Using Controller-Synthesis Techniques to Build Property-Enforcing Layers

Karine Altisen[1], Aurélie Clodic[2], Florence Maraninchi[1], and Eric Rutten[3]

[1] VERIMAG Centre Equation, 2 Av. de Vignate – F38610 GIERES
www-verimag.imag.fr – {Karine.Altisen,Florence.Maraninchi}@imag.fr
[2] LAAS-CNRS, 7 av. Colonel Roche – F-31077 TOULOUSE
www.laas.fr – Aurelie.Clodic@laas.fr
[3] INRIA Rhône-Alpes, MONTBONNOT – F-38334 ST ISMIER Cedex
www.inrialpes.fr – Eric.Rutten@inrialpes.fr

Abstract. In complex systems, like robot plants, applications are built on top of a set of components, or devices. Each of them has particular individual constraints, and there are also logical constraints on their interactions, related to e.g., mechanical characteristics or access to shared resources. Managing these constraints may be separated from the application, and performed by an intermediate layer.
We show how to build such property-enforcing layers, in a mixed imperative/declarative style: 1) the constraints intrinsic to one component are modeled by an automaton; the product of these automata is a first approximation of the set of constraints that should be respected; 2) the constraints that involve several components are expressed as temporal logic properties of this product; 3) we use general controller synthesis techniques and tools in order to combine the set of communicating parallel automata with the global constraint.

1 Introduction

Consider the programming of a small robot made of two devices: an elevator table and a rotating arm placed on it. The elevator has a motor than can be switched on and off, in either direction, and two sensors at its extreme positions. The rotating arm also has a motor with commands on and off, and a choice between two speeds. The requests for moving up or down, and rotating the arm, come from an application program in charge of performing some given sequence of tasks with the robot.

At a low level, independently of any particular application, the programming of the robot has to ensure safety properties related to the characteristics of the devices composing the robot, and the way they interact. These can concern the mechanics, or the access to shared resources. For instance, the motor of the elevator should be turned off when the elevator reaches one of its extreme positions. This type of local constraint can be specified independently of the behavior of the arm. Similarly, the arm motor should be turned off before a change of speeds can be performed.

We may also have to take into account some global constraints, concerning their interactions, like "the arm should not be turning at its highest speed while the elevator is moving up". There are several methods we can think of for ensuring such properties in the running application:

- The responsibility could be left to the application; the code ensuring the safety properties related to the mechanics of the robot has to be included in all application programs; it may be difficult to intertwine with the proper code of the application. Even if we can provide powerful static verification tools to check the properties before running the application on the actual robot, this solution should be avoided, because it makes writing the application very difficult.
- A solution that allows to separate the code of the application and the code that is in charge of ensuring the safety properties, is to introduce an *intermediate layer*. The application does not talk directly to the robot but to this *layer*, that may delay or reject its requests to the actuators of the robot. This layer is in charge of enforcing the safety properties, and may be reused with various applications. Using this architecture means that the application is aware of the fact that its requests may be postponed or canceled. This is where an acknowledge mechanism is needed.

In all cases, note that we cannot rely on *monitoring* techniques and dynamic checks, because we are mainly interested in embedded systems. These systems should not raise exceptions at runtime. Our aim is not to reject faulty programs, either statically or dynamically, but to help in designing them correctly.

In this paper we formalize the general intermediate layer approach, thereby allowing for the automatic generation of such *property-enforcing layers* from a mixed-style description of the properties: several automata for the individual properties of the devices, and temporal-logic formulas for the global properties. Controller synthesis techniques are used as a compilation technique here.

2 The Approach

Expressing Individual Constraints and Global Constraints The individual constraints on the behavior of the devices can be conveniently modeled as simple reactive state machines with the sensors from the physical devices and requests from the application (`sensor`, `req`) as inputs and the commands to the actuators (`start`, `stop`) as outputs (see figure 1-a). Each automaton records significant states of the corresponding device, e.g., `I` for idle, and `A` for active. The automaton of figure 1-a enforces the following property: "a request is ignored if it happens while a previous request is being treated.". Note that we may think of various protocols between the application and the intermediate layer: it may be useful to send an acknowledgment (`ack`) on the transition that stops the motor, meaning: "the request has been executed". In particular, it is not sent when a request is ignored.

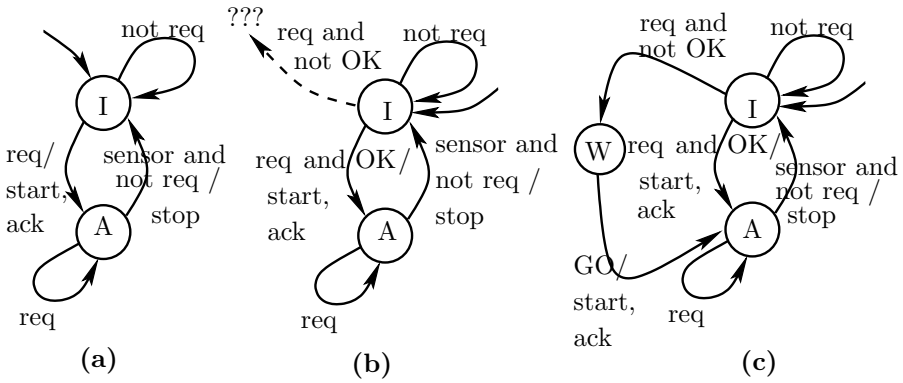


Fig. 1. Expressing Individual Constraints

The parallel composition of all the individual automata models all the *individual constraints*. In terms of these parallel automata, a *global property* like "the arm should not be turning at its highest speed while the elevator is moving up" means that one particular global state (or, perhaps, a *set* of global states) should be unreachable. More generally, we are interested in *safety* properties of the parallel composition (see [9] for the distinction between safety and liveness properties).

