1	2	6	3	-5	12	7	-3	-8	13	...
Y := X + 1	0	3	7	4	-4	13	8	-2	-7	14	...
ZY := Y$1 init 0	0	0	3	7	4	-4	13	8	-2	-7	...
PY := ZY when ZY>0			3	7	4		13	8			...
Z := PY default (0 when (event X))	0	0	3	7	4	0	13	8	0	0	...

The rest of the language is built upon this kernel . A structuring mechanism is proposed in the form of **process** schemes, defined by a name, typed parameters, input and output signals, a body, and local declarations. Instances of processes in a program are expanded by a pre-processor of the compiler. Derived operators have been defined from the primitive operators, providing programming comfort. E.g., `synchro{X,Y}` constrains the signals `X` and `Y` to be synchronous, i.e. their clocks to be equal. The process `CB := when B` gives the clock `CB` of occurrences of the logical signal `B` at the value **true**. The process `Y := X cell B` memorizes values of `X` and outputs them when `B` is true. In the process `C := # X`, `C` is an integer counter of the occurrences of signal `X`. Delays can be made of `N` instants, or on windows of `M` past values. Arrays of signals and of processes are available as well.

The SIGNAL compiler performs the analysis of the consistency of the system of equations (absence of causal cycles), and determines whether the synchronization constraints between the signals are verified or not. If the relational program is constrained enough to be a function computing a deterministic solution, then executable code can be produced automatically (in C, FORTRAN or ADA). The compiler can also produce output for the SYNDEX system, where the quantitative analysis of computation times can be performed, as well as the distribution of the code on distributed, multi-processor architectures. The complete programming environment also features a graphical user interface with a block diagram oriented editor, a proof system for dynamical properties of programs, and the synthesis of VHDL for the compilation into integrated circuits.

As was said before, the SIGNAL language manipulates series of values called signals. This fact influences the programming style, which consists in writing systems of equations on these signals, thereby constraining their occurrences and values. This style corresponds to the specification of control systems given by control theorists and engineers in terms of equations over time-indexed variables. For example, a filter defined by equation $y_t = \frac{x_t + x_{t-1} + x_{t-2}}{3}$ is written in a style very close to this in SIGNAL: `Y := (X + X$1 + X$2)/3`. The selection operator **when** can be used to condition computations upon the value of a boolean. E.g., in the case of the filter given above, if we have: `X := Z when Z>0`, then the value will be computed only on the occurrences of `Z` with positive values. This way, the filter computation can be *suspended* when this condition is absent or false.

Another example[2] is that of a memory cell, of which the value is read on signal `V`, and modified (i.e.,

---

[2] This very simple example is meant to describe the basics of the way SIGNAL works; it should be noted that variables and counters are provided to the users as language constructs, and do not have to be re-defined by them at this lower level.
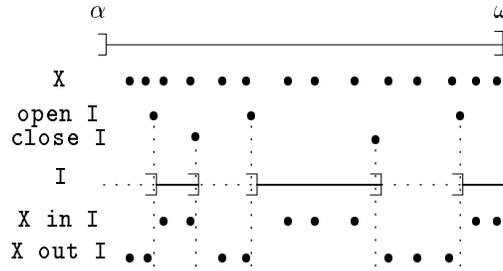
Figure 7: Time intervals.

written) by signal **V_IN**. It can be specified as follows:

```
process CELL= (V_0) {?V_IN, Clk !V}
  (| V := V_IN default (V$1 init V_0) | synchro{V, V_IN default Clk} |)
end
```

The first equation specifies that the output (read) value **V** is equal to the input (write) value **V_IN** when there is one and otherwise to the former value **V$1**. It is initialized at the parameter value **V_0**. As such, this equation does not specify **V** completely: the instants at which it is present are the *union* (by the operator **default**) of those at which it is written and those at which it is read. However, the clock of the readings is needed in order to know the clock of **V**; therefore, the **CELL** has a second input **Clk**, and the second equation defines the presence of **V** by the **synchro** operation. This process is in fact the expansion of the SIGNAL operator **cell**, more precisely **V := V_IN cell Clk**.

