CGMurphi: Automatic Synthesis of Numerical Controllers for Nonlinear Hybrid Systems [*]

Giuseppe Della Penna
Department of Computer Science, University of L'Aquila, Italy[†]

Benedetto Intrigila
Department of Mathematics, University of Roma "Tor Vergata", Italy

Daniele Magazzeni
Department of Informatics, King's College London, UK

Igor Melatti and Enrico Tronci
Department of Computer Science, University of Roma "La Sapienza", Italy

May 6, 2013

Abstract

In the last years, the use of controllers has become very common, thus much work is being done to create automatic controller synthesis tools. When dealing with critical systems, most of the times such controllers are required to be *optimal* and *robust*, i.e., they must achieve their goal with minimal resource consumption and be able to handle also unexpected situations. All these requirements, which are intrinsically difficult to satisfy, become even more challenging when dealing with *hybrid systems*, which represent a wide range of real world systems.

In this paper we propose a model checking based tool, namely CGMurphi, which assists in the the generation of optimal and robust numerical controllers for systems having complex dynamics, possibly hybrid systems. The tool provides a complete controller generation solution, being also able to effectively compress the controllers and encode them so that they can be directly embedded in software/hardware systems.

The tool has been widely experimented with very promising results. In particular, the present paper reports the complete experimentation results relative to two academic case studies, and the preliminary achievements obtained by applying CGMurphi to an industrial critical system.

Keywords: Numerical controllers, Hybrid Systems, Model Checking

[*] A preliminary version of this paper appears in [16, 17].
[†] Corresponding author. Address: Department of Computer Science, University of L'Aquila, Via Vetoio, Coppito, 67100 L'Aquila, Italy. Tel.: +390862433740. Email: Giuseppe.DellaPenna@univaq.it

1 Introduction

A control system (or, shortly, *controller*) is a hardware/software component that controls the behaviour of a larger system, called *plant*. In a closed loop configuration, the controller reads the plant state (looking at its *state variables*) and adjusts its *control variables* in order to keep it in a particular state, called *setpoint*, which represents its *normal* or *correct* behaviour.

In the last years, the use of sophisticated controllers has become very common in robotics, critical systems and, in general, in the hardware/software *embedded systems* contained in a growing number of everyday products and appliances. In particular, much work is being done to provide methodologies for the automatic (or semi–automatic) synthesis of controllers directly from the plant specifications (see Section 1.3).

Numerical controllers are tables indexed by the plant states, whose entries are commands for the plant. The commands are used to set the control variables in order to reach the setpoint from the corresponding states.

Beyond the creation of the correct controllers, there are several other important tasks that have to be considered. First, most of the times the controller has to be *optimal*, i.e., from any state it has to select the *best* control path to the setpoint, with respect to a cost function that is often the time-to-setpoint (*time optimality*). Such optimal controller generation task is obviously harder than the previous one, since optimisation commonly requires more complex calculations and larger working data storage.

Moreover, a major problem of numerical controllers is their *robustness*, i.e., the state read from the plant may not be in the controller table, although it may be *close* to some states in the table. This commonly happens when the controller drives a physical system that is subject to approximation and disturbances.

Finally, since controllers have often to be embedded in small devices, their size can be an issue. Indeed, the size of a numerical controller can be huge, especially if it has to be both optimal and robust. Therefore, suitable compression techniques should be exploited, but these may impact the controller access time, that is another important issue.

All the issues above become even more challenging when dealing with *hybrid systems*, i.e., systems described by both continuous and discrete components. This kind of system is actually very common: for example systems with relays or switches, motion controllers, constrained robotic systems, flight control systems, analog/digital circuit design, biological applications, etc. Unfortunately, hybrid systems often present a very complex dynamics, thus the (optimal) controller generation is harder to achieve, and the robustness cannot be obtained using simple interpolation techniques, as happens for continuous systems (e.g., see [30]), thus more complex approaches must be applied (e.g. see [38, 28]).

1.1 Motivations

Unfortunately, there are wide classes of real-world systems that are hardly tractable with the current controller generation techniques. In particular,

- when the system dynamics is very complex and does not satisfy the Lipschitz condition ([1, 14]), like it is common for hybrid systems (i.e., systems described by both continuous and discrete variables), the control problem cannot be solved using analytical methods like [30, 18].
- if the system dynamics is nonlinear, the control problem cannot be solved by a dynamic programming approach ([10, 6]), since it requires a backward decomposition of the cost function.
- finally, local heuristics (e.g., fuzzy rules) perform poorly in in nonlinear or hybrid systems, since the presence of discontinuities may make a good *local* often action not suitable for the final goal.

Therefore, for nonlinear and/or hybrid systems, even the synthesis of a good numerical controller (not to say an optimal or a robust one) seems to be computationally difficult. To the best of our knowledge, there is no automatic tool available that is able to generate optimal and robust controllers for such systems. Moreover, all the current controller generation techniques rarely address the issue of controller size and access time, leaving this problem to the software and hardware manufacturers.

1.2 Contribution

In this paper we propose an automatic model checking based methodology for the generation of optimal and robust numerical controllers for systems having complex dynamics, possibly nonlinear and hybrid systems. In particular, we focus on plants whose state is fully observable, i.e., all the variables which define the plant state can be read by the controller at any time. However, this is not a big limitation, since it allows us to model realistic systems such as the ones presented in the case studies (Section 7).

Symbolic (i.e., OBDD based) model checking techniques have been already successfully applied to generate controllers for a wide class of systems. However, they do not perform well when applied to hybrid systems, which usually have a very large state space. Thus, our idea is to use *explicit* model checking techniques to perform a *reachability analysis*, which allows to compute the exact reachable region of the system state space, i.e., all the states reachable from the initial ones through some action sequence. This region has usually a complex structure, thus is not trivial to define without automatic support. However, since the reachable region is often composed only of a small fraction of the possible states specified by the system state variables, it allows to build a representation of the system dynamics which focuses only on the actual behaviour of the system, which can be easily analysed to generate the controller.

Moreover, we also perform a *probabilistic* analysis of the plant state space in order to effectively apply a *strengthening algorithm* which extends the initial controller making it robust, that is able to cope with disturbances.

Finally, as a further contribution, we present an effective controller compression technique, namely an OBDD-based controller encoding, which allows to reduce the size of a numerical controller table up to 10% of its original size, while preserving small access times.

1.3 Related Works

A survey of methods for synthesising controllers is out of scope for this paper, we refer the interested reader to the following books: [4] for PID-based techniques, [32] for what concerns the synthesis of robust controllers and [22] for mathematical methods and in particular Lyapunov-based methods. Finally, the control handbook [31].

Furthermore, here we want to mention some approaches similar to ours.

First, one of the most versatile and widely used technique is *dynamic programming*, which is very suitable for the generation of optimal controllers [6, 30]. In [18], we presented a comparison between model checking and dynamic programming techniques for the synthesis of optimal controllers. However, differently from our approach that provides an automatic methodology, the dynamic programming often requires the definition of design functions which have to be chosen case by case. Moreover, it requires the inversion of the dynamical behaviour of the system which can be hard to compute.

Cell mapping, originated by Hsu[26, 24] as a computational technique for analysing the global behaviour of nonlinear systems, has been used to generate control tables (see e.g. [25]). However, since cell mapping requires a global analysis of the system, when a high precision is required, it is hard to apply.

A widely used approach is the one based on *symbolic* model checking, (see, e.g. [46], the Pnueli's works [3, 2] or the UPPAAL-TIGA tool [5, 47]), which however differs from ours since we use an explicit approach.

Furthermore, the problem of synthesis of controllers can be viewed as a two-players game between the controller and its environment. For this approach, see, e.g. [39, 34, 44], where, however, authors consider timed automata [29].

Finally, there is a class of "on the fly" algorithms (see, e.g. [45] or the CIRCA project [35, 21])

However, to the best of our knowledge, this is the first time that explicit model checking techniques are successfully applied for the automatic synthesis of controllers for hybrid systems.

1.4 Summary

The rest of the paper is organised as follows. In Section 2 we provide some background notions on hybrid systems and define the corresponding control problem. Then, in Section 3 we present out controller generation tool, namely CGMurphi, whose algorithms are detailed in Sections 4,5 and 6. A number of validating case studies are described in Section 7. Finally, Section 8 concludes the paper.

2 Basic Definitions and Statement of the Problem

In this Section we give some formal definitions required to understand the controller generation algorithm. The interested reader can refer, e.g., to [33, 43] for further information on the theoretical arguments introduced in this section.

2.1 Discrete Time Hybrid Systems

Roughly speaking, hybrid systems are ensembles of interacting discrete and continuous systems. The discrete system operates on a discrete state and performs discontinuous state changes at discrete time points, while the continuous system operates on a continuous state which evolves continuously.

More formally, we have the following definition.

Definition 1 *A* Discrete Time Hybrid System *(DTHS) is a tuple $\mathcal{H}$ = (X,Q, U, W, I, f, p) where:*

- X = $\times_{i=1}^{n}[a_i, b_i]$*, with* $[a_i, b_i]$ *a bounded interval of the reals* $\mathbb{R}$*.*
- Q = $\times_{i=1}^{k}[c_i, d_i]$*, with* $[c_i, d_i]$ *a finite subset of the integers* $\mathbb{Z}$*.*
- U = $\times_{i=1}^{m}[\alpha_i, \beta_i]$*, with* $[\alpha_i, \beta_i]$ *a bounded interval of the reals* $\mathbb{R}$*.*
- W = $\times_{i=1}^{r}[\gamma_i, \mu_i]$*, with* $[\gamma_i, \mu_i]$ *a finite subset of the integers* $\mathbb{Z}$*.*
- I *is a subset of* $X \times Q$*.*
- f *is a function from* $X \times Q \times U \times W$ *to* X *s.t. for each* $q \in Q$*,* $w \in W$*,* $\lambda xu\ [f(x, q, u, w)]$ *is a continuous function of* (x, u) *(where* λ *is the abstraction operator).*
- p *is a function from* $X \times Q \times U \times W$ *to* Q*.*

The state space of $\mathcal{H}$ is S = $X \times Q$. A *state* for $\mathcal{H}$ is a pair s = (x, q) in S, where $x \in X$ and $q \in Q$.

A *run* for the DTHS $\mathcal{H}$ is a (possibly infinite) sequence of states and actions $(x(0), q(0), u(0), w(0)), \ldots (x(t), q(t), u(t), w(t)), \ldots$ s.t. for all t we have:

- $(x(0), q(0)) \in I$
- $x(t + 1) = f(x(t), q(t), u(t), w(t))$
- $q(t + 1) = p(x(t), q(t), u(t), w(t))$

If π = $(x(0), q(0), u(0), w(0))$, $(x(1), q(1), u(1), w(1)), \ldots$ is a run of $\mathcal{H}$ we denote with $\pi(t)$ the t-th state element of π. That is $\pi(t)$ = $(x(t), q(t))$. Furthermore we write $\varphi(x(t), q(t), u(t), w(t))$ for ($f(x(t), q(t), u(t))$, $p(x(t), q(t), w(t))$).

A state $s \in X \times Q$ is said to be *reachable* iff there exist a path π and an integer t s.t. $s = \pi(t)$.

To convey to reader the motivations behind our formalism, we make the following observations.

First we observe that $x \in X$ is the vector of the *continuous components of the state*, $q \in Q$ is the vector of the *discrete components of the state*, $u \in U$ is the vector of the *continuous components of the control actions*, and $w \in W$ is the vector of the *discrete components of the control actions*. I is the set of *initial states*.

Moreover, to each discrete state q_i is assigned, via the function f, a region X_i in the *continuous state space* X and a dynamics which acts on the region X_i when the discrete state is q_i. Roughly speaking, to every discrete state corresponds a *mode* of the system.

2.2 Control Problem for DTHS

In order to model our control problem, we extend the definition of DTHS assuming that a *setpoint* $G \subseteq X \times Q \neq \emptyset$ has been specified. In this context, setpoint refers to the normal condition the controller should bring (or maintain) the plant to. We call *goal states* (or *goals*) the states in G.

Moreover, we can tolerate a given *approximation* in reaching the goal, more precisely given $\epsilon > 0$, we say that a state (x,q) is an *ϵ-approximation of the goal* iff for some goal state (x_g, q_g), $|x - x_g| < \epsilon$ and $q = q_g$.

Now we are in position to state the *control problem* for a given hybrid system $\mathcal{H}$ with respect to a setpoint G and an ϵ-approximation of the goal.

Definition 2 *A* Control Problem *(CP) is a triple ($\mathcal{H}$, G, ϵ) where: $\mathcal{H}$ = (X, Q, U, W, I, f, p) is a DTHS, G is a setpoint and ϵ is the tolerated goal approximation. A solution to a CP is a map $\mathcal{K}$ from $X \times Q$ to $U \times W$ s.t. for all (x, q) $\in I$ there exist $k \in \mathbb{N}$ and a run π of $\mathcal{H}$ s.t.: for all $t < k$, $\pi(t+1) = \varphi(\pi(t), \mathcal{K}(\pi(t)))$, and $\pi(k)$ is an ϵ-approximation of a goal state in G.*

Note that the definition 2 handles the problem of driving a system to the setpoint G, while it does not explicitly address the problem of keeping the system in G (*stabilisation*), which is very common in the control theory. However, stabilisation can be addressed as well by extending the set of initial states I with all the states in the neighbourhood of G (i.e., all the possible ϵ-approximations of G). Indeed, in this case the problem solution, devised as described above, will also contain the information needed to bring back (and keep) the plant to its setpoint when it moves within the region defined by ϵ around the setpoint itself.

Since we have no restrictions on the dynamics of the system, the problem of determining if a state is controllable to the setpoint has no algorithmic solution, even for very simple (non linear) dynamics [42]. Therefore, we must consider suitable restrictions, which however should not compromise the usefulness of the approach.

First of all, we assume *effectiveness* of all functions. That is, we assume that (the characteristic function of the set of) initial states, the DTHS transition relation (i.e. f and p in Definition 1) as well as any other function can be computed with any degree of accuracy. Note that, while we need this requirement (which corresponds, e.g., to type-2 computability in [48]) to solve the DTHS control problem in general, it seems likely that in every specific instance of the problem only a finite precision should be required. Further requirements on the computability of the transition functions are pointed out below.

Therefore we only consider *effective* DTHS, i.e. DTHS that satisfy the above effectiveness condition.

