Proceedings of the
44th IEEE Conference on Decision and Control, and
the European Control Conference 2005
Seville, Spain, December 12-15, 2005

MoA01.1

A generic approach to the control of discrete event systems

André Arnold, Xavier Briand, Gérald Point and Aymeric Vincent

Abstract— In this paper, we present extensions of the framework of the μ-calculus which allow us to handle in a very generic and extensible way many control problems. The fundamental new tool is a division operator, and two new modalities are given as examples which allow us to handle observability and distinguishability. Furthermore, all this gives rise to a method for the synthesis of controllers which is implemented in a tool presented here.

I. INTRODUCTION

We discuss the problem of the synthesis of a controller to enforce a given process (or *plant*) to satisfy a given specification. We would like to present here our method [1], which extends the well-known framework of Ramadge and Wonham [2] by providing a very generic framework in which we can solve a variety of synthesis problems.

The most important part of the extension of the Ramadge and Wonham framework is that specifications are given in a logic (the modal μ-calculus). It is used to specify properties of a controlled plant but also restrictions on controller, as for example *a controller must accept an uncontrollable event at any point in time*. The approach via μ-calculus formulas permits specification of more general properties such as *an event is controllable until a failure event occurs*.

This allows us to express in a uniform way a wide variety of control problems. Moreover, finding their solutions can also be done in a uniform way: namely finding a model of a formula.

We will first present our method when no observability requirements are made on the controller. Most notably, this section will introduce a quotient operator which allows us to translate a synthesis problem into a problem of extracting a model from a formula. By generalizing this quotient operator, we obtain a framework which allows us to compute decentralized controllers.

In the next section, we describe a tool we have developed around this framework, and we give pointers to the algorithms we have chosen to implement.

Handling of unobservable events will be postponed until section IV, because we need to extend the μ-calculus in order to embed the observability constraints in our framework, and this comes with a few technical difficulties that would clutter a first presentation.

II. OUR METHOD

A. Processes

The plant to be controlled and the controller will be modelled as deterministic transition systems. By deterministic,

we mean that in any given state, any action (from a fixed set Σ) can lead to at most one state. Furthermore, in order to distinguish some states, every state will be tagged by a set of labels (from a fixed set Λ). Formally, a labelled process is defined as follows:

Definition 1: A *process* is a tuple $\langle Q, q^0, \delta, \lambda \rangle$ where Q is a finite set of states, $q^0 \in Q$ is the initial state of the process, $\delta : Q \times \Sigma \to Q$ is its transition function, and $\lambda : Q \to \mathcal{P}(\Lambda)$ is its labelling function.

An *execution* of a process is a sequence of actions that a process can take (a path in the transition system). In the framework of Ramadge and Wonham specifications talk only about the set of executions. In our framework, we will talk about the tree of executions (unfolding of the transition system).

From an automata-theoretic point of view, we are switching from word automata to tree automata. Rather than automata themselves we will use here an equivalent formalism that is better suited for our purposes. We will work with the propositional μ-calculus which is a modal propositional logic with fixpoint operators.

Because in our framework both a controller and a plant are processes, we need to model the action of the controller on the plant. This is done by defining the *synchronized product* of two processes. Basically, the synchronized product of two processes is itself a process which keeps track of the states in which the two processes are and allows only the actions which both processes allow.

Definition 2: The *synchronized product* $P = \langle Q, q^0, \delta, \lambda \rangle$ of two processes $\langle Q_1, q_1^0, \delta_1, \lambda_1 \rangle$ and $\langle Q_2, q_2^0, \delta_2, \lambda_2 \rangle$ is defined as:

$Q = Q_1 \times Q_2$, $\forall (q_1, q_2) \in Q, \forall a \in \Sigma, \delta((q_1, q_2), a) = (\delta(q_1, a), \delta(q_2, a))$ if $\delta(q_1, a)$ and $\delta(q_2, a)$ are defined and is undefined otherwise. The initial state is the pair of initial states: $q^0 = (q_1^0, q_2^0)$, and the labelling is taken here as the union of both labellings: $\forall (q_1, q_2) \in Q, \lambda((q_1, q_2)) = \lambda_1(q_1) \cup \lambda_2(q_2)$.

