Controlling Contractors with Monads for Hybrid Dynamical Systems

Gilles Chabert, Rémi Douence

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Controlling Contractors with Monads for Hybrid Dynamical Systems

Gilles Chabert — Rémi Douence

N° 7451

Novembre 2010

Distributed Systems and Services

ISSN 0249-6399 ISRN INRIA/RR--7451--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Controlling Contractors with Monads for Hybrid Dynamical Systems

Gilles Chabert*, Rémi Douence†

Theme : Distributed Systems and Services
Équipe-Projet Ascola

Rapport de recherche n° 7451 — Novembre 2010 — 20 pages

Abstract: Physical systems with intelligent behaviors result from inter-actions of different fields: sensor networks, robotics, optimization, reasoning, etc. Rooted in this philosophy of interdisciplinary, this paper makes a connexion between hybrid dynamical systems, interval-based constraint propagation and functional programming. It shows how to build a monadic program in Haskell to control contractors (constraint propagators) for the state estimation of multi-model (hybrid) dynamical systems, subject to partial and uncertain measurements. The example of system taken here is an elevator that can either be moving upward, downward or stopped. The altitude is measured directly and the estimation problem is simply to track its motion. The purpose of the Haskell library is to offer both a high-level and flexible framework for building propagation strategies based on user knowledge or user requirements.

Key-words: hybrid dynamical systems, continuous constraints, monads

* Contraintes, Mines de Nantes, LINA, CNRS, UMR 6241, 4 rue Alfred Kastler 44300 Nantes, France, gilles.chabert@mines-nantes.fr

† Ascola, Inria, Mines de Nantes, 4 rue Alfred Kastler 44300 Nantes, France remi.douence@inria.fr

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Contrôler des contracteurs avec des monades pour les systémes hybrides dynamiques

Résumé : L'élaboration de sytèmes autonomes est le résultat d'interactions entre différents domaines: réseau de capteurs, robotique, optimisation, raisonnement automatique, etc. Cet article, ancré dans cette philosophie pluridisciplinaire, montre une connexion entre les systèmes hybrides dynamiques, la propagation de contraintes sur intervalles et la programmation fonctionnelle. Plus précisément, il montre comment concevoir un programme monadique en Haskell pour contrôler les contracteurs (propagateurs de contraintes) permettant l'estimation d'état de systèmes dynamiques multi-modéles (hybrides), sujets à des mesures partielles et incertaines. L'exemple de système pris ici est un ascenseur qui peut monter, descendre ou être stopé. L'altitude est mesurée directement et le problème d'estimation est simplement de suivre son mouvement. L'objectif de la bibliothéque Haskell est d'offrir un cadre à la fois flexible et de haut niveau pour construire des stratégies de propagation basées sur les connaissances ou les pré-requis de l'utilisateur.

Mots-clés : systèmes hybrides dynamiques, contraintes continues, monades

1 Introduction

Many real-life systems are continuous by nature but controlled by discrete inputs (switches, valves, digital devices). The typical example is a gas burner with a boolean state (*leaking/not leaking*) and continuous variables representing timers or quantity of leaked/stored gas.

In these systems, two kinds of dynamics are thus interacting: the continuous one, governed by physics laws, and the discrete one, unpredictable or subject to randomness. These systems are usually referred to as *hybrid systems* [6]. One problem with hybrid systems is called *observation*, that is, estimating the system state from both the knowledge of its dynamics and measurements supplied by sensors.

This paper proposes a first constraint-based approach for the state estimation of hybrid systems, i.e., observable dynamical systems subject to configuration changes. A simple example of an elevator with elastic cable is used all along the paper to illustrate the concepts.

Estimation is made by contracting (filtering) domains with respect to the physics equation in a branch & bound process where branching are choices in the set of possible configurations.

To be controlled in a suitable way, constraint propagation requires in this context different levels of abstraction. We will identify three of them; basically, the first corresponding to the discrete events, the second to the continuous dynamics and the last to the integration of measurements. Each level has its own strategy and the solver is obtained as a composition of them.

Therefore, the goal is to generate a dedicated solver by composing contractor strategies acting at different levels. Programming this with an imperative language would clearly be very tedious and doomed to result in a poorly maintainable software. Our contribution is a framework written in a functional language, Haskell, where the power of *monads* is used to properly address the required composability.

This paper is *not* about interval computations but about how to control them in a specific (but significant) context: hybrid dynamical systems. For that reason, the Haskell program itself does not perform numerical computations. The job is left to an interval constraint library, the details of which being not relevant for this paper. The Haskell program solely requires the interface of each operation to match the definition of *contractor* (see below).

The interval library used for our experiments was `Ibex` [3], but the above requirement on the interface is so simple that any other library could be easily plugged as well. This makes our approach generic.

In Section 1.1, we briefly review the different existing approaches for hybrid systems and point out the benefits of the constraint paradigm in this area. In Section 1.2, we introduce how contractors (or propagators) are used so far in dynamical systems. We motivate then why building an upper layer on top of contractors is necessary to tackle hybrid systems and why Haskell is a good choice for this aim. For simplicity, the paper focuses then on a particular example: the tracking of an elevator and the expected features of the Haskell framework, in regard to this application, are listed in §1.4.

The technical contribution of the paper is in Section 2. All the features are implemented in Haskell and the results obtained are shown graphically. The Haskell implementation is detailled in Section 3.

Perspectives of this work are finally discussed in a conclusion.

1.1 Hybrid Systems

Hybrid systems in the literature are often modeled by *hybrid automata* where vertices represent configurations (with associated dynamics law) and transitions configuration switches. These transitions are tagged with explicit constraints, making the system deterministic. Formal verification methods exist typically to check safety properties (reachability analysis) of such systems [7][2]. Estimation can be achieved by tuning solvers dedicated to differential equations like VNode [11] since hybrid automata are not far different from piecewise EDOs. Note that solvers for global optimization (e.g., Baron [15]) cannot be employed in dynamical systems because the number of variables resulting from time discretization is too big, even with finite degrees of freedom.