Second, we assume a *finite temporal horizon*. That is we require that the setpoint is reached within a given *maximum number of control actions*. Note that in most practical applications we always have a maximum time allowed to complete the execution of system run. Thus this restriction, although theoretically quite relevant, has a limited practical impact. Note that with this restrictions, *piecewise linear dynamics* becomes decidable [41, 40].

We consider the following control problem.

Definition 3 *A* Finite Horizon Control Problem *(FHCP) is a quadruple ($\mathcal{H}$, G, ϵ, T) where:*

- *$\mathcal{H}$ = (X, Q, U, W, I, f, p) is an effective DTHS,*
- *$G \subseteq X \times G$ is a set of goal states,*
- *$\epsilon > 0$ is the tolerated goal approximation,*
- *$T \in \mathbb{N}$ is a temporal horizon.*

A solution *to a FHCP is a map $\mathcal{K}$ from $X \times Q$ to $U \times W$ s.t. for all (x, q) $\in I$ there exist $k \leq T$ and a run π of $\mathcal{H}$ s.t.: for all $t < k$, $\pi(t+1) = \varphi(\pi(t), \mathcal{K}(\pi(t)))$, and $\pi(k)$ is an ϵ-approximation of a goal state in G.*

In the following, we will write $\mathcal{K}_\rho(s)$ to mean a path π starting at s (i.e. $\pi(0) = s$) and s.t. there exists $k \leq T$ s.t. for all $t < k$, $\pi(t+1) = \varphi(\pi(t), \mathcal{K}(\pi(t)))$, and $\pi(k)$ is an ϵ-approximation of a goal state in G. Moreover, we write $|\mathcal{K}_\rho(s)|$ to mean the k above, i.e. the number of steps required to drive s to (an ϵ-approximation of) a goal state.

A more general control problem is obtained when a *cost function* is defined on the transitions of the system.

Definition 4 *An* Optimal Finite Horizon Control Problem *(OFHCP) is a 5-tuple ($\mathcal{H}$, G, ϵ, T, C) where:*

- *$\mathcal{H}$ = (X, Q, U, W, I, f, p), G, ϵ and T are the same as in Def. 3*
- *$C : X \times Q \times U \times W \to \mathbb{R}^+$ is a* cost function, *s.t. for each $q \in Q$, $w \in W$, $\lambda xu\, [C(x, q, u, w)]$ is a continuous function of (x, u).*

A solution *to an OFHCP is a solution $\mathcal{K}$ for the FHCP $\mathcal{P}$ = ($\mathcal{H}$, G, ϵ, T, C) s.t., for all other solutions $\mathcal{K}'$ for $\mathcal{P}$, the following holds. For all $s \in I$, consider $\pi' = \mathcal{K}'_\rho(s)$ and $\pi = \mathcal{K}_\rho(s)$, then $\sum_{t=0}^{|\mathcal{K}_\rho(s)|-1} C(\pi(t), \pi(t+1)) \leq \sum_{t=0}^{|\mathcal{K}'_\rho(s)|-1} C(\pi'(t), \pi'(t+1))$.*

2.3 Discretisation of DTHS

In order to address the control problem using model checking techniques, we need to extract from the DTHS a Finite State System (FSS) through a suitable discretisation of the continuous variables.

Definition 5 *Let $\mathcal{H}$ = (X,Q, U, W, I, f, p) be a DTHS and $d \in \mathbb{R}^+$ be a* discretisation step. *Then a* discretisation *$D = (D_s, D_c)$ with step d of $\mathcal{H}$ is a pair of functions $D_s : X \to Z^n \cap X$ and $D_c : U \to Z^m \cap U$ where $Z = \{dz | z \in \mathbb{Z}\}$. Function D_s is defined as $D_s(x) = y$ s.t. $y \in Z^n \cap X$ and $|x - y|$ is minimal. Function D_c is defined as $D_c(u) = y$ s.t. $y \in Z^m \cap U$ and $|u - y|$ is minimal.*

Moreover, if δ is the smallest real value such that for all x, $|x - D_s(x)| \leq \delta$, we call δ the radius *of the discretisation D_s.*

Observe that, being X bounded, the discretised states in X are finite (by an abuse of language we denote also by D_s this set). The same holds for the values of the discretised control actions (correspondingly, we denote by D_c this set). We use also the notation $\hat{x}$ ($\hat{u}$) to denote elements of D_s (resp. D_c).

Given a discretisation $D = (D_s, D_c)$ and an approximation ϵ, the *discretised ϵ-approximations of the goal state* (shortly, *ϵ-goals*) are all the ϵ-approximations of the goal states, whose continuous components are in D_s. We shall always require that the discretisation has been chosen in such a way that the set of discretised ϵ-approximations of the goal state is *not empty*, i.e. the discretisation radius is less than or equal to ϵ.

The discretisation step, the temporal horizon and the precision of the approximation of the goal states are three *design parameters* that the designer can choose to get the solution to the OFHCP which is best suited w.r.t. the constraints to be fulfilled.

Now we are in position to associate to our hybrid system $\mathcal{H}$ = (X, Q, U, W, I, f, p) a suitable *finite state system* as follows:

Definition 6 *Given a discretisation $D = (D_s, D_c)$ and a DTHS $\mathcal{H}$ = (X,Q, U, W, I, f, p), the* finite state system *(FSS in the following) $\mathcal{F}_\mathcal{H} \equiv (S, A, F_T, I_S)$ associated with $\mathcal{H}$ is defined as follows (with $\hat{x} = D_s(x), \hat{u} = D_c(u)$):*

1. *the set S of states of $\mathcal{F}_\mathcal{H}$ is defined as the set of all $(\hat{x}, q)$ with $\hat{x} \in D_s$ and $q \in Q$;*
2. *the set A of actions of $\mathcal{F}_\mathcal{H}$ is defined as the set of all $(\hat{u}, w)$ with $\hat{u} \in D_c$ and $w \in W$;*
3. *the transition function $F_T : S \times A \longrightarrow S$ is defined as follows: $F_T(\hat{x}, q, \hat{u}, w) = (\hat{x}', q')$, where:*
   - *$x' = f(\hat{x}, q, \hat{u}, w)$,*
   - *$q' = p(\hat{x}, q, \hat{u}, w)$;*
4. *the set I_S of the initial states of $\mathcal{F}_\mathcal{H}$ is defined as $I_S = \{(\hat{x}, q) | (x, q) \in I\}$.*

We can now reformulate the control problem above as a problem relative to the finite state system $\mathcal{F}_\mathcal{H}$:

Definition 7 *An* Optimal Finite Horizon Finite State Control Problem *(OFHFSCP) is a 5-tuple ($\mathcal{F}_\mathcal{H}$, G, ϵ, T, C) where:*

- *$\mathcal{F}_\mathcal{H} = (S, A, F_T, I_S)$ is the FSS associated with the DTHS $\mathcal{H}$ as defined in Def. 6*

- *ϵ and T are the same as in Def. 3*
- *$G \subseteq S$ is a set of goal states*
- *$C : S \times A \to \mathbb{R}^+$ is a* cost function

A solution *to an OFHFSCP is the minimum-cost one (w.r.t. the cost function C, as shown in Def. 4) among all the possible maps $\mathcal{K}$ from S to A s.t. for all $(\hat{x}, q) \in I_S$ there exist $k \leq T$ and a run π of $\mathcal{F}_{\mathcal{H}}$ s.t.: for all $t < k$, $\pi(t+1) = F_T(\pi(t), \mathcal{K}(\pi(t)))$, and $\pi(k)$ is an ϵ-approximation of a goal state in G.*

In the following sections, we present a general methodology to solve this problem. As we will see, our algorithm brings into (an epsilon approximation of) a goal state not only each initial state, but also all the states which are backward reachable from the goal in at most T steps.

3 The CGMurphi Tool

The CGMurphi tool [13] is an extended version of the CMurphi model checker [12]. It is based on an explicit enumeration of the state space, originally developed to verify protocol-like systems.

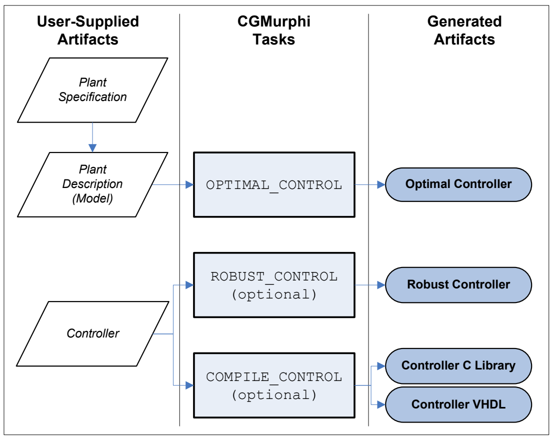


Figure 1: CGMurphi usage pattern.

We added to CMurphi a set of new functionalities that take advantage from the verifier's base algorithms and data structures, extending and enhancing them in order to create a complete *press-button* controller generation framework.

In particular, an overall view on the CGMurphi functions and produced artifacts is shown in Figure 1. As the reader can see, the only input required by CGMurphi is a plant description (including the setpoint) derived from its specifications. Then, the tool is able to automatically

- Synthesise an optimal numerical controller for the given plant (the `OPTIMAL_CONTROL` task of Figure 1). The output of this task is a binary encoded state-action table.
- Optionally, strengthen the numerical controller by refining the control in selected areas of the plant state space (the `ROBUST_CONTROL` task of Figure 1). The output of this task is new, possibly larger, state-action table. The state space areas that are subject to the strengthening are automatically chosen by a probabilistic analysis process based on a set of user defined parameters.
- Optionally, compile the controller table into a compact OBDD-based representation and then outputs executable code that implements this representation, for an easy and quick embedding of the controller in any hardware or software device (the `COMPILE_CONTROL` task of Figure 1).

The process is entirely automatic, and required no user further assistance after the model has been created and fed to the tool together with the controller generation settings and preferences.

In the following, we first describe the CGMurphi input language. Then, we give details about the algorithms used to implement all of the three tasks listed above, namely, `OPTIMAL_CONTROL`, `ROBUST_CONTROL` and `COMPILE_CONTROL`. A final worked example will then show, step by step, the tool usage from the end-user point of view.

3.1 CGMurphi Input Language

The CGMurphi input consists of the definition of a FSS $\mathcal{F}_{\mathcal{P}}$, representing the plant $\mathcal{P}$ to be controlled, and including the definition of the set of states in the setpoint, i.e. the states that the controller should bring (or maintain) the plant to. Definitions are stored in a file that we call *CGMurphi model*.

The plant model is described in CGMurphi using the *CGMurphi modelling language*, which is basically the same as is a high-level programming language for finite-state asynchronous concurrent systems, which includes many features found in common high-level programming languages such as Pascal or C, such as has user-defined data types, procedures and parametrisation of descriptions.

A CGMurphi model consists of

- a set of declarations of constants, types, global variables and procedures,
- a collection of transitions rules,
- a description of the initial states,
- a set of properties.

The behavioural part of the model is a collection of transition rules. Each transition rule is a guarded command consisting of a condition (i.e., a boolean expression on global variables) and an action (i.e., a statement that can modify the global variables values).

Moreover, the CMurphi modelling language offers two important functionalities that are essential to cope with complex and hybrid systems:

- **Finite Precision Real Numbers**. The type `real(m, n)` can be used to represent real numbers with m digits for the mantissa and $\overline{n} = log_{10}\lfloor n \rfloor + 1$ digits for the exponent. The type `real(m, n)` is actually finite, with a cardinality of $2 \times 9 \times 10^{m-1} \times 2 \times 10^{\overline{n}} = 36 \times 10^{m+\overline{n}-1}$. Thus this extension has no impact on synthesis algorithms, but makes it easier to model hybrid systems within CGMurphi.

- **External C/C++ Functions**. It is possible to call externally defined C/C++ functions in the modelling language. Therefore we can use the C/C++ code to model the plant dynamics: this makes possible, for instance, to directly include (with some arrangement) in the CGMurphi model a simulator for the plant under analysis, since plant simulators are often available and almost always written in C/C++. In this way, creating the CGMurphi model for a plant becomes a really simple process.

Finally, in order to define the setpoint, in CGMurphi we extended the modelling language above with the `setpoint` construct. Such construct has the following syntax:

$$\langle\text{setpoint}\rangle ::= \texttt{setpoint}\ [\ \langle\text{string}\rangle\ ]\ \langle\text{expr}\rangle$$

where <expr> is a boolean expression that is true in a state s if and only if s satisfies the setpoint property, optionally named with a <string>.

Section 7 contains more examples that show how the CGMurphi input language can be used to model real systems.

4 Optimal Controller Generation

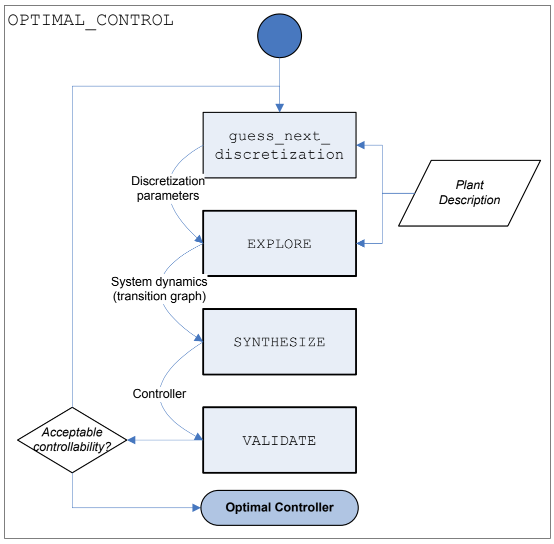
In this Section we present a model checking based algorithm used by CGMurphi for the synthesis of optimal numerical controllers.

In particular, we will take advantage from the *reachability analysis* algorithms used by explicit model checkers, and adapt them in order to efficiently compute the controller table. Indeed, model checking starts by computing the exact *reachable region* of the system by means of the so called reachability analysis. Such region consists of all states reachable from the *initial states* by some action sequence, and it has usually a complex structure, being composed only by a small fraction of the possible states specified by the state variables.

Reachability analysis can be performed in a *backward* or in *forward* way, starting from the goals or from the initial states, respectively. We adopt a *breadth-first* forward search algorithm, since this approach is viable also when the dynamics of the system is difficult to invert, as often happens with hybrid systems.

The overall optimal controller generation algorithm is represented by the `OPTIMAL_CONTROL` procedure depicted in Figure 2.