B. Logic

The logic we will use is the modal μ-calculus [3]. We will present this logic in three steps: the core of the logic is made up of the classical boolean operators and constants, then atomic propositions and *modalities* allow to express properties on a bounded part of the behaviours of processes, and finally, by grouping these expressions in systems of equations, we can express many properties on finite or

infinite behaviours of processes. For the reader familiar with other logics in the verification world, this logic is more expressive than CTL*.

The basis of the logic is the propositional calculus

$$\varphi ::= \bot \mid \top \mid p \mid \neg p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$$

where p ranges over the set of labels.

The semantics of a formula in a given state q of a given process $P = \langle Q, q^0, \delta, \lambda \rangle$ is given by

$P, q \not\models \bot$; $P, q \models \top$;
$P, q \models p$ if $p \in \lambda(q)$; $P, q \models \neg p$ if $p \notin \lambda(q)$;
$P, q \models \varphi_1 \vee \varphi_2$ if $P, q \models \varphi_1$ or $P, q \models \varphi_2$ holds;
$P, q \models \varphi_1 \wedge \varphi_2$ if $P, q \models \varphi_1$ and $P, q \models \varphi_2$ hold.

For every action in Σ, we introduce modalities which allow us to check if any or all of the neighbours of state q satisfy a given subformula.

$$\varphi ::= \langle a \rangle \, \varphi \mid [a] \, \varphi$$

where $a \in \Sigma$. We have
$P, q \models \langle a \rangle \, \varphi$ if $\exists q', q' = \delta(q, a)$ and $P, q' \models \varphi$
$P, q \models [a] \, \varphi$ if $\forall q', q' = \delta(q, a)$ implies $P, q' \models \varphi$

With this logic, we can describe requirements on the behaviours of processes, but only to a size which is bounded, roughly by the size of the formula.

For example, $[a] \, (\langle b \rangle \top \wedge [c] \bot)$ describes the states in which if an action a is possible, then after that action the process is in a state in which b must be possible and no c can occur.

In order to be able to express properties like "a possible execution of the system reaches a property p", we need more expressive power. We think that the most convenient way to extend the logic for this purpose is to introduce recursivity in the form of a system of equations.

An equation will be of the form $x = \varphi(x)$, where x is a variable introduced in the equation. Intuitively, and semantically, x represents a set of states which satisfy the fixpoint equation. For example, in order to describe the set of states from which there exists a path of b actions which leads to a property p, we can characterize this set as a solution of the following equation:

$$x = p \vee \langle b \rangle \, x$$

However, several sets of states satisfy this equation: for example in a process where there is a loop consisting of b's, the set of states in the loop will satisfy the equation even if there is no way to reach a state where p holds.

This motivates the introduction of a way to select two particular solutions among the possible ones: the least (set-wise) solution and the greatest solution are the two solutions we will consider. Here, if we want to enforce that at some point p is reached, we need to consider only the least fixpoint. If we were interested in finding states which lead to p or loop infinitely in b's then the greatest fixpoint would be our solution of choice.

We denote the least and greatest solutions of an equation respectively by placing a μ or a ν above the 'equal' sign. E.g.: $x \stackrel{\mu}{=} p \vee \langle b \rangle \, x$ or $x \stackrel{\nu}{=} p \vee \langle b \rangle \, x$.

In this article, we would like to avoid the intricacies needed to present the full power of this logic, but the reader has to know two things that allow to use its full expressiveness. Basically, equations can be used within ordered equations systems. This allows to compute a first set of states and then use it in another equation, to express properties like "from this set of states it is possible to reach a state such that from that state yet another state is not reachable".

Finally, it is also possible to make the equations mutually reference themselves. Mutually referencing equations of different kinds (greatest and least fixpoint) is what gives rise to the full expressive power of the μ-calculus; in order to express liveness properties such as "from this state, action b happens infinitely often", this mutual recursion is needed.

Results on the μ-calculus: The μ-calculus is studied for more than twenty years (see e.g. [4]). Among the results, one is of special interest to us: given a μ-calculus formula, it is possible to extract all the finite processes which satisfy this formula. We refer the interested reader to the original paper [1] covering all the details. This important fact means that if we can have a specification given by a μ-calculus formula we can find all the controllers we are interested in. Building such a formula is the aim of the next sub-section.