This example shows how state information can be memorized and managed. Combined with the selection operator, it does enable the suspension or activation of reactions depending on the state, hence the specification of sequential behaviours in SIGNAL. The same applies to other data-flow languages like LUSTRE , but imperative languages like ESTEREL and ARGOS are better suited for pure sequencing.

5.2   Time intervals and nested preemptive tasks in SIGNAL *GTi*

SIGNAL *GTi* extends SIGNAL with constructs for the activation, triggering, suspension and interruption of data-flow processes. We propose a language-level integration of the data flow and preemption paradigms, by extending SIGNAL with two kinds of *tasks* (suspendable or interruptible) which associate a process with the *time interval* on which it is active [28]. This extension is integrated to the SIGNAL environment as a pre-processor to the compiler [30]. It has been applied to an active robot vision system [20], and a control system for a power transformer station [21].

A new type is introduced in the language: **interval**. It can take two values: **inside** or **outside**. The purpose of intervals is to sub-divide the global interval of an application: $]\alpha,\omega]$ into *slices of time* (see Figure 7). The construction of an interval with initial value **I0** is noted: $\boxed{\texttt{I :=]B,E] init I0}}$, where **B** and **E** are begin and end events. Repeatedly, **I** opens at the first occurrence of **B**, and is **inside** until it closes at the next occurrence of **E**, and then it is **outside** until the next opening, etc ... Bounding events are given by, respectively: $\boxed{\texttt{O := open I}}$ and $\boxed{\texttt{C := close I}}$. Intervals are *left-open / right-closed*: transitions (going in and out of the interval) occur in reaction to an event, and depending on the current state, resulting in the new state only *after* the reaction instant (like in reactive automata). Time intervals can be composed in expressions such as union $\boxed{\texttt{I := I1 union I2}}$, complement $\boxed{\texttt{I := comp J}}$, or intersection $\boxed{\texttt{I := I1 inter I2}}$. The restriction of signal **X** to time intervals is $\boxed{\texttt{XI := X in I}}$ for occurrences of **X** inside **I**, and $\boxed{\texttt{XO := X out I}}$ for those outside **I**. Note that **X out I** is **X in (comp I)**, and that **open I** is **B out I**, and **close I** is **E in I**.

(a) Task `on` interval I, *splitting* time.

(b) Task `each` interval I, *replicating* time.
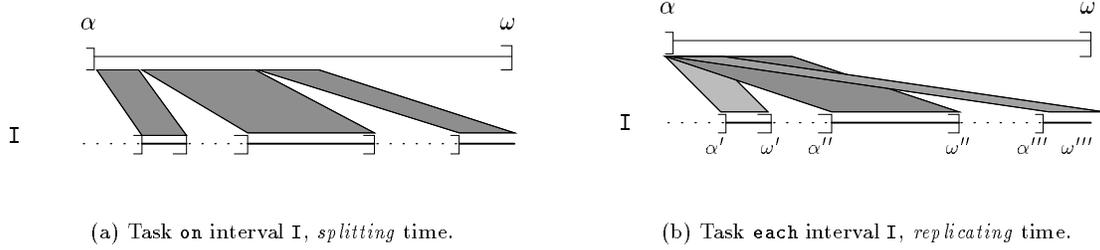
Figure 8: Tasks, and how the interval affects the time of the process.


```
(| I  := ]B,E] init outside
 | (| J  := ]R,S] init inside
   | (| C:= # X |)  on J
   |) each I              |)
```

(a) Specification in SIGNAL *GTi*

X	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
I	o	o	i	i	i	i	i	i	i	i	i	o	o	i	i	
J			i	i	i	o	o	i	i	o	i			i	i	o
C			1	2	3			4	5		6			1		
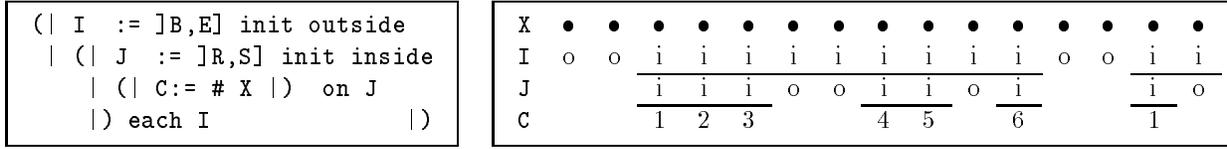
(b) A trace of the counter.