In particular, an *iterative approach* is used to find the best suited discretisation of the given DTHS $\mathcal{H}$: we start with a tentative discretisation chosen as a very coarse quantisation of the variable domain. Then, function



OPTIMAL_CONTROL
guess_next_ discretization
Discretization parameters
Plant Description
EXPLORE
System dynamics (transition graph)
SYNTHESIZE
Controller
Acceptable controllability?
VALIDATE
Optimal Controller

Figure 2: Structure of the OPTIMAL_CONTROL Procedure.

`guess_next_discretisation` is iteratively used to generate finer discretisation for the continuous variables by increasingly refining their quantisation.

The devised discretisation D and the corresponding OFHCP definition $\mathcal{F}$ are passed to the `EXPLORE` procedure, which builds the dynamics of the system, and then the `SYNTHESISE` procedure is called to create the controller `CTRL`.

Both procedures work on the DTHS definition using the full machine precision for the calculations involving continuous values: the discretisation is only applied to the reachable states. Moreover, the presented algorithm works directly on an implicit description of the given DTHS, i.e., it does not require a complete *state space generation*, but generates on-the-fly only the reachable states, saving both memory and time.

Finally, the `VALIDATE` procedure is used to check if `CTRL` guarantees a suitable controllability. This is accomplished by checking that the discretised trajectories are a satisfactory approximation of the real ones. A similar iterative approach is advocated in the well-known *cell mapping* approach [38, 37]. If the validation fails, the whole process is repeated with a finer discretisation. re obtained with finer discretisations.

In the following sections we describe the `EXPLORE`, `SYNTHESISE` and `VALIDATE` procedures in more detail.

4.1 The `EXPLORE` Procedure

The `EXPLORE` procedure in Fig. 3 is directly derived from a forward reachability analysis algorithm. It visits the DTHS $\mathcal{H}$ up to T steps, building a representation of the system dynamics.

More in detail, the procedure uses the following data structures.

```
EXPLORE(OFHCP $\mathcal{F}$, discretisation $D$) {
 let $\mathcal{F}$ = ($\mathcal{H}$, $G$, $\epsilon$, $T$, $C$);
 let $\mathcal{H}$ = ($X$, $Q$, $U$, $W$, $I$, $f$, $p$);
 states_current_level := 0;
 foreach $s \in I$ {
  $(x_s, q_s)$ := $D(s)$; //discretize the state
  Enqueue(Q_S, $(x_s, q_s)$);
  Insert(HT, $(x_s, q_s)$);
  states_current_level := states_current_level + 1;
  if ($(x_s, q_s)$ is an $\epsilon$-goal) {
   Enqueue(Q_G, $(x_s, q_s)$);
   HT[$(x_s, q_s)$].cost := 0;
  }
 }
 current_BFS_level := 1;
 states_next_level := 0;
 let $\mathcal{F}_{\mathcal{H}} = (S, A, F_T, I_S)$ be defined as in Def. $6$;
 while ((Q_S $\neq \emptyset$) $\wedge$ (current_BF_level $\leq$ T)) {
  $(x, q)$ := Dequeue(Q_S);
  foreach $(y, r)$ $\in \{F_T(x, q, u, w) \mid (u, w) \in A\}$ {
   if ($(y, r)$ $\notin$ HT) {
    Insert(HT, $(y, r)$);
    if ($(y, r)$ is an $\epsilon$-goal) {
     Enqueue(Q_G, $(y, r)$);
     HT[$(y, r)$].cost := 0;
    }
    else {
     Enqueue(Q_S, $(y, r)$);
     states_next_level := states_next_level + 1;
    }
   }
   PT[$(y, r)$] := PT[$(y, r)$] $\cup$ $(x, q)$;
  }
  //BFS level calculation
  states_current_level := states_current_level - 1;
  if (states_current_level = 0) {
   states_current_level := states_next_level;
   states_next_level := 0;
   current_BFS_level := current_BFS_level + 1;
  }
 }
}
```

Figure 3: The EXPLORE Procedure.

- A **Hash Table** HT used in order to store already visited states;
- A **Queue** Q_S containing the states to be expanded;
- A **Predecessors Table** PT containing, for each visited state, a list of its directed predecessors;
- A **Queue** Q_G where we put the ϵ-goal states encountered during the visit;

EXPLORE takes as input an OFHCP $\mathcal{F}$ = ($\mathcal{H}$, G, ϵ, T, C) and a discretisation D, and fills the structures HT, PT and Q_G with the reached states, discretised through D. To this aim, it performs a BF visit of the DTHS state space, starting from all the (discretised) initial states.

Namely, the initial states are inserted in the queue Q_S (line 7) after being discretised through D (line 6). As usual in the BF strategy, we expand the discrete state (x, q) in the front of the queue by computing (line 20) the set of the successor states of (x, q), i.e. $\{F_T(x, q, u, w)|(u, w) \in A\}$. Note that this is performed on the continuous dynamics function of the DTHS, exploiting the full machine precision.

Each successor (y, r) of (x, q) that has not been already visited (i.e., is not in HT), is inserted in HT (line 22).

If (y, r) is a ϵ-goal state, it is also inserted in Q_G, with cost set to zero. (line 24. Otherwise, it is inserted in the BFS queue Q_S .

Finally, (x, q) is added to the predecessor list of (y, r) in table PT. In this way, for all $(x, q) \in S$ which are reachable from a discretisation of an initial OFHCP state (i.e. from a (x_s, q_s) s.t. $\exists s \in I\ D(s) = (x_s, q_s)$), PT[$(x, q)$] contains all the states which may go in (x, q) by means of an action, i.e. (y, r) $\in$PT[(x, q)] iff $(y, r) = F_T(x, q, u, w)$ for some $(u, w) \in A$.

4.2 The SYNTHESISE Procedure

The SYNTHESISE procedure in Fig. 4 makes a BF visit of the inverted graph resulting from the FSS $\mathcal{F}_{\mathcal{H}}$. To this end, SYNTHESISE uses the information in Q_G, HT and PT prepared by EXPLORE. Namely, we start from the ϵ-goal states in Q_G and we navigate each edge backward via the table PT.

The procedure takes as input the DTHS $\mathcal{H}$, a discretisation D and a *cost function* $C : S \times A \to \mathbb{R}^+$ (where S and A are the discretisation by D of states and actions, see Def. 6), and returns as output the *controller table* CTRL, containing (state,action) pairs.

Of course, if only *time optimality* is required, then we simply fix $C = 1$, giving a *unitary* cost for all transitions of the system, so that the problem of finding an optimal controller reduces to select the shortest paths between each state and the goal state.

The BF visit queue Q_S is initialised with the ϵ-goal states in Q_G. Then, in the generic iteration we first extract a state (x, q) from Q_S and, for all predecessors (y, r) of (x, q) (line 9) we perform the following steps.

First, we pick a local_action among the control actions $(u, w) \in A$ such that $F_T(y, r, u, w) = (x, q)$ and $C(y, r, u, w)$ is minimum (lines 10-12).

```
1  SYNTHESISE(DTHS $\mathcal{H}$, discretisation D, cost_function $C$) {
2   let $\mathcal{H}$ = ($X$, $Q$, $U$, $W$, $I$, $f$, $p$);
3   let $\mathcal{F}_\mathcal{H} = (S, A, F_T, I_S)$ be defined as in Def. $6$;
4   CTRL := $\emptyset$;
5   Q_S := Q_G; //this erases the previous content of Q
6   while (Q_S $\neq \emptyset$) {
7    $(x,q)$ := Dequeue(Q_S);
8    previous_cost := HT[$(x,q)$].cost; //0 if HT[$(x,q)$].state is a goal
9    foreach $(y,r) \in$ PT[$(x,q)$] {
10    local_cost := $\min_{(u,w)\in A \mid F_T(y,r,u,w)=(x,q)} C(y,r,u,w)$;
11    U := $\{(u,w) \in A \mid F_T(y,r,u,w) = (x,q) \wedge C(y,r,u,w) =$local_cost$\}$;
12    local_action := pick an action in U;
13    if (CTRL[$(y,r)$] = $\emptyset\vee$HT[$(y,r)$].cost > previous_cost + local_cost){
14     CTRL[$(y,r)$] := local_action;
15     HT[$(y,r)$].cost := previous_cost + local_cost;
16     Enqueue_in_Order(Q_S, $(y,r)$);
17  } } }
18  return CTRL;
19 }
```

Figure 4: The SYNTHESISE Procedure.

Then, if either the control action for (y,r) has not been defined yet, or `local_action` leads to better results than the already computed one (line 13), `CTRL[`(y,r)`]` and `HT[`(y,r)`].cost` are properly updated (lines 14 and 15, resp.), and (y,r) is enqueued in `Q_S` in ascending order w.r.t. cost C to be later expanded. This phase ends when the queue `Q_S` is empty, and the final controller table `CTRL` is returned.

Note that the `SYNTHESISE` procedure is the same as Dijkstra algorithm except that the nodes are not enqueued in `Q_S` before the main loop. Namely, nodes (states) with infinite cost are not enqueued since as soon as a state (y,r) is found to have a finite cost, (y,r) is enqueued in `Q_S`. Since, in the Dijkstra algorithm, if the queue is left with states with infinite costs only, no further modifications take place in the shortest path tree, the algorithm in Figure 4 is indeed the same of Dijsktra algorithm. This implies the optimality of the returned controller.

4.3 The `VALIDATE` Procedure

Finally, the `VALIDATE` procedure, shown in Fig. 5, is used to check if the chosen discretisation has generated a controller that fits the user-defined tolerance.

In particular, we measure such tolerance by means of two important parameters:

- the *trajectory control*, i.e., the percentage of states belonging to the *real* trajectories (generated using the full machine precision) that are controlled to the set-point by the *pseudo-trajectories* (generated using the given discretisation). This parameter is used to estimate the *control errors* introduced by the discretisation.
- the *trajectory delay*, i.e., the ratio between the length of the pseudo-trajectories and the one of the corresponding real trajectories. This parameter is used to estimate how *slower* are the pseudo-trajectories due to their approximation.

```
int LengthPseudoTraj(table CTRL,state $(x_1, q_1)$, int max_steps,double $\varepsilon$)
{
 steps := 1;
 while (steps < max_steps) {
  $(u, w)$ := find_action(($\hat{x}_1, q_1$),CTRL);
  $(\hat{x}_2, q_2)$ := $F_T(\hat{x}_1, q_1, u, w)$; //using discretisation
  if (($\hat{x}_2, q_2$) is a $\epsilon$-goal)
   return steps;
  steps++;
  $(\hat{x}_1, q_1)$ := $(\hat{x}_2, q_2)$;
 }
 return -1;
}
int LengthRealTraj(table CTRL,state $(x_1, q_1)$, int max_steps,double $\varepsilon$)
{
 steps := 1;
 while (steps < max_steps) {
  $(u, w)$ := find_action(($\hat{x}_1, q_1$),CTRL);
  $(x_2, q_2)$ := $(f(x_1, q_1, u, w), p(x_1, q_1, u, w))$; //full precision
  if (($x_2, q_2$) is a $\epsilon$-goal)
   return steps;
  steps++;
  $(x_1, q_1)$ := $(x_2, q_2)$;
 }
 return -1;
}
bool VALIDATE(table CTRL,int $T$,double $\varepsilon$,int tdelay, double tcontrol)
{
 foreach state $(x, q)$ $\in$CTRL {
  t1 := LengthPseudoTraj($(x, q)$,$T$,$\varepsilon$);
  if (t1 > -1) {
   t2 := LengthRealTraj($(x, q)$,$t1 + t1*$tdelay,$\varepsilon$);
   if (t2 > -1)
    controllable := controllable + 1;
   else
    real_out := real_out + 1;
  } else pseudo_out := pseudo_out + 1;
 }
 return ((controllable/(controllable+real_out+
          pseudo_out))>=tcontrol);
}
```

Figure 5: The VALIDATE Procedure.

The VALIDATE procedure takes as input a controller table CTRL, the design parameters T (horizon) and ε (goal approximation), the required trajectory delay ($tdelay$) and control ($tcontrol$), and returns true if the latter two parameters are satisfied.

For each state in CTRL , VALIDATE determines if the state is controlled both by the real and discretised trajectories, and the corresponding lengths. These values are aggregated and finally compared with the user-defined tolerances to determine the return value of the function.

5 Robust Controller Generation

The controller CTRL generated by the OPTIMAL_CONTROL algorithm in Section 4 is an*optimal* controller, but may not be *robust*.

Given a set of range of disturbances $\Delta = \{\delta_1, \ldots, \delta_n\}$ on state variables and control variables (commonly due to sensor noise and approximation of continuous variables), we say that a controller is robust if it is able to cope with Δ-disturbed trajectories, which may lead to unknown states, although *close* to some states in the table.

Since this is an always desirable feature, in many cases the optimal controller should be refined to make it robust. Obviously, it is impossible to correctly handle every possible disturbance, thus even a robust controller will be actually unable to control the plant in some cases. In particular, in this paper we will apply a statistical method to select a set of interesting plant states, in order to concentrate the refinement process *around* them.

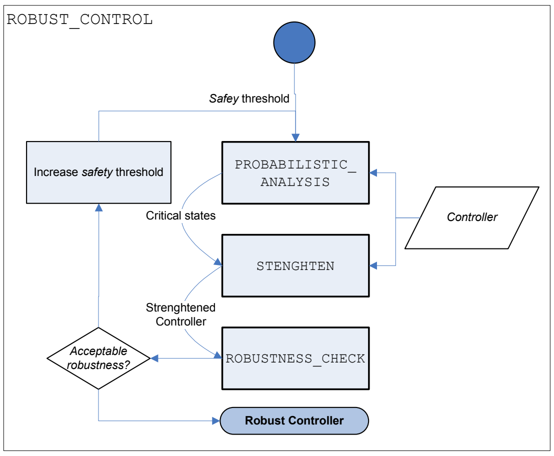


Figure 6: Structure of the ROBUST_CONTROL Procedure.

The overall robust controller generation algorithm is represented by the ROBUST_CONTROL procedure depicted in Figure 6.

The idea is to *strengthen* the controller by adding new states in order to increase its robustness degree. However, since the strengthening would require a huge amount of time and memory to be performed, we do not apply this procedure to all the states of the controller. Indeed, we first apply a probabilistic analysis in order to select the most important system states. In this way, the controller strengthening is performed only on these states, ensuring a high degree of robustness with a relatively small growth of the controller table.

