Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 84192, 17 pages
doi:10.1155/2007/84192

Research Article

A Domain-Specific Language for Multitask Systems, Applying Discrete Controller Synthesis

Gwenaël Delaval[1] and Éric Rutten[2]

[1] *INRIA Rhône-Alpes, 38334 Saint Ismier Cedex, France*
[2] *Laboratoire d'Informatique Fondamentale de Lille, INRIA Futurs, 59655 Villeneuve d'Ascq Cédex, France*

Received 30 June 2006; Revised 15 December 2006; Accepted 3 January 2007

Recommended by S. Ramesh

We propose a simple programming language, called Nemo, specific to the domain of multitask real-time control systems, such as in robotic, automotive, or avionics systems. It can be used to specify a set of resources with usage constraints, a set of tasks that consume them according to various modes, and applications sequencing the tasks. We automatically obtain an application-specific task handler that correctly manages the constraints (if there exists one), through a compilation-like process including a phase of discrete controller synthesis. This way, this formal technique contributes to the safety of the designed systems, while being encapsulated in a tool that makes it usable by application experts. Our approach is based on the synchronous modelling techniques, languages, and tools.

1. CONTEXT AND MOTIVATION

1.1. Embedded control systems

Embedded control systems are implementing automatic control laws or signal processing, such as in robotic, automotive, or avionics systems, or even more widely available portable devices processing voice and image signals. These systems are reactive, working in close interaction with their environment, including the controlled process which has its own dynamics (typically, following the laws of physics), imposing real-time management. Their global behavior results from this very interaction.

They are typically designed in terms of continuous models, and then implemented in a discretized form, as a *cyclic* computation upon sensor input data, producing extracted information, or control values towards actuators. This combination of computations and resource usage (sensors, processors, memory, power, actuators) defines a level of abstraction which we call a *task*.

For a complex system, with a number of different resources and meant to fulfill a variety of functionalities, several control *modes* or *phases* can be designed, and the switching between them has to be handled and controlled properly [1]. This can be intricate and the risk of errors is important, because of the complexity of systems and of requirements, particularly with respect to constraints on resource usage and interaction with the environment. Task handlers can be seen as property-enforcing layers [2]. Instances of systems structured this way can be found in robotics, for example, in robot programming environments like ORCCAD [3]. Programming languages for such purposes typically combine data flow and sequencing [4–7]. The same kind of abstraction level, considering the control of tasks independently of the encapsulated computation, is considered in the reactive language Electre [8].

We address the difficulty of designing such complex controllers safely by proposing a method applying safe design techniques to the domain of embedded control systems.

1.2. Safety-critical systems

These systems are in interaction with their environment, in such a way that malfunction can lead to disastrous consequences, be it material, financial, or human. Hence, their design has to be safe, so that they are fully validated before operation.

Formal methods and verification are a way to design safety-critical systems with an explicit care for their validation. The design is based on models of requirements, architectures,

properties to be satisfied. A common practice consists of building up a specification, and then using *formal verification* techniques (e.g., model checking of temporal logic properties on a transition system-based abstraction of the system) to assess whether given properties are satisfied or not. In the latter case, when a bug is detected, the verification technique can give indications or a diagnosis on its origin, and the designer has to go back to the design and modify it, and verify again.

Such techniques are considered difficult to use, amongst other things because of the competence required in formal techniques. Much effort is devoted to *make them more user-friendly*, because they are to be applied by engineers specialized in the systems under design. A general notion of *invisible formal methods* advocates for fully automated techniques, integrated into a design process and tools. Approaches exist in methods for correctness by construction [9], where components assembly preserves essential properties like deadlock freedom.

Some *programming languages* have compilers integrating verification, for example, the synchronous languages for reactive systems [10–12] check for each program whether static properties are satisfied, regarding the coherence of event synchronizations. Explorations of dynamical behaviors in the reachable state space, integrated in the compilation [13, 26] is applied less currently, for example, for optimization purposes with respect to dead code [14], or interface computations [15].

For the control systems that we consider, what has to be verified is the correctness of the controller handling of tasks and resources. We propose to use a formal technique, targeted at the level of these controllers, integrated in a user-friendly design framework.

1.3. Control of reactive multitasks systems

1.3.1. Robotics systems as multitasks systems

A common approach in the design of robotics systems is the *discrete/continuous control* approach. This framework divides a system in two separate layers:

(i) the *continuous control* layer is a set of purely computational tasks, each one implementing a specific atomic action on the system (e.g., control laws for a specific move);
(ii) the *discrete control* layer is dedicated to the schedule of these computational tasks.

The Orccad tool [3], for instance, implements this approach. Within this tool, the tasks of the automatic control layer, named *robot tasks*, can be described as a dataflow graph of algorithmic modules, implemented with a general-purpose programming language. To each task corresponds a single automaton managing starting, termination, and exceptions. The discrete control layer is then described either by means of the synchronous reactive language Esterel [16], either with MaestRo [17], a domain-specific language designed as a front end for Orccad . MaestRo features imperative sequencing constructs, applied to the defined robot tasks. Its compilation generates a controller in Esterel , which interacts with the tasks' automata. The behavior of this controller is therefore fully defined, and its temporal properties can be checked by the use of a model-checking tool.

1.3.2. Automated synthesis of task handlers

Discrete controller synthesis [19] can be defined in the framework of formal languages, or finite state automata or transition systems. It consists of, given a property as objective, computing the constraint (i.e., the controller) on transitions, if there exists one, such that the resulting constrained (i.e., controlled) behavior satisfies the property. It can be defined on algorithmic bases similar to those for model checking. It differs from verification in that it is more constructive, and proposes a solution.[1] The technique has been studied and implemented in the synchronous framework [20].

It has been applied to the *modelling and control of multitask systems* [9, 21–23], where the set of tasks is modelled as a transition system, and a controller has to be found that handles the preservation of constraints regarding the resources and sequencing. It can then be seen as the automatic generation of a *property-enforcing layer* [2] for a given system, or of an *application-specific scheduler* [24].

Our aim is to adapt these models and applications of discrete controller synthesis to multitask control systems, in a way such that it is encapsulated into a simple domain-specific language, and an automatic generation framework.

1.4. Our approach

We propose a *domain-specific language*, called Nemo , whose compilation encapsulates controller synthesis for multitask systems. Its constructs describe domain-specific notions of resources and their constraints, tasks and their control, particular ordering constraints to be enforced, and applications built upon them. It is defined in terms of transition systems, temporal properties, and synthesis objectives. We produce, through a compilation-like process including a phase of discrete controller synthesis (i.e., *automatically*), a *correct application-specific task handler* that satisfies the constraints (if there exists one). This way, this formal technique contributes to the safety of the designed systems, while being encapsulated in a tool that makes it usable by programmers. We use synchronous languages, modelling techniques and tools, particularly the Mode Automata language [6] and the Sigali synthesis tool [20].

Our contribution is in the proposal of this language, and the "hidden" use of discrete controller synthesis in its compilation cycle. We used for that only existing discrete controller synthesis tools, in a fairly basic way. This article shows the complete definition and implementation of this language, whose basic principle have been exposed in [25].

In the remainder of this paper, Section 2 exposes motivations for the language's constructs. Section 3 gives an

[1] "*Verification is autopsy*" [18].

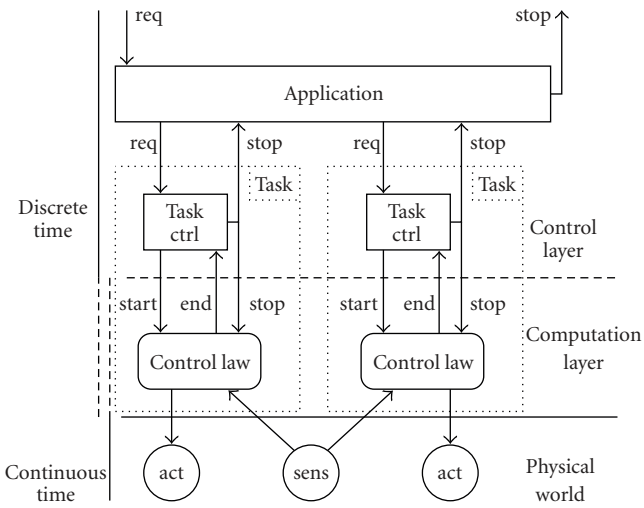


FIGURE 1: Control system composed of computations encapsulated into tasks, topped with an application.

informal overview of this language. Section 4 gives a detailed description of all the language constructs, with associated transitions systems and synthesis objectives: preliminary definitions are given in Section 4.1. Section 4.2 describes how tasks are modelled, followed by resources in Section 4.3. Section 4.4 describes temporal properties and Section 4.5 describes applications. In Section 5, the implementation using synchronous tools is presented, and performance aspects are discussed. Section 6 illustrates the approach with an example. Section 7 concludes.