Figure 9: Time intervals and nested tasks: example of a counter.

Time intervals and processes are associated to form *tasks*. The process is *active* inside the time interval, it is *triggered* by the opening event. Outside the time interval, it is absent (in a sense, it is out of time: its clock is cut off). Time intervals do condition the existence and activity of the processes; the other way around, time intervals can be defined by the activity of the associated process. Tasks switch between activity and inactivity when the interval switches between the values `inside` and `outside`. When the interval re-opens, the process re-starts; it can do so in one of the two following ways:

suspendable tasks are noted $\boxed{P \text{ on } I}$, where the process P is *suspended* on each closing of I, and on its opening it re-starts from the *current state* of its state variables. The behaviour of P is *split* on windows in time where I is `inside` (as illustrated by Figure 8(a)).

interruptible tasks are noted $\boxed{P \text{ each } I}$, where a process P is *interrupted* on each closing of I, and on its opening it re-starts from the *initial state* of its state variables (according to their declaration). The behaviour of P is *replicated* on each time window inside I. (see Figure 8(b)).

The process in a task can be decomposed into sub-processes, which can be tasks themselves. Hierarchies of tasks and sub-tasks can be built that way. In particular, when a task is built with `each`, re-entering the interval involves re-initializing all sub-intervals and all sub-tasks recursively.

Figure 9 gives the example of a counter with output C counting occurrences of its input X: `C := # X`. It is initially inactive, started by B and terminated by E; in the meantime it can be suspended by S and resumed by R. Figure 9(a) shows how the counter is associated in a suspendable task with interval J, and how this task is itself asociated with interval I into an interruptible task. Figure 9(b) shows a possible execution trace of this task, illustrating suspensions and interruptions of the counting.

Parallelism between several tasks is obtained naturally using the composition "|" when tasks share the same interval, or overlapping intervals. *Sequencing tasks* then amounts to constraining the intervals of the tasks, by constraining their bounding events. Each time interval holds some state information, and events cause transitions between these states. With hierarchies of tasks, it is then possible to specify hierarchical parallel place/transition systems.
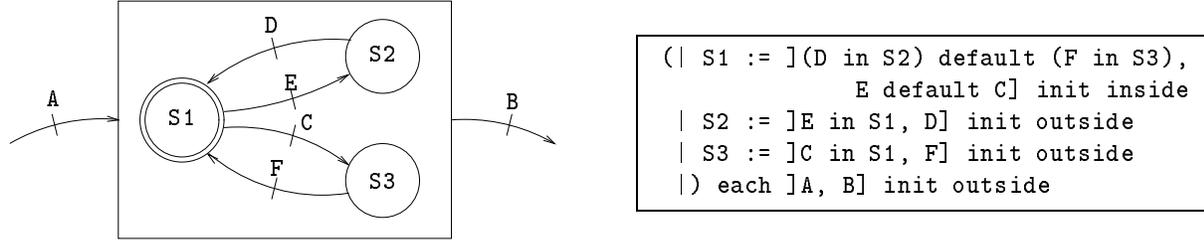
(a) Hierarchical place/transition system.   (b) Specification in SIGNAL *GTi*.

Figure 10: Sequencing, concurrency and nested preemption in SIGNAL *GTi*.

For example, in the behaviour illustrated in Figure 10(a), a transition leads from the initial place S1 to place S2 on the occurrence of an event E, except if the event C occurs before, leading in place S3. If E and C happen synchronously or are constrained to be equal, then both places S2 and S3 are entered. This is a sub-behaviour attached to a place entered upon event A and left upon event B. This can be coded by a task and intervals such that the closing of the one is the opening of the other, as in Figure 10(b).