C. Quotient and generalized quotient

Basically, the previous statements mean that if we have a system of equations φ, we can compute a process C such that $C \models \varphi$. However, in the controller synthesis problem, we are interested in finding a C such that, given a plant P, $P \times C \models \varphi$. This problem can be reduced to the former, and for this purpose we introduce a quotient operator defined in such a way that:

$$P \times C \models \varphi \iff C \models \varphi/P$$

and φ/P is itself a system of equations.

Technically speaking, φ/P should rather be seen as a kind of product of φ with P, in the sense that the quotient is a system of equations which will have a variable for each possible pair (ψ, s) where ψ is a subformula of φ and s is a state of P.

The idea in defining this operator is that the new system of equations should take into account all the information that is available from P. More specifically, given a process $P = \langle Q, q^0, \delta, \lambda \rangle$ and a system of equations φ, every pair (s, ψ) of a state and a subformula of φ is replaced like this in the following cases:

(s, p) $\rightsquigarrow$ $\top$ if $p \in \lambda(s)$
                           $\bot$ otherwise
$(s, [a] \, \psi')$ $\rightsquigarrow$ $\top$ if $\delta(s, a)$ doesn't exist
                           $(s, [a] \, \psi')$ otherwise
$(s, \langle a \rangle \, \psi')$ $\rightsquigarrow$ $\bot$ if $\delta(s, a)$ doesn't exist
                           $(s, \langle a \rangle \, \psi')$ otherwise

Following this idea, it is also possible to define the quotient of two systems of equations φ/ψ, such that

$$\mathrm{Mod}(\varphi/\psi) = \bigcup_{P \in \mathbf{Mod}(\psi)} \varphi/P$$

where $\mathrm{Mod}(\varphi)$ denotes the set of processes which satisfy φ.

D. Application to the synthesis of controllers

Given a plant P, the aim of the synthesis of a controller is to find a process C called a controller which, when supervising P forces it to satisfy a control objective, given that for example the controller may not be able to prevent some of the events from occurring.

This problem can be recast in our framework as follows: the supervision of P by C is modelled as the synchronized product of P and C, and the control objective is given as a system of equations φ. Finding a controller C such that $P \times C \models \varphi$ can be achieved by extracting a process satisfying formula φ/P, as seen in the previous sub-section. It so happens that the fact that some events are uncontrollable can be expressed in the μ-calculus. Given a set Σ_{uc} of uncontrollable events, a controller which satisfies the following formula will not be able to prevent the occurence of events in Σ_{uc}:

$$x \overset{\nu}{=} \bigwedge_{a \in \Sigma_{uc}} \langle a \rangle\, x \,\wedge\, \bigwedge_{a \in \Sigma \setminus \Sigma_{uc}} [a]\, x$$

In other words, given a controllability constraint ψ on the controller, a process P, and a control objective φ, we can extract a controller from the system of equations $\varphi/P \wedge \psi$.

Decentralized control problems can also be treated in our framework thanks to the extended quotient operator, and the problem can be stated as follows: given a plant P, a control objective φ, and controller constraints ψ_1 and ψ_2, find two controllers C_1 and C_2 such that $P \times C_1 \times C_2 \models \varphi$. And this can be solved like this:

$$\begin{cases} P \times C_1 \times C_2 \models \varphi \\ C_1 \models \psi_1 \\ C_2 \models \psi_2 \end{cases} \iff \begin{array}{l} C_1 \models \varphi/\psi_2/P \wedge \psi_1 \\ C_2 \models \varphi/(P \times C_1) \wedge \psi_2 \end{array}$$

This generalizes to any number of controllers.

III. THE 'SYNTHESIS' TOOL

The presentation of our framework in this article is very simplistic. However, the whole method has been implemented in a tool which will be made available shortly on `http://altarica.labri.fr/` for free.

A. The method in practice

The `Synthesis` tool [5] is a command-line interpreter providing the user with commands for handling the objects involved in the synthesis of controllers:

- *Processes* describe plants and controllers. They are described in Mec 4 format [6].