2. DOMAIN SPECIFICITIES

This section gives motivations for the language constructs in terms of the targeted class of systems.

2.1. Computations, tasks, applications

We consider control systems composed of two layers, as exposed in Section 1.3.1.

(i) The *computation layer* performs data transformation algorithms, for example, numerical computations, in an infinite loop. Such computation can be implemented in another general language (e.g., as C code), and, as shown in the lower part of Figure 1, has basic control points:
   (a) it can be *started*, which can involve initializations;
   (b) it can *notify that it has reached its end*, that is, that it is ready to stop: for example, a control law that has reached its objective within a given precision range (it may yet continue controlling the actuator around the objective);
   (c) it can be *stopped*;
   (d) it can be *suspended*, that is, it ceases computation and interaction, until it is resumed.

(ii) The *control layer* manages these computations' starts and stops, by encapsulating them into *tasks*, each provided with a local controller. As shown in the middle part of Figure 1, this controller makes the relation between requests and starts, and between ends and stops: as will be discussed below, several variants may make sense. A task can also involve several *modes of activity*, where the computation is different, as well as the resources engaged.
These tasks can then be composed into *applications*, using structures such as loops, sequences, or parallel statements, in order to define a partial control over execution of tasks. As shown in the upper part of Figure 1, an application can be requested, and send in turn, according to its control structure, requests to underlying tasks. It can also eventually stop. Such applications correspond to the control layer, defined with the MAESTRO language within ORCCAD [17]; but while a MAESTRO program entirely defines the behavior of this control layer, applications in NEMO will define some control points to be constrained by means of discrete controller synthesis.
This whole *control* layer is a discrete event system, we see it as a synchronous reactive system [12].

2.2. Resources

Such computations are related to *resources*:

(i) for their computation: typically processors, memories, communication links;
(ii) and in the embedded system: typically sensors and actuators.

These resources involve *constraints* which are implicit properties such as:

(i) exclusivity,
(ii) bounds on the number of users,
(iii) bounds on the available capacity,
(iv) the need to be always under control.

Within a task, modes can correspond to several different configurations with respect to resource consumption, for example, with choices between time and memory consumption, degraded modes with lower quality level but also lower consumption.

The application of discrete controller synthesis defines a sequence of tasks by constructing a global controller that interacts with the tasks' local controllers by mean of controllable points. This global controller's aim is to *preserve* the properties of the resources.

2.3. The points to be controlled

We will describe here different possible articulations between ends, stops, requests, and starts, which will motivate the constructs of the NEMO language presented further. They correspond to different kinds of computations that can be seen in applications.

2.3.1. Controlling the termination of a computation

As we said, the end reports that some termination condition has been reached, the stop being the actual termination of the computation. Controlling the termination of a computation involves relating *stops* and *ends*.

Stop coming before end

Some computations may be stopped without having yet reached their complete termination: for example, anytime algorithms, characterized by an incremental construction where each intermediary result can be delivered as a result, be it of intermediary quality. Such tasks can hence be interrupted before having reached an end: their *stop can be triggered*.

Stop coming after end

Some computations reach their objective, and they can continue cyclically in order to maintain it: an example is a control law, always giving the correction to be applied by actuators in order to near the objective. When the latter is reached, continuing will just maintain the situation. This can be useful and even necessary, for example, in Orccad [3], the "robot tasks" encapsulating a control law have a "transition phase" when the task is finished, but the next task is not yet started. The task executes then a "degraded mode" until the start of the next task, thus allowing the operation of actuators that have to be always under control. Such tasks can hence be sustained beyond their end: their *end can be rejected*, the stop will occur at a later occurrence of the end, or *delayed*: the stop occurs at a later point, even without reoccurrence of the end.

2.3.2. Controlling the beginning of a computation

As for the termination, controlling the beginning of a computation involves relating *requests* and *starts*.

Start coming after request

When a request is made for a task, it might not be started, typically because of a resource not being available yet. Then, the request may be memorized for later treatment, or not. The *request* can be *rejected*: the start will occur at a later occurrence of the request, or *delayed*: the start occurs at a later point, even without occurrence of the request.

Start coming without request

Some computations may be called without an explicit request, for example, default control tasks for an actuator that must always be under control: their *start* can be *triggered*.

2.3.3. Controlling the modes during a computation

Modes are different ways to achieve the functionality of a task, which vary in the resources they consume, the time they take, the quality of service they achieve. For example, on an architecture composed of two processors P_1 and P_2, some tasks can be executed on either P_1 or P_2. So, we can say that this task is composed of two modes, each of them corresponding to the execution of the task on a given processor. Another example is a computation that can be performed by several algorithms, each of them using different amounts of the available resources.

Switching modes can have the effect of, for example, release parts of a bounded resource for other tasks to be able to begin, to switch to a better quality and more costly mode, or unlocking a task waiting upon an exclusive resource. The mode switches are part of the points controllable by the controller to be synthesized.

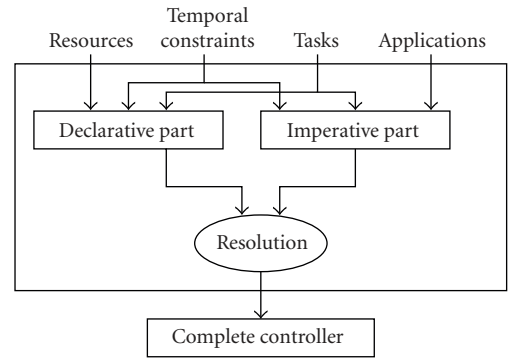


Figure 2: Compilation of Nemo .

3. OVERVIEW OF THE LANGUAGE

This section gives an informal overview of the Nemo language, taking into account Section 2.

3.1. Programming multitasks systems with Nemo

The Nemo language is devoted to build control layers. It allows the description of an abstraction of the computation layer, that is, the *resources* used, the ways computations can be controlled, that is, *tasks*, as well as some *explicit temporal properties* between tasks. These declarations are used to specify *applications*, in terms of an imperative sequencing of tasks.

From these elements, two basic parts are derived, as shown in Figure 2:

(i) a *declarative part*, grouping
   (a) the constraints corresponding to resources,
   (b) the explicit properties to be enforced,
   (c) the declared consumptions of tasks;

(ii) an *imperative part*, grouping
   (a) the observers for the explicit properties,
   (b) the behaviors of the tasks,
   (c) the behaviors of the applications.

These two parts constitute a partial specification. The imperative part features behaviors not satisfying a priori the constraints in the declarative part, with some controllable points. Therefore, obtaining the complete controller satisfying these properties involves applying some *resolution*.

This requires the use of formal models and algorithms, in order for the process to be automated, and encapsulated into a compiler-like tool. In our approach, the models will be automata, and the resolution will take the form of discrete controller synthesis.

In the following, we will introduce the programming constructs of the language in an informal way, from a user's view. Further sections will give the definitions in terms of transition systems and synthesis objectives.

3.2. Resources: implicit properties

A resource is declared using the keywords resource and `end resource`, with properties as follows. These properties are all optional, and can be specified in any order.

Bounded number of users

This maximum number of tasks which can use the resource at the same time is stated by `usable by` *n* `tasks`, where *n* is a natural integer. For example, a resource declared as

```
resource actuator:
    usable by 2 tasks;
end resource;
```

can be used by no more than 2 tasks. *Exclusivity* is the particular case where $n = 1$.

Bounded capacity

A quantifiable resource, composed of elements which can be distributed among the tasks using it (e.g., a memory), is declared by mean of the construct `composed of` *n* `elements`. Uses of such a resource will be quantified, and bounded by *n*.

Steady control

Some resources have to be controlled by *at least one task*, for example, actuators in a robotic system. This is stated by `steady control`.

Example

The following example shows the definition of a resource usable by at least one task, at most 3 tasks, and decomposable in 42 elements:

```
resource r:
    usable by 3 tasks;
    composed of 42 elements;
    steady control;
end resource;
```

3.3. Tasks

A task is declared by `task`, followed by the task name and a colon, and `end task`. This construct encloses a list of task properties, separated by semicolons, as follows.

3.3.1. Activity

A task's *activity* involves three notions.

Properties of the beginning of the task

As explained in Section 2.3.2, a task can be stated as *beginning triggerable*, *beginning delayable,* and *beginning rejectable*. They are specified by the keyword `start`, followed by one or more of `triggerable`, `delayable`, and `rejectable`. Due to their incompatible meanings, the two options `delayable` and `rejectable` are exclusive.

A *beginning-triggerable* and *beginning-delayable* task `go` will then be specified as

```
task go:
    start triggerable, delayable;
    ...
end task;
```

Properties of the end of the task

They are specified by the use of the keyword `stop`, followed by the same keywords as for beginning properties.