This last example illustrates a hierarchy of tasks and intervals; it could also have featured data-flow equations. This is the advantage of embedding such constructs into a data-flow language and environment: it enables the integration of the two aspects for the specification of hybrid applications.

6   Using SIGNAL *GTi* for behavioural animation

6.1   Data-flow between perception, decision and action

At the global level, where the loop of *perception-decision-action* is handled, SIGNAL provides naturally the data-flow structures needed, especially through its graphical editing interface, as shown in the figure 11.

Across the levels, data-flow processes in SIGNAL can be organised in hierarchies, where a process is decomposed into a network of connected subprocesses itself. The process in the left part of the figure 11 is the car driver, controlling the sensor by defining its position, orientation and opening angle. The vehicle model is also controlled by the driver by determining the value of the guidance and motor torques, which are the inputs of the mechanical model of the vehicle. The driver is decomposed into sub-processes managing each one a particular aspect of the model (itinerary, obstacle detection, decisional part, ...). Also, the possibility is given to declare process models, that can be instanciated in different places in a program, thus introducing re-usability.

The specification of multi-rate computations is handled by the multi-clocked aspect in SIGNAL: signals can be under-sampled and merged, hence leading to the presence of signals and computations on them having different clocks. This can be achieved as the decomposition of a basic clock into sub-clocks for the activation of sub-modules at different frequencies. However it should be noted that the SIGNAL compiler performs a resolution of clocks where they are ordered into an inclusion hierarchy; therefore, the global clock or base clock can be *synthesized* by the compiler, and does not have to be managed by the programmer. In the example of the figure 11, each of the three modules possesses its own clock (HP, HC and HM) and then by using the **cell** operator, data generated at one particular frequency can be read at another one. For example, the frequency of the driver module is 10 Hz while the one of the vehicle model is 50 Hz, thus each value of the motor and guidance torques is used five times.

In a network of data-flow processes, the activation of computations in a process P_1 depends on the presence of the inputs, which can be themselves outputs of another process P_0. In that case there should
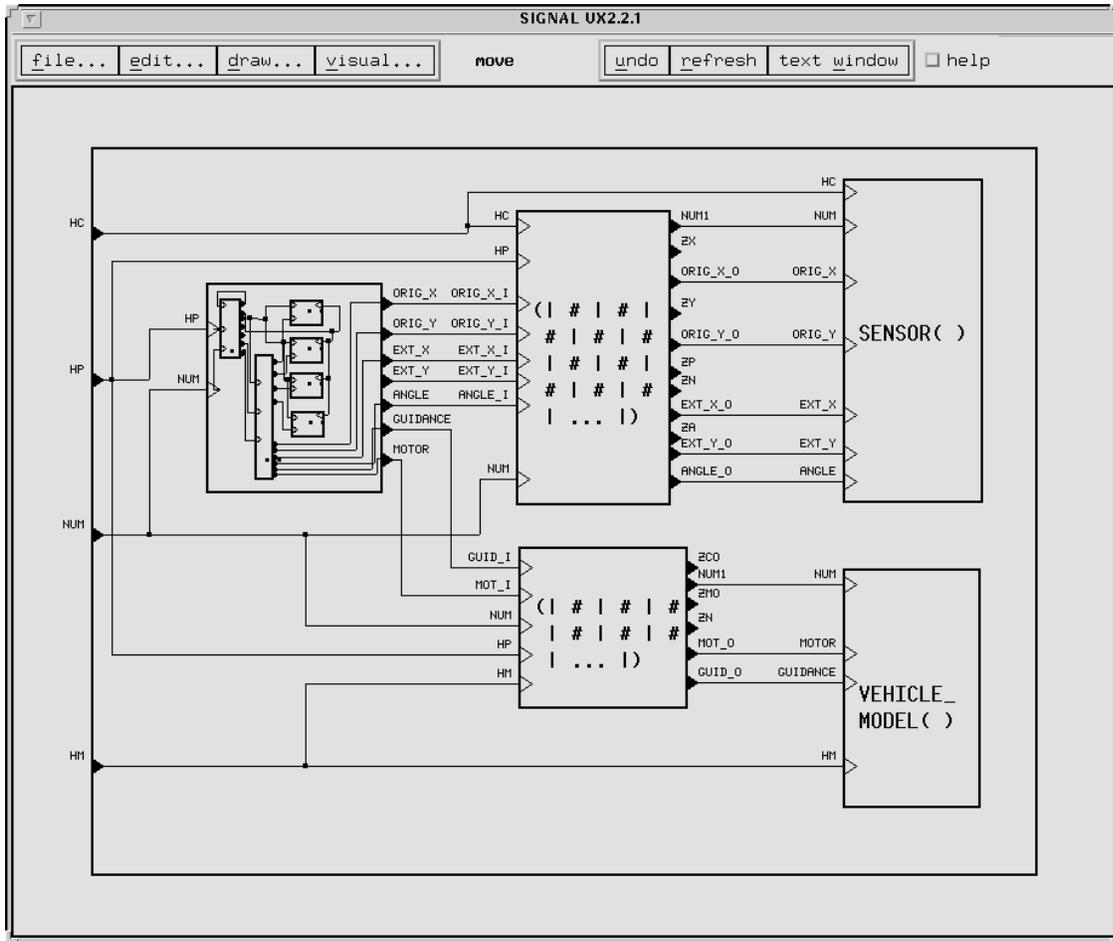
Figure 11: The hierarchical data-flow structure of a SIGNAL process (car driver).