Hybrid automata do not integrate observability (measurements) nor the fact that transitions may depend on unpredictable events (obstacles, etc.). For non-deterministic systems interacting with the environment, estimation can be made in the setting of Markov decision processes [1]. In this case, the measurement noise is modeled explicitly (e.g., by a Gaussian distribution) and transition have some associated probability. Classical Bayesian observers (e.g., Kalman filter) can then be adapted to manage the possible transitions in the prediction and update of state distributions.

The probabilistic approach has fundamental advantages: uncertainties are more finely represented by a probability than by two simple bounds. Propagation of uncertainties is therefore more informative. This is blatant if measurements are potentially very inaccurate but with a small deviation and if the system behavior is poorly constrained. Under these conditions, intervals can get quickly arbitrary large, i.e., irrelevant, whereas probabilities supply expected values with good confidence.

However, distributions are not compatible with constraint reasoning and backtracking schemes: they can only be propagated safely with linear constraints and cannot be bisected anyway. On the contrary, intervals allow to integrate external constraints that typically arise when the system has a perception of the environment. An example is a robot detecting objects that have only a fixed number of occurrences. This knowledge can be integrated in the estimation process as a global constraint [5], providing significant contraction. Intervals also supply validated solutions, including with nonlinear equations.

1.2 Contractors

The concept of contractor in interval constraints (i.e., constraints over the reals) is similar to the concept of *propagator* with discrete domains. We first give the definition of a contractor and then show the main existing variants. Contractors for dynamical systems will be our base ingredient afterwards.

The following notations will be used. All the unknowns are real and have their domains represented by intervals. The set of real intervals is denoted by $\mathbb{IR}$. A vector of intervals in $\mathbb{IR}^n$ is called a *box*. Intervals and boxes are represented by bracketed symbols, *e.g.,* $[x]$.

Contractor A contractor is an operator $C : \mathbb{R}^n \to \mathbb{R}^n$ such that[1]

$$\forall [x] \in \mathbb{IR}^n,\ \mathcal{C}([x]) \subseteq [x].$$

1.2.1 Contractors in classical CSPs.

Interval constraints have been successfully used in global optimization; especially for finding guaranteed solutions of nonlinear equations since this problem fits in well with the framework of CSP (Constraint Satisfaction Problems). The problem is to find the set of vectors x such that

$$f(x) = 0,\ \text{where } f : \mathbb{R}^n \to \mathbb{R}^m\,. \tag{1}$$

To solve (1), contractors have been developed to satisfy

$$\forall [x] \in \mathbb{IR}^n,\ \forall x \in [x],\ (f(x) = 0) \Longrightarrow x \in C([x]),$$

that is, all the values removed from $[x]$ are not solutions. A survey of techniques used for building contractors is given in [12]. From the language perspective, a formalism to build complex contractors from basic ones is proposed in [4] through the concept of *contractor programming*. As an example, a complex contractor can encapsulate a fine control on the constraint processing order.

Solving (1) is performed then by a "branch & contract" process. A search loop, or *paver*, is therefore easy to supply. Note that, once the branching heuristic is set up, one can decide precisely the form under which the output has to be produced, simply by programming contractors in a suitable way [4].

Hence, there is not much a need for controlling contractors apart from choosing the branching heuristic (*i.e.,* how to select the next current box and how to bisect it). Our point is that such a unique all-in-one algorithm is unlikely to exist for hybrid dynamical systems.

1.2.2 Dynamical Systems.

The potential of interval constraints is far to be restricted to optimization-like problems. Contractors are also successfully employed for calculating validated enclosures of the solutions of nonlinear ordinary differential equations [11]. The approach is of great interest including for equations involving input vectors and measurement vectors, that is, for systems studied in automation. The main application is the state estimation of nonlinear dynamical systems [9]. We will not consider input vectors in the sequel. Under this restriction, the problem is to characterize in a time window $[t_0, t_N]$ the set of functions $x : [t_0, t_N] \to \mathbb{R}^n$ such that:

$$x(t_0) = x_0 \quad \text{and} \quad \forall t \in [t_0, t_N] \begin{cases} \dot{x}(t) = f(x(t)) \\ y(t) = g(x(t)) \end{cases} \tag{2}$$

where x_0 is the initial state and $y(t)$ the observation. We shall consider henceforth a discretization $(t_0, t_1, \ldots, t_N)$ of $[t_0, t_N]$ with a fixed time step $\delta_t = t_i - t_{i-1}$, $\forall i > 0$. Measurements usually only provide imprecise values of $y(t)$ for t in a subset of the t_i's.

[1] There are actually some additional properties in the definition, but with no incidence here.

Definition (Model-compatibility). Given a motion model, *i.e.,* a function f, a couple (x_{i-1}, x_i) is said to be *f-compatible* if there exists a trajectory $z : [0, \delta_t] \to \mathbb{R}^n$ such that $z(0) = x_{i-1}$, $z(\delta_t) = x_i$ and $\dot{z}(t) = f(z(t))$ for all $t \in [0, \delta_t]$.

Remark. Since f does not depend on t (the system is *autonomous*), we see that model-compatibility between states at two successive instants does not actually depend on the instants (*i.e.,* the subscript i) but only on the states themselves. This will have consequences further.

Similarly, we will say that x_i is compatible with the i^{th} measurement if either there is no measurement at $t = t_i$ or $g(x_i) \in \tilde{y}_i \pm \delta_i$, where $\tilde{y}_i$ stands for the measurement value (got from the sensors) and δ_i a bound on the measurement error. If we set $[y]_i := \tilde{y}_i \pm \delta_i$ and $[y]_i := (-\infty, +\infty)$ in case of unavailable measurement, we see that, at all events, $g(x) \in [y]_i$.