Suspensibility

It means that computation can be suspended at any instant by the task controller. Thus, resources used can be declared as used only when the computation is actually running, or always used (i.e., kept reserved and busy during the suspension). A suspensible task is specified by `suspensible`:

```
task think:
    suspensible;
end task;
```

3.3.2. Resources used by a task

Usage of resources is specified by `uses`, followed by the names of the resources used. If the resource is used even when the task is suspended, it is specified by the keyword `always`. The use of a decomposable resource is specified by *n* `of`, followed by the name of the resource.

The following example shows how a task using three resources can be declared. This task, named `go_forward`, uses a resource named `wheels`, 50 elements of a resource named `cpu`, and always 20 elements of the resource `memory`. This example also illustrate the possible meaning of decomposability of a resource:

```
task go_forward:
    uses wheels;
    uses always 20 of memory;
    uses 50 of cpu;
end task;
```

3.3.3. Modes composing a task

A definition of a set of modes is surrounded by the keywords `modes` and `end modes`. Thus, orthogonal aspects of a task can be specified by means of different sets of modes, with several such constructs.

The definition of each mode

It is composed of its name, and what resources it uses (using `uses`).

Transitions between modes

Every transition between modes is not necessarily possible. For example, in the case of three modes for high, medium and low values of some characteristics, one can go from high to low only through medium. So each transition has to be specified explicitly. Also, we make the choice that transitions are all bidirectional, and unconditioned (i.e., controlled entirely by the controller to be synthesized). A transition between the two modes *A* and *B* is specified as `trans A <-> B;`.

Example

The following example is the specification of a task encapsulating a computation which can be performed by three different algorithm versions, here named `high`, `medium`, and `low`. It shows also how to deal with compromises such as CPU versus memory use, by having the synthesized controller decide when to use what algorithm version, that is, in this example, to switch between the `high` and `medium` algorithms, and between the `medium` and `low` ones:

```
task calc1 :
  start rejectable;
  modes
    high : uses 100 of CPU,
           always 50 of Memory;
    medium : uses 80 of CPU,
             always 70 of Memory;
    low : uses 50 of CPU,
    always 80 of Memory;
    trans high <-> medium;
    trans medium <-> low;
  end modes;
end task;
```

In that specific case, the expected behavior of the synthesized controller is that, if another task is undelayable and requires 50% of CPU, then it will ensure that the system will never get in the `high` mode of the `calc1` task. Indeed, in this mode, the system could not go back directly to the `low` mode in case of the request of the other task.

3.4. Temporal constraints: explicit properties

Until now, the properties considered were all seen through the use constraints declared with the resources. Some other constraints could be required, for reasons not directly related to any declared resource, but having to do with some knowledge about the environment, for example, the possible incompatibility between some activities for reasons not modelled here (waiting for a temperature to cool down, rinsing brushes between painting in two different colours, etc.). Therefore, we introduce a few constructs enabling the specification of explicit temporal constraints. Properties expressible in Nemo will be safety temporal properties.

The basic events of those properties are tasks executions: a "task execution" runs from the emission of its "start" signal to the emission of its "stop" signal. The temporal properties will then be expressed in term of observers [26] on these events. Nemo provides five elementary property patterns, and two logical operators to compose properties.

Always between

Between the executions of the two specified tasks `t1` and `t2`, a third specified task `t3` must always be executed. This is expressed as

```
property
    always t3 between (t1,t2)
end property;
```

Always before

The execution of a specified task must always be preceded by the execution of another specified task. We can imagine, as an example, a physical resource `r` which has to be initialized by executing a task named `init` before any use of `r`. Then, if a task `t` is declared as using `r`, we have to explicitly declare that `init` must be executed before `t`. This is expressed as `always init before t`:

```
property
    always init before t
end property;
```

Always during

During any execution of a task `t1`, `t2` must be executed: `always t2 during t1`:

```
property
    always t2 during t1
end property;
```

Always while

Task `t1` must always be executed while `t2` is executed: `always t1 while t2`:

```
property
    always t1 while t2
end property;
```

Never while

The executions of tasks `t1` and `t2` are mutually exclusive: `never t1 while t2`:

```
property
    never t1 while t2
end property;
```

Or, and

They are the classical ones. The "not" operator cannot be allowed, so as to stay within safety properties.

3.5. Applications

Applications are the imperative part of Nemo . Their purpose is the expression of an order of execution of the tasks, for a functionality to be produced.

Once we have defined, through the declarative part of the language, a set of resources, tasks, and properties, the interface provided by the system obtained to its environment (i.e., its set of uncontrollable inputs) is, for each task, two request signals, respectively, requesting the start and the end of the task. These signals can then be received from the environment to the system at any time and in any order.

Applications define an intermediate layer emitting starting requests to task controllers, and waiting for ends of tasks, as shown in Figure 1. Compared to the usual intuition in imperative languages, sending a request does not mean immediate activation of the task: the sequencing is to be interpreted as being soft.

The definition of an application is surrounded by the keywords `application`, followed by the application name and a colon, and `end application`. It encloses the application statement as follows.

Task or application request

It will simply be denoted by its name (this simple application will just emit the signal `req`, and terminate when it receives the signal `stop`).

Sequence

The sequence of two applications app_1 and app_2 requests app_1, then upon termination app_2. It is denoted by

$$app_1; app_2. \tag{1}$$

Parallel

Parallel composition of two applications app_1 and app_2 requests app_1 and app_2 simultaneously. It terminates when both are terminated. This is denoted by

$$app_1 \,||\, app_2. \tag{2}$$

Alternative

Alternative between two applications app_1 and app_2 executes either app_1, or app_2, and terminates with the chosen application. The choice is left free, for the controller to decide, either at run-time, if both are potentially possible, or offline, if the preservation of properties excludes one. This is denoted by

$$app_1 \mid app_2. \tag{3}$$

Trigger

Trigger of an application *app* by a signal *s* awaits the occurrence of *s*, then requests *app*. It is denoted by

$$s \texttt{ triggers } app. \tag{4}$$

Loop

Loop of an application *app*, executed repeatedly until the occurrence of a signal *s* requests *app*; once *app* terminates, if *s* is absent, it requests *app* again, and so on. The condition required to get out of a loop may appear rather restrictive, but one can see that they could be emitted from observers [26, 27] in a general way. This is denoted by

$$\texttt{loop } app \texttt{ until } s. \tag{5}$$

Here is a small example of use of this application language: the application `A2` is a loop, which executes repeatedly the task `T3`, then the tasks `T4` and `T5` in any order, and then the application `A1` which executes either `T1` or `T2`:

```
application A1 :
    T1 | T2
end application;

application A2 :
    loop
        T3 ; (T4 || T5) ; A1
    until kill_A2
end application;
```

3.6. Compiling a Nemo program

Now that the Nemo language is defined, its compilation towards a complete controller will have to start from a set of resources, temporal constraints, tasks, and applications, as shown in Figure 3, which can be seen as a refinement of Figure 2.

Based on such a program, a compilation-like process constructs an *automaton*-based model of behaviors to be controlled, featuring *free variables* for the points to be controlled, and a set of properties derived from the declared constraints which are *objectives* for the synthesis. *Discrete controller synthesis* is applied on them. It computes the *controller*, that is, the constraints on the control points of the tasks which are necessary for the properties or objectives to be fulfilled. This way, the *controlled automaton*, such that the properties are satisfied, can be used through the *coexecution* or cosimulation engine.

This way, we achieve a form of *hidden formal method*, thus relieving users from heavy prerequisites in technicalities of the formal method applied.

4. MODELLING BEHAVIORS AND PROPERTIES

This section presents the formal modelling of Nemo . The behavioral aspects will be defined in terms of labelled transition systems, with a synchronous composition operator as

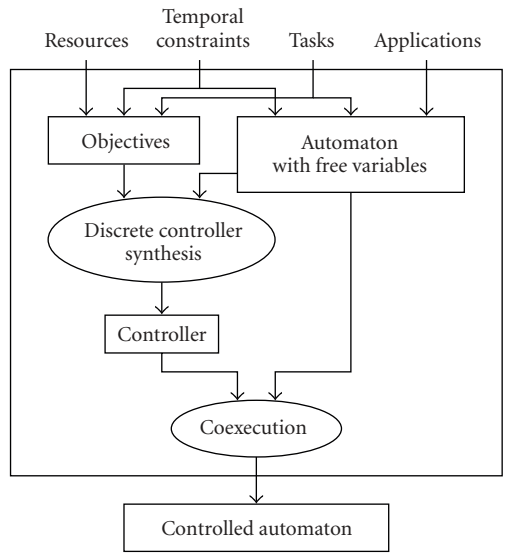


Figure 3: Compilation of Nemo : models and algorithms.

in synchronous languages [10–12]. Properties on states and trajectories will be translated to synthesis objectives. When such properties concern sequences of transitions or states, they are defined in terms of synchronous observers, embodied as transition systems composed with the previous ones [2].

4.1. Preliminary definitions

We briefly recall the notions used here, in a classical way, and which are detailed elsewhere [2].