Mixing the two kinds of Constraints Of course, if we start from a set of automata $A = A_1||A_2|| \ldots ||A_n$ that were designed in isolation, and impose a global safety property ϕ, it is very likely that A does not satisfy ϕ. For example, if the application requires that the arm motor be switched on, while the elevator is moving up, nothing can be done to avoid the faulty situation.

When global constraints appear, due to the joint use of several devices, the automaton describing one component has to be designed in a more flexible way. For instance, if obeying a request from the application immediately is forbidden by a global constraint, given the states of the other devices, the request has to be either rejected or delayed.

We choose to introduce an additional component (i.e. a *controller*), that knows about the global safety property to be ensured, and may constrain the individual automata about the transitions they take in order to ensure this property. Then, we re-design the individual automata in a more controllable way, allowing them to respond to events from the application, the physical device, *and* the controller. The transitions that were labeled by "`req`" are now labeled by "`req AND ok`", meaning that the request is taken into account only if the controller allows it (see Figure 1-b). But then, what happens when "`req AND NOT ok`"? The missing transition may be a loop, meaning that the request is simply canceled. In this case, the application is likely to apply a protocol that maintains the request as long as it is not taken into account.

Another solution is to memorize the request. Instead of responding directly to a request by the appropriate command to the physical device, the automaton enters a *waiting* state `W`, hence postponing the request until it can be obeyed without violating the global safety property. This gives the machine of Figure 1-c, where label "`GO`" corresponds to the controller releasing the waiting request.

We could model even more sophisticated behaviors. For instance, the application might cancel its requests; or several requests might be queued, etc.

Again, writing the controller by hand may be hard to do if there are a lot of individual devices and global properties. It can even be the case that such a controller does not exist.

The solution we propose is to let the general *controller synthesis* technique do the job for us. Instead of programming the controller and the communications between the machines by hand, we state this control objective in a very declarative way (as a logical formula on the set of states). Then we let the *controller synthesis* technique generate the controller that, put in parallel with the individual machines, will ensure the global property.

Summarizing the Method Consider an application program A and a physical system under control, e.g., a robot R. The latter requires that a property ϕ be respected, i.e. $A||R \models \phi$. Our method is the following:

- First, design A with a software architecture that introduces an intermediate *layer* I_ϕ to ensure ϕ: $A = A'||I_\phi$. The problem becomes: $A'||I_\phi||R \models \phi$. A' is easier to write, and I_ϕ is reusable.
- I_ϕ includes properties that can be expressed for each component or device independently of the others, and also global constraints. ϕ is of the form $\phi_1 \wedge \phi_2 \ldots \wedge \phi_n \wedge \phi_{glob}$:
- For the individual constraints, propose a set of *automata* $A_1, A_2, \ldots A_n$ (like the ones presented in Figure 1), composable with a controller, i.e. able to respond to an application and to a controller, and corresponding to the properties $\phi_1, \phi_2, \ldots \phi_n$.
- For the global constraints ϕ_{glob}, express them as safety properties, and let the controller synthesis technique build the controller. This gives the *most permissive* controller, that has to be made deterministic since we want to use it as a program. We will use techniques from *optimal controller synthesis* [14] to reduce the non-determinism and to impose some kind of *progress*.

If a controller exists, the final picture is: $I_\phi = A_1||A_2|| \ldots ||A_n||C_{\phi_{glob}}$, and $A'||I_\phi||R \models \phi$, by construction, for all A'. If there exists no controller, it means that some of the automata have to be redesigned, introducing more "controllability" (e.g., `OK` and `GO` inputs, waiting states) so that the controller should be able to ensure the property.

The paper Section 3 sets a formal framework in which our approach can be explained together with the main results of controller synthesis. An example taken from robotics is described in section 4, with a list of global constraints one may want to ensure for this kind of systems. Section 5 gives some quick hints

on the implementation of the approach. Section 6 comments on the method. Section 7 reviews related work, and section 8 is the conclusion.

3 Framework

Our work uses general controller synthesis results (see [18]): we present them in a unified formal framework by using synchronous Mealy machines from synchronous languages (see, for instance, [11]), augmented with state *weights*. A presentation of controller synthesis with Mealy machines can also be found in [2], with similar motivations: Mealy machines give programs straightforwardly.

3.1 Synchronous Automata with Outputs and Weights

Definition 1 (Automaton). *An* automaton $\mathcal{A}$ *is the tuple* $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$ *such that* $\mathcal{Q}$ *is the set of states,* $s_{init} \in \mathcal{Q}$ *is the initial state,* $\mathcal{I}$ *and* $\mathcal{O}$ *are the sets of Boolean input and output variables respectively,* $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times 2^{\mathcal{O}} \times \mathcal{Q}$ *is the set of transitions, and* $\mathcal{W} : \mathcal{Q} \longrightarrow \mathbb{N}$ *is a function that labels states with natural* weights. $\mathcal{B}ool(\mathcal{I})$ *denotes the set of Boolean formulas with variables in* $\mathcal{I}$*. For* $t = (s, \ell, O, s') \in \mathcal{T}$*,* $s, s' \in \mathcal{Q}$ *are the source and target states,* $\ell \in \mathcal{B}ool(\mathcal{I})$ *is the triggering condition of the transition, and* $O \subseteq \mathcal{O}$ *is the set of outputs emitted whenever the transition is triggered. We consider that the Boolean formulas used as input labels are conjunctions of literals and their negation. Disjunctions lead to several transitions between the same two states.*

Definition 2 (Reactivity and Determinism). *Let* $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$ *be an automaton.* $\mathcal{A}$ *is* reactive *iff* $\forall s \in \mathcal{Q}, \bigvee_{(s,\ell,O,s') \in \mathcal{T}} \ell$*.* $\mathcal{A}$ *is* deterministic *iff* $\forall s \in \mathcal{Q}, \forall t_i = (s, \ell_i, O_i, s_i) \in \mathcal{T}, i = 1, 2 \,.\, \ell_1 = \ell_2 \implies (O_1 = O_2) \wedge (s_1 = s_2)$ *.* [4]

Every automaton in this paper is reactive but is not necessarily deterministic. The automata of figure 1 are of this kind. However, in the concrete syntax, we often omit the transitions that are loops and do not emit anything. When the weights on states are omitted, they are 0.