In this context, we usually resort to

1. a *model contractor* M that takes as input a $(2 \times n)$-box $([x]_{i-1}, [x]_i)$ and yields a subbox $([x]'_{i-1}, [x]'_i)$ such that all model-compatible couples (x_{i-1}, x_i) of $[x]_{i-1} \times [x]_i$ belong to $[x]'_{i-1} \times [x]'_i$.

2. a *measurement contractor* M^{μ}_i that takes as input a box $[x]_i$ and yields a box $[x]'_i$ such that all the values consistent with the i^{th} measurement, if any, are kept.

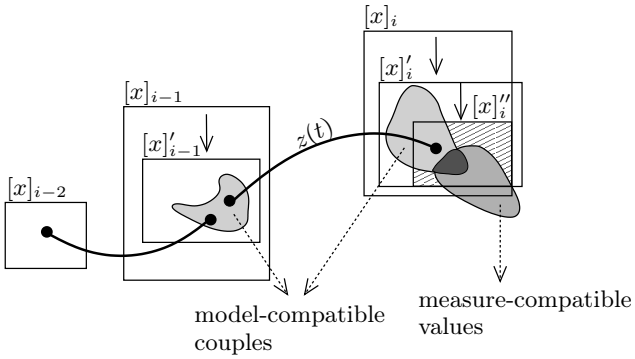

Figure 1: Contractors in dynamical systems. The box $[x]_i$ is contracted by the model to $[x]'_i$ .The latter is guaranteed to contain all the extremities $z(\delta_t)$ (marked with black circles) of the trajectories $z(t)$ obeying the differential equation and whose other extremity $z(0)$ is inside $[x]_{i-1}$. The box $[x]'_i$ can be contracted in turn thanks to a measurement to obtain $[x]''_i$.

This is illustrated in Figure 1. Examples of recent methods to build model contractors are in [14]. To solve (2), we can still use a all-in-one algorithm that basically calculate the fix-point of model contractors and measurement contractors (bisection is prohibited in general). Note, however, that a fix-point is dramatically time-consuming for a large number of time steps.

1.3 Motivating Example

The example chosen in this paper is an elevator that can either be moving upward (`up`), downward (`down`) or stopped (`stop`), the latter resulting into oscillations around the halting position. In this example, the state x has three components: the altitude of the elevator, its speed and the cable stretch. However, we will only observe the first one, the altitude. Whether the elevator example corresponds to a real-life hybrid system or not does not matter.

The goal is to track the elevator, *i.e.,* to calculate its altitude with time, from noisy and partial measurements. Of course, configuration changes are unknown. Model contractors are used to make consistent two consecutive positions with respect to a configuration of the elevator (either `up`, `down` or `stop`) at the corresponding time. So there is a different contractor for each configuration, and one must decide which contractors are enforced in turn to obtain an overall *feasible* trajectory, that is, as thin as possible and consistent with all the measurements. Let us call this a *strategy*. Of course, a choice point is necessary when two different models are compatible with measurements (either for real or apparently) at a given time step. So a strategy acts as a solver. The less backtracks, the better the strategy.

1.4 Contribution of the Paper

This paper is not intended to give a turn-key strategy for the state estimation of the elevator. This would be of little interest because the success of a strategy strongly depends on the problem. A good strategy for our toy example may not be applicable elsewhere.

The purpose is to describe a framework for the development of strategies. More precisely, the program in the next section can be viewed as a *kit* to build your own strategy.

The main qualities of our framework are modularity and expressiveness. First, it gives the ability to develop a strategy by working at different levels. Second, each level can be programed in a flexible and independent way. There are three levels, each corresponding to a different abstraction of the problem.

1.4.1 level 1.

The behavior of the system is often constrained *globally*. For instance, one may know that the scenario is a sequence of `up`, followed by a sequence of `stop` and a sequence of `down` (but not the times when switches occur). The system may also have some *latency*: once a switch occurs, the new model remains valid for a lapse of time. This type of knowledge is modeled with an automaton and Level 1 refers to the integration of such automata into strategies.

1.4.2 level 2.

Once a sequence of models is enforced, one may decide to propagate contractors in different ways. Model and measurement contractors can be interleaved in a specific order, some contractors can be called twice, etc. Level 2 refers to the choices made in a strategy for calculating an enclosure of the propagation fix-point.

1.4.3 level 3.

This level contains the structure and functions at the core of the search process: choice points, fails and backtracks.

Our program will be illustrated on the elevator tracking example. The target strategy will have to be based on the following requirements, one for each level:

1.4.4 req 1.

It is known that the elevator starts by `up` and necessarily stops between the configurations `up` and `down` (in this order and in the other way around). Furthermore, each model has a latency of 4 time steps.

1.4.5 req 2.

When one estimation is sharpened by a measurement, all the estimations at the previous time steps have to be smoothed accordingly.

1.4.6 req 3.

When two models can be applied, the one that leads to an estimation that maximizes likelihood with respect to the measurement should be tried first.

Do we have to extend contractor programming to achieve this goal? Probably not. The purpose of contractor programming is to build complex contractors from basic ones, not to control them. This is precisely why branching heuristics were overlooked in [4]. Branching is a matter of controlling contractors, as said above. Furthermore, controlling is more complex in the case of hybrid dynamical systems and involves user knowledge in a similar way as *business rules*. It cannot be reduced to the parameterization of a monolithic algorithm.

1.4.7 A Good Candidate: Haskell

We have decided to program our framework in Haskell [10]. Haskell is a functional, lazy, higher-order and typed programming language. The main implementation of Haskell, GHC relies on the C compiler `gcc` which makes it simple to link an Haskell program and a C program (such as Ibex). It has recently been successfully used to control discrete constraint solvers such as `Gecode` [16, 17]. Higher-order and lazy features of Haskell makes it possible to define domain specific languages as sets of combinators. For instance, the monadic parser library [8] provides BNF-like operators. They enable users to program backtracking parsers simply by defining grammars. This library strongly inspired us to define (backtracking) controllers as grammars. In the next section, we present our framework.