Finally, to check the actual controller robustness, the `CHECK_ROBUSTNESS` procedure is applied to the final controller. If the robustness degree is not satisfying, we repeat the `STRENGTHENING` algorithm on a larger set of states, obtained from a call to `PROBABILISTIC_ANALYSIS` with stricter parameters.

In the following sections we describe both `PROBABILISTIC_ANALYSIS` and `STRENGTHENING` procedures in more detail.

5.1 The `PROBABILISTIC_ANALYSIS` Procedure

In this phase, we want to select the *most significant states* of the plant under analysis, in order to focus the strengthening only on these states. Indeed, some states are rarely reachable by the plant, or may represent ”no way out” states, where any kind of recovery fails, and thus it is useless to strengthen these states.

Therefore, the selection process is performed using a *probabilistic analysis* algorithm described in the following.

For each controlled state s (i.e., in `CTRL`), we first calculate the probability $p_c(s)$ that from s, after any sequence of allowed actions, we are still in a state of the controller. This gives us a measure of how much the states deriving from s can be handled by our controller.

In particular, since we actually cannot take into account any possible sequence of actions, we *approximate* our algorithm by considering only sequences σ of a given length k_c. Note that the choice of k_c depends on the required degree of accuracy. In the following, we suppose k_c to be fixed.

The selected sequences form a tree $\mathcal{T}$ rooted on s. For a given sequence σ in $\mathcal{T}$, we define $|\sigma|$ as the minimum value (if it exists) of i, with $1 \leq i \leq k_c$, such that the action $\sigma(i)$ leads to a state in the controller. We leave $|\sigma|$ undefined otherwise.

Now let $\mathcal{T}^*$ be the set of all sequences σ such that $|\sigma|$ is defined. The value of $p_c(s)$ is computed by the following equation:

$$p_c(s) = \begin{cases} \sum_{\sigma \in \mathcal{T}^*} \frac{1}{|A|^{|\sigma|}} & \text{if } \mathcal{T}^* \text{ is not empty;} \\ 0 & \text{otherwise.} \end{cases} \qquad (1)$$

where $|A|$ is the number of actions in the FSS (see Definition 6).

We say that a state s is *dead* if $p_c(s) < M_c$, where the *safety threshold* M_c is a small value, say below 10%. Otherwise, s is *live*. *Live* states represent the normal states of the plant, whereas *dead* states correspond to *extreme* states, that are practically uncontrollable in case of disturbances.

Therefore, our idea is to identify the set of *live* states $\mathcal{S} = \{s \mid p_c(s) \geq M_c\}$ and concentrate the strengthening process on them.

Figure 7 shows the `PROBABILISTIC_ANALYSYS` algorithm. The probability $p_c(s)$ is calculated *incrementally* using again a breadth-first visit starting from s. This

```
PROBABILISTIC_ANALYSIS(controller table CTRL,int $k_c$, double $M_c$)
{
  foreach $s \in$ CTRL {
    queue Q := $s$;
    $p_c(s) := 0$;
    while ((Q$\neq \emptyset$) && ($BF\_Level \leq k_c$)) {
      $s$ := Dequeue(Q);
      foreach $t \in F_T(s)$ {
        if ($t \in$ CTRL) $p_c(s) = p_c(s) + \frac{1}{n^{BF\_Level}}$;
        //n is the number of actions
        else Enqueue(Q,$t$);
    } }
    if ($p_c(s) < M_c$) $s$ is $dead$;
    else $s$ is $live$;
} }
```

Figure 7: The PROBABILISTIC_ANALYSIS Procedure.

time the exploration of each path is stopped after k_c levels or when a state of the controller is reached. Indeed, from Equation (1) we know that the remaining paths do not contribute to the increment of $p_c(s)$. The exploration also stops on *error states* (that are clearly *unrecoverable*), which may or may not exist according to the system under consideration.

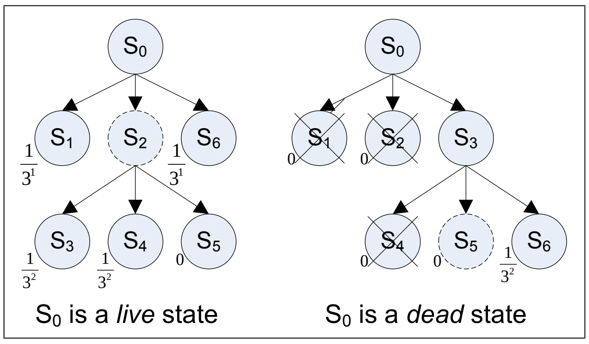


Figure 8: Examples of probability computation.

Two examples of this computation are shown in Figure 8, where solid circles are the states in CTRL, dashed circles are the states not in CTRL, barred circles are error states and, for each leaf-node, the contribution to $p_c(s)$ is indicated. Supposing that $M_c = 20\%$, in the left case we have that s_0 is *live* since $p_c(s_0) = \frac{8}{9} \geq M_c$, whereas in the right case s_0 is *dead* since $p_c(s_0) = \frac{1}{9} < M_c$.

Once we have built the set $\mathcal{S}$, we can apply the strengthening only on the *live* states.

5.2 The STRENGTHENING Procedure

As described in the previous Section, the *strengthening* of the controller CTRL is applied only on the set of *live* states $\mathcal{S}$. The STRENGTHENING algorithm is shown in Figure 9.

```
STRENGTHENING(controller table CTRL, safe states $\mathcal{S}$, int $k_s$, double $M_l$,
 double $M_h$)
{
  foreach $s \in \mathcal{S}$ {
    $s_1, \ldots, s_k$ := rand_dist$_1$($s$),...,rand_dist$_k$($s$);
    foreach $s_j \in \{s_1, \ldots, s_k\}$ {
      queue Q := $s_j$;
      $\alpha :=$ NULL; $p_c(\alpha) := 0$;
      while ((Q$\neq \emptyset$) && ($BF\_Level \leq k_s$)) {
        $s$ := Dequeue(Q);
        foreach $t \in F_T(s)$ {
          if ($t \in$ CTRL) {
            if ($p_c(t) \geq M_l$) {
              add path from $s_j$ to $t$ in CTRL; break;
            } else if ($p_c(t) \geq p_c(\alpha)$) {
               $\alpha := t$; $p_c(\alpha) := p_c(t)$;
            }
          } else Enqueue(Q,$t$);
      } }
      if ($p_c(\alpha) \geq M_h$)
        add path from $s_j$ to $\alpha$ in CTRL;
      else $s_j$ is $dead$;
} } }
```

Figure 9: The STRENGTHENING Procedure.

In particular, to ensure the robustness of the controller, we explore a larger number of states obtained by *randomly perturbing* the states in $\mathcal{S}$. Such random changes simulate the possible control errors and state disturbances that may happen in the real plant dynamics but cannot be described by the plant model.

That is, for each state $s \in \mathcal{S}$ we apply a set of small random changes and obtain a set of new states which, generally speaking, are not in the controller. Then, from each new state s', we start a breadth-first visit of the plant state space, stopping it after a given number k_s of levels or when we find a state s'' such that $p_c(s'') \geq M_l$, that is a *sufficiently safe* controlled state. Note that, in a sense, here we use the safe states of the controller as *an extended setpoint*. Finally, let $N(s')$ be the set of visited states during this visit and let t be a state in $\mathcal{S} \cap N(s')$ such that $\forall t' \in \mathcal{S} \cap N(s') : p_c(t) \geq p_c(t')$. If $p_c(t) > M_h$, the path from s' to t is stored in CTRL, otherwise we declare s' *dead* (see Section 5.1).

Note that, again, the choice of the constants k_s, M_l and M_h depends on the required controller accuracy. In particular, M_l is the *minimum probability value* that we accept to consider a state *near* to the controller and M_h the *maximum probability value* that we consider *too low* to for state to be *safe*. Obviously we require that $M_h < M_l$.

After some iterations of this strengthening process, we have that the final controller CTRL is able to drive $\mathcal{P}$ from any reasonable system state to the *best* near state of the optimal controller and, from there, reach a goal. That is, CTRL has been augmented with new $(state, action)$ pairs in order to deal with a larger number of possible plant states. This makes it *robust* without affecting too much its optimality.

Note that both the probabilistic analysis and the strengthening algorithms are highly parallelisable. Indeed, the controller states can be partitioned in several subsets and processed simultaneously by different processes (possibly on different machines).

5.3 The ROBUSTNESS Test

Finally, in order to check the robustness of the final controller, one can apply the following *robustness test*:

1. consider, for each live state in the controller, a trajectory starting from it;
2. for each state s occurring in a given trajectory, apply a random disturbance on the state variables (within a user-defined range of possible disturbances), generating a new state s^p;
3. apply to s^p the rule associated to the controller state s' that is nearest to s^p.

A trajectory is *robust* if, applying the disturbances above, it eventually reaches the setpoint. Using this test, it is possible to iterate the strengthening process until the required robustness degree for the controller is reached.

6 Controller Compilation

Embedding and querying the obtained numerical controller within a small hardware or software device is also an issue that may be addressed using CGMurphi.

Indeed, the simplistic solution of embedding the controller table together with a lookup procedure in the target device is realistic only for very small controllers, e.g., with a size that does not exceed a megabyte. Also in this case, saving some space may be a valuable result.

To this aim, CGMurphi is able to encode the controller table using a representation based on Ordered Binary Decision Diagrams [11]. As we will see, this representation is extremely compact, if compared with the original table, and easy to translate in artifacts, such as VHDL architectures or C libraries, that are directly embeddable in most of the hardware/software systems.

This phase is called controller compilation and its overall algorithm is represented by the COMPILE_CONTROL procedure depicted in Figure 10.

Here, the numerical controller table (note that this phase could be applied on *any* numerical controller table that is written in the CGMurphi binary format) is fed to the COMPRESS procedure, which complies it in a OBDD that compactly represents the state-action binary relation encoded in the table. Then, the decision diagram is manipulated by the SPLIT procedure to obtain a set of OBDDs that encode state-action-bit relations, i.e., binary functions that return a single bit of the action associated to the given state. Finally, the state-action-bit relations can be rewritten as a set of C functions or as a VHDL architecture, ready to be embedded in a software or hardware project, respectively.

6.1 The COMPRESS Procedure

OBDDs are directed acyclic graphs that represent boolean functions in a canonical form. It is possible to reduce dramatically the size of an OBDD representation by

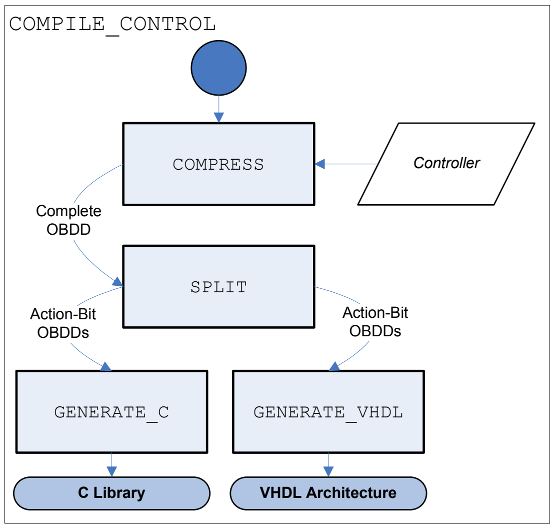


Figure 10: Structure of the COMPILE_CONTROL Procedure.

means of the elimination of: duplicate terminals nodes, duplicate nonterminals nodes and redundant tests.

A useful way to see BDDs, that will be used in this paper, is that they *encode the compressed representation of a relation*. However, unlike other compression techniques, the actual operations on BDDs are performed *directly on that compressed representation*, i.e. without decompression.

On the other hand, a controller table containing a set of (state,action) pairs represents a relation $R = \{(s, a) | a \text{ is the action associated to } s \text{ in the controller table}\}$ between states and actions. Since BDDs encode formulas, it may be useful to represent R through its characteristic function C_R defined as follows:

$$C_R(s, a) = \begin{cases} T & \text{if } (s, a) \in R \\ F & \text{otherwise} \end{cases}$$

Now, to write a definition of C_R as a boolean formula, we first have to represent its arguments, i.e., states and actions, in terms of logic variables. To this aim, we expand them to their binary memory representation.

Let suppose that states are n-bit values and actions are m-bit values. We write $s[i]$ and $a[i]$ to denote the ith bit of state s and action a,respectively.

Let x_i, $i = 1 \ldots n$ and y_j, $j = 1 \ldots m$ be $n + m$ boolean variables. A state s is then be represented by the formula

$$f_s(x_1, \ldots, x_n) = \bigwedge_{i=1 \ldots n} l_i \text{ where } l_i = \begin{cases} x_i & \text{if } s[i] = 1 \\ \bar{x}_i & \text{if } s[i] = 0 \end{cases}$$

Each f_s is a boolean formula in n variables that is true if and only if its variables are assigned with the bits of s (denoting, as usual, the boolean true with 1 and the boolean false with 0). In the same way, an action a corresponds to the formula

$$f_a(y_1, \ldots, y_m) = \bigwedge_{i=1\ldots m} l_i \text{ where } l_i = \begin{cases} y_i & \text{if } a[i] = 1 \\ \bar{y}_i & \text{if } a[i] = 0 \end{cases}$$

Therefore, the controller characteristic function C_R can be encoded by the boolean formula

$$f_R(x_1, \ldots, x_n, y_1, \ldots, y_m) = \bigvee_{(s,a)\in R} (f_s \wedge f_a)$$

f_R is a boolean formula in $n + m$ variables that is true if and only if the variable assignment corresponds to the encoding of a (s, a) pair for which $R(s, a)$ holds.

For example, assume that the controller table contains the following 2-bit states $s = 00, s' = 01, s'' = 10$, with the following associated 1-bit actions $u = 0, u' = 0, u'' = 1$. Then the formula for the characteristic relation would be $f_R = \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{y}_1 + \bar{x}_1 \cdot x_2 \cdot \bar{y}_1 + x_1 \cdot \bar{x}_2 \cdot y_1$.

Moreover, we have to fix some BDD encoding parameters, namely the variable ordering in the boolean formulas and the dynamic reordering method used by the BDD package.

Indeed, the BDD structure and therefore the compression ratio can be influenced by the original ordering of the variables in the boolean formulas presented to the BDD package. In particular, we recall that the variables in our BDDs are the state bit variables, namely $x_i, i = 1 \ldots n$, and the action bit variables, $y_i, i = 1 \ldots m$. Thus, we may consider the variable orderings arising from all the possible combinations of the following conditions:

- the state bit variables and the action bit variables can be ordered with different endianness, that is from the most significant bit to the least or vice-versa;
- the state bit variables can be placed before the action bit variables, after them or interleaved.