4.1.1. Transition systems

The labelled transition systems we use in this paper are Mealy automata. An automaton $\mathcal{A}$ is a tuple $\mathcal{A} = \langle \mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$ where $\mathcal{Q}$ is the states set, $s_{\text{init}} \in \mathcal{Q}$ the initial state, $\mathcal{I}$ and $\mathcal{O}$ the input and output variables sets, and $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times 2^{\mathcal{O}} \times \mathcal{Q}$ the transitions set. $\mathcal{B}ool(\mathcal{I})$ is the set of Boolean expressions with variables in $\mathcal{I}$. We denote by $\mathcal{A}_1 \| \mathcal{A}_2$ the synchronous composition [2, 10–12] of the two transition systems $\mathcal{A}_1$ and $\mathcal{A}_2$, as defined in [2]. If $\mathcal{A} = \mathcal{A}_1 \| \cdots \| \mathcal{A}_n$ and $s_i \in \mathcal{Q}_i$, we denote by $\underline{s_i}$ the subset of $\mathcal{Q}$ whose projection in $\mathcal{Q}_i$ is equal to s_i. $\overline{s_i}$ denotes the complementary of $\underline{s_i}$ in $\mathcal{Q}$. A states set $\mathcal{Q}' \subseteq \mathcal{Q}$ is an *invariant set for* $\mathcal{A}$ if and only if every eligible transition outgoing from states of $\mathcal{Q}'$ leads to a state of $\mathcal{Q}'$.

4.1.2. Weight functions

We are going to describe properties of states by means of *weight functions*. These weight functions will later be used to specify the number or quantity of resources used in states. A weight function on a states set $\mathcal{Q}$ is a function $f : \mathcal{Q} \to \mathbb{N}$ assigning a value to each state of the transition system. We will manipulate such functions using addition and multiplication of two functions, and product of a function by a scalar λ, with the usual meaning: $(\lambda \cdot f)(q) = \lambda \cdot f(q)$, $(f + g)(q) = f(q) + g(q)$ and $(f \cdot g)(q) = f(q) \cdot g(q)$. Furthermore, all the weight functions considered below apply on the states set resulting from the general composition of all the automata.

4.1.3. Discrete controller synthesis

We simply use the classical notion [2, 19], without modifying it in this work. The aim of the discrete controller synthesis is, from an automaton $\mathcal{A}$, to compute a controller $\mathcal{C}$ such that $\mathcal{A} \| \mathcal{C}$ satisfies a property P, called *synthesis objective*, not satisfied a priori by $\mathcal{A}$. For $\mathcal{A} = \langle \mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$, and a partition of its inputs variables into two subsets $\mathcal{I}^c$ (the *controllable* inputs) and $\mathcal{I}^u$ (the *uncontrollable* ones), a *controller* of $\mathcal{A}$ is an automaton $\mathcal{C} = \langle \mathcal{Q}, s_{\text{init}}, \mathcal{I}^u, \mathcal{O} \cup \mathcal{I}^c, \mathcal{T}' \rangle$ such that $\exists (s, \ell_u \wedge \ell_c, O, s') \in \mathcal{T} \Leftrightarrow \exists \gamma \subseteq \mathcal{I}^c \wedge \exists (s, \ell_u, O \cup \gamma, s') \in \mathcal{T}'$, where ℓ_u (resp., ℓ_c) holds on only variables of $\mathcal{I}^u$ (resp., $\mathcal{I}^c$) and γ holds at most the controllable inputs of ℓ_c. This means that the computed controller will control a fixed subset $\mathcal{I}^c$ of the inputs. The values of inputs of $\mathcal{I}^c$ will be computed by this controller from the current state and $\mathcal{I}^u$ values.

Tools and algorithms exist, for example, in the synchronous approach [20], which we use as they are. A detailed discussion of their principles is beyond the scope of this paper. We use only the *invariance* objective, that is, the properties to be satisfied are all invariance properties: for an automaton $\mathcal{A}$, we denote by Inv(S) the controller $\mathcal{C}$ such that S is invariant for $\mathcal{A} \| \mathcal{C}$.

For the sake of readability, in the following representations of automata within this paper, controllable inputs will be represented in bold faces.

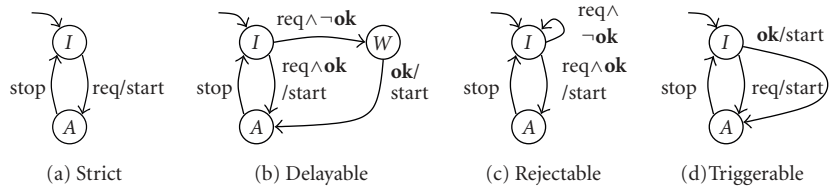
4.2. Tasks: behaviors

We are building, from the properties of a task t, an automaton $\mathcal{A}^t$ modelling the behavior of t.

4.2.1. Beginning of the task

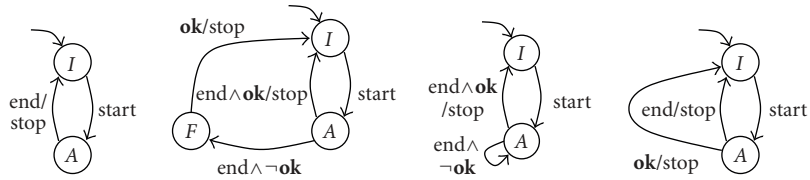
It is related to a request, coming from, for example, the application automaton, as will be defined further, and being an uncontrollable signal `req`. The task controller will emit a `start` signal to actually launch the computation encapsulated in the task. In order to express the controllability on the beginning of the task, we introduce a *controllable* signal named `ok`, which will control the emission of `start` of tasks with a nonstrict beginning (i.e., delayable or rejectable). Furthermore, in every subsequent automata depicted, the state names I, A, W, and F stand, respectively, for "Idle," "Active," "Waiting" and "Final."

The behavior of a task with a *strict* beginning is defined by the automaton on Figure 4(a). The `start` signal is emitted when `req` occurs; there is no means to inhibit it.



(a) Strict (b) Delayable (c) Rejectable (d) Triggerable

Figure 4: Models of tasks beginnings (bold inputs are controllable ones).



(a) Strict (b) Delayable (c) Rejectable (d) Triggerable

Figure 5: Models of tasks ends.

A "non-strict" beginning means that the request can be:

(i) *delayable*: the task controller will memorize it, as in Figure 4(b), and actually launches the task on the occurrence of a controllable signal `ok`;

(ii) *rejectable*: the task can only begin in the presence of a request. It is rejected if the controllable signal `ok` is false, and will in that case wait for the next occurrence of `start`, as in Figure 4(c).

A "*triggerable*" beginning means that the computation can be launched by the task controller, by mean of the controllable signal `ok`, without request from the environment, as in Figure 4(d). This property is very useful to define default task-controlling resources which have to be steadily controlled.

To this automaton $\mathcal{A}^t_{\text{beg}}$, we associate a weight function

$$\mathcal{W}^t_{\text{beg}}(q) = \begin{cases} 1 & \text{if } q \in \underline{A}, \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

4.2.2. End of the task

It involves a similar modelling: a signal called `end` is received (e.g., from the computation itself). The task controller will emit a `stop` signal to actually stop the computation, and report the actual end of the task to the environment. We use the same *controllable* input `ok` to control the emission of `stop` with respect to occurrences of `end` and properties of the task. Figure 5 show the automaton modelling the ends of tasks. This automaton is $\mathcal{A}^t_{\text{end}}$.

4.2.3. Suspensibility of the task

It is modelled by the automaton $\mathcal{A}^t_{\text{susp}}$ of Figure 6(b), to be composed with the previous ones. A new *controllable* input `susp` allows for switching between the "active" and the "suspended" states. We associate to it a weight function $\mathcal{W}^t_{\text{susp}}$ defined as

$$\mathcal{W}^t_{\text{susp}}(q) = \begin{cases} 1 & \text{if } q \in \underline{A}, \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

4.2.4. Modes

They are modelled as a new automaton, in parallel with the previous ones, with an "idle" state, one state s_m for each mode m, and controllable transitions between the modes reachable from each other.

Figure 6(a) shows the automaton representing the set of modes as defined in the example of Section 3.3.3: `l`, `m`, and `h` are *controllable inputs* which allow the task controller to control the mode in which the task is executed. We use here one input per mode for the sake of readability, but for determinism we actually assign to each mode m an expression ℓ_m, such that for all $(m_1 \neq m_2) \neg(\ell_{m_1} \wedge \ell_{m_2})$.