The semantics of an automaton $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$ is given in terms of input/output/state *traces*.

Definition 3 (Trace). *Let* $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$ *be an automaton. A sequence of tuples* $t = \{(v_i, O_i, s_i)\}_i$ *where the* v_i *are valuations of the inputs, the* O_i *are subsets of outputs, and the* s_i *are states, is a trace of* $\mathcal{A}$ *iff*

$$\begin{cases} s_1 = s_{init} \\ \forall n \quad \exists (s_n, \ell, O_n, s_{n+1}) \in \mathcal{T} \text{ such that } \ell \text{ has value } \texttt{true} \text{ in } v_n \,. \end{cases}$$

In state s_i*, upon reception of input valuation* v_i*, the automaton emits* O_i *and goes to* s_{i+1}*. We note Trace*$(\mathcal{A})$ *the set of all traces of* $\mathcal{A}$*.*

[4] The equality $\ell_1 = \ell_2$ stands for syntactical equality since there is no disjunction in labels.

Definition 4 (Trace with hidden inputs). *Let $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$ be an automaton, and let $J \subseteq \mathcal{I}$ be a set of input variables to be hidden. A* trace of $\mathcal{A}$ with hidden values J *is a sequence of tuples $t_{\mathcal{I}\setminus J} = \{(v'_i, O_i, s_i)\}_i$ where $\forall i \,.\, v'_i : \mathcal{I} \setminus J \longrightarrow \{\mathtt{true}, \mathtt{false}\}$, $O_i \subseteq \mathcal{O}$ and $s_i \in \mathcal{Q}$ such that there exists a trace $t = \{(v_i, O_i, s_i)\}_i \in Trace(\mathcal{A})$ and $\forall i \,.\, \forall x \in \mathcal{I} \setminus J \,.\, v'_i(x) = v_i(x)$.*

The trace with hidden inputs J built from a trace $t \in Trace(\mathcal{A})$ is noted $t_{(\mathcal{I}\setminus J)}$ as above. And we note the set of all traces with hidden inputs J: $Trace_{(\mathcal{I}\setminus J)}(\mathcal{A})$.

Definition 5 (Synchronous Product). *Let $\mathcal{A}_1 = (\mathcal{Q}_1, s_{init1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1, \mathcal{W}_1)$ and $\mathcal{A}_2 = (\mathcal{Q}_2, s_{init2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2, \mathcal{W}_2)$ be automata. The* synchronous product *of $\mathcal{A}_1$ and $\mathcal{A}_2$ is the automaton $\mathcal{A}_1 || \mathcal{A}_2 = (\mathcal{Q}_1 \times \mathcal{Q}_2, (s_{init1} s_{init2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T}, \mathcal{W})$ where $\mathcal{T}$ is defined by: $(s_1, \ell_1, O_1, s'_1) \in \mathcal{T}_1 \wedge (s_2, \ell_2, O_2, s'_2) \in \mathcal{T}_2 \iff (s_1 s_2, \ell_1 \wedge \ell_2, O_1 \cup O_2, s'_1 s' 2) \in \mathcal{T}$; $\mathcal{W}$ is defined by: $\mathcal{W}(s_1 s_2) = \mathcal{W}_1(s_1) + \mathcal{W}_2(s_2)$ (more general composition of weights may be defined if needed).*

The synchronous product of automata is both commutative and associative, and it is easy to show that it preserves both determinism and reactivity.
Encapsulation makes variables local to some subprogram and enforces synchronization; the following definition is taken from ARGOS [11]. In general, the encapsulation operation does not preserve determinism nor reactivity. This is related to the so-called "causality" problem intrinsic to synchronous languages (see, for instance [3]). However, these problems can appear only if two parallel components communicate in both directions, in the same instant. We will use encapsulation only in simple cases for which this is not necessary.

Definition 6 (Encapsulation). *Let $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$ be an automaton and $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ be a set of inputs and outputs of $\mathcal{A}$. The* encapsulation *of $\mathcal{A}$ w.r.t. Γ is the automaton $\mathcal{A} \backslash \Gamma = (\mathcal{Q}, s_{init}, \mathcal{I} \setminus \Gamma, \mathcal{O} \setminus \Gamma, \mathcal{T}', \mathcal{W})$ where $\mathcal{T}'$ is defined by: $(s, \ell, O, s') \in \mathcal{T} \wedge \ell^+ \cap \Gamma \subseteq O \wedge \ell^- \cap \Gamma \cap O = \emptyset \iff (s, \exists \Gamma \,.\, \ell, O \setminus \Gamma, s') \in \mathcal{T}'$.*

ℓ^+ is the set of variables that appear as positive elements in the monomial ℓ (i.e. $\ell^+ = \{x \in \mathcal{I} \mid (x \wedge \ell) = \ell\}$). ℓ^- is the set of variables that appear as negative elements in the monomial l (i.e. $\ell^- = \{x \in \mathcal{I} \mid (\neg x \wedge \ell) = \ell\}$).