be a sequencing of P_0 *before* P_1. This kind of sequencing of computations *within* a reaction is motivated by data dependencies but not by a real temporal precedence (from one instant to the other). This is scheduled by the compiler on the base of a graph representing actual data-dependencies between operations.

In SIGNAL programs, it is possible to call external functions; that is, some of the processes in the data flow network can be declared as being externally defined: only their interface (inputs, outputs and parameters) has to be known. This way, SIGNAL programs are open, and enable links with external modules written in C or in FORTRAN (e.g. numerical functions generated by mathematical systems, operating systems, windowing systems, ...). In the example of the figure 11, SENSOR() and VEHICLE_MODEL() are external functions.

6.2 SIGNAL *GTi* for hierarchical parallel transition systems and concurrent data-flow tasks

An example featuring sequencing, concurrency and nested preemption in SIGNAL *GTi* was presented in Figure 10. It must be noted that in synchronous languages, the concepts of "asynchronous" events is handled in a way perfectly coherent with the general notion of events; therefore they should preferably be called *sporadic* events instead, in our framework. The reactive aspect of transitions is handled in the same way as the data-flow aspect.

Transition systems are given by time intervals and the constraints between their bounds. Indeed, each

interval holds some partial state information. Leaving one interval and opening another one on the occurrence of the same event constitutes a sequencing from the one to the other. The fact that time intervals are inherently parallel makes that constrained time intervals support the definition of place/transition systems (*à la* Petri nets) rather than state/transition systems (sequential finite state automata).

The construction of hierarchies of behaviours, featuring definitive or temporary preemption of sub-tasks, is handled by the constructors **on** and **each**. They can be used to specify the re-initialization of local modules or agents upon the occurrence of events from the global controller: this is a case for the application of the **each** constructor, where the task interval is entered upon this event from the global controller. It must be noted that events can come as well from inside of the agents as from outside them.

Some behaviours require to first have a given data-flow network, continuously treating data in a certain way. Then, the occurrence of an event causes a change. This event can be either received from the external environment or caused by an internal event, like reaching a threshold. A new data-flow network, i.e. a new continuous interaction with the environment has to be installed. This sequencing of tasks *between* configurations of the network, involving a transition between states, is obtained simply by associating the processes of these tasks with intervals constrained accordingly.