For each mode m, we denote by t_m the task concerning m, and we associate to m a weight function $\mathcal{W}_m$ defined as (s_m stands for the state representing the mode in the automaton modelling the set of modes which contains m):

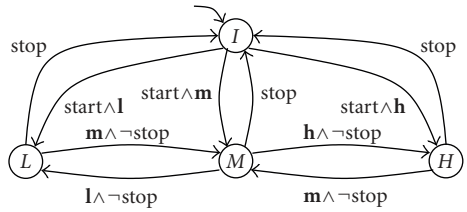
$$\mathcal{W}_m(q) = \begin{cases} 1 & \text{if } q \in \underline{s_m}, \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

4.2.5. Global behavior of the task

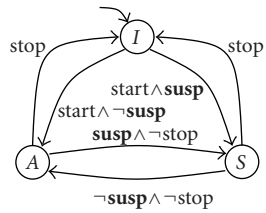
The definition of the automaton $\mathcal{A}^t$, modelling the global behavior t is then deduced from above as:

$$\mathcal{A}^t = \mathcal{A}^t_{\text{beg}} || \mathcal{A}^t_{\text{end}} || \mathcal{A}^t_{\text{susp}} || \mathcal{A}^t_{M_1} || \cdots || \mathcal{A}^t_{M_n}, \tag{9}$$

where the $\mathcal{A}^t_{M_i}$ are the modelling each modes set M_i of the task t.



(a) Model of three modes of a task



(b) Model of the suspension of a task

Figure 6: Models of modes and suspension.

4.3. Resources: implicit properties and synthesis objectives

Now that the behavior of tasks is specified in term of transition systems, we are going to show what implicit properties can be deduced from the resources and tasks properties, and are translated in terms of synthesis objectives.

4.3.1. Notations

We will consider that $\mathcal{T}$ and $\mathcal{R}$ are, respectively, the sets of the tasks and resources composing the system. For $r \in \mathcal{R}$, $\mathcal{T}^r$ is the set of tasks which use r, partitioned in two subsets $\mathcal{T}^r_{alw}$ and $\mathcal{T}^r_{act}$ the sets of tasks using r, respectively, *always* (specified by `always` with `suspensible`), and *only when active*. In the same way, we denote by $\mathcal{M}$ the set of all modes, $\mathcal{M}^r$ the set of modes using r partitioned as well in two subsets $\mathcal{M}^r_{\text{alw}}$ and $\mathcal{M}^r_{\text{act}}$.

For all decomposable resources r, we define the functions $\text{ET}^r : \mathcal{T}^r \to \mathbb{N}$ and $\text{EM}^r : \mathcal{M}^r \to \mathbb{N}$ where $\text{ET}^r(t)$ (resp., $\text{EM}^r(m)$) is the number of elements of r used by the task t (resp., in the mode m).

We also define the following two functions:

(i) $\mathcal{N} : \mathcal{R} \to \mathbb{N}$ where for $r \in \mathcal{R}$, $\mathcal{N}(r)$ is the maximum number of tasks which can use r at the same time;

(ii) $\mathcal{E} : \mathcal{R} \to \mathbb{N}$ where for $r \in \mathcal{R}$, $\mathcal{E}(r)$ is the number of elements composing r.

4.3.2. Synthesis objectives

Bounded number of resource users

It is handled by a function $\mathcal{U}^r : \mathcal{Q} \to \mathbb{N}$, associating a weight $\mathcal{U}^r(q)$ on each state q of the global system, representing the number of tasks using r in the state q. $\mathcal{U}^r$ can easily be computed by means of the functions introduced above:

$$\mathcal{U}^r = \sum_{t \in \mathcal{T}^r_{\text{alw}}} \mathcal{W}^t_{\text{beg}} + \sum_{t \in \mathcal{T}^r_{\text{act}}} \mathcal{W}^t_{\text{susp}} + \sum_{m \in \mathcal{M}^r_{\text{alw}}} \mathcal{W}_m + \sum_{m \in \mathcal{M}^r_{\text{act}}} \mathcal{W}_m \cdot \mathcal{W}^{t_m}_{\text{susp}}. \tag{10}$$

Then, to preserve the implicit property that the resource r will not be used by more tasks than it can hold, the computed controller must ensure that the system will stay within the set of states q such that $\mathcal{U}^r(q) \leq \mathcal{N}(r)$.

So, the synthesis objective to ensure the property of bounded number of users of r is

$$\text{Inv}\left(\{q \in \mathcal{Q} \mid \mathcal{U}^r(q) \leq \mathcal{N}(r)\}\right). \tag{11}$$

Exclusiveness

It is just the special case where $\mathcal{N}(r) = 1$.

Distinguishing this case is worthwhile, as the synthesis objective can be expressed only in terms of states exclusiveness, into Boolean formulas, computable much more efficiently.

Steady control

Steady control for r corresponds to the objective

$$\text{Inv}\left(\{q \in \mathcal{Q} \mid \mathcal{U}^r(q) \geq 1\}\right). \tag{12}$$

There again, it can be formulated in a Boolean formula.

Decomposable resources

r is handled with a function $\mathcal{C}^r : \mathcal{Q} \to \mathbb{N}$, where $\mathcal{C}^r(q)$ giving the total amount of elements of r consumed by the tasks using r in the state q:

$$\mathcal{C}^r = \sum_{t \in \mathcal{T}^r_{\text{alw}}} \text{ET}^r(t) \cdot \mathcal{W}^t_{\text{beg}} + \sum_{t \in \mathcal{T}^r_{\text{act}}} \text{ET}^r(t) \cdot \mathcal{W}^t_{\text{susp}} + \sum_{m \in \mathcal{M}^r_{\text{alw}}} \text{EM}^r(m) \cdot \mathcal{W}_m + \sum_{m \in \mathcal{M}^r_{\text{act}}} \text{EM}^r(m) \cdot \mathcal{W}_m \cdot \mathcal{W}^{t_m}_{\text{susp}}. \tag{13}$$

Then, the controller is

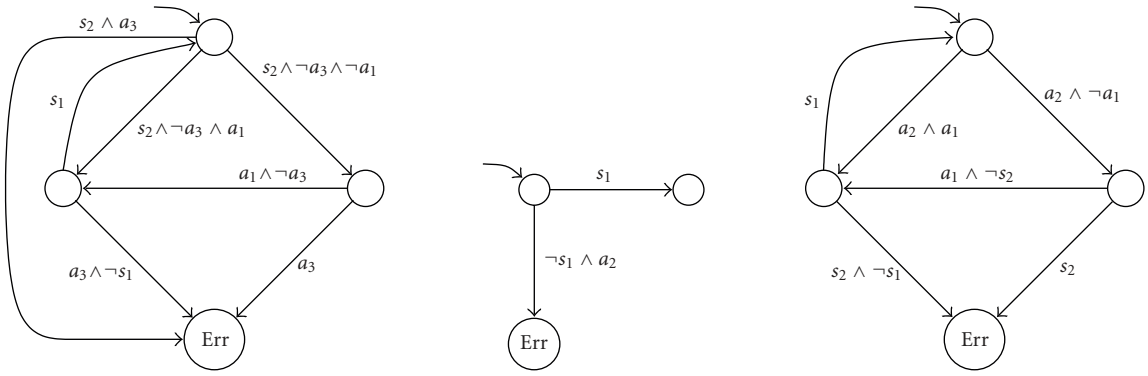
$$\text{Inv}\left(\{q \in \mathcal{Q} \mid \mathcal{C}^r(q) \leq \mathcal{E}(r)\}\right). \tag{14}$$

4.4. Temporal constraints: explicit properties

Observers and objectives

They are translated into observer automata, describing sequences leading to an "error" state Err where they are violated. The synthesis objective is the invariance of the state set deprived of this "error" state: $\text{Inv}(\overline{\text{Err}})$.

The observers are placed in parallel with the automata managing the tasks. They will take as inputs the "start" (a_i) and "stop" (s_i) signals of the tasks they observe.



(a) `Always t1 between (t2,t3)` (b) `Always t1 before t2` (c) `Always t1 during t2`

Figure 7: Observers for the temporal constraints.

"Always between" observer

Figure 7(a) depicts the observer for the property

```
always t1 between (t2,t3)
```

The state "Err" is reached when a_3 occurs after s_2, unless a_1 then s_1 are emitted in between.

"Always before" observer

Figure 7(b) depicts the observer for the property

```
always t1 before t2
```

The "Err" state of this observer is unreachable once `t1` has been executed.

"Always during" observer

Figure 7(c) depicts the observer for the property

```
always t1 during t2
```

This observer is very similar, and can easily be compared with the observer of the "between property" (see Figure 7(a)). The differences are due to taking into consideration that a task lasts at least one instant.