Example 1. In figure 2 two automata $\mathcal{A}$ and $\mathcal{B}$ are composed by a synchronous product, and then $\{b\}$ is encapsulated. The typical use of an encapsulation is to enforce the synchronization between two parallel components, by means of a variable which is an input on one side, and an output on the other side. In the product, this variable appears in both the triggering condition and the output set of transitions.

Definition 7 (Temporal Properties of the Automata). *Let $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$ be an automaton, let $S \subseteq \mathcal{Q}$ be a set of states and let $t = \{(v_i, O_i, s_i)\}_i \in Trace(\mathcal{A})$ be a trace of $\mathcal{A}$. The properties ϕ we are interested in are the two CTL [7] formulas defined below.*

Invariance of S: $\phi = \forall \Box(S)$. A trace t satisfies $\Box(S)$ (noted $t \vDash \Box(S)$) iff $\forall i \,.\, s_i \in S$. For automata: $\mathcal{A} \vDash \forall \Box(S) \iff \forall t \in Trace(\mathcal{A}) \,.\, t \vDash \Box(S)$.

Reachability of S: $\phi = \forall \Diamond(S)$. A trace t satisfies $\Diamond(S)$ iff $\exists s \in S \,.\, \exists i \,.\, s = s_i$. For automata: $\mathcal{A} \vDash \forall \Diamond(S) \iff \forall t \in Trace(\mathcal{A}) \,.\, t \vDash \Diamond(S)$.

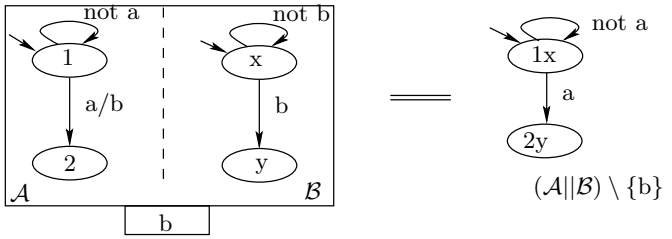


Fig. 2. An encapsulation example

3.2 Controllers and Controller Synthesis

Controllers Let $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$ be an automaton. We partition the set $\mathcal{I}$ of inputs into a set $\mathcal{I}_u$ of *uncontrollable* inputs (those coming from the application or from the physical devices, like `req`, `sensor` in figure 1) and a set $\mathcal{I}_c$ of *controllable* inputs (i.e. inputs coming from the controller, like `OK`, `GO`).

Definition 8 (Controller of an Automaton). *A* controller of $\mathcal{A}$ *is an automaton* $\mathcal{C} = (\mathcal{Q}, s_{init}, \mathcal{I}_u, \mathcal{O} \cup \mathcal{I}_c, \mathcal{T}', \mathcal{W}')$ *such that* $\exists t = (s, \ell_u \wedge \ell_c, O, s') \in \mathcal{T} \iff \exists \gamma \subseteq \mathcal{I}_c \wedge \exists t' = (s, \ell_u, O \cup \gamma, s') \in \mathcal{T}'$*, where* ℓ_u *(resp.* ℓ_c*) is only written with variables in* $\mathcal{I}_u$ *(resp.* $\mathcal{I}_c$*) and* $\gamma \subseteq \mathcal{I}_c$ *at most contains the controllable inputs involved in* ℓ_c *(i.e.,* $\ell_u^+ \cup \ell_u^- \subseteq \mathcal{I}_u$*, and* $\gamma \subseteq \ell_c^+ \cup \ell_c^- \subseteq \mathcal{I}_c$*). Moreover* $\forall s \in \mathcal{Q} \,.\, \mathcal{W}'(s) = 0$.

Notice that one $t \in \mathcal{T}$ may define several $t' \in \mathcal{T}'$ as defined above. The controller $\mathcal{C}$ of an automaton $\mathcal{A}$ has the same structure (states and transitions). The controllable variables are *inputs* in $\mathcal{A}$, whereas they are *outputs* of the controller. This means that the role of the controller is to choose whether controllable variables should be emitted, depending on uncontrollable inputs and states.

The automaton $(\mathcal{A}||\mathcal{C}) \setminus \mathcal{I}_c$ represents the *controlled automaton of* $\mathcal{A}$ *by* $\mathcal{C}$: the interaction between the controller and its automaton is formalized by a synchronous product ($\mathcal{A}$ and $\mathcal{C}$ execute in parallel, communicating *via* $\mathcal{I}_c$ variables) and the $\mathcal{I}_c$ variables are kept as local variables (and so encapsulated).

Properties 1 *Let* $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$ *and let its input variables* $\mathcal{I}$ *be partitioned into the two subsets* $\mathcal{I}_c$ *and* $\mathcal{I}_u$*. Let* $\mathcal{C}$ *be a controller of* $\mathcal{A}$*.*

1. *Trace*$((\mathcal{A}||\mathcal{C}) \setminus \mathcal{I}_c) \subseteq$ *Trace*$_{\mathcal{I}_u}(\mathcal{A})$.
2. *If* $\mathcal{A}$ *is reactive, then* $\mathcal{C}$ *is reactive, by construction. But* $\mathcal{C}$ *may not be deterministic even if* $\mathcal{A}$ *is deterministic.*
3. $(\mathcal{A}||\mathcal{C}) \setminus \mathcal{I}_c$ *is reactive and, if* $\mathcal{C}$ *is deterministic, then so is* $(\mathcal{A}||\mathcal{C}) \setminus \mathcal{I}_c$*. This holds because the encapsulation is used in a case for which causality problems do not occur.*

The first property means that every trace of the controlled automaton of $\mathcal{A}$ by $\mathcal{C}$ is also a trace of $\mathcal{A}$ with hidden variables $\mathcal{I}_c$: the controller restricts the execution of the automaton. 2 and 3 are specific to the way we build controllers. Reactivity and determinism are required if we want to obtain *programs* with this method.