2 Controlling Contractors in Haskell

Let us first restate the problem. A hidden command sequence (say, `up-up-down-stop`-etc.) or *scenario* has given a trajectory to the elevator. At some moments, the state of the elevator can be estimated from measurements. There may be different scenario compatible with these measurements and our goal is to find

one sequence, *i.e.,* one of the solutions. This solution is not necessarily the hidden sequence.

The problem can formally be cast into a CSP as follows. There are two sets of variables: $\mathcal{X} = \{x_0, \ldots, x_N\}$ and $\mathcal{U} = \{u_1, \ldots, u_N\}$. For all time step, x_i is the i^{th} state vector and u_i the *command*, that is, the model which applies between t_{i-1} and t_i. For all i, domains for x_i and u_i are $[x]_i = (-\infty, +\infty)^3$ and $\{\texttt{up}, \texttt{stop}, \texttt{down}\}$ respectively.

Let us denote c_i^μ the constraint $y_i = g(x_i)$ and $c_i^{\texttt{UP}}$, $c_i^{\texttt{STOP}}$ and $c_i^{\texttt{DOWN}}$ the constraints that (x_{i-1}, x_i) are compatible with `up`, `stop` and `down` respectively (see the definition of model-compatibility above).

The set of constraints is therefore:

$$\begin{array}{c}(c_0^\mu \wedge \ldots c_N^\mu) \wedge \\ (c_1^{\texttt{UP}} \wedge \neg c_1^{\texttt{STOP}} \wedge \neg c_1^{\texttt{DOWN}} \iff u_1 = \texttt{up}) \wedge \ldots \\ \ldots \wedge (c_N^{\texttt{DOWN}} \wedge \neg c_N^{\texttt{UP}} \wedge \neg c_N^{\texttt{STOP}} \iff u_N = \texttt{down}).\end{array}$$

It makes no sense to look for solutions in terms of $(\mathcal{X},\mathcal{U})$ since (1) the problem is extremely under-constrained (partial and noisy measurements) and (2) we cannot check satisfiability for model constraints (only contraction is possible). The goal is rather to find solutions in terms of $\mathcal{U}$ only. A *solution* is an instantiation of $\mathcal{U}$ such that propagation leads to a nonempty set.

An alternative way of proceeding is to consider the set of 3^N CSP:

$$\begin{array}{c}(c_0^\mu \wedge \ldots c_N^\mu) \wedge (c_1^{\texttt{UP}} \wedge c_2^{\texttt{UP}} \wedge \ldots \wedge c_N^{\texttt{UP}}), \ldots, \\ \ldots, (c_0^\mu \wedge \ldots c_N^\mu) \wedge (c_1^{\texttt{DOWN}} \wedge c_2^{\texttt{DOWN}} \wedge \ldots \wedge c_N^{\texttt{DOWN}}),\end{array}$$

each scenario being identified to a unique scenario $\mathcal{U}$. The idea is then to find the first consistent scenario.

Such a generate and test solution avoids disjunctions or reified constraints. However the number of scenarios is unacceptable. There are many scenarios that should be discarded thanks to a priori knowledge on the system behavior (as said in §1.4). Moreover, there are many redundant computations: propagation for two scenarios sharing a subchain of constraints should be factorized.

2.1 A Search Tree with Incremental Propagation

Let us take an example. We know that the elevator starts by moving up and keeps moving up until it is stopped. It remains then stopped indefinitely. A grammar modeling this behavior is

$$\begin{array}{ll}(R_1) & U \to \texttt{up}\ U \\ (R_2) & U \to \texttt{up}\ S \\ (R_3) & S \to \texttt{stop}\ S\end{array}$$

The basic idea is to build a search tree that generates all the words accepted by the automaton (see Figure 2) and to remove the inconsistent ones as soon as possible.

A node at depth i in this tree is therefore the i^{th} terminal that can be read and corresponds to a *choice*, that is, the instantiation of u_i. An edge between u_i and u_{i+1} represents the rule applied. Propagation is performed along the way: each time a choice is made (a new node u_i appears in the tree), this choice is propagated by adding a new contractor on-the-fly. Of course, the effect of the

new contractor is canceled when backtracking. This is illustrated in Figure 2, with $N = 3$ (three time steps). The consistency at the grey-filled node is checked by propagating the choice $u_3 = \texttt{stop}$ in a contractor list inherited from the node with dashed contour. This list was built with $(u_1, u_2) = (\texttt{up}, \texttt{up})$ and consistency was enforced by the node with dashed contour.

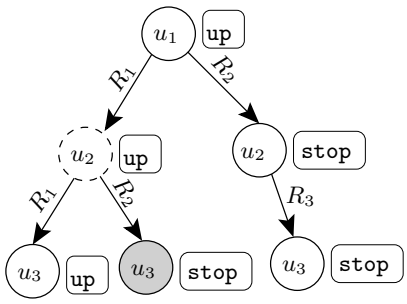

Figure 2: Research Tree for the Grammar (R_1, R_2, R_3).

Copying all the $[x]_i$'s when branching is clearly non affordable and trailing has to be used. Hence, the solver state (not to be confused with the state x) contains a *trail*: a list that incrementally records the changes brought to the $[x]_i$'s. It also contains the choices made so far $(u_1, \ldots)$ and a *clock* that indicates on which time step the next choice will operate.