"Always while" property

This property does not need adding an observer. Actually, the property "t_1 is executed always while t_2 is executed" is strictly equivalent to "the model automaton of t_1 is in an execution state $\Rightarrow$ the model automaton of t_2 is in an execution state." Let A_1 and A_2 be, respectively, the execution states of t_1 and t_2 (actually, the states labelled A of the automaton of Figure 4). Then the controller to synthesize must ensure the property

$$\text{Inv}\,(\overline{A_1} \cup \underline{A_2}). \tag{15}$$

"Never while" property

There again, the property can be straightforwardly translated in a synthesis objective of invariance state set

$$\text{Inv}\,(\overline{A_1} \cup \overline{A_2}). \tag{16}$$

4.5. Applications

4.5.1. Control automaton for an application

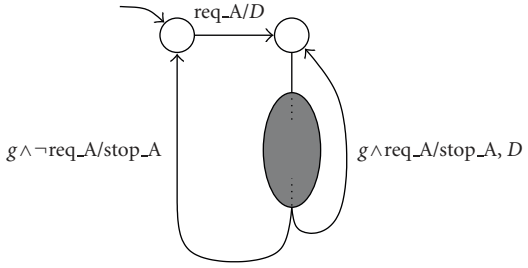
For each application, a control automaton can be constructed. Each declared application has a name, and is launched on the occurrence of a signal `req_name`. The automaton of an application named `A` is shown in Figure 8(a). On this figure, the two signals `req_A` and `stop_A` are, respectively, the requesting signal from the environment to launch the application and the signal emitted by the application to report its end. It can be launched again at the instant it terminates. This way, the synchronous paradigm is kept: this application can be called in a loop, for example.

D and g are, respectively, a set of requesting signals, and a guard. They are structurally computed from the structure of the body of the application, as well as the core of the automaton, represented in the figure by the grey ellipse. The structural translation uses "protoautomata," an intermediary form of automata, to represent each statement translated, and which will be composed together.

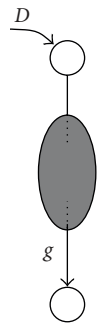
A protoautomaton is an automaton as shown in Figure 8(b), with a labelled initial transition and a "final state," that is, a sink state, with only one incoming transition, with a guard g and no emission:

(i) the initial state is the state where the statement *is launched*;
(ii) D is the set of request signals to be emitted at the launch of the statement;
(iii) the final state is where the statement is finished.

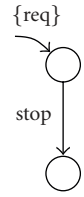
In the structural construction, the labelled initial transition, the final state and its incoming transition, will eventually be suppressed in the composition with other protoautomata.



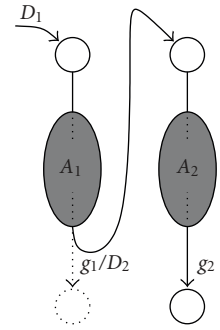
(a) Application



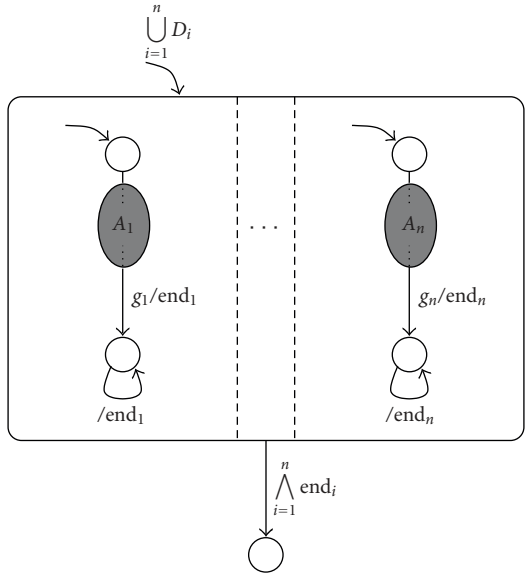
(b) Proto automaton



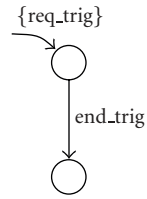
(c) Task



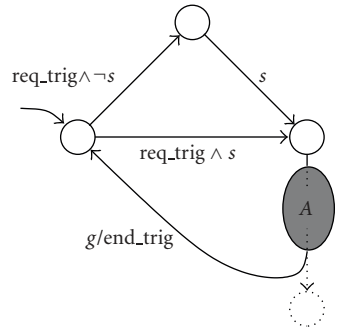
(d) Sequence



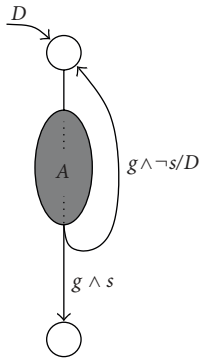
(e) Parallel



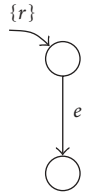
(f) Trigger



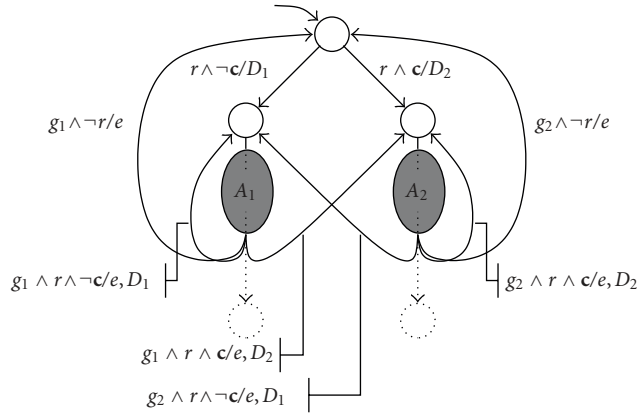
(g) Triggered statements (in parallel)



(h) Loop



(i) Alternative



(j) Alternative statements (in parallel)

Figure 8: Models of applications.

4.5.2. Control automata for statements

Request for a task or application

It consists in emitting the request, and waiting for the stop, as in Figure 8(c): the signals "req" and "stop" are, respectively, the signal requesting the launch of the task (or the application), and the signal reporting its end. The requesting signal is to be emitted immediately, at the first instant of the statement. This statement terminates once the "stop" signal occurs.

Sequence

Sequence of two statements concatenates their protoautomata as in Figure 8(d), representing $A_1; A_2$. The initial state and the initial signals set of the result are those of A_1. When A_1 terminates, requests D_2 for A_2 are emitted, going to the initial state of A_2. The final state is that of A_2. The final state of the A_1 protoautomaton is removed in the structural construction.

Parallel composition

The resulting proto automaton encapsulates the synchronous composition of the composed proto automata into an initial state. Then, synchronization on termination of all branches causes terminating the parallel construct. Figure 8(e) gives the result of this construction.

Trigger statement

Trigger involves taking into account that the trigger signal can occur at the very instant the statement is launched. Two parallel automata are set up, one for the trigger statement itself (see Figure 8(f)) and, in parallel, one for the triggered statement (see Figure 8(g)).

Loop

Loop involves the proto automaton of Figure 8(h). When A terminates, if the signal s is not present, A is launched again, with emission of signals D.

Alternative

Similarly to the trigger, this statement involves two proto automata: as we have to take into account the signals present at the instant when the application is launched, we have to disjoin the automaton managing the alternative, and the proto automaton actually representing the application.

They represent the statement itself (see Figure 8(i)), and the alternative between A_1 and A_2 (see Figure 8(j)), in parallel with the rest of the global application. `c` is a new controllable signal for the choice of the actually launched statement.

4.6. Global model

For a given Nemo program, a global automaton is built by synchronous composition of all relevant automata, as shown in Figure 9. Relevant weight information is associated to states. Also, invariance synthesis objectives are generated, corresponding to the various constraints given by resources and tasks, and observers.

At this level of granularity, abstracting away from the computations in the tasks, the automaton is already quite large. Two perspectives are to be explored: finer grain in modelling the tasks can lead to a finer resolution of the control; better abstractions and compositionality [9] and using coordination code (glue) for some of the properties would alleviate from part of the synthesis cost.

5. COMPILATION OF NEMO: IMPLEMENTATION

5.1. Implementation

Nemo is implemented as illustrated in Figure 10, which is a concretization of Figure 3. The compiler takes a Nemo program, and produces, according to each entity (resource, temporal constraint, task, application), the behavioral parts, in terms of Mode Automata, in its textual syntax (`.targos`), and the more declarative parts, weights and objectives, in terms of the equational format for the Sigali tool[2] (`.z3z`) [20]. The global automaton is compiled using the Matou Mode Automata compiler[3] [6], into an equational representation (`.z3z`), for synthesis purposes, and into the EC[4] format (`.ec`) for execution and simulation.

Discrete controller synthesis is performed by Sigali on the basis of the transition system and the objectives. The resulting controller, available in the `.res` and `.sim` formats, is fed to a resolver, encapsulated into the interactive tool SigalSimu [2], which performs a *co-simulation* of the EC representation with the controller.