Namely, we can write the function f_R with any of the ten variable orderings $O1 \ldots O10$ shown in Table 1. Note that in $O9$ and $O10$ we assume $n > m$.

Moreover, variables can be dynamically reordered by the BDD package during the construction of the final BDD. In our experiments, we used the fourteen dynamic reordering methods offered by the CUDD package.

The `COMPRESS` algorithm, whose pseudocode is shown in Figure 11, implements the technique described above.

After reading the number of bits in the controller states and actions, the `BDD_encoding` procedure creates the corresponding set of boolean variables x_i and y_i, respectively.

Then, for each entry of the controller table, a new BDD E is created as the appropriate conjunction of the state and action variables. In particular, the code checks every bit in the state and adds to the BDD f_s a conjunction with the corresponding

O1	$x_1 \cdots x_n y_1 \cdots y_m$
O2	$x_1 \cdots x_n y_m \cdots y_1$
O3	$x_n \cdots x_1 y_1 \cdots y_m$
O4	$x_n \cdots x_1 y_m \cdots y_1$
O5	$y_1 \cdots y_m x_1 \cdots x_n$
O6	$y_1 \cdots y_m x_n \cdots x_1$
O7	$y_m \cdots y_1 x_1 \cdots x_n$
O8	$y_m \cdots y_1 x_n \cdots x_1$
O9	$x_1 y_1 x_2 y_2 \cdots x_m y_m x_{m+1} \cdots x_n$
O10	$x_n, y_m x_{n-1} y_{m-1} \cdots x_{m-n} y_1 x_{m-n-1} \cdots x_1$

Table 1: Possible initial variable orderings

```
BDD COMPRESS(controller_table CTRL) {
 read number $N$ of entries in CTRL;
 read number $n$ of bits in each state of CTRL;
 read number $m$ of bits in each action of CTRL;
 foreach $j = 1 \ldots n$ create boolean variable $x_j$;
 foreach $j = 1 \ldots m$ create boolean variable $y_j$;
 BDD $f_R$;
 foreach $i = 1 \ldots N$ {
  BDD E, $f_s$, $f_a$;
  foreach $j = 1 \ldots n$ //encode state bits
   if (bit(CTRL[i].state,j) == 1) $f_s$ = $f_s$ $\wedge$ $x_j$;
   else $f_s$ = $f_s$ $\wedge$ $\bar{x_j}$;
  foreach $j = 1 \ldots m$ //encode action bits
   if (bit(CTRL[i].action,j) == 1) $f_a$ = $f_a$ $\wedge$ $y_j$;
   else $f_a$ = $f_a$ $\wedge$ $\bar{y_j}$;
  E = $f_s$ $\wedge$ $f_a$;
  $f_R$ = $f_R$ $\vee$ E; //disjunction of entries
 }
 return $f_R$;}
```

Figure 11: The COMPRESS Procedure.

variable, in its positive or negated form, based on the value of the bit. The process is then repeated for the action bits in the BDD f_a, and finally E is obtained as $f_s \wedge f_a$.

The BDD E is then added to the final BDD f_R using a disjunction. When all the controller entries have been processed, `BDD_encoding` returns f_R.

As we can see, the algorithm is actually very simple, since all the BDD manipulation is done by the external BDD package. In particular, our implementation uses the CUDD [15] BDD manipulation package. Such package provides a large set of operations on BDDs and many variable dynamic reordering methods, which are crucial to gain the highest possible compression factor.

Note that, to ensure the correctness of our approach, we also developed a parallel-query algorithm that tests for correctness and completeness the BDD-encoded controller f_R with respect to the original numerical controller $CTRL$. This procedure simply compares the results obtained by querying the uncompressed and the compressed controller with all the states in the controller table.

6.2 The `SPLIT` Procedure

The BDD generated by the `COMPRESS` procedure is compact, has a relatively slow access time. Indeed, to find the action a associated to a given state s, we first need to fix the BDD variables $x_1, \dots, x_n$ to the value of the corresponding state bits (i.e., perform a BDD *restriction*), and then find the unique satisfying assignment $y_1, \dots, y_n$ for the obtained decision diagram, that corresponds to the (binary representation of the) action a. Both the BDD operations above are nontrivial and require the complete support of a BDD package like CUDD to be easily performed.

Therefore, to make the compressed controller standalone, we have to further simplify its BDD encoding. To this aim, split the BDD f_R i n a set of BDDs ${f_R}^i$, $i \in [1 \dots m]$, one for each bit of the action, defined as follows:

$$\begin{aligned} {f_R}^i(x_1, \dots, x_n) = \; & \exists y_1, \dots, y_{i-1}, y_{i+1}, \dots, y_m \\ & f_R(x_1, \dots, x_n, y_1, \dots, y_{i-1}, 1, y_{i+1}, \dots, y_m) \end{aligned}$$

In other words, ${f_R}^i$, if applied to the logical encoding of a particular state $(x_1, \dots, x_n)$ is true if and only if there is an action in the controller table, associated with such state, which has an 1 in its i-th bit. However, since we know that such an action, if exists, is unique, we can reformulate the definition of ${f_R}^i$ as follows: "${f_R}^i(x_1, \dots, x_n)$ is true if and only if the i-th bit of the action associated with the state represented by $x_1, \dots, x_n$ is 1".

Thus, the new set of BDDs can be used to get the action associated to a given state by simply calculating their value with respect to the variable assignment corresponding to the state binary encoding. This operation is extremely easy and fast to implement and does not need any advanced BDD manipulation algorithm.

6.3 The `GENERATE` Procedures

Due to their simplicity, the BDDs above can be easily rewritten as boolean expressions in any programming language. Indeed, the translation process is very straightforward and requires only a visit of the OBDD graph.

Let n be a node of the OBDD. If it is a constant (true or false) node, the translation is trivial. Otherwise, let $V(n)$ be the logic variable associated to the node, and let $T(n)$ and $E(n)$ be the two children of n for $V(n) = true$ and $V(n) = false$, respectively. We can define the function CT of n as follows:

$$CT(n) = \begin{cases} n = \text{true constant} & true \\ n = \text{false constant} & false \\ & \texttt{IF}\ (V(n)) \\ \text{otherwise} & \texttt{THEN}\ CT(T(n)) \\ & \texttt{ELSE}\ CT(E(n)) \end{cases}$$

where we use the common IF-THEN-ELSE construct to encode the OBDD structure. Such construct is indeed present in almost any programming language. Moreover, note that the expression obtained through this simple translation does not need any further simplification, since all the possible reductions to the underlying decision tree have been already done by the OBDD package.

In particular, we can translate the OBDD in a set of C source files containing functions like `char get_action_bit_i(char* state)`, generated through the translation process above applied to the root node of the OBDDs ${f_R}^i$, a main function `char *control(char* state)`, which collects the results of the calls to each `get_action_bit_i` and returns the complete action associated to the given state. Note that both actions and states are treated as generic byte arrays.

This translation is linear in terms of the required space, and the resulting representation can be easily embedded in a (small) hardware device resulting in good time performances.

Actually, the effective implementation of this encoding is optimised and, in particular, it looks for common sub-diagrams in order to avoid recursive calls. Thus, for example, from the OBDD representing `x XOR y XOR z` we obtain the following code:

```
tmp1 = y ? z : !z;
tmp2 = x ? tmp1 : !tmp1;
result = tmp2;
```

Moreover, internally the representation can be further optimized and restructured to accommodate the C compiler limitations: for example, the functions are split in sub-functions (possibly included in different source files) if the contained logical expression exceeds the compiler stack. To this aim, in our implementation the GCC [20] compiler, version 3.4.4, has been used as a reference.

Similarly to the C translation, we can translate the compressed controller table in a VHDL definition. We start by defining the whole controller (or, better, its characteristic function) as a module through an *entity* definition as the following:

```
entity control is
 port (
  state : in bit_vector (n downto 0 )
  action : out bit_vector (m downto 0 )
 )
end control;
```

The module has a set of input lines, representing the state bits, and a corresponding set of output lines, composing the action bits.

Then, we give the behavioral description of the actual circuits that compute the module outputs. Action bits are computed by a separate *processes*, that are executed in parallel in an *architecture*, as defined in the following:

```
architecture controller of control is
 begin
  process (state)
   action(0) <= code for the first bit
  end process
  ...
  process (state)
   action(m) <= code for the m-th bit
  end process
end controller
```

The actual code that computes the value of each bit and assigns it to the corresponding output line is the same of the C translation, since VHDL supports the IF-THEN-ELSE statements.

7 Case Studies and Experimentation

To show the effectiveness of our methodology and the usability of our tool, in this section we present the experimental results related to three case studies. Namely, we first consider the following case studies.

- the *inverted pendulum on a cart* problem shows a variant of the inverted pendulum problem which presents an highly nonlinear dynamics;
- the *truck and trailer obstacles avoidance* problem shows the ability of our technique to deal with very complex hybrid problems.
- finally, the *turbogas control* problem (that is only summarised in this paper) shows a case where the huge system state space would be impossible to analyse with other tools based on explicit state space generation.

7.1 The Inverted Pendulum on a Cart Problem

As a first example, we consider the inverted pendulum on a cart problem (Fig. 12), according to the formulation presented by Junge and Osinga in [27].

7.1.1 Problem Formulation

The system consists of a planar inverted pendulum on a cart that moves under an applied horizontal force u, constituting the control. The position x_1 of the pendulum is measured relative to the position of the cart as the offset angle from the vertical upright position. The motion of the pendulum is modelled by the following equations:

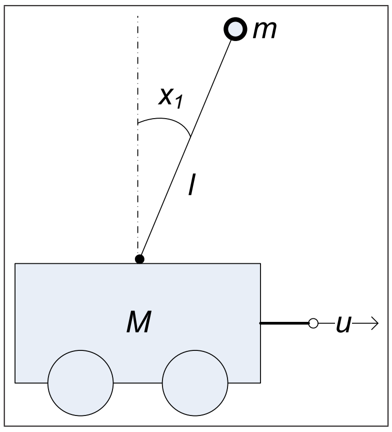


Figure 12: Inverted Pendulum on a Cart.

$$
\begin{aligned}
\dot{x}_{(1)} &= x_{(2)} \\
\dot{x}_{(2)} &= \frac{\frac{g}{l}\sin(x_1) - \frac{1}{2} m_r x_2^2 \sin(2x_1) - \frac{m_r}{ml}\cos(x_1)u}{\frac{4}{3} - m_r \cos^2(x_1)}
\end{aligned}
$$

where the parameters are as in [27], i.e. the mass of the cart is $M = 8$kg, the mass of the pendulum in $m = 2$kg, the distance between the center of mass and the pivot is $l = 0.5$, the mass ratio is $m_r = m/(m + M)$, and the gravitational constant is $g = 9,8$m/s^2.
The instantaneous cost is $q(x, u) = \frac{1}{2}(0.1x_1^2 + 0.05x_2^2 + 0.01u^2)$.

7.1.2 CGMurphi Model Description

Now, we can start to build our CGMurphi model by defining the state of the system, which is represented by the set of the *state variables*. As shown in Fig. 13, CGMurphi allows one to define the type `real(m,n)` in order to represent real numbers with m digits for the mantissa and $\bar{n} = log_{10}\lfloor n \rfloor + 1$ digits for the exponent.

```
type  real_type : real(4,9);
var   x1: real_type; --pendulum angle
      x2: real_type; --angular velocity
```

Figure 13: CGMurphi state definition for the Inverted Pendulum on a Cart.

Note that, since CGMuphi is not able to deal with differential equations such as the ones describing the pendulum motion shown in the previous section, to create the model we discretised the continuous dynamics and we used discrete time steps and step functions for the state values. In particular, we approximated x_1 to 0.01, x_2 to 0.1 and t to 0.1. These approximations were experimentally validated as adequate to provide valid solutions for the original problem.

Region of Interest.

As second step, we have to define the region of interest for the continuous variables. The boundaries are imposed by physical, theoretical, and/or design constraints.

For instance, when dealing with angles, as in the inverted pendulum case, a typical range would be $[-\pi/2, \pi/2]$ rad. For the angular velocity, instead, the boundary is directed defined by the safety constraint of the problem, so that the range is $[-8, 8]$ rad/sec.

Finally, physical properties of the control device define the range of the control inputs. In this case, as function g suggests, we can assume $u \in [-0.7, 0.7]$.

Thus, we can add to the model the upper bounds for the state variables making use of constants and data types as shown in Fig. 14.

```
const
 MIN_X1: -4; MAX_X1 : 4;
 MIN_X2 : -8; MAX_X2 : 8;
 MIN_FORCE: -0.7; MAX_FORCE: 0.7; Step_Force: 0.1;
 TOLL_X1 : 0.04; TOLL_X2 : 0.3;
 TIME : 0.1;
type
 real_type: real(4,9);
 force_type: MIN_FORCE..MAX_FORCE;
 interval_force : 0..((MAX_FORCE-MIN_FORCE)/Step_Force);
 x1_type : -4..4;  x2_type : -8..8;
```

Figure 14: CGMurphi user-defined data types for the Inverted Pendulum on a Cart.

Setpoint and Guarded Transition Rules.

As final step, we have to define the Setpoint and the Guarded Transition Rules.

For the inverted pendulum problem, the setpoint is the set of states near the upright equilibrium position ($x_1 = 0, x_2 = 0$). So we extend our model with the setpoint definition as shown in Fig. 15.

```
function Equilibrium(x1:real_type; x2:real_type) : boolean;
begin
  return (x1 <= 0.0 + TOLL_x1 & x1 >= 0.0 - TOLL_x1 &
          x2 <= 0.0 + TOLL_x2 & x2 >= 0.0 - TOLL_x2);
end;

setpoint "Upright Equilibrium" (Equilibrium(x1, x2));
```

Figure 15: CGMurphi setpoint definition for the Inverted Pendulum on a Cart.