Example 2. Let us observe figure 3. The right part of the figure depicts a controller $\mathcal{C}$: the set of controllable inputs is $\mathcal{I}_c = \{\texttt{OK}\}$, whereas `req` and `stop` are uncontrollable. The controller shown here enforces the fact that the task always has to wait before executing. This is done by deciding which of the controller transitions do emit `OK`, in such a way that the transition to `wait` remains, whereas the transition to `EX` disappears, in the product.

The controlled automaton $(\mathcal{A}||\mathcal{C}) \setminus \mathcal{I}_c$ is shown in figure 4, where the synchronous product of the automaton $\mathcal{A}$ and of its controller $\mathcal{C}$ has been performed (left part), and where the encapsulation of the controllable input `OK` has been realized (right part).

In the transition from `stopped` to `wait`, `OK` appears as a negative element in the triggering condition of $\mathcal{A}$; the controller chooses not to emit it in the corresponding transition, hence the transition remains in the encapsulated product. Conversely, from `stopped` to `execute`, `OK` appears as a positive element in $\mathcal{A}$; since the controller does not emit it, the transition disappears in the encapsulated product.

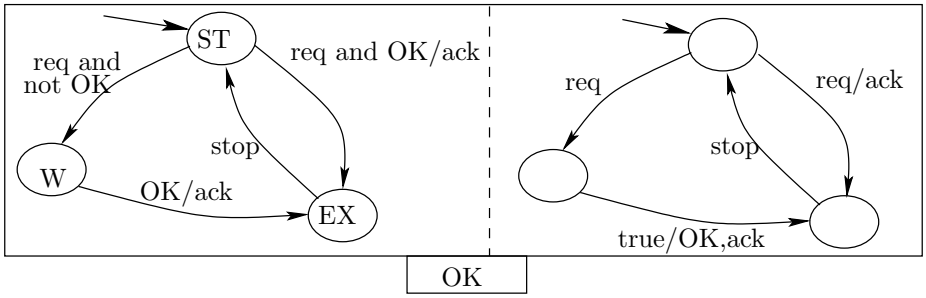


Fig. 3. An automaton and a controller for it

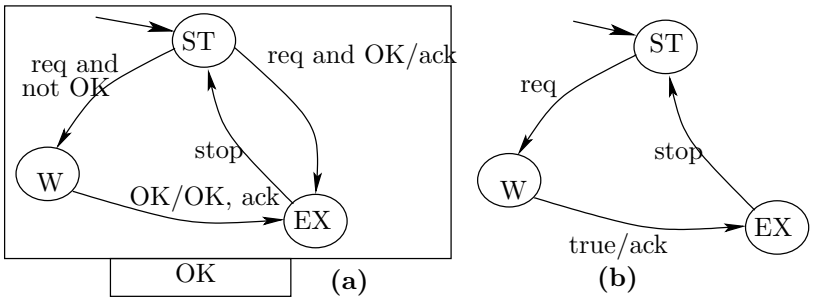


Fig. 4. The controlled automaton obtained from figure 3

Controller Synthesis Problem Let $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$ be a *deterministic* and reactive automaton $\mathcal{I} = \mathcal{I}_c \cup \mathcal{I}_u$. Let ϕ be one of the two CTL properties on $\mathcal{A}$ given by definition 7.

Problem 1 (Controller Synthesis). The controller synthesis problem consists in finding a controller $\mathcal{C}$ of $\mathcal{A}$ such that the controlled automaton of $\mathcal{A}$ by $\mathcal{C}$, satisfies the property ϕ: $(\mathcal{A}||\mathcal{C}) \setminus \mathcal{I}_c \vDash \phi$.

The problem may have several solutions but has a greatest solution, called the *most permissive* controller: if $\phi = \forall\Box(S)$ (resp. $\phi = \forall\Diamond(S)$), the controller $\mathcal{C}$ is the most permissive iff if $\exists t \in \mathit{Trace}(\mathcal{A})$ such that $t \vDash \Box(S)$ (resp. $t \vDash \Diamond(S)$) then $t_{(\mathcal{I}_u)} \in \mathit{Trace}((\mathcal{A}||\mathcal{C}) \setminus \mathcal{I}_c)$.

Reducing Non-Determinism in the Controller We are interested in deterministic controllers because our aim is to build a *program*.

Let $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$ be a deterministic and reactive automaton with $\mathcal{I} = \mathcal{I}_c \cup \mathcal{I}_u$. Let ϕ be one of the two CTL properties on $\mathcal{A}$ defined in definition 7. Let $\mathcal{C}$ be a solution of the controller synthesis given by $\mathcal{A}$ and ϕ.

We are looking for a controller $\mathcal{C}'$ which is a solution of the same controller synthesis problem, and which is more deterministic. First, we impose that $\mathcal{C}$ and $\mathcal{C}'$ have the same set of states and outputs but not necessarily the same set of transitions and inputs.

Second, we want to ensure the property $\mathit{Trace}_{\mathcal{I}_u}(\mathcal{C}') \subseteq \mathit{Trace}_{\mathcal{I}_u}(\mathcal{C})$ since it guarantees that: $\mathit{Trace}_{\mathcal{I}_u}((\mathcal{A}||\mathcal{C}') \setminus \mathcal{I}_c) \subseteq \mathit{Trace}_{\mathcal{I}_u}((\mathcal{A}||\mathcal{C}) \setminus \mathcal{I}_c)$ and then $(\mathcal{A} \,||\, \mathcal{C}) \setminus \mathcal{I}_c \vDash \phi \implies (\mathcal{A}||\mathcal{C}') \setminus \mathcal{I}_c \vDash \phi$, i.e. if $\mathcal{C}$ is a solution of the above controller synthesis problem then also is $\mathcal{C}'$. We give two ways of building $\mathcal{C}'$ from $\mathcal{C}$: *static* or *dynamic* reduction of non-determinism.