- *Modal automata* are systems of fixpoint equations with modalities very similar to that presented in the previous section. Parities (natural numbers) or vectors of parities can be associated to equations instead of the μ and ν specifiers. Vectors are used to describe multi-parity conditions.
- *Two-player games with parity conditions* are used to compute processes encoding the controllers (see below).

Given a plant, a control objective and controller constraints, the user has to describe all steps of the method (this approach gives the user the opportunity to handle intermediate objects).

In a first step the user has to compute the modal automaton that must be satisfied by the controller. Here, one applies quotient and product operations as described in the previous section; if both operands are modal automata the result is a modal automaton equipped with a multi-parity condition.

The second step consists in transforming the multi-parity modal automaton into a parity game. This operation is mandatory and, unfortunately, generates an exponential blow-up in the size of the transformed object. This blow-up comes from the unfolding of the multi-parity condition using records similar to L.A.R [7]. The user can do the transformation directly or in several steps; he can change separately the multi-parity condition into a parity-condition and the nature of the object.

Finally the user computes a winning strategy in the parity-game obtained from the modal automaton specifying the controller. With this strategy, `Synthesis` produces a process encoding the expected controller; as explained in the previous section, in the case of decentralized supervision, this controller is reused in the first step of the method to compute the next controller. Winning strategies are obtained using the algorithm presented in [8]. This algorithm works in $\mathcal{O}(n^{d/2+1})$ in time and $\mathcal{O}(nd\log(n))$ in space where n is the number of positions of the game and d the number of different parities labelling positions.

B. A simple example

To illustrate the use of `Synthesis` we consider the plant P depicted on the figure 1. This process can do three actions a, c and d; only c and d are controllable. From its initial state 0, these actions might put P on a wrong way modeled by the state 5 (labelled with WW).
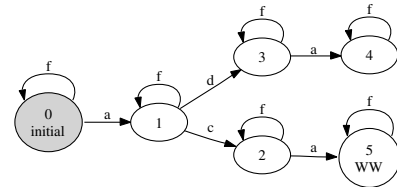

Fig. 1.   The plant P

We want to prevent P to go the wrong way. We will use two controllers. The first one, C_1, is a low-cost controller which can break down on an event f. In order to model the

fact that f can happen at any time, we have to handle f in each state of P; so we add an f-loop to each state of its transition system (see figure 1).

To fulfil the safety objective, we need a second controller C_2 which will be used as a standby redundancy. C_2 never fails and is started only when event f occurs.

The control objective ϕ that must be fulfilled by $P \times C_1 \times C_2$, specifies that any action puts the supervised system into a state where WW is not true:

$$\phi : x \overset{\nu}{=} \neg WW \wedge \left(\begin{array}{ll} & \langle a \rangle\, x \wedge [c,d,f]x \\ \vee & \langle c \rangle\, x \wedge [a,d,f]x \\ \vee & \langle d \rangle\, x \wedge [a,c,f]x \\ \vee & \langle f \rangle\, x \wedge [a,c,d]x \end{array} \right)$$

$(e_1, \ldots, e_n)x$ is a shortcut for $(e_1)x \wedge \ldots (e_n)x$ where $(.)$ is $\langle . \rangle$ or $[.]$).

Now we have to describe the controller constraints: ψ_1, the constraint for C_1, specifies that events c and d are under control until the occurence of event f:

$$\psi_1 : \left\{ \begin{array}{l} x \overset{\nu}{=} \left(\begin{array}{ll} & \langle a \rangle\, x \wedge [c,d]x \wedge \langle f \rangle\, y \\ \vee & \langle a,c \rangle\, x \wedge [d]x \wedge \langle f \rangle\, y \\ \vee & \langle a,d \rangle\, x \wedge [c]x \wedge \langle f \rangle\, y \\ \vee & \langle a,c,d \rangle\, x \wedge \langle f \rangle\, y \end{array} \right) \\ y \overset{\nu}{=} \langle a,c,d,f \rangle\, y \end{array} \right.$$