2.2 A Two-Layered Structure of Contractors

Each new choice leads to a sequence of contractions. Let us consider the grey-filled node in Figure 2 and denote by $>>$ the composition of two contractors. The basic sequence is

$$M_3^{\texttt{STOP}} >> M_3^{\mu}, \tag{3}$$

that is, the state interval $[x]_3$ is estimated by applying the model contractor for `stop` on $([x]_2, [x]_3)$ and subsequently updated with the measurement contractor. However, if the measurement entails a significant contraction, one might also decide to refine the past state estimations by propagating backward with the contractors. This would give here:

$$M_3^{\texttt{STOP}} >> M_3^{\mu} >> M_3^{\texttt{STOP}} >> M_2^{\texttt{UP}} >> M_1^{\texttt{UP}}. \tag{4}$$

Since a composition of contractors is a contractor, sequences (3) and (4) form two different versions of a compound contractor $\mathcal{C}_3^{\texttt{stop}}$ resulting from the choice $u_3 = \texttt{stop}$. Hence, the overall propagation for the scenario (u_1, u_2, u_3) corresponding to the grey-filled node is a sequence of first-level contractors

$$\mathcal{C}_1^{\texttt{up}} >> \mathcal{C}_2^{\texttt{up}} >> \mathcal{C}_3^{\texttt{stop}}, \tag{5}$$

each first-level contractor $\mathcal{C}_i$ corresponding to a choice for u_i and being itself a composition of second-level contractors, as shown above.

Why such a structure? The first-level sequence reflects the scenario and is therefore impacted by the behavior of the system. On the contrary, the

second-level sequence basically only results from tradeoffs between efficiency and accuracy. It is not directly related to the system. Both levels should be programed in a complete separate way.

2.3 First-Level Strategy

As said in the remark of §1.2, time is a *free* variable in our system. This means that the user knowledge makes never appear time explicitly. Take the example of *switches with latency* (cf. **Req 1**). The strategy tells that when the system switches from `up` to `stop` then `stop` is valid for 4 additional time steps. The subscript i is never specified. Furthermore, first-level sequences like (5) are synchronized with the clock so that time could be made implicit.

The idea is to drop subscripts and to let first-level contractors manage the *ticks* (clock increments). The first-level sequence (5) becomes in the syntax of our framework:

$$\texttt{stepUp} >> \texttt{stepUp} >> \texttt{stepStop}. \tag{6}$$

A first-level contraction simply applies the sequence of second-level contractors and increments the clock (see next section).

The nice point is that, under the form (6), a scenario is both an operational program (that check consistency of the scenario) and a word of Σ^*, where Σ is $\{\texttt{up}, \texttt{stop}, \texttt{down}\}$. This will give us the ability to generate scenarios from grammars (that is, to build our solver) in the fashion of parsers.

But instead of building a parser for these grammars that would generate solvers with obscure code, we shall use *monads* (cf. §2.5), a very powerful feature of functional languages that allows to build *executable* grammars like (6). Let us give now an example of a valid strategy that fulfills our first requirement.

(Req 1). The following grammar introduces latencies of 4 time steps and eliminate the forbidden transitions (from `up` to `down` and conversely):

```
up  = repeat 3 stepUp » up'
up' = stepUp » (end `or` (up' `or` stop))
stop  = repeat 3 stepStop » stop
stop' = stepStop » (end `or` (stop' `or` down `or` up))
down  = repeat 3 stepDown » down'
down' = stepDown » (end `or` (down' `or` stop))
```

Indeed, the first non terminal `up` starts by a sequence of 3 `stepUp`, then (`»`), the non terminal `up'` also starts by one `stepUp`. The operator `or` composes two executable grammars. It is similar to the choice operator | of BNF. For instance, in this grammar, `up'` leads either to the terminal `end` (which succeeds when the clock is equal to the final time step), or it loops to `up'`, or it leads to `stop`.

The graphical result of this controller for the elevator is shown in Figure 3. This figure represents the possible altitudes of the elevator in function of time, calculated by our strategy. The initial position of the elevator has been measured and is quite precise (almost reduced to a point). Then a model (here `up`) predicts the next position of the elevator, and so on. Positions get less and less precise until a new measurement comes up and constrains the position again.

We can first observe visually that the shape of the trajectory follows the hidden scenario, shown by the dashed arrows in the figure. However, the uncertainties make other scenarios possible. In the solution given by our strategy, the first switch occurs at $t = 52$ instead of $t = 46$. The figure suggests that this

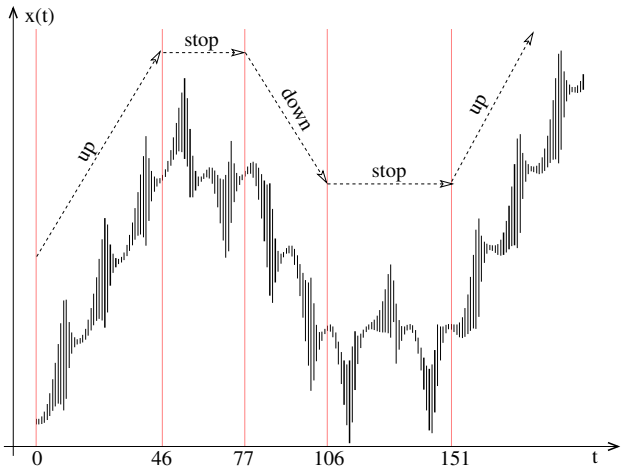

Figure 3: Inference of a scenario. Model switches of the hidden scenario are marked by the vertical lines. The inferred scenario differs significantly. Some switches are just shifted (*e.g.,* `stop` at $t = 111$ instead of 106) but superfluous switches also appear at $t = 56$, 65, etc. The stability of the solution is improvable simply by increasing the latency.

shift is probably not imputable to the strategy but to the inherent uncertainty (that is, the lack of measurements). Indeed, it is apparent that the first model switch can be postponed a few more steps after 46. The sequence of switches returned by our strategy is:

```
number of backtracks = 5059
models are [(up,0),(stop,52),(down,56),(stop,65),
(up,69),(stop,73),(down,77),(stop,111),(up,116),
(stop,127),(down,131),(stop,142),(up,146)]
```

From the time step 0 to the time step 51 the model up has been applied, then from 52 to 55 the elevator has stopped, then from 56 to 64 it has moved down, etc.