Static reduction of non-determinism: $\mathcal{C}'$ only differs from $\mathcal{C}$ by its transition set, $\mathcal{T}_{\mathcal{C}'}$: $\mathcal{T}_{\mathcal{C}'} \subseteq \mathcal{T}_{\mathcal{C}}$, where $\mathcal{T}_{\mathcal{C}}$ is the set of transitions of $\mathcal{C}$. In this paper, we use a very particular case of this approach: $\mathcal{T}_{\mathcal{C}'}$ may be obtained from $\mathcal{T}_{\mathcal{C}}$ by a local optimization based on state weights: $\forall t = (s, \ell, O, s') \in \mathcal{T}_{\mathcal{C}'} \;.\; \mathcal{W}(s') = \min\{\mathcal{W}(s'') \mid \exists (s, \ell, O'', s'') \in \mathcal{T}_{\mathcal{C}}\}$. Notice that this operation may not completely suppress the non-determinism of the controller.

Dynamic reduction of non-determinism: $\mathcal{C}'$ differs from $\mathcal{C}$ by its set of inputs $\mathcal{I}_{\mathcal{C}'}$ such that $\mathcal{I}_u \subseteq \mathcal{I}_{\mathcal{C}'}$, and by its triggering conditions. The idea is to add special inputs called *oracles*: from a state S, if there are two transitions labeled by the same input ℓ, then one of them becomes $\ell \,.\, i$ and the other one becomes $\ell \,.\, \bar{i}$, where i is the oracle. In general we need several oracles (see [10]). We obtain a deterministic automaton (or a "program") that has to be run in a environment that decides on the values of the oracle inputs. See sections 5 and 6.

4 Example System and Methodology

4.1 A robot system

We illustrate the proposed methodology with a case study [5] concerning a robot system: an automated mobile cleaning machine, designed by ROBOSOFT[5]. It can learn a mission, with trajectories to be followed, and starting and stopping

[5] `www.robosoft.fr`

of cleaning tools at pre-recorded points. It can play them back, using sensors like odometry, direction angle and laser sensors to follow the trajectories and detect beacons. One of the tools is a brush, mounted on an articulated arm, under the robot body, that can achieve vertical translation (in order to be in contact with the floor or not), horizontal translation (in order to reach corners), and rotation.

The constraints are the following: 1) the brush should rotate only when on the floor, in low position, because otherwise, when in high position, it might damage the lower part of the mobile robot; 2) it can be moved laterally only when on the floor, and not rotating, for the same reason.

4.2 Modeling the brush individual constraints

The brush individual constraints are modeled in terms of three automata, each one representing the activation of control laws for one degree of freedom of the brush: vertical movement, horizontal movement and rotation.

Vertical movement: the initial state `up` in Figure 5(a) represents the brush being in high position. Upon reception of a request from the application to move down, `r_down`, either the controller accepts it, in the absence of any conflict at global level, by `okV`, or not. If yes, then the `going_down` state is reached, with emission of the acknowledgement `start_down`. Otherwise, the request is memorized by going to state `wait_down`. The controller may then authorize the activation from state `wait_down` to state `going_down`, by `okV` with emission of `start_down`. When the uncontrollable event `sensor_down` occurs from the physical device, corresponding to the reaching of the low position, the state `down` is reached, with emission of `stopV`. Movement upwards follows a symmetrical scheme, also subject to controller authorization.

Horizontal movement: the automaton for horizontal movements follows exactly the same scheme as for vertical movement.

Rotation: the automaton for rotation follows a different scheme (figure 5(b)): there is no intermediate state going to the rotation state. State `imm` designates an immobile brush. A request for rotating, `r_rot`, is either accepted directly by `okR`, which leads into the `rotate` state, or not, which leads to the `wait_rot` state. Going back from rotation to immobility is done through a request `r_imm`, and follows the same scheme as before, with a waiting state `wait_imm` in case not authorized, and a deceleration state `going_imm`.

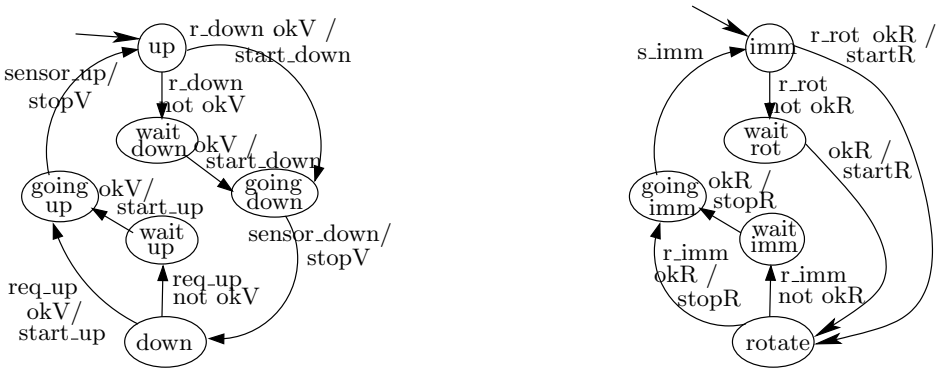
4.3 Safety properties to be ensured by the controller

We introduce a notation to define sets of global states in terms of local states. Let $\mathcal{A}_1, ..., \mathcal{A}_n$ be n automata. $(\mathcal{A}_i = (\mathcal{Q}_i, s_{\text{init}\,i}, \mathcal{I}_i, \mathcal{O}_i, \mathcal{T}_i, \mathcal{W}_i)$. Let $\mathcal{A} = \mathcal{A}_1 || \mathcal{A}_2$ ... $|| \mathcal{A}_n = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{W})$. Let $s_i \in \mathcal{Q}_i$ be a state of automaton $\mathcal{A}_i$. We note $\underline{s_i}$ for all the states of $\mathcal{A}$ whose projection of $\mathcal{Q}_i$ is equal to s_i: $\underline{s_i} = \{s \in \mathcal{Q} \mid s = (s'_1, s'_2, ..., s'_n)\ .\ s'_i = s_i\}$ The set of global states excluding s_i is noted $\overline{s_i}$ for $\mathcal{Q} \setminus \underline{s_i}$. The set of global states excluding $S \subseteq \mathcal{Q}$ is noted $\overline{S}$.