5.2. Performances

The significance of our approach strongly depends on the performance of the synthesis tool used. Indeed, while the compilation of the language to automata and synthesis objectives is straigthforward, the actual bottleneck is the discrete controller synthesis, the complexity of which being, just like for model-checking verification, in the worst case polynomial in the size of the state space to handle. The state space of a system produced by the compilation of a Nemo program grows exponentially with the number of tasks and applications, as each of these is modelled by an automaton, and the comprehensive system being the synchronous product of all these automata. It is also to be expected that the performance, in the worst case, will increase dramatically with the size of the model in term of number of tasks and applications. However, we claim that the size of manageable systems is already meaningful, and rising, just like for

[2] http://www.irisa.fr/vertecs/Logiciels/sigali.html.
[3] http://www-verimag.imag.fr/~synchron.
[4] Executable Code.

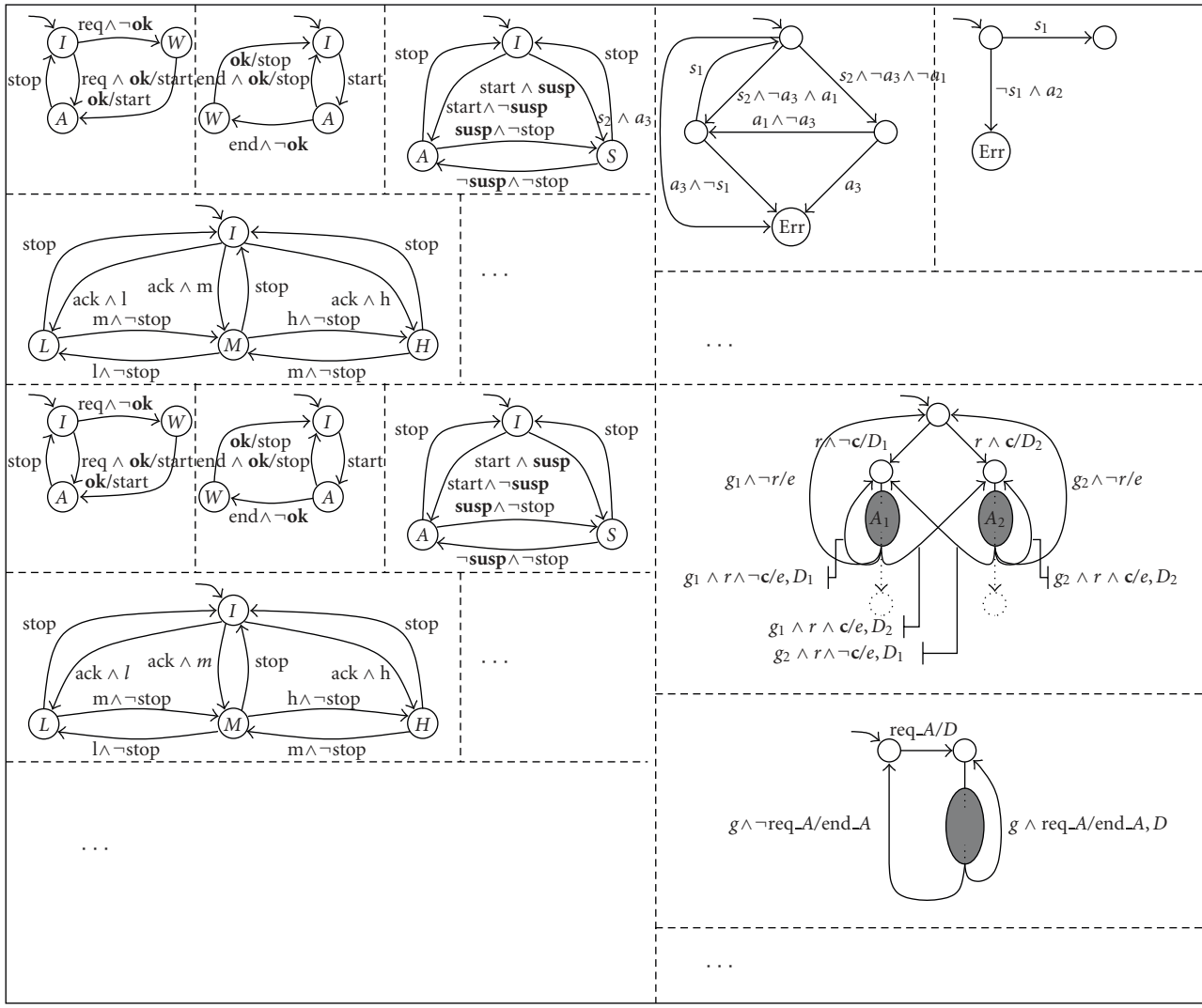


Figure 9: Complete system model.

model-checking verification, especially when compared to what can be achieved brainually.[5]

In order to evaluate the pertinence of the approach, we have measured the time taken by the controller synthesis on models composed of n start-rejectable tasks, for several values of n. The results of this experiment are presented in Table 1.

The actual experimental results presented above show that our approach is relevant for the studied domain, that is, control systems. The synthesis tool has indeed been able to handle in a few seconds systems composed of more than 30 tasks.

Nevertheless, one can notice several severe losses of performance, for example, for systems composed of 24, 28, or 32 tasks. Indeed, the computation time of the controller synthesis is difficult to predict, as it rests on manipulation of symbolic equations implemented as BDDs.[6] However, the synthesis time depends on many parameters, several of them being difficult to master, as, for example, the order of the variables. This problem is a recurrent one in the framework of symbolic computation using BDDs, and discussing it further is beyond the scope of this paper; our work can benefit from results of the very active ongoing research in this area.

Finally, a noteworthy point is that our approach comes as a substitute to the traditional one of design then verification then correction. Thus, even if the discrete controller synthesis operation on some systems is not cheap, it is to be reminded that the verification on a similar system would be evenly expensive, as verification and synthesis are based on the same algorithmic basis. The size of the systems possibly

[5] We gratefully thank Albert Benveniste for this neologism, formed from the two words "brain" and "manually." It intends to capture the fact that such work generally involves more the "brain" than the "hands."

[6] More precisely, (ternary decision diagrams) TDDs for Sigali .

Table 1

Nb of tasks	1	· · ·	10	· · ·	15	· · ·	20
Synthesis time	0.01 s	· · ·	0.34 s	· · ·	4.09 s	· · ·	4.74 s
Nb of tasks	· · ·	23	24	25	26	27	28
Synthesis time	· · ·	4.80 s	53 min · 50 s	20.60 s	10.79 s	9.61 s	> 24 h
Nb of tasks	29	30	31	32	· · ·		
Synthesis time	27.88 s	24.21 s	22.96 s	≈ 19 h	· · ·		

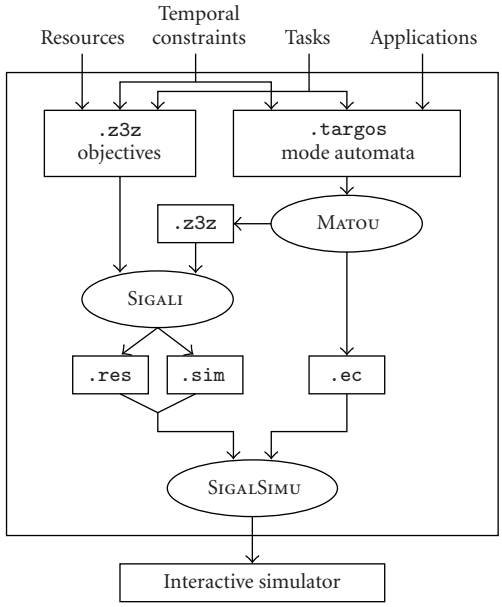


Figure 10: Compilation of Nemo : synchronous tools-based implementation.

managed by the two approach will also be the same. Furthermore, it must be considered that the controller synthesis does provide a solution, and in that sense it is to be compared with hours, or days, of brainual design.

6. TYPICAL EXAMPLE

The example proposed in Figure 11 gathers some of the interesting features of Nemo . The system modelled is a robotic arm aimed at moving some objects from a place to another. The two resources modelled are the actuator (the moving arm), and a CPU. The actuator is obviously an exclusive resource, and we also want its steady control. The elements of the CPU can be viewed as the percentage of use of it. The system encloses two categories of tasks: `hold_arm` ("default" task holding the arm in its current position), `grab`, `move`, and `release` actually manipulate the arm, whereas `check` and `compute_move` are computing tasks. We assume that the objects are brought toward the arm, for example, by a conveyor belt. Some of these objects are delicate, so at any time, the operator can trigger a checking computation to insure that the system is currently able to handle such objects properly. The event `end_checking` signals that from now, all objects are standard, and the check computation will not need to be performed any more.

In this example, the synthesis will compute a controller that will enforce the activation of the `hold_arm` task every time none of the other manipulating tasks is executed.

It will also force the second branch of the alternative of `move_object`, as long as the task `check` can be triggered, because the CPU would not have the capacity to handle the three tasks `compute_move`, `move` and `check` at the same time. This does show the interest of using discrete controller synthesis, as the choice involves a look ahead in the possible paths.