2.4 Second-Level Strategy

To obtain (3), we write in Haskell

```
stepUp = do
  contract up
  contract measurements
  tick
```

where `contract up` and `contract measurements` are simply monadic wrappers for $M_i^{\mathtt{UP}}$ and M_i^{μ}, where i is the current time (see §2.5). The `tick` increments the solver's clock.

(Req 2). To perform backward propagation as in (4), we write:

```
stepUp = do
  choice m
```

```
  contract up
  r ← contract measurements
  tick
  when (r==signif) backwardPropag
```

It is quite similar to the previous version. Additionally, it adds the model `m` to the list of recorded choices and triggers a backward propagation when the result of the measurement contractor is significant.

```
backwardPropag = do
  cs ← choices
  backwardPropag' (concat cs)

backwardPropag' ((m,i):es) = do
  r ← contractAt m i
  when (r==signif) (backwardPropag' es)
```

`backwarPropag` just flattens the list of choices made so far and `backwardPropag'` applies the model contractor again for each choice, in the reverse chronological order. This backward propagation proceeds as long as a model contractor significantly contracts boxes as denoted by the statement `when` in the recursive definition of `backwardPropag'`. Each sublist of `choices` contains the sequence of model contractors between choice points. In order to limit the backward propagation from the current time to the previous choice point we simply can replace `concat` by `head` in `backwardPropag` in order to select the first sublist of `choices`.

We can see in Figure 3 that estimations ahead of measurements have been smoothed by the backward propagation.

2.5 Monadic Operators

Haskell provides a formal framework to define and compose effects: monads [13]. For instance, the `IO` monads encapsulates side effects (such as calling a C function) and the `Maybe` monads enables us to encapsulate failures and backtracks. In our case, we define a new monad for executable grammars with the type[2]

```
type Ibex a = State → IO (State,Maybe a)
```

It can be read as follows: an executable grammar takes a state (containing the clock, the choices and the trail) as a parameter and applies a sequence of contractors (*i.e.,* calls C contractors in Ibex, whence the `IO`). The result is a modified state (*e.g.,* the clock has been incremented) and a return value of type (`Maybe a`) which indicates if the strategy (executable grammar) was successful.

A monad defines two functions: `return` and `(»)`. In our case, the function `return` defines an empty grammar that applies no contractor. The function `(»)` defines the sequential composition of two grammars.

Our grammars provide choice points and backtrack based on trailing. Our monad provides two more functions: `mzero` and `or`. The function `mzero` defines a grammar that fails and terminates a sequence of contractors. The function `or` composes two executable grammars and introduces a choice point. When `m1 ‘or‘ m2` is executed, first `m1` is executed. If `m1` fails (*i.e.,* calls `mzero`), then the trail is used to restore the domains and `m2` is executed. If `m1` succeeds,

[2]For the sake of clarity this is a simplified type, see Section 3.1 for a full definitiion.

then `m2` is never executed. Note that we never explicitly call `mzero` but a `guard` statement does when an invariant is violated. For instance:

```
contract m = do
  i ← time
  r ← contractC m i
  guard (r/=unfeas)
```

In this definition, `contract` gets the current time and calls the C contractor (`contractC`) defined in Ibex. If fails and backtracks when the contractor returns *unfeasibility* (the box gets empty). Otherwise, the execution proceeds in sequence.

2.6 Third-Level Strategy

Our monadic operators hide implementation details about side-effects and backtrack by trailing. Our second level strategies hide implementation details about the time and backward propagation. Our approach is all about encapsulation and providing the right level of abstraction to the user. When the programmer write `up' `or` stop`, he hopes `up'` more likely succeeds than `stop`. When the programmer does not have such a piece of information, he can rely on heuristics.

(Req 3). We can design an alternative operator `orH` in answer to this requirement:

```
orH (model1,g1) (model2,g2) = do
  bm ← try measurements
  b1 ← try model1
  b2 ← try model2
  if (dist bm b1 < dist bm b2) then g1 `or` g2
                               else g2 `or` g1
dist boxMeasurement boxModel =
    abs (mid boxMeasurement - mid boxModel) +
    abs (radius boxMeasurement - radius boxModel)
```

The operator takes two models and two grammars as parameters. It applies the measurement contractor, saves the contracted box in `bm` and restores the original box. The same is done for `model1` and `model2`. The Hausdorff distances with `bm` are then compared for both results so that the most probable grammar is executed first.

This new operator enables us to find a solution that compares advantageously to the one found with the `or` operator in §2.3. With `orH`, the solution is less jumpy: it is a sequence of only 10 models (instead of 13) obtained for the cost of 2664 backtracks (instead of 5059).

3 Monadic Operators Implementation

In Section 2.5, we gave a quick overview of the monad used in our framework. We now dive into more technical details. First, in Section 3.1 we detail our monad definition. Second, in Section 3.2 we discuss the implementation of the backtrack by trailing. Third, in Section 3.3 we show how monadic expression can be rewritten dynamically to satisfy monadic laws.

3.1 Monadic Parsers Revisited

We have designed and implemented a monadic library in Haskell strongly inspired from the monadic parser libraries. However, our library does not parse sequences of commands but it enumerates them. At the same time it enumerates the models in a sequence, the sequence is tested by applying the corresponding constraints. If a constraint makes an interval empty, the sequence is declared not valid and the next sequence is enumerated and tested. This requires first to undo the side effects of the constraints of the first sequence. This backtrack is implemented by a trailing mechanisms. Indeed, a constraint has a local side effects on the C array: it modifies the variables at index i and $i+1$ only, where i is the current time step. So, it would be to inefficient to copy the full array of variables for each choice point.

The type of our monad `Ibex a` is defined as a function that takes the current `State` as a parameter and returns a pair with the updated state and possibly a result (*i.e.,* `Maybe a`):

```
newtype Ibex a =
  Ibex { runIbex :: State → IO (State,Maybe a)
}
```

Note that, even when there is no result (*i.e.,* `Nothing`) the state is returned. Indeed, the state contains the growing trail which enables backtracking. Our monad is used to call foreign C functions so its type is based on `IO`.