Finally, we have to define the *guarded transition rule*, that is the core of our model, since it regulates the evolution of the system. We take from step 2 we have the condition for which the transition can take place. Namely, given a state s, we are interested in the successor states of s only if s is within the region of interest and s is not a goal. In this case, the set of successor states of s is computed by applying the transition rule *for* *each* control input we are considering, this is done by using the `ruleset` construct, as shown in Fig. 16.

```
function InRange(x1:real_type; x2:real_type) : boolean;
begin
  return (x1 <= MAX_x1 & x1 >= MIN_x1 &
          x2 <= MAX_x2 & x2 >= MIN_x2);
end;
ruleset f : interval_force do
 rule "Transition" (!Equilibrium(x1, x2)) ==>
  var
   tmp_x1 : extensive_type; tmp_x2 : extensive_type;
   tmp_force : force_type;
  begin
   tmp_force := MIN_FORCE + (f*Step_Force);
   tmp_x1 := next_x1(x1, x2, TIME);
   tmp_x2 := next_x2(x1, x2, tmp_force, TIME);
   if InRange(tmp_x1, tmp_x2) then
    x1 := tmp_x1;
    x2 := tmp_x2;
   endif;
  end;
 end;
end;
```

Figure 16: CGMurphi transition rule for the Inverted Pendulum on a Cart.

Model Compilation.

Once the model has been defined, we can start the controller synthesis. First, we have to compile the CGMurphi model through the `mu` compiler which takes as input the file `model_name.m` containing the model description. This generates a file `model_name.C`, containing the C++ code implementing the body of rules, start states, setpoint, functions and procedures, plus other stuffs.

Note that the option `--ctrl` enables the model compilation for controller generation, while without parameters it will be performed a verification (this mode can be used to verify the generated controller).

As final step, we have to compile the file `model_name.C` with the standard C++ compiler and launch the executable `model_name.C.o`. In this step, the user can specify the amount of memory for the hash table (and thus determining the maximum number of states which can be visited) and the amount of memory for the controller table, that is the maximum number of controllable states. Furthermore, it is possible to fix the maximum level of bfs to be explored (this option is used to set a finite time horizon).

7.1.3 Optimal Controller Generation

To apply our approach, we used the `OPTIMAL_CONTROL` procedure of Fig. 2 to find a suitable discretisation of this DTHS and compute the corresponding controller. We require (at least) a trajectory control of 98% and a trajectory delay of 5%. Table 2 shows some different discretisation tried and the corresponding validation results, where the columns show the main evaluation parameters described in Section 4.3. The chosen discretisation approximates the values of x_1 and x_2 to $1/32$ (radians) and $5/128$ (radians per second), respectively. With this discretisation applied, the problem state space contains $385 \times 501 = 192,885$ states and 17 control actions, thus a total of $3,279,045$ transitions.

Discretisation		Uncontrolled	Discretised Trajectories	Real Trajectories	*Traj. control*	*Traj. delay*
x_1	x_2	States	out of horizon	out of delay		
$1/16$	$5/64$	39,363	0	39,363	74%	5%
$1/32$	$5/128$	3,476	0	3,476	98%	5%

Table 2: Discretisation of the Inverted Pendulum on a Cart

```
=========================================================================
EXPLORE
Progress Report:
 1000 states explored in 0.14s, with 894 states in the queue
 1798 predecessors. 0 goal states reached, current level: 3.
 ....
Final Report:
 192885 states explored in 240.65s.
 2206920 predecessors. 63 goal states reached.
========================================================================
SYNTHESISE
Final Report:
 151394 controlled states
========================================================================
```

Figure 17: CGMurphi execution.

The results of the synthesis are reported in Table 3. The corresponding execution trace is shown in Figure 17.

States	Transitions	CTRL Rules	Time (sec.)	CTRL Size
192885	3,277,974	151,394	241	1,478 Kb

Table 3: Controller Synthesis for the Inverted Pendulum on a Cart

Fig. 18 shows an example of how the controller drives the pendulum to the upright equilibrium position.

7.1.4 Robust Controller Generation

Due to some structural properties of the inverted pendulum on a cart problem, robustness can be achieved in this controller by simply using interpolation. In other words, when the controller is presented with a state that is not in the controller table, it should simply choose the nearest states in the table and use them to interpolate the corresponding action. For this reason, we instructed CGMurphi to skip the robust controller generation phase and proceed directly with the compilation of the optimal controller.

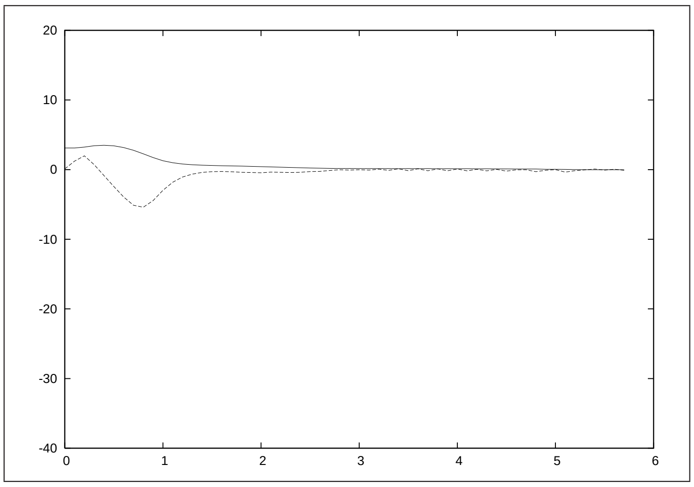


Figure 18: Inverted Pendulum on a Cart trajectory with initial condition $x = [3.1, 0.1]$ (x_1(——),x_2($- - -$) versus time).

7.1.5 Controller Compilation

Table 4 shows the compression results for the optimal controller, both using LZ on the binary controller table and using the CGMurphi `COMPRESS` procedure.

	Normal	LZ	BDD
Entries	151394		
Size	1,478 Kb	90 Kb (6.1%)	215 Kb (14.6%)
Time	3 ms	206 ms	1 ms

Table 4: Inverted Pendulum on a Cart controller compression results

We see that on a small controller the BDD compression has a lower compression ratio than LZ, but always better access times (1ms vs. 206ms), since it does not require any decompression to read the table entries.

7.2 The Truck and Trailer Obstacles Avoidance Problem

The goal of the controller is to back a truck with a trailer up to a specified parking place starting from any initial position in the parking lot. Moreover, the parking lot contains some obstacles, which have to be avoided by the truck while maneuvering to reach the parking place. The obstacles position and geometry are given in a tabular way, i.e. each obstacle is a composition of bidimensional figures defined through the position of their vertexes relative to the parking lot origin. This is a reasonable representation that could be automatically generated, e.g. by analysing an image of the parking lot. We also disallow *corrective maneuvers*, that is the truck cannot move forward to *backtrack* from an erroneous move.

In this setting, finding a suitable maneuver to reach the goal from any starting position is a hard task. On the other hand, as pointed out in the Introduction, finding an *optimal* maneuver is a *very* complex problem, that cannot be modelled and resolved using common mathematical or programming strategies.

Moreover, note that the states of this system contain both continuous variables (e.g., the truck position) and discrete ones (e.g., the boolean variable that indicates if the truck has hit an obstacle). In such a *hybrid system* interpolation techniques cannot be use to obtain a robust controller.

In the following, after giving more details about the truck and trailer model, we show the results obtained by using the CGMurphi tool to synthesise an optimal and robust numerical controller for this problem.

7.2.1 Problem Formulation

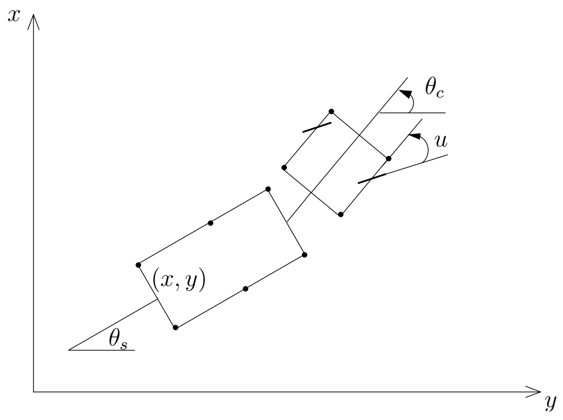


Figure 19: Truck and Trailer System Description.

Our model of the truck and trailer is based on the set of equations presented in [36]. Moreover, in our setting the *parking region* is an open bounded region of $\mathbf{R^2}$, delimited by a set of obstacles. We call Γ the parking region. The system has four state variables relative to its position in Γ: the coordinates of the center rear of the trailer $(x, y \in [0, 50])$, the angle of the trailer w.r.t. the x-axis $(\theta_S \in [-90°, 270°])$ and the angle of the cab w.r.t the x-axis $(\theta_C \in [-90°, 270°])$. Moreover the system has a *status variable* q, which has the following possible values: `normal`, when the truck lies in Γ and the jackknife adjustment must be not applied, `jackknife` when the truck lies in Γ and the jackknife adjustment must be applied, `stop` when the truck is parked and then it must not move anymore, `forbidden` if the truck is outside Γ or if it hits an obstacle.

The dynamics for the truck and trailer is shown in Figure 20 as an hybrid automaton [23]. Each state of the automaton represents a mode of the DTHS (i.e. a value for q). When no action is described, it is assumed that the plant does not change its state.

We assume that the truck moves backward with constant speed of $2m/s$, so the only control variable is the steering angle $u \in [-70°, 70°]$. We also allow u to take values

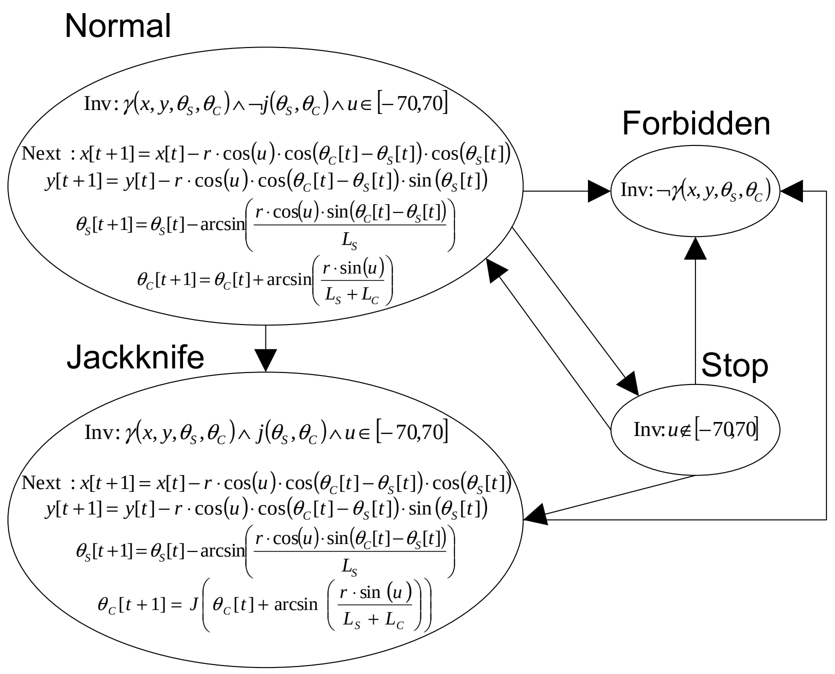

Normal
Inv: $\gamma(x, y, \theta_S, \theta_C) \wedge \neg j(\theta_S, \theta_C) \wedge u \in [-70,70]$
Next : $x[t+1] = x[t] - r \cdot \cos(u) \cdot \cos(\theta_C[t] - \theta_S[t]) \cdot \cos(\theta_S[t])$
$y[t+1] = y[t] - r \cdot \cos(u) \cdot \cos(\theta_C[t] - \theta_S[t]) \cdot \sin(\theta_S[t])$
$\theta_S[t+1] = \theta_S[t] - \arcsin\left(\frac{r \cdot \cos(u) \cdot \sin(\theta_C[t] - \theta_S[t])}{L_S}\right)$
$\theta_C[t+1] = \theta_C[t] + \arcsin\left(\frac{r \cdot \sin(u)}{L_S + L_C}\right)$
Forbidden
Inv: $\neg\gamma(x, y, \theta_S, \theta_C)$
Jackknife
Inv: $\gamma(x, y, \theta_S, \theta_C) \wedge j(\theta_S, \theta_C) \wedge u \in [-70,70]$
Next : $x[t+1] = x[t] - r \cdot \cos(u) \cdot \cos(\theta_C[t] - \theta_S[t]) \cdot \cos(\theta_S[t])$
$y[t+1] = y[t] - r \cdot \cos(u) \cdot \cos(\theta_C[t] - \theta_S[t]) \cdot \sin(\theta_S[t])$
$\theta_S[t+1] = \theta_S[t] - \arcsin\left(\frac{r \cdot \cos(u) \cdot \sin(\theta_C[t] - \theta_S[t])}{L_S}\right)$
$\theta_C[t+1] = J\left(\theta_C[t] + \arcsin\left(\frac{r \cdot \sin(u)}{L_S + L_C}\right)\right)$
Stop
Inv: $u \notin [-70,70]$

Figure 20: Truck and Trailer Hybrid Automaton.

outside this range to mean that the truck is parked and it must not move anymore. Figure 19 shows a schematic view of the truck and trailer system with its state and control variables. We single out ten points on the truck and trailer border (displayed in the Figure 19 by bold points) as *representative* of the truck and trailer position.