ψ_2 specifies that C_2 does not take into account events c and d until the failure of C_1 when f occurs:

$$\psi_2 : \left\{ \begin{array}{l} x \overset{\nu}{=} \langle a,c,d \rangle\, x \wedge \langle f \rangle\, y \\ y \overset{\nu}{=} \left(\begin{array}{ll} & \langle a,f \rangle\, y \wedge [c,d]y \\ \vee & \langle a,c,f \rangle\, y \wedge [d]y \\ \vee & \langle a,d,f \rangle\, y \wedge [c]y \\ \vee & \langle a,c,d,f \rangle\, y \end{array} \right) \end{array} \right.$$

It remains to ask `Synthesis` to compute the controllers; this task is achieved by executing the tool with the script listed and explained below.

First we load the plant `P` and the formulas `phi`, `psi1` and `psi2`. The next step computes the modal automaton `SpecC1`$= (\phi/\psi_2/P) \wedge \psi_1$. Then we compute `C1` as follows: `pgame` produces the parity game associated with `SpecC1`, `strategy` computes a winning strategy which is used by `control` to generate the controller. We use the command `minimize` in order to reduce the number of states of the controller.

```
load plant.mec specs.fam

SpecC1 := product (quotient phi psi2 P) psi1
C1 := minimize (control (strategy \
                 (pgame SpecC1))
```

The second controller is obtained in a similar way, except that we start by computing the system `P_C1` supervised by `C1`. This process is used to compute the modal automaton `SpecC2`$=\phi/(P \times C_1) \wedge \psi_2$ that must be satisfied by C_2.

```
P_C1 := sync P C1
SpecC2 := product (quotient phi P_C1) psi2
C2 := minimize(control (strategy \
                  (pgame SpecC2)))
```

Finally we ask `Synthesis` to compute the whole supervised system `P_C1_C2` and to output it with C_1 and C_2 using the `dot` graph format [9].

```
P_C1_C2 := sync P_C1 C2

dot C1 C2 P_C1_C2 > result.dot
```

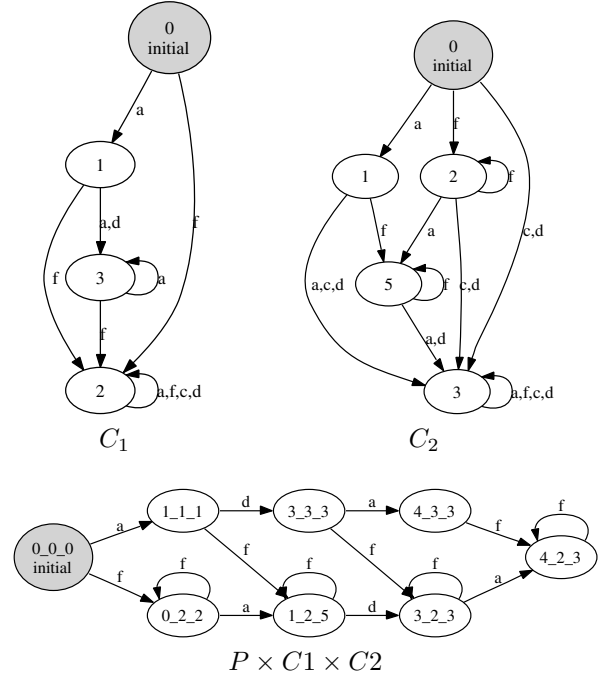The two controllers and the supervised system are depicted on figure 2.

Fig. 2. The expected controllers and the supervised system. On the left, C_1, it can be partially blocked on an event f. On the right is depicted C_2 the redundant controller. Below, the supervised system which never executes action c which would otherwise lead to the wrong way.

IV. PARTIAL INFORMATION AND UNDECIDABILITY

In the area of controller synthesis, it is very often considered that the controllers do not know all the behaviors (i.e. sequences of actions) of the plant but have only *partial information* about it. It would be nice to be able to express this kind of constraints in our systems of equations, and if it were the case, we would immediately handle it in our framework.

A. Notion of unobservability

In most of the studies, the notion of partial information is the case of *unobservable events*: the controller does not see all the events which the plant reacts to. Therefore, it amounts to saying that upon receipt of this event, the controller is not allowed to change state. A logic which could state such a property would not be "bisimulation invariant": it could distinguish between processes which have the same behaviour but not exactly the same structure. As it is known that the μ-calculus *is* bisimulation invariant, we cannot express unobservability constraints as-is in our framework.