The monadic operators are defined as follows:

```
instance Monad Ibex where
    return a = Ibex $ λs → return (s,Just a)
    m >>= k = Ibex $ λs → do
                      res ← runIbex m s
                      case res of
                        (s1,Nothing) → return (s1,Nothing)
                        (s1,Just a) → runIbex (k a) s1
```

The operator `return` just returns the current state and a result. The bind operator `>>=` runs its first argument with the current state. (Here `runIbex` is the deconstructor of the type `Ibex a`). When there is no result, the second argument of the bind operator is ignored and the current state is returned. When there is a result, the second argument is evaluated as for the classic state monad.

Our monad is also a monad plus that provides the notions of choice point and failure with the function `mplus` (*a.k.a.* `or` in the previous sections) and `mzero`:

```
instance MonadPlus Ibex where
    mzero = Ibex $ λs → return (s,Nothing)
    mplus m1 m2 = Ibex $ λs → do
      (s1,r1) ← runIbex m1 (commit s)
      case r1 of
        Just a1 → return (s1,r1)
        Nothing → do s2 ← backtrack s1
                     (s3,r2) ← runIbex m2 (commit s2)
                      case r2 of
                        Just a2 → return (s3,r2)
                        Nothing → do s4 ← backtrack s3
                                     return (s4,r2)
```

Its `mzero` operator simply returns the current state and no result. Its `mplus` operator first performs the computation corresponding to its first argument. If there is a result, it is returned and the second computation is never evaluated. If there is no result, the second branch is evaluated. The auxiliary function `commit` is called before each computation to record a new trail in the state and the function `backtrack` is called each time a computation has no result to replay the trail backward (*i.e.,* undo side effects in Ibex and remove the corresponding trail from the state). The definitions of the functions `commit` and `backtrack` depend on the representation of the `State`. We postpone them to the next section.

Our monad provides access to the low-level C functions with the following lifting operator:

```
liftIO :: IO a → Ibex a
liftIO m = Ibex $ λs → do
             a ← m
             return (s,Just a)
```

For instance, we define

```
contract :: Model → Time → Ibex Return
contract m i = liftIO $ c_contract m i
```

3.2 Backtrack by Trailing

The state of our monad is defined as follows:

```
data State = State { clock   :: Time
                   , back    :: Int
                   , trailS  :: TrailS
                   , choiceS :: ChoiceS }
```

It encapsulates a `clock` (*i.e.,* an integer corresponding to the current index in the C array), a counter of backtracks for profiling, a trail that registers the actions to be undone in order to restore a previous state, and the list of past chosen models (*i.e.,* which constraint has been applied at what time).

Both the trail and the chosen models are structured as list of list: each sublist represents the actions, or the models, between two choice points.

```
type TrailS = [[Action]]
type ChoiceS = [[(Model,Time)]]
```

There are two kinds of action corresponding to modifications of the clock or modification of the C array by a constraint.

```
data Action = Tick | Undo Event
```

When the clock is incremented by `tick` a corresponding action is added to the trail:

```
tick :: Ibex ()
tick = do
  t ← time
  modify $ λs → s { trailS=addAction Tick (trailS s)
                  , clock=clock s+1 }
```

When a constraint is to be applied, the column of the C arrays it can modify is saved by `trailAt`

```
trailAt :: Time → Ibex ()
trailAt i = do
  e ← liftIO $ saveEvent i
  modify $ λs → s { trailS=addAction (Undo e) (trailS s) }
```

This function relies on `saveEvent` that reads a column of the C array and returns it as a Haskell tuple. Then this tuple is added to the first sublist of `trailS`.

```
data Event = Event {
      t::Time
    , d0x_lb::Value , d0x_ub::Value
    , d1x_lb::Value , d1x_ub::Value
    , d2x_lb::Value , d2x_ub::Value
    }
```

Such a tuple is created and added to the trail by calling the function `choice` each time a constraint is chosen.

```
choice :: Model → Ibex ()
choice m = do
  i ← time
  trailAt i
  modify $ λs → s { choiceS=addChoice (m,i) (choiceS s) }
```

The model is also added to the first sublist of `ChoiceS`.

Finally, our backtrack mechanism is implemented by the two functions `commit` and `backtrack` called by the function `mplus`. The function `commit` introduces a new choice point by simply adding a new empty sublist to the trail and the choices. The function `backtrack` uses the most recent sublist of the trail to undo the most recent actions and restore a state corresponding to the last choice points. This is performed by the function `rollback` that decrements the clock for `Tick` and restores the state of Ibex variables for `Undo`.

```
commit :: State → State
commit s = s {trailS=[]:trailS s,choiceS=[]:choiceS s}

backtrack :: State → IO State
backtrack s = do
    let as:ts=trailS s
    foldM rollback (s{trailS=ts,choiceS=tail (choiceS s)
                   ,back=1+back s}) as

rollback :: State → Action → IO State
rollback s Tick = return $ s {clock=clock s-1}
rollback s (Undo u) = do
  restoreEvent u
  return s
```

3.3 Left Distribution in Monadic Expressions

Monads come with laws. In particular, a monad plus must satisfy the left distribution law:

```
(m1 `mplus` m2) >> m3 = (m1 >> m3) `mplus` (m2 >> m3)
```

Unfortunately our monad `Ibex` does not satisfy this law. Indeed, if `m1` succeeds then its result is returned and the second branch `m2` is discarded. So, this second branch `m2` is lost when `m3` fails (the correct behavior would be to backtrack and try `m2 >> m3` as specified by the right hand side of the law).