If the values of the state variables at time t are $q[t]$, $x[t]$, $y[t]$, $\theta_S[t]$ and $\theta_C[t]$, and the steering angle is u, then the new values of state variables at time $t+1$ are determined by the following equations:

$$
\begin{array}{rcl}
x[t+1] & = & \begin{cases} x[t] - B * cos(\theta_S[t]) & \text{if } q[t] \in \{\texttt{normal}, \texttt{jackknife}\} \\ x[t] & \text{otherwise} \end{cases} \\
y[t+1] & = & \begin{cases} y[t] - B * sin(\theta_S[t]) & \text{if } q[t] \in \{\texttt{normal}, \texttt{jackknife}\} \\ y[t] & \text{otherwise} \end{cases} \\
\theta_S[t+1] & = & \begin{cases} \theta_S[t] - arcsin\left(\frac{A * sin(\theta_C[t] - \theta_S[t])}{L_S}\right) & \text{if } q[t] \in \{\texttt{normal}, \texttt{jackknife}\} \\ \theta_S[t] & \text{otherwise} \end{cases} \\
\theta_C[t+1] & = & \begin{cases} \theta_C[t] + arcsin\left(\frac{r * sin(u)}{L_S + L_C}\right) & \text{if } q[t] = \texttt{normal} \\ J\left(\theta_C[t] + arcsin\left(\frac{r * sin(u)}{L_S + L_C}\right)\right) & \text{if } q[t] = \texttt{jackknife} \\ \theta_C[t] & \text{otherwise} \end{cases} \\
q[t+1] & = & \begin{cases} \texttt{normal} & \text{if } \gamma(x[t], y[t], \theta_S[t], \theta_C[t]) \wedge \neg j(\theta_S[t], \theta_C[t]) \wedge \\ & \wedge u \in [-70, 70] \wedge q[t] \in \{\texttt{normal}, \texttt{stop}\} \\ \texttt{jackknife} & \text{if } \gamma(x[t], y[t], \theta_S[t], \theta_C[t]) \wedge j(\theta_S[t], \theta_C[t]) \wedge \\ & \wedge u \in [-70, 70] \wedge q[t] \neq \texttt{forbidden} \\ \texttt{stop} & \text{if } u \notin [-70, 70] \wedge q[t] \in \{\texttt{stop}, \texttt{normal}\} \\ \texttt{forbidden} & \text{otherwise} \end{cases}
\end{array}
$$

where $A = r * cos(u)$, $B = A * cos(\theta_C[t] - \theta_S[t])$, $r = 1$ is the truck movement length per time step, $L_S = 4$ and $L_C = 2$ are the length of the trailer and cab, respectively (all the measures are in meters), $\gamma(x[t], y[t], \theta_S[t], \theta_C[t])$ returns true iff the truck-and-trailer system is in Γ, $j(\theta_S[t], \theta_C[t])$ returns true iff $|\theta_S[t] - \theta_C[t]| \leq 90°$ and $J(\theta_C)$ computes the correct value for θ_C when the current position does not respect the jackknife constraint.

7.2.2 CGMurphi Model Description

In the CGMurphi model we use real values to represent the state variables x and y, whilst for the angle values (i.e., θ_S, θ_C and u) it is sufficient, w.r.t. the system dimensions, to use integer values. Fig. 21 shows the description of the data types and the state in CGMurphi.

```
const MIN_ANGLE:-90; MAX_ANGLE:270;
      SP_X:10; SP_Y:0; TOLL_X:1; TOLL_Y:1;
      SP_THETA_S:90;  TOLL_THETA_S:5;
      MIN_U:-70; MAX_U:70; STEP_U: 5;
      ...

type  angle_type : MIN_ANGLE..MAX_ANGLE;
      real_type : real(5,99);
      interval_u : 0..((MAX_u-MIN_u)/STEP_u);
      ...

var   pos_x : real_type;
      pos_y : real_type;
      theta_s : angle_type;
      theta_c : angle_type;
```

Figure 21: Truck and Trailer State within the CGMurphi Model.

Moreover, we define some *tolerance* constants to set up a range of admissible final positions and angles for the center rear of the trailer. These tolerances are used to define the CGMurphi setpoint, as shown in Fig. 22.

```
setpoint "Parked"
  (pos_x <= SP_X + TOLL_X & pos_x >= SP_X - TOLL_X &
   pos_y <= SP_Y + TOLL_Y & pos_y >= SP_Y - TOLL_Y &
   angle>=SP_THETA_S-TOLL_THETA_S &
   angle<=SP_THETA_S+TOLL_THETA_S );
```

Figure 22: Setpoint for the Truck and Trailer within the CGMurphi Model.

Fig. 23 shows the main rule of the model. This rule (by means of the `ruleset` construct) computes *all* the next positions of the truck by considering all the defined control actions u (i.e. the possible maneuvers). The computation is performed by the external C functions `next_[x,y,theta_s,theta_c]` and `jackknife(tmp_theta_s,tmp_theta_c)`. In this way the description of the system dynamics is directly embedded in the external C library.

To embed the obstacles in the model, we approximate them through their bounding rectangles (or rectangle compositions). Then we consider the *representative* points of the truck-trailer position and, each time a new truck position is computed, we use

```
ruleset u : interval_u do
 rule (!Parked(pos_x, pos_y, theta_s)) ==>
 var tmp_x:real_type; tmp_y:real_type;
     tmp_theta_s:angle_type; tmp_theta_c:angle_type;
     tmp_u:u_type;
 begin
  tmp_u:=MIN_u + (u*STEP_u);
  tmp_x:=next_x(pos_x, theta_c, theta_s, tmp_u, R);
  tmp_y:=next_y(pos_y, theta_c, theta_s, tmp_u, R);

  tmp_theta_s:=next_theta_s(theta_s,theta_c,tmp_u,R,L_S);
  tmp_theta_c:=next_theta_c(theta_c,tmp_u,R,L_S,L_C);
  tmp_theta_c:=jackknife(tmp_theta_s,tmp_theta_c);

  if (!isForbidden(tmp_x,tmp_y,tmp_theta_s,tmp_theta_c,
                   M,L_S,L_C,MIN_X,MAX_X,MIN_Y,MAX_Y))
  then
   pos_x := tmp_x;
   pos_y := tmp_y;
   theta_c := tmp_theta_c;
   theta_s := tmp_theta_s;
  endif;
 end;
end;
```

Figure 23: Transition Rule for the Truck and Trailer within the CGMurphi Model.

the function `isForbidden()` to check if any of these points has hit the parking lot obstacles or borders. Therefore, our controller synthesis algorithm considers only feasible maneuvers to the goal state.

Moreover, in order to obtain a more *robust* controller we also considered the maneuvering errors due to the truck-trailer complex dynamic properties (e.g., friction, brakes response time, etc.) that cannot be easily embedded in the mathematic model. We used such errors to draw a *security* border around each obstacle and used these augmented obstacles in the collision check described above.

To estimate maximum maneuvering error we applied a *Monte Carlo's method* described as follows. We consider a large set of valid parking lot positions $S = \{s_k | 1 \leq k \leq 500,000\}$. Given a position $s_k \in S$, (1) we apply a random maneuver m_k obtaining the new position $\bar{s}_k$. Then (2) we randomly perturb s_k generating the position s_k^p and apply the same maneuver m_k on s_k^p obtaining the position $\bar{s}_k^p$. Finally, (3) we compute the distance of the selected truck points P_i between the positions s_k^p and $\bar{s}_k^p$. This process is repeated 200 times for each position in S, thus analysing 100 millions of perturbations. The security border size is the highest distance measured for a point in the step (3). We found out that this distance is 0.98m.

7.2.3 Optimal Controller Generation

We tested our methodology using several obstacles topologies. As an example, we consider the map shown in Figure 24, where the black shapes represent the obstacles and the security borders are drawn grey. In the following we present the results of each phase of our controller generation methodology.

To synthesise the optimal controller we approximate real variables rounding x and

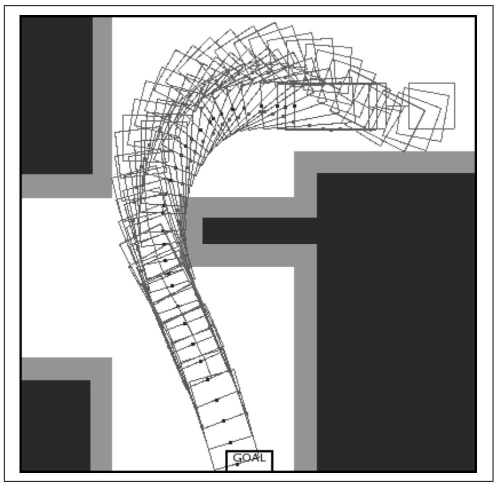


Figure 24: Truck and Trailer optimal trajectory generated by CGMurphi from initial position $x = 12$, $y = 16$, $\theta_S = 0$, $\theta_C = 0$.

States	Transitions	CTRL Rules	Time (sec.)	CTRL Size
12,227,989	364,334,756	1,749,586	13,708	37,589 Kb

Table 5: Controller Synthesis for the Truck and Trailer

y to 0.2 meters and θ_S, θ_C and u to the nearest degree. Table 5 shows the result of the synthesis (performed using a 2.8GHz Intel Xeon workstation with 4GB of RAM) and states that CGMurphi is able to deal with systems having millions of states.

7.2.4 Robust Controller Generation

Then, we single out the *live* states in the optimal controller by calculating the probability p_c on each state (see Sect. 5.1). Using Eq. 1 with $k_c = 4$ and $n = 29$ we obtain the distribution of probability p_c shown in Figure 25. The graph shows the number of states having a given value for p_c. It is clear that most of the states are *live* (high values of p_c), whereas there is a little but consistent set of *dead* states. If we set M_c to 0.1, we have that $|S| = 1,493,876$ and so in the last phase we have to consider only 85% of states in the optimal controller.

Note that, in the system under consideration, there are two kinds of *extreme* positions: (a) when the truck is very near to obstacles and (b) when the truck and trailer are in the *jackknife* position (i.e. when $|\theta_S - \theta_C| = 90°$). Indeed, in the case (a) only a very little number of actions are safe for the truck, whereas the other ones make it crash on an obstacle. On the other hand, in the case (b), it is very difficult to bring the truck outside the jackknife position, since the truck could follow an exceedingly long *almost circular* trajectory. Therefore this phase correctly identifies as *dead* all the states corresponding to positions of case (a). On the other hand, positions of case (b) could be identified as *dead* only after the third phase.

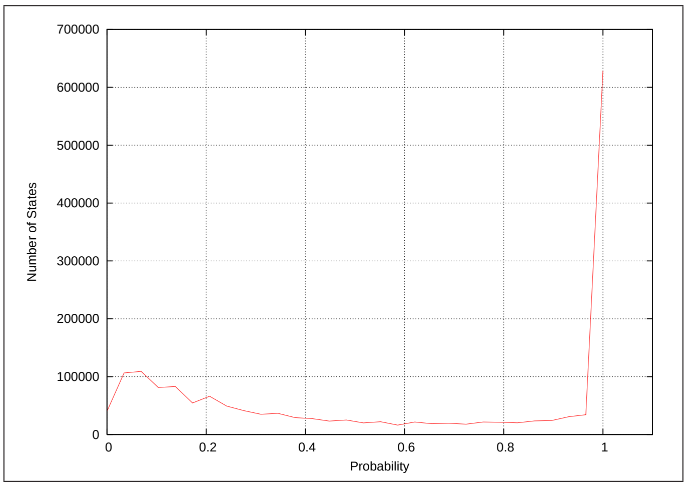


Figure 25: Distribution of p_c on the optimal controller states for the Truck and Trailer controller.

Finally, we consider the set S and we perform the strengthening of the optimal controller using the algorithm of section 5.2 and choosing $k_s = 4$, $M_l = 0.7$ and $M_h = 0.2$ as the constant values.

After the strengthening phase, a significant number of entries (1,725,529) has been added to the controller to make it robust, due to the complexity of the truck-trailer dynamics. On the other hand, 218,260 more states of the controller have been marked as *dead*: these states correspond to jackknife positions of the truck and trailer. The final controller now handles a total of 3,256,855 states and its size is 71,650Kb.

Note that the last two phases were *parallelised* by partitioning the controller states in 9 subsets and performing the analysis and strengthening separately for each of these partitions using different workstations. In this way, generating the robust controller took less than an hour.

In order to check the robustness of final controller, we considered, from each *live* state in the controller, a trajectory starting from it. For each state s occurring in a given trajectory, we applied a random disturbance on the state variables, generating a new state s^p, and then we applied to s^p the rule associated to the controller state s' that is nearest to s^p. A trajectory is *robust* if, applying the disturbances above, it eventually reaches the goal state.

Range of Disturbances for x,y	Range of Disturbances for θ_S,θ_C	Robust Trajectories
$\pm$ 0.1m	$\pm$ 5°	95%
$\pm$ 0.25m	$\pm$ 5°	94%
$\pm$ 0.5m	$\pm$ 5°	91%

Table 6: Truck and Trailer results about controller robustness

As shown in Table 6, we obtain completely satisfying percentages of robust trajectories: even in the presence of big disturbances (0.5 meters for x and y and 5 degrees for θ_s and θ_c) the controller robustness is more than 90%.

7.2.5 Controller Compilation

	Normal	LZ	BDD
Entries	3,256,855		
Size	71,650 Kb	22,644 Kb (31.6%)	7,038 Kb (9.8%)
Time	89 ms	3173 ms	108 ms

Table 7: Truck and Trailer controller compression results

As final step, we compressed the controller using the scheme presented in Section 6. Results are in Table 7. As we can see, the controller has a very big size. However, the best OBDD compression scheme is able to reduce the size of the controller up to 90.2% space savings, that is 21.8% more than using LZ77 compression. Moreover, the OBDD compression wins also with respect to the access time.

7.3 The Turbogas Control Problem

In this section we sketch our experimental results on using CGMurphi for the generation of the controller for a *real world* hybrid system, namely the gas turbine of a 2MW *electric co-generative power plant* (*ICARO*) in operation at the ENEA Research Center of Casaccia (Italy). The generated controller has to bring the plant to its *setpoint* by modifying the opening of the *fuel valve*. In the following, unless otherwise stated, all our data (e.g. block diagrams, parameter values, etc) are taken from the ICARO documentation [19].

Due to its complexity, this case study is still in progress, thus in the following we will show only the result from the Optimal Controller Generation phase.

7.3.1 Problem Formulation

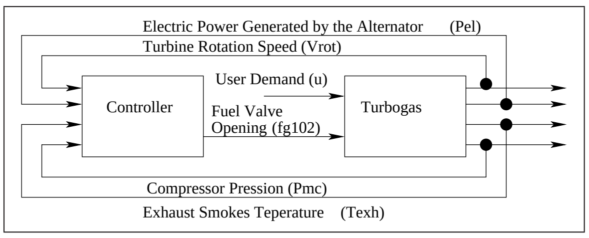

Electric Power Generated by the Alternator (Pel)
Turbine Rotation Speed (Vrot)
User Demand (u)
Controller
Fuel Valve Opening (fg102)
Turbogas
Compressor Pression (Pmc)
Exhaust Smokes Teperature (Texh)

Figure 26: High level block diagram of ICARO Turbogas Control System.

Actually, ICARO plant consists of many subsystems. Here we only focus on one of the many subsystems of ICARO (e.g., see [9, 7, 8]). Namely we focus on the *Gas Turbine* ICARO subsystem, which corresponds to the block named *Turbogas* in Fig. 26. As a matter of fact, this module consists, in turn, of many subsystems (e.g. the compressor, the combustion chamber, the turbine itself and the generator). For our purposes here we can simply use it as a black box and look at its input-output model.

The Gas Turbine module has the following input variables.