These tasks can be connected in a data-flow network; their outputs and inputs can be shared, provided this does not impose incoherent constraints on their presence. E.g., in the driving simulation application, part of the driver requires the "integration" of the flows of results "proposed" by various concurrent tasks

```
(| % sequencing %
   (| I  := ]A, B]
    | J  := comp I
    |)
 | % tasks %
   (| C1 := P1{X} on I
    | C2 := P2{X} each J
    | C3 := P3{X}
    |)
 | % integration %
   C := integrate{C1,C2,C3}
 |)
```
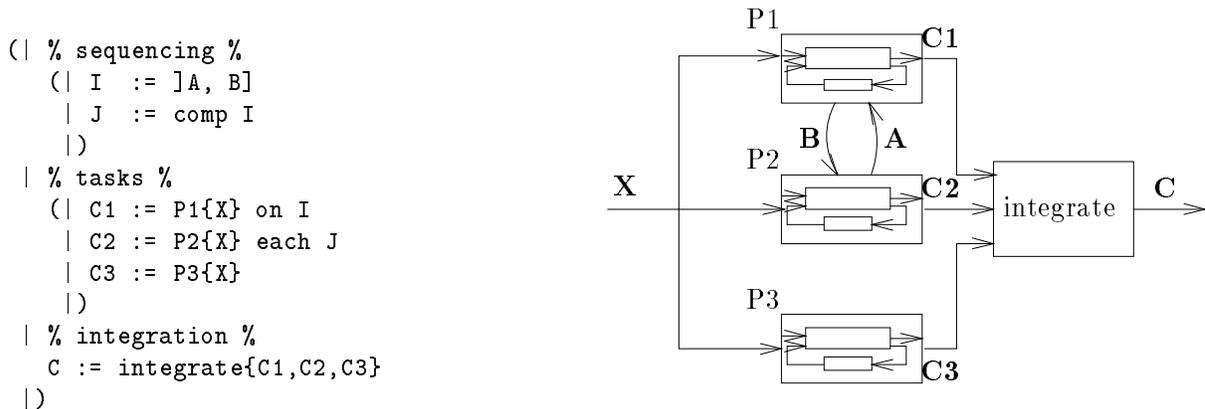

Figure 12: Structural separation between sequencing, data-flow, and integration.

The specification of such behaviours in SIGNAL *GTi* provides a methodologically clean separation between the following aspects, that are specified in parallel:

- the sequencing of tasks, which is achieved by the constraining of time intervales (see Section 4.2)

- the data-flow processes, associated to the intervals into tasks using **on** or **each**, and connected into a network

- the integration processes taking as inputs the flows of results of the tasks, which may have overlapping time intervals. The integration process composes them in order to produce the unique result for the upper level (choice, averadge, sum, max, ...)

Figure 12 gives a sketch of how SIGNAL *GTi* code can be structured this way. There can be an arbitrary recursive interleaving of data-flow and sequencing, i.e. data-flows between communicating automata or transitions between data-flow processes. This is because processes and tasks are unified into a coherent language framework.

7 Conclusions and perspectives

We have presented in this paper the behavioural model, and more precisely the decisional part of this model. A formal model of a Hierarchical Parallel Transition System has been presented to describe *Realistic* behaviours, which requires different programming paradigms: reactivity, concurrency, data-flow and hierarchical preemption. These paradigms are all integrated in SIGNAL *GTi* which is a tasking extension of the declarative synchronous language SIGNAL and we have outlined how this model can be described in SIGNAL *GTi* considering this work is currently in progress.

This model will allow us to describe, in a same way, different kinds of living beings, and to simulate them in the same virtual environment, while most of behavioural models are actually restricted to the animation of one model in a specific environment. Another important point is that our behavioural model has been constructed to generate dynamic entities which are both autonomous and controllable. allowing us to use the same model in different contexts and moreover with different level of control.

Perspectives for SIGNAL and SIGNAL *GTi* that are relevant to the domain of behavioural animation systems concern both the specification and the implementation aspects. On the side of the programming language and implementation of the systems:

- it would be interesting to be able to declare external tasks "driven" from *GTi* (asynchronous tasks started, stopped, suspended, resumed upon the occurrence of corresponding events); this would enable the linking with external tasks written in other languages.

- the former could be done by connecting *GTi* to a real-time operating system, taking advantage of lower-level functionalities for the preemption of external tasks

- work is currently in progress around SIGNAL in order to compile programs so as to produce distributed executable code; given the complexity of computations involved in graphics, it is beneficial to have a dixtributed implementation of systems. In the SIGNAL approach, the semantical equivalence between specification and distributed implementation would be guaranteed, which constitutes a great help for ensuring program correctness.