Our examples in this paper do not exhibit this problem because their definitions match the right-hand side of the law only. However, it could be convenient to use the left-hand side pattern too. For instance, the sequences of a given model can be defined as

```
up'' = stepUp >> (up'' `mplus` empty)

stop'' = stepStop >> (stop'' `mplus` empty)
```

where `empty` that applies no model and succeeds is simply defined as `return unsign`. In this case, an expression such as `up''>>stop''` is structured as the left-hand side of the law.

A work around consists in defining a data type `IbexS` to represent monadic expression.

```
{-# LANGUAGE ExistentialQuantification #-}

data IbexS a = R a | forall b. B (IbexS b) (b → IbexS a) |
               Z | P (IbexS a) (IbexS a) | A (Ibex a)
```

In this definition `R` stands for return, `B` for bind, `Z` for zero, `P` for plus and `A` for atomic expression of type `Ibex` with no inner choice (*e.g.,* call a C function). The type of the bind operator `>>=` requires existential quantification to define its corresponding constructor `B`.

There is no need of parser to construct a term of this type but we simply define a monad plus and its operators so that they return a term.

```
instance Monad IbexS where
    return a = R a
    m >>= k = m `B` k

instance MonadPlus IbexS where
    mzero = Z
    mplus m1 m2 = m1 `P` m2
```

Finally, we define an interpreter `run` that restructures the terms and eventually calls the actual operators of `Ibex`.

```
run :: IbexS a → Ibex a
run ((m1 `B` k1) `B` k) = run (m1 `B` (λb → ((k1 b) `B` k)))
run ((m1 `P` m2) `B` k) = run ((m1 `B` k) `P` (m2 `B` k))
run (m1 `P` m2)         = (run m1) `mplus` (run m2)
run (m `B` k)           = (run m) >>= (λb → run (k b))
run (R a)               = return a
run Z                   = mzero
run (A e)               = e

eval :: IbexS a → IO a
eval m = evalIbex (run m)
```

Where `evalIbex` is the usual function that evaluates a monadic expression by passing it the initial `State`.

4 Conclusion

In this article, we have proposed a framework to control contractors for the state estimation of hybrid dynamical systems. It promotes the design of tailor-made constraint solvers. It is composed of three layers that offer different levels of abstraction. The top-level layer enables the user to declaratively yet operationally specify research strategies as ambiguous grammar. The second layer provides abstractions for controlling the propagation heuristic. Finally, the third layer implements choice points and backtrack with a trail. Our framework is implemented as a monadic library in Haskell on top of the Ibex contractor library in C. It benefits from the expressiveness and safety of Haskell and from the efficiency of C. We have introduced and illustrated our framework with an elevator running example, but it can be used for any hybrid dynamical system as long as a list of models is handled (instead of just three).

Opportunities for future work are manyfold. Here are two of them. First, our backtrack mechanism relies on trailing. However, the longer the system lives, the longer the trail gets. For a scale up, the third layer of our framework could be modified to integrate a sliding time window that keeps only a limited trail of the past. Second, the approach could be extended to estimation problems in mobile robotics where strategies are more complicated: the unknowns include the environment (in addition to the state of the robot) and new kinds of contractors (relating the state of the robot and that of detected objects) are involved.

References

[1] E. Altman and V. Gaitsgory. Asymptotic Optimization of a Nonlinear Hybrid System Governed by a Markov Decision Process. *SIAM Journal on Control and Optimization*, 35(6):2070–2085, 1997.

[2] A. Balluchi, A. Casagrande, P. Collins, A. Ferrari, T. Villa, and A.L. Sangiovanni-Vincentelli. Ariadne: a Framework for Reachability Analysis of Hybrid Automata. In *Proceedings of the International Syposium on Mathematical Theory of Networks and Systems*, 2006.

[3] G. Chabert. *IBEX*. http://www.ibex-lib.org, 2009.

[4] G. Chabert and L. Jaulin. Contractor Programming. *Artificial Intelligence*, 173(11):1079–1100, 2009.

[5] G. Chabert, L. Jaulin, and X. Lorca. A Constraint on the Number of Distinct Vectors with Application to Localization. In *15th International Conference on Principles and Practice of Constraint Programming (CP'09)*, volume 5732 of *Lecture Notes in Computer Science*, pages 196–210. Springer-Verlag, 2009.

[6] R.L. Grossman, A. Nerode, and A. P. Ravn. *Hybrid Systems*, volume 736 of *LNCS*. Springer-Verlag, 1993.

[7] J. Guéguen, H. Zaytoon. On the Formal Verification of Hybrid Systems. *Control Engineering Practice*, 12(10):1253–1267, 2004.

[8] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Jour. of Functional Programming*, 8(4):437–444, 1998.

[9] L. Jaulin. Interval Constraint Propagation with Application to Bounded-Error Estimation. *Automatica*, 36(10):1547–1552, 2000.

[10] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[11] N.S. Nedialkov, K.R. Jackson, and G.F. Corliss. Validated Solutions of Initial Value Problems for Ordinary ifferential Equations. *Applied Mathematics and Computation*, 105(1):21–68, 1999.

[12] A. Neumaier. *Complete Search in Continuous Global Optimization and Constraint Satisfaction*, pages 1–99. Cambridge University Press, 2004.

[13] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *POPL '93*, pages 71–84. ACM, 1993.

[14] N. Ramdani, N. Meslem, and Y. Candau. A Hybrid Bounding Method for Computing an Over-Approximation for the Reachable Space of Uncertain Nonlinear Systems. *IEEE Tran. Automatic Control*, 54(10):2352–2364, 2009.

[15] N. Sahinidis. *BARON, Branch-And-Reduce Optimization Navigator*. http://www. andrew.cmu.edu/user/ns1b/baron/baron.html.

[16] Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic Constraint Programming. *Journal of Functional Programming*, 19(6):663–697, 2009.

[17] Pieter Wuille and Tom Schrijvers. Monadic Constraint Programming with Gecode. In *8th Int. Workshop on Constraint Modelling and Reformulation*, pages 171–185, 2009.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399