- Variable $fg102$ takes value in the real interval [0,1]. This variable gives the opening fraction of the turbogas fuel gas valve (namely valve FG102). It takes value 0 when the valve FG102 is fully closed (no fuel can flow trough the valve) and value 1 when the it is fully opened. This is a *control variable*, i.e. a variable whose value can be chosen so as to achieve the control goals.
- Variable u models the *User Demand* of electric power. This variable has to be considered as a *disturbance*, i.e. a variable whose value we (i.e. the controller) cannot choose. However, since our controller generation framework works on deterministic plant models, in our experiment we always set u to its nominal value, i.e. $\frac{MAX\_U}{2}$.

The output variables of the module are the following.

- P_{el}, the *Electric power* generated by the alternator.
- V_{rot}, the *Rotation speed* of the gas turbine.
- T_{exh}, the *Temperature* of the exhaust smokes.
- P_{mc}, the *Pressure* of the compressor.

$$\boxed{\begin{array}{rcl}
\dot{P}_{el}(t) & = & \alpha_{1,1}P_{el}(t) + \alpha_{1,2}fg102(t) + \alpha_{1,3}u(t) \\
\dot{T}_{exh}(t) & = & \alpha_{2,1}T_{exh}(t) + \alpha_{2,2}fg102(t) + \alpha_{2,3}(P_{el}(t) - P_{el}^{0}) \\
 & + & \alpha_{2,4}(P_{mc}(t) - P_{mc}^{0}) \\
\dot{V}_{rot}(t) & = & \alpha_{3,1}V_{rot}(t) + \alpha_{3,2}fg102(t) + \alpha_{3,3}(P_{el}(t) - P_{el}^{0}) \\
P_{mc}(t) & \in & [MIN\_P_{mc}, MAX\_P_{mc}] \\
\mid \dot{P}_{mc}(t) \mid & \leq & MAX\_D\_P_{mc} \\
u(t) & \in & [0, MAX\_U] \\
\mid \dot{u}(t) \mid & \leq & MAX\_D\_U
\end{array}}$$

Figure 27: Turbogas ODE model used for our analysis.

For the purposes of our analysis we used the ODE (*Ordinary Differential Equation*) model, shown in Fig. 27, to link the turbogas input variables with output variables. Of course such a model is only valid in a neighbourhood of the setpoint.

```
const -- constant declarations
  SAMPLING_FREQ: 100.0; -- Inverse of the sampling time
                        -- (Hz)
  MAX_U: 200.0;  -- Max user demand value (kW)
  MAX_D_U: 10.0; -- Max of time derivative of user demand
  MAX_D_P: 0.1;  -- Max of time derivative of compressor
                 -- pression
  MAX_PRES_COMPR: 13.0; -- Max compressor pression (bar)
  MIN_PRES_COMPR: 11.0; -- Min compressor pression (bar)
  Power_setpnt: 2000.0; -- Setpoint of Electric Power
                        -- (kW)
  Texh_setpnt: 552; -- Setpoint of exhaust smokes
                    -- temperature (C)
  Vrot_setpnt: 75;  -- Setpoint of rotation speed (RPM)
  Pow_v_coef: Power_setpnt;     -- $\alpha_{1,1}$ in Fig. 27
  Texh_v_coef: 0.1*Texh_setpnt; -- $\alpha_{2,1}$ in Fig. 27
  Vrot_v_coef: 2*Vrot_setpnt;   -- $\alpha_{3,1}$ in Fig. 27
  FREQ_1: 100; -- frequency injection disturbances

type -- type declarations
  Disturbance_type : -1..1;
  real_type : real(4, 2); -- used for all real variables
  longint_type : -50000 .. +50000;  -- used for counters

var -- (global) variable declarations
  step_counter : longint_type; -- initialized to 0
    -- We do: step_counter := (step_counter + 1)  FREQ_1-- at each time
     stepPower :  real_type; -- Generated Electric Power
```

Figure 28: A glimpse of the CGMurphi declarations used in the Turbogas controller model.

Note that, according to the model in Fig. 27, the compressor pressure P_{mc} can change value *nondeterministically* as long as it satisfies the constraints given in Fig. 27. We do not need a more detailed model here since the compressor pressure is only used as input to the fuel gas valve controller whose requirements do not involve the compressor pressure.

Finally, the plant setpoint, that is the set of goal states of our plan, is given by the following values of the output variables:

- Electric Power setpoint value: P^0_{el}=2000 (KW).
- Exhaust Smokes Temperature setpoint value: T^0_{exh}=552 (C).
- Turbine Rotation Speed setpoint value: V^0_{rot}=75 (RPM)
- Compressor Pressure setpoint value: P^0_{mc}=12 (Bar)

7.3.2 CGMurphi Model Description

In order to use CGMurphi, we discretise the ODEs given above with sampling time 0.01 seconds, as suggested in [19], and truncating real valued variables to 3 digits of mantissa and 2 of exponent. An example of the CGMurphi code used in the declaration section of our model is in Fig. 28.

Note that in this way the discretised state space consists of approximately 2^{180} states. This rules out all brute-force methods consisting in explicitly enumerating the state space.

7.3.3 Optimal Controller Generation

Reachable States	Transitions	Controller Rules	Time (sec.)	Used Memory	Controller Size
11,240	112,400	11,240	141	10,115 Kb	549 Kb

Table 8: Turbogas controller synthesis results

Table 8 shows the results of the controller synthesis. As we can see, CGMurphi was able to complete the controller generation with 10 Megabytes of RAM, since the system reachable states are indeed only $11,240$, compared to the 2^{180} states resulting from the model specification. We are still working on this system, and we expect that the controller validation will show that the controller needs to be strengthened by adding more states to the reachable region. However, this first result is very promising and shows how reachability analysis can help in the generation of controllers for complex systems.

8 Conclusions

In this paper we presented CGMurphi, an automatic tool for the generation of numerical controllers. The tool exploits explicit model checking techniques, in particular the reachability analysis algorithm, to sensibly reduce the efforts needed to analyse the dynamics of very complex systems, such as nonlinear and hybrid systems, which are often out of scope for the current controller generation tools.

Moreover, CGMurphi is also able to apply a suitably adapted Dijkstra algorithm during the system state space exploration, to compute *optimal* controllers, and to make them robust thanks to an iterative *strengthening* algorithm which incrementally add new control actions to the controller table in order to handle unexpected system states due to disturbances.

Finally, since numerical controller tables may contain millions of state-rule pairs, CGMurphi includes a OBDD encoder for such tables, which generates a very compact still functional representation of the original controller, which can be in turn exported in C or VHDL language, to be easily embedded in software/hardware devices.

We extensively experimented our methodology on a variety of academic case studies, including the *inverted pendulum on a cart* and the *truck and trailer with obstacles avoidance*, but also on real industrial applications, like the *turbogas control system*, whose development is still in progress.

The experimental results show that CGMurphi is a very versatile product that can be used as a complete and effective controller generation tool for challenging complex systems.

References

[1] V.I. Arnol'd. *Ordinary Differential Equations*. Springer-Verlag, Berlin, 1992.

[2] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *Proc. System Structure and Control*, pages 469–474. IFAC, Elsevier, July 1998.

[3] Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems*, pages 1–20, 1994.

[4] Astrom K.J and Hagglung. T. *Advanced PID Control*. International Society for Measurement and Con; 2nd edition, 2005.

[5] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *CAV*, pages 121–125, 2007.

[6] D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 2005.

[7] A. Bobbio, E. Ciancamerla, G. Franceschinis, R. Gaeta, M. Minichino, and L. Portinale. Methods of increasing modelling power for safety analysis, applied to a turbine digital control system. In Stuart Anderson, Sandro Bologna, and Massimo Felici, editors, *Computer Safety, Reliability and Security, 21st International Conference, SAFECOMP 2002, Catania, Italy, September 10-13, 2002, Proceedings*, volume 2434 of *Lecture Notes in Computer Science*, pages 212–223. Springer, 2002.

[8] A. Bobbio, E. Ciancamerla, M. Gribaudo, A. Horvath, M. Minichino, and E. Tronci. Model checking based on fluid petri nets for the temperature control system of the icaro co-generative plant. In Stuart Anderson, Sandro Bologna, and Massimo Felici, editors, *Computer Safety, Reliability and Security, 21st International Conference, SAFECOMP 2002, Catania, Italy, September 10-13, 2002, Proceedings*, volume 2434 of *Lecture Notes in Computer Science*, pages 273–283. Springer, 2002.

[9] A. Bobbio, S.Bologna, M. Minichino, E. Ciancamerla, P.Incalcaterra, C.Kropp, and E. Tronci. Advanced techniques for safety analysis applied to the gas turbine control system of icaro co generative plant. In *Proc. of X Convegno TESEC*, Genova, Italy, June 2001.

[10] F. Borrelli, M. Baotic, A. Bemporad, and M. Morari. Dynamic programming for constrained optimal control of discrete-time linear hybrid systems. *Automatica*, 41:1709–1721, October 2005.

[11] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug 1986.

[12] Cached Murphi Web Page. http://www.di.univaq.it/gdellape/murphi/cmurphi.php.

[13] CGMurphi Web Page. http://www.di.univaq.it/magazzeni/cgmurphi.php.

[14] E. A. Coddington and N. Levinson. *Theory of Ordinary Differential Equations*. McGrawHill, 1955.

[15] CUDD Web Page. http://vlsi.colorado.edu/∼fabio/.

[16] G. Della Penna, B. Intrigila, N. Lauri, and D. Magazzeni. Fast and compact encoding of numerical controllers using obdds. In *Informatics in Control, Automation and Robotics: Selected Papers from ICINCO 2008*, pages 75–87. Springer, 2009.

[17] G. Della Penna, D. Magazzeni, A. Tofani, B. Intrigila, I. Melatti, and E. Tronci. Automatic synthesis of robust numerical controllers. In *ICAS '07*, page 4. IEEE Computer Society, 2007.

[18] Giuseppe Della Penna, Benedetto Intrigila, and Daniele Magazzeni. Synthesis of optimal control systems: a comparison between model checking and dynamic programming techniques. In Springer-Verlag, editor, *Proceedings of International Conference on Industrial Electronics, Technology and Automation*, pages 54–59, 12 2007.

[19] ENEA. *Proprietary ICARO Documentation*.

[20] GCC Compiler. http://gcc.gnu.org/.

[21] R. Goldman, D. Musliner, and M. Pelican. Incremental verification for on-the-fly controller synthesis. In *In Proceeding of the Third Workshop on Model Checking and Artificial Intelligence (MoChArt05).*, 2005.

[22] Wassim M. Haddad and Vijaysekhar Chellaboina. *Nonlinear Dynamical Systems and Control: A Lyapunov-Based Approach*. Princeton University Press, 2008.

[23] Thomas A. Henzinger. The theory of hybrid automata. In *LICS'96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1996.

[24] C. S. Hsu and R. S. Guttalu. An unravelling algorithm for global analysis of dynamical systems - An application of cell-to-cell mappings. *ASME Transactions Series E Journal of Applied Mechanics*, 47:940–948, December 1980.

[25] C.S. Hsu. A discrete method of optimal control based upon the cell state space concept. *J. Optim. Theory Appl.*, 1985.

[26] C.S. Hsu. *Cell-to-Cell Mapping*. Springer-Verlag, 1987.

[27] O. Junge and H. Osinga. A set oriented approach to global optimal control. *ESAIM Control Optim. Calc. Var., 10(2):259–270 (electronic), 2004.*, 2004.

[28] Henry Kautz, Wolfgang Thomas, and Moshe Y. Vardi. 05241 executive summary – synthesis and planning. In Henry Kautz, Wolfgang Thomas, and Moshe Y. Vardi, editors, *Synthesis and Planning*, number 05241 in Dagstuhl Seminar Proceedings, 2006.

[29] Kaynar D.K., Lynch N., Segala R., and Vaandrager F. *The Theory of Timed I/O Automata*. Morgan & Claypool Publishers, 2006.

[30] G. Kreisselmeier and T. Birkholzer. Numerical nonlinear regulator design. *IEEE Transactions on Automatic Control*, 39(1):33–46, jan 1994.

[31] W.S. Levine. *The Control Handbook*. Jaico Publishing House, 2005.

[32] Feng Lin. *Robust Control Design: An Optimal Control Approach*. Wiley-Interscience, 2007.

[33] Jan Lunze and Franoise Lamnabhi-Lagarrigue. *Handbook of Hybrid Systems Control*. Cambridge University Press, 2009.

[34] Oded Maler, Dejan Nickovic, and Amir Pnueli. On synthesizing controllers from bounded-response properties. In *CAV*, pages 95–107, 2007.

[35] David J. Musliner, Edmund H. Durfee, and Kang G. Shin. CIRCA: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23:1561–1574, 1993.

[36] D. Nguyen and B. Widrow. The truck backer-upper: an example of self learning in neural networks. In *In Proceeding of IJCNN.*, volume 2, pages 357–363, 1989.

[37] M. Papa, H. Tai, and S. Shenoi. Cell mapping for controller design and evaluation. *IEEE Control Systems*, 17(2):52–65, 1997.

[38] M. Papa, J. Wood, and S. Shenoi. Evaluating controller robustness using cell mapping. *Fuzzy Sets and Systems*, 121(1):3–12, 2001.

[39] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.

[40] E. D. Sontag. Nonlinear regulation: The piecewise linear approach. *IEEE Trans. Autom. Control AC*, 26(2):346–358, 1981.

[41] E. D. Sontag. Remarks on piecewise-linear algebra. *Pacific Journal of Mathematics*, 98(1):183–201, 1982.

[42] E. D. Sontag. From linear to nonlinear: some complexity comparisons. In *Proc. of IEEE Conference on Decision and Control*, pages 2916–2920, 1995.

[43] Paulo Tabuada. *Verification and Control of Hybrid Systems*. Springer, 2009.

[44] Claire Tomlin, John Lygeros, and Shankar Sastry. Synthesizing controllers for nonlinear hybrid systems. In *HSCC*, pages 360–373, 1998.

[45] Stavros Tripakis and Karine Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In *World Congress on Formal Methods (1)*, pages 233–252, 1999.

[46] E. Tronci. Automatic synthesis of controllers from formal specifications. In *ICFEM '98: Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, page 134, Washington, DC, USA, 1998. IEEE Computer Society.

[47] UPPAAL TIGA web page. http://www.cs.aau.dk/ adavid/tiga/.

[48] K. Weihrauch. *Computable Analysis: An Introduction*. Texts in Theoretical Computer Science. Springer, 2000.