One can also notice that the explicit property "always move between grab and release" will not need any controller synthesis, as it is enforced by the application. In this case, the controller synthesis computation behaves as a verification: if the property is satisfied by the transition system, the controller gives no additional constraint. If it had not been the case, in another application, the synthesis would have failed, because the tasks are not controllable.

7. CONCLUSION AND PROSPECTS

7.1. Results

Results presented are a *domain-specific language*, devoted to multitask systems, and its *implementation*. Its definition involves a model of multitask control as transition systems, and the application of *discrete controller synthesis techniques*. The framework is an instance of user-friendly, "invisible" formal method, where the final user need not know about the underlying technicalities.

The approach presented was shown to be well adapted to the application area considered, that is, the design of robotic controllers. Indeed, this approach shares many concepts currently manipulated by existing tools of this area, such as Orccad where robot controllers are designed by mean of robot tasks and robot procedures (sharing the same role as, resp., tasks and applications in Nemo). Moreover, the performances of the underlying controller synthesis tool gives good hopes for scaled applications. Furthermore, it can be noted that even if Nemo is aimed at the design of robotics systems, it indeed can be applied to any control system expressible in terms of computing tasks, topped with a control layer on which one want to ensure temporal Boolean properties.

```
resource actuator_arm:
    exclusive;
    steady control;
end resource;

resource CPU:
    composed of 100 elements;
end resource;

task hold_arm:
    start triggerable,rejectable;
    stop triggerable,rejectable;
    uses actuator_arm, 5 of CPU;
end task;

task grab:
    uses actuator_arm, 20 of CPU;
end task;

task move:
    uses actuator_arm, 20 of CPU;
end task;

task release:
    uses actuator_arm, 20 of CPU;
end task;

task check:
    uses 40 of CPU;
end task;

task compute_move:
    modes
        Precise_move : uses 80 of CPU;
        Rough_move : uses 50 of CPU;
        trans Precise_move <-> Rough_move;
    end modes;
end task;

property
    always move between (grab,release)
end property;

application move_object:
    (grab;(compute_move||move);release)
  | (grab;compute_move;move;release)
end application;

application main:
 loop (move;move_object) until end_app ||
 loop (sig_check triggers check) until
       end_checking
end application;
```

Figure 11: Example of Nemo program.

7.2. Prospects

Although the language presented and its implementation are consistent, it is only an outline to suggest what could be an actually useable, more general-purpose language. Therefore, some work still remains.

First short-term prospects concern *usability* of the framework. In the case of absence of solution, the current choice is a short error message pointing out what objective is not synthesizable. It is worthwhile to think about what kind of diagnosis could be useful to the end users: counter examples, properties to add or remove in the current model. Also, problems of usability include *integration* in the aimed application area. Until now, the resulting controller has only a simulable form. A future implementation should be able to connect the resulting controller to a run-time executive, together with computations encapsulated in tasks.

A different and orthogonal concern is the extension of the *multitasks system model* used. The reasoning about, for example, relations between requests and actual beginning of tasks has to be generalized, as its criticalness may depend on the context of the request itself. Model of tasks can, for example, be extended with notion of successive execution phases, allowing more sophisticated models of tasks to be more easily described. Given a more specific application area (e.g., fault-tolerant design [28]), model of resources can also be augmented with some ad hoc primitives in order to allow the building of more elaborated environment models (e.g., where resources can fail or wear off).

Finally, the exclusive use of invariant synthesis objectives may be restrictive, and besides generally leads to a controller still allowing some control (as the controller synthesis computes the *most permissive controller*). In such a controller, choices between eligible values for controllable inputs have to be made at execution time. From this point of view, it could also be interesting to consider other kinds of synthesis objectives, qualitative ones such as reachability or attractivity, or quantitative ones such as optimal synthesis on paths. A further study could also be to integrate in the approach some alternatives for the current synthesis tool: solving some constraints with coordination code, or other synthesis techniques such as compositional controller synthesis.

REFERENCES

[1] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[2] K. Altisen, A. Clodic, F. Maraninchi, and É. Rutten, "Using controller synthesis to build property-enforcing layers," in *Proceedings of European Symposium on Programming (ESOP '03)*, Warsaw, Poland, April 2003.

[3] J.-J. Borrelly, E. Coste-Manière, B. Espiau, et al., "The ORCCAD architecture," *International Journal of Robotics Research*, vol. 17, no. 4, pp. 338–359, 1998.

[4] É. Rutten, "Programmation sûre des systèmes de contrôle/commande: le séquencement de tâches flot de données dans les langages réactifs," Document d'Habilitation à Diriger des Recherches, IFSIC, Université de Rennes 1, Rennes Cedex, France, décembre 1999.

[5] F. Maraninchi, Y. Rémond, and É. Rutten, "Effective programming language support for discrete-continuous mode-switching control systems," in *Proceedings of the 40th IEEE Conference on Decision and Control (CDC '01)*, pp. 3296–3301, Orlando, Fla, USA, December 2001.

[6] F. Maraninchi and Y. Rémond, "Mode-automata: a new domain-specific construct for the development of safe critical systems," *Science of Computer Programming*, vol. 46, no. 3, pp. 219–254, 2003.

[7] J.-L. Colaço, B. Pagano, and M. Pouzet, "A conservative extension of synchronous data-flow with state machines," in *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT '05)*, pp. 173–182, Jersey City, NJ, USA, September 2005.

[8] F. Cassez and O. Roux, "Compilation of the *ELECTRE* reactive language into finite transition systems," *Theoretical Computer Science*, vol. 146, no. 1-2, pp. 109–143, 1995.

[9] K. Altisen, G. Gößler, and J. Sifakis, "Scheduler modelling based on the controller synthesis paradigm," *Journal of Real-Time Systems*, vol. 23, no. 1, pp. 55–84, 2002.

[10] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic, Boston, Mass, USA, 1993.

[11] N. Halbwachs, "Synchronous programming of reactive systems, a tutorial and commented bibliography," in *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98)*, vol. 1427 of *Lecture Notes in Computer Science*, pp. 1–16, Vancouver, BC, Canada, June 1998.

[12] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[13] P. Le Guernic, "Compilation involving model-checking and controller synthesis," personal communication, 1996.

[14] D. Potop-Butucaru and R. de Simone, "Optimizations for faster execution of Esterel programs," in *Proceedings of the 1st ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '03)*, pp. 227–236, Mont-Saint-Michel, France, June 2003.

[15] A. Chakrabarti, L. de Alfaro, T. Henzinger, and F. Mang, "Synchronous and bidirectional component interfaces," in *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, pp. 414–427, Copenhagen, Denmark, July 2002.

[16] G. Berry, "The foundations of Esterel," in *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds., pp. 425–454, MIT Press, Cambridge, Mass, USA, 2000.

[17] E. Coste-Manière and N. Turro, "The MAESTRO language and its environment: specification, validation and control of robotic missions," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '97)*, vol. 2, pp. 836–841, Grenoble, France, September 1997.

[18] P. Darondeau, "Verification is autopsy," personal communication, October 2004.

[19] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[20] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic, "Synthesis of discrete-event controllers based on the signal environment," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 10, no. 4, pp. 325–346, 2000.

[21] H. Marchand and É. Rutten, "Managing multi-mode tasks with time cost and quality levels using optimal discrete control synthesis," in *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS '02)*, pp. 241–248, Vienna, Austria, June 2002.

[22] C. Kloukinas, C. Nakhli, and S. Yovine, "A methodology and tool support for generating scheduled native code for real-time Java applications," in *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT '03)*, vol. 2855 of *Lecture Notes in Computer Science*, pp. 274–289, Philadelphia, Pa, USA, October 2003.

[23] É. Rutten and H. Marchand, "Automatic generation of safe handlers for multi-task systems," Rapport de Recherche 5345, INRIA, Le Chesnay Cedex, France, October 2004, http://www.inria.fr/rrrt/rr-5345.html.

[24] C. Kloukinas and S. Yovine, "Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS '03)*, pp. 287–294, Porto, Portugal, July 2003.

[25] G. Delaval and É. Rutten, "A domain-specific language for task handlers generation, applying discrete controller synthesis," in *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC '06)*, vol. 1, pp. 901–905, Dijon, France, April 2006.

[26] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," in *Proceedings of the 3rd International Conference on Algebraic Methodology and Software Technology (AMAST '93)*, pp. 83–96, Twente, The Netherlands, June 1993.

[27] L. J. Jagadeesan, C. Puchol, and J. V. Olnhausen, "Safety property verification of Esterel programs and applications to telecommunications software," in *Proceedings of the 7th International Conference on Computer Aided Verification (CAV '95)*, vol. 939 of *Lecture Notes in Computer Science*, pp. 127–140, Liège, Belgium, July 1995.

[28] A. Girault and É. Rutten, "Discrete controller synthesis for fault-tolerant distributed systems," in *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '04)*, Linz, Austria, September 2004.