Another aspect of the synchronous approach, although also based on its formal grounds, is that the environments include tools for the automated analysis of formal properties of the specified systems and their behaviour (absence of deadlocks, reachability of states, or on the contrary non-reachability of a "bad" state). This point is important:

- for development purposes: it is a complement to tests in the debugging phase, in order to verify that the systems really has the expected or required behaviour.

- for the certification of the safety of the controllers and behaviours, which is particularly meaningful regarding safety-critical applications like in the domain of transportation.

References

[1] O. Ahmad, J. Cremer, S. Hansen, J. Kearney, and P. Willemsen. Hierarchical, concurrent state machines for behavior modeling and scenario control. In Proceedings of *Conference on AI, Planning, and Simulation in High Autonomy Systems*, Gainesville, Florida, USA, 1994.

[2] F. Arbab and E. Rutten. MANIFOLD: a programming model for massive parallelism. In Proceedings of *Proceedings of the Working Conference on Massively Parallel Programming Models*, pages 151–159, IEEE, Berlin, German, Sep. 1993.

[3] B. Arnaldi, R. Cozot, and S. Donikian. Virtual urban environment for the simulation of an automated electrical cars platoon in the praxitele project. In Proceedings of *Second Eurographics Workshop on Virtual Environments*, Monte Carlo, Jan. 1995.

[4] B. Arnaldi and G. Dumont. Vehicle simulation versus vehicle animation. In Proceedings of *Third Eurographics Workshop on Animation and Simulation*, Cambridge, Sep. 1992.

[5] N. I. Badler, C. B. Phillips, and B. L. Webber. *Simulating Humans : Computer Graphics Animation and Control*. Oxford University Press, 1993.

[6] N. I. Badler, B. L. Webber, J. Kalita, and J. Esakov, editors. *Making them move: mechanics, control, and animation of articulated figures*. Morgan Kaufmann, 1991.

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

[8] M. Booth, J. Cremer, and J. Kearney. Scenario control for real-time driving simulation. In Proceedings of *Fourth Eurographics Workshop on Animation and Simulation*, pages 103–119, Politechnical University of Catalonia, Sep. 1993.

[9] P. de Reffye, C. Edelin, J. Francon, M. Jaeger, and C. Puech. Plant models faithful to botanical structure and development. In Proceedings of *Computer Graphics (SIGGRAPH '88 Proceedings)*, J. Dill, editor, pages 151–158, Aug. 1988.

[10] S. Donikian. Les modèles comportementaux pour la génération du mouvement d'objets dans une scène. *Revue Internationale de CFAO et d'Infographie*, 9(6):847–871, 1994. Numéro Spécial 1re journées AFIG Groplan.

[11] S. Donikian and B. Arnaldi. Complexity and concurrency for behavioral animation and simulation. In Proceedings of *Fifth Eurographics Workshop on Animation and Simulation*, G. Hégron and O. Fahlander, editors, Oslo, Norvège, Sep. 1994.

[12] S. Donikian and G. Hégron. A declarative design method for 3d scene sketch modeling. In Proceedings of *EUROGRAPHICS'93 Conference Proceedings*, Barcelona, Spain, Sep. 1993.

[13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, 1994.

[14] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[15] G. Hégron and B. Arnaldi. *Computer Animation : Motion and Deformation Control*. Eurographics'92 Tutorial Notes, Eurographics Technical Report Series, Cambridge (Grande-Bretagne), Sep. 1992.

[16] D. Kalra and A. Barr. Modeling with time and events in computer animation. In Proceedings of *Eurographics*, A. Kilgour and L. Kjelldahl, editors, pages 45–58, Blackwell, Cambridge, United Kingdom, Sep. 1992.

[17] J. Kearney, J. Cremer, and S. Hansen. Motion control through communicating, hierarchical state machines. In Proceedings of *Fifth Eurographics Workshop on Animation and Simulation*, G. Hegron and O. Fahlander, editors, Oslo, Norway, Sep. 1994.