Global states must be avoided where the properties mentioned in section 4.1 are violated. To define them, we identify states where:

– the brush turns, which can happen when in states `rotate`, `wait_imm` and `going_imm`, as a decelerating brush is still in motion. This is expressed by the set of states: $Rotating = \underline{rotate} \cup \underline{wait\_imm} \cup \underline{going\_imm}$;
– the brush arm is in low position, i.e. in state `down` and also `wait_up`: $Low = \underline{down} \cup \underline{wait\_up}$;
– the brush arm is moving laterally i.e., in states `going_out` and `going_back`: $Lateral = \underline{going\_out} \cup \underline{going\_back}$.

The set of safe states wrt properties described in section 4.1 is then given by: $S = (\overline{Rotating} \cup Low) \cap (\overline{Lateral} \cup \overline{Rotating} \cap Low)$. Finally, we compute a controller for the property $\forall\Box(S)$.



(a) vertical move (b) rotation

Fig. 5. The brush control tasks.

4.4 Result of the controller synthesis phase

The result is a non-deterministic controller (see section 3.2). In particular, the choice remains between staying in a `wait` state and moving to the active state, both being safe with respect to the property. This is the usual problem when specifying a system by safety properties: a very simple way of respecting them is to do nothing. Hence, some *progress* should be expressed and taken into account.

We propose to use the weights associated with states. The weight of the waiting states is set to 1, and the weight of all other states is set to 0. The static reduction of non-determinism produces a controller where, whenever there are two transitions sourced in the same state, with the same inputs, only the transitions that go to the states with minimal weight are kept.

In the example, this is sufficient for ensuring that at least one component leaves its waiting state when it is possible. This does not yield a deterministic automaton, as some global states might have the same weight due to the composition of local weights. Dynamic reduction of non-determinism can then be used. In the framework of our case study, we worked in a context of interactive simulation: the values for the oracles are given by the end-user.

5 Implementation

The current implementation of the method, which has been used for the example, relies on the chain of Figure 6: the individual constraints are described using a synchronous formalism called "mode-automata" [12]; Yann Rémond provided the compiler into `z3z`, the input format of the synthesis tool *Sigali*. The global properties and the weights are expressed into `z3z` by the means of *Sigali* macros. *Sigali* [13] is a tool that performs model-checking, controller synthesis for logical goals, and optimal controller synthesis.

The result of *Sigali* is a controller, in the form of an executable black-box. Y. Rémond and K. Altisen developed the tool *SigalSimu*, to simulate the behavior of the controlled system. This corresponds to a dynamic resolution of the non-determinism, where the human being plays the role of the oracle.

The next step will be to transform the interpretation chain into a *compilation* chain, producing the controlled system as an explicit automata that can then be compiled into C code (see below).

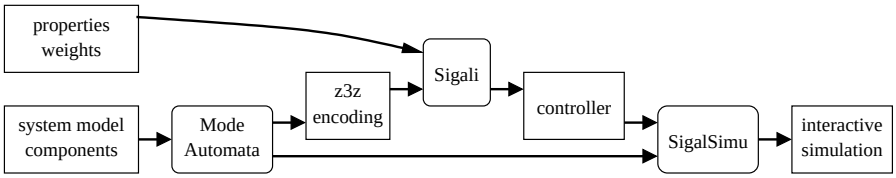


Fig. 6. Implementation of the approach: the tools involved.

6 Evaluation of the Method

Patterns for the individual constraints: The whole approach requires that the mechanical devices (or, more generally, the "resources" for which we build an application program) be modeled as small automata. We suggest that a set of reusable *patterns* — in the spirit of "design patterns" [8] — be designed for that purpose. It is likely to be specific of a domain.

Cost: The algorithms involved in controller synthesis techniques are expensive. If the whole intermediate layer had to be built as the result of a synthesis, starting from a declarative specification only, it could be too costly to be considered as a viable implementation technique. The reason why it is reasonable in our case is that part of the specification is already given as a set of automata. Controller synthesis is used only to further restrict the possible behaviors of the product.

Towards a compilation chain (non-determinism, progress and fairness): First, we want programs, so we have to determinize the controllers. Second, a specification S made of safety properties only, leads to trivial solutions that do not *progress*. Hence we have to *specify* progress properties, and to remove transitions in the controller obtained from S, so that only progressing behaviors remain. Third, what about *fairness*? Our example system is an instance

of a *mutual exclusion* problem, and we defined *critical sections* as sets of states in each of the individual automata. In some sense, the effect of the controller-synthesis phase is to add the protocol between the components, so that mutual exclusion is respected. This is a typical case where a non-deterministic choice remains, for choosing the component to serve. Usually, a dynamic scheduler is used to ensure fairness between the concurrent processes willing to access the shared resource.