B. Notion of indistinguishability

One can also take into account the case of *indistinguishable events*: the controller can detect the occurrence of an event but is not able to distinguish it from certain events. That comes down to saying that upon receipt of indistinguishable events, the controller is not allowed to reach different states. Remark that, for each state of the controller, we can have various classes of indistinguishable events. As for unobservable events, we cannot express indistinguishability constraints with the standard modal μ-calculus.

C. Framework extension

This is the motivation behind the introduction of new modalities, the loop modality, denoted by $LOOP_a$ and the convergence modality, denoted $CONV_{a,b}$. A state s of a process satisfies the modality $LOOP_a$ if it has an a-transition from itself to itself, i.e. if $\delta(s,a) = s$. Moreover, s satisfies the modality $CONV_{a,b}$ if it has an a-transition and a b-transition such that $\delta(s,a) = \delta(s,b)$.

It so happens that these extensions don't break the major results of the μ-calculus we are interested in, and that with this extension, we can express the observability and indistinguishability constraints a controller has to satisfy.

For instance, with these new modalities, one can specify that a subset of events $\Sigma_{uo} \subset \Sigma$ are unobservable:

$$x \stackrel{\nu}{=} \left(\bigwedge_{a \in \Sigma_{uo}} LOOP_a \vee [a] \bot \right) \wedge \bigwedge_{a \in \Sigma \setminus \Sigma_{uo}} [a] x$$

One can also specify a partition $\{\Sigma_1, \Sigma_2, \ldots, \Sigma_n\}$ of Σ of indistinguishable events, i.e., events are indistinguishable if they belong to the same Σ_i:

$$x \stackrel{\nu}{=} \left(\bigwedge_{1 \leq i \leq n} \bigwedge_{a,b \in \Sigma_i} CONV_{a,b} \vee [a] \bot \vee [b] \bot \right) \wedge \bigwedge_{a \in \Sigma} [a] x$$

Everything we have explained so far extends seemlessly to these extensions except for the quotient operator φ/ψ which is defined only if ψ is a "classical" system of equations, without any LOOP and CONV modalities.

D. Undecidability

As soon as two controllers have to satisfy observability *or* indistinguishability constraints, the problem of knowing if two such controllers exist is undecidable. This is proved in [1] by reducing the Post correspondance problem to a control problem with two controllers having to satisfy unobservability constraints.

REFERENCES

[1] A. Arnold, A. Vincent, and I. Walukiewicz, "Games for synthesis of controllers with partial observation," *Theoretical Computer Science*, vol. 303, no. 1, pp. 7–34, June 2003. [Online]. Available: http://www.sciencedirect.com/science/article/B6V1G-487DRHS-1/2/314a88f9636f4887be6660a19eec19ad

[2] P. Ramadge and W. M. Wonham, "The control of discrete event systems," in *Proceedings of the IEEE*, vol. 77, Jan. 1989, pp. 79–98.

[3] D. Kozen, "Results on the propositional μ-calculus," *Theoretical Computer Science*, vol. 27, pp. 333–354, 1983.

[4] A. Arnold and D. Niwiński, *Rudiments of μ-calculus*, ser. Studies in Logic and the Foundations of Mathematics. North-Holland, 2001.

[5] G. Point, "The Synthesis Toolbox - From modal automata to controller synthesis," LaBRI, Tech. Rep. RR-1342-05, 2005.

[6] A. Arnold, D. Bégay, and P. Crubillé, *Construction and analysis of transition systems with MEC*. World Scientific, 1994.

[7] S. Dziembowski, M. Jurdziński, and I. Walukiewicz, "How much memory is needed to win infinite games?" in *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science*, Warsaw, Poland, 1997, pp. 99–110.

[8] J. Bernet, D. Janin, and I. Walukiewicz, "Permissive strategies: from parity games to safety games," *Theoretical Informatics and Applications (RAIRO)*, vol. 36, no. 3, pp. 261–275, 2002.

[9] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software — Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.