[18] P. Le Guernic, M. Le Borgne, T. Gautier, and C. Le Maire. Programming real time application with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[19] C. Lecerf. *Contrôle du mouvement de systèmes mécaniques en animation*. PhD thesis, Université de Rennes 1, Sep. 1994.

[20] E. Marchand, E. Rutten, and F. Chaumette. *Applying the Synchronous Approach to Real Time Active Visual Reconstruction*. Research Report 2383, INRIA, Oct. 1994. (ftp `ftp.inria.fr`, file `/INRIA/publication/RR/RR-2383.ps.Z`).

[21] H. Marchand, E. Rutten, and M. Samaan. *Specifying and verifying a transformer station in* SIGNAL *and* SIGNAL*GTi*. Publication Interne 916, IRISA, March 1995.

[22] M. Parent and P. Daviet. Automatic driving for small public urban vehicles. In Proceedings of *Intelligent Vehicle Symposium*, Tokyo, Japon, July 1993.

[23] M. Parent and P. Texier. A public transport system based on light electric cars. In Proceedings of *Fourth International Conference on Automated People Movers*, Irving, Texas, U.S.A., March 1993.

[24] P. Prusinkiewicz, M. S. Hammel, and E. Mjolsness. Animation of plant development. In Proceedings of *Computer Graphics (SIGGRAPH '93 Proceedings)*, J. T. Kajiya, editor, pages 351–360, Aug. 1993.

[25] C. W. Reynolds. Flocks, herds, and schools: a distributed behavioral model. In Proceedings of *Computer Graphics (SIGGRAPH '87 Proceedings)*, M. C. Stone, editor, pages 25–34, July 1987.

[26] G. Ridsdale and T. Calvert. Animating microworlds from scripts and relational constraints. In Proceedings of *Computer Animation '90 (Second workshop on Computer Animation)*, N. Magnenat-Thalmann and D. Thalmann, editors, pages 107–118, Springer-Verlag, Apr. 1990.

[27] M. Rozier, V. Abrassimov, F. Armand, M. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the chorus distributed operating system. In Proceedings of *Usenix Symposium on micro-kernels and other kernels architectures*, pages 39–69, Seattle, Apr. 1992.

[28] E. Rutten and P. L. Guernic. *Sequencing data flow tasks in* SIGNAL. Research Report 2120, INRIA, Nov. 1993. (FTP site `ftp.inria.fr`, file: `INRIA/publication/RR/RR-2120.ps.Z`).

[29] E. Rutten and P. Le Guernic. The sequencing of data flow tasks in SIGNAL. In Proceedings of *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Orlando, Florida, June 1994.

[30] E. Rutten and F. Martinez. SIGNAL*GTi*: implementing task preemption and time intervals in the synchronous data flow language SIGNAL. In Proceedings of *Proceedings of the 7th Euromicro Workshop on Real Time Systems*, (IEEE Publ.), 1995.

[31] H. Sun and M. Green. The use of relations for motion control in an environment with multiple moving objects. In Proceedings of *Graphics Interface*, pages 209–218, Toronto, Ontario, May 1993.

[32] X. Tu and D. Terzopoulos. Artificial fishes: physics, locomotion, perception, behavior. In Proceedings of *Computer Graphics (SIGGRAPH'94 Proceedings)*, pages 43–50, Orlando, Florida, July 1994.

[33] M. van de Panne and E. Fiume. Sensor-actuator networks. In Proceedings of *Computer Graphics (SIGGRAPH '93 Proceedings)*, J. T. Kajiya, editor, pages 335–342, Aug. 1993.

[34] T. Widyanto, A. Marriott, and M. West. Applying a visual perception model to a behavioural animation system. In Proceedings of *Eurographics Workshop on Animation and Simulation*, pages 89–98, Vienna, Austria, Sep. 1991.

[35] J. Wilhelms and R. Skinner. A "notion" for interactive behavioral animation control. *IEEE Computer Graphics and Applications*, 10(3):14–22, May 1990.