In our case, we would like to obtain a deterministic controller and to compile it into a single program. This can be done by 1) determinizing the controller with oracles (see section 3.2); 2) adding in parallel an automaton *Or* that sends the oracle values, and encapsulating the oracle variables. *Or* is responsible for the fairness of the whole system.

7 Related work

Previous Work In previous work [15, 16], an approach is proposed for the modeling of robot control tasks, using simple pre-defined control patterns, and generic logical properties regarding their interactions. A teleoperation application is considered, as an illustration of a safety-critical interactive system. An extension of this work concerns multi-mode tasks [14], where each task has several activity modes or versions, distinguished by weights capturing quality (as in e.g., image processing) and cost (typically: execution time). Optimal control synthesis is then used to obtain the automatic control of mode switchings according to objectives of bounded time and maximal quality. The approach in this paper is a generalization of this more specialized work.

On "Property-Enforcing" techniques A number of approaches has been proposed for enforcing properties of programs, but they mainly rely on dynamic checks. In [6], a program transformation technique is presented, allowing to equip programs with runtime checks in a minimal way. Temporal properties are taken into account, and abstract interpretation techniques are used in order to avoid the runtime checks whenever the property can be proven correct, statically. In the general case, the technique relies on runtime checks, anyway.

The approach described in [17] is a bit different because it does not rely on program transformation. The authors propose the notion of *security automaton*. Such a security automaton is an observer for a safety property, that can be run in parallel with the program (performing an on-the-fly synchronous product). When the automaton reaches an error state, the program is stopped.

On the use of Controller-Synthesis Techniques In [4], the authors use controller synthesis techniques to help in designing component interfaces.

In their sense, an interface is a (possibly synchronous) black box that is specified by input and output conditions (input and output behaviors). Interfaces may be composed as far as they are compatible, i.e. as far as there exist some inputs for which the composition works. Compatibility is computed by a

controller synthesis algorithm which finds the most permissive application (wrt input and output conditions) under which the composed interfaces may work.

The unusual thing here (regarding the use of controller synthesis) is that they constrain the application, i.e. the environment of the interfaces, in order to fit input and output conditions, whereas, in usual controller synthesis framework, we use it to make the system work whatever the application/environment does.

In [1], the authors use controller synthesis techniques to build real-time schedulers. A layered modeling methodology is also provided here. First, real-time processes are individually modeled by timed transition systems; then a synchronization layer is built ensuring functional properties; finally, a scheduler is computed by controller synthesis, ensuring the non functional properties of the layer.

8 Conclusion and Further Work

We presented a method that automates partially the development of *property-enforcing* layers, to be used between an application program and a set of resources for which safety properties are defined and should be respected by the global system (the application, plus the intermediate layer, plus the set of resources).

We illustrated the approach with a case-study where the set of resources is a robot. However, the method can be generalized to other kinds of applications.

The method relies on two ideas. First, the specification of the properties to be respected often comes in a mixed form: simple and "local" properties, typically those imposed by one mechanical device in isolation, are better given as simple automata; on the contrary, the interferences between the devices, and the situations that should be avoided, are better described in a declarative way, for instance with trace properties in a temporal logic. Second, mixing the two parts of the specification is not easy, and we show how to use very general controller-synthesis techniques to do so. The technique that enforces the safety part of the specification has to be complemented, in order to ensure some progress. We adapted the notion of optimal synthesis to obtain progress properties.

Further work has to be devoted to the notion of "*progress*" in layers that enforce safety properties. We encountered the problem and solved it only in a very particular case. The first questions are: what kind of progress properties do we need? How can they be expressed in terms of optimal synthesis goals?

We really believe that optimal control synthesis is the appropriate method because, in the contexts we are interested in, the "progress" properties are often related to a notion of "quality". The states in the individual automata might be labeled by weights related to CPU time, memory use, energy consumption, quality of service, etc. (additivity of weights in parallel components may not fit all the needs, of course). In these cases, progress means "improve the quality".

Acknowledgments: The authors would like to thank Hervé Marchand, the author of *Sigali*, and Yann Rémond, the author of mode-automata, for their work on the implementation.

References

1. K. Altisen, G. Gößler, and J. Sifakis. Scheduler modelling based on the controller synthesis paradigm. *Journal of Real-Time Systems*, 23(1), 2002.
2. S. Balemi. *Control of Discrete Event Systems: Theory and Application*. PhD thesis, Automatic Control Laboratory, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, May 1992.
3. G. Berry. The foundations of esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.
4. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In *CAV 2002: 14th International Conference on Computer Aided Verification*, LNCS. Springer Verlag, 2002.
5. Aurélie Clodic. Tâches multi-modes et génération automatique de contrôleurs. Master thesis report, Institut National Polytechnique de Grenoble, June 2002. *(in French)*.
6. Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*. ACM Press, January 19–21 2000.
7. E.A. Emerson and E. Clarke. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. Workshop on Logic of Programs*. LNCS 131, Springer Verlag, 1981.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
9. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.
10. F. Maraninchi and N. Halbwachs. Compositional semantics of non-deterministic synchronous languages. In *European Symposium On Programming*, Linkoping (Sweden), April 1996. Springer verlag, LNCS 1058.
11. F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001.
12. F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, to appear, 2002.
13. H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the Signal environment. *Discrete Event Dynamical System: Theory and Applications*, 10(4), October 2000.
14. Hervé Marchand and Éric 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*, 2002.
15. Eric Rutten. A framework for using discrete control synthesis in safe robotic programming and teleoperation. In *Proceedings of the IEEE International Conference on Robotics and Automation, ICRA'2001*, 2001.
16. Eric Rutten and Hervé Marchand. Task-level programming for control systems using discrete control synthesis. Research Report 4389, INRIA, February 2002.
17. Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.
18. W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal on Control and Optimization*, 25(3):637–659, May 1987.