Controller Synthesis for Autonomous Systems: a Constraint-Based Approach

Cédric Pralet, Michel Lemaître, Gérard Verfaillie, Guillaume Infantes

ONERA, Toulouse, France
e-mail: firstname.lastname@onera.fr

Abstract

Controller synthesis can be seen as an alternative to controller verification, in which controllers are directly synthesized from a model of the possible evolutions of a system to be controlled and from properties to be satisfied. This paper presents the application of controller synthesis to observation satellites, and more precisely to two case studies: the control of a digital signal processor, and the control of connexions between detector lines of an observation instrument and the mass memory. The approach used is model-based, and models are defined using the flexibility of constraint programming languages.

1 Introduction

In the very large sense, a controller is a tool able to react continuously to every possible situation in its domain of control using appropriate actions. More precisely, a controller takes as input observation data from the system it controls and returns as outputs actions supposed to guarantee that some desired properties will always be satisfied. Such properties can be for example safety properties or liveness properties. The former express that some condition on the system state must always hold. The latter express that some condition must hold at a future step.

A standard approach to the definition of controllers is a two-step procedure in which, first, the complete behavior of the controller is specified (e.g. using deterministic automata), and second, the desired properties are checked. If property checking fails, the initial specification is revised, properties are checked again, and so on, until a valid controller is found. An alternative way of building valid controllers is controller synthesis, a one-step procedure in which a full controller is directly synthesized from a declarative model representing the possible evolutions of the system to be controlled and from the set of properties to be satisfied. Such a synthesis can be difficult to achieve, but the resulting controller is directly valid by construction.

Several proposals for controller synthesis have been developed recently [1, 2, 9, 11, 10]. This paper presents the application of one these approaches [10] to observation satellites. More precisely, we show how controller synthesis can be applied to two case studies: the control of a Digital Signal Processor (DSP), and the management of connexions between an observation instrument and the mass memory. The framework and methodology described are however not reduced to observation satellites and can be applied to autonomous systems in general.

The main features of the model-based framework and algorithm used are:

- *a constraint-based nature*, in the sense that the desired properties and models of systems to be controlled are expressed using a constraint programming language. This constraint-based nature has two mains advantages: first, a greater flexibility for modelers (*e.g.* integer-valued attributes and complex constraints can be used), and second, the potential reuse of efficient constraint programming techniques;
- *the ability to handle non determinism* in the evolution of the system controlled;
- *the ability to handle partial observability* of the real state of the system controlled (some attributes of the state may remain unobserved);
- *the ability to synthesize finite-state controllers*: the controllers built are allowed to maintain a kind of internal memory, similar to the current state in a finite-state automaton; such finite-state controllers are more general than memoryless controllers mapping observations to controller outputs.

The paper is organized as follows: we first present the constraint-based framework used (Section 2), and then apply it to the two case studies mentioned earlier: the DSP (Section 3) and the controller of connexions (Section 4). We conclude with some comparisons and perspectives (Section 5).

2 Controller synthesis framework

Control models This section recalls the main definitions presented in [10]. We model the attributes of the controlled system using a finite set S of variables. This set is partitioned into a set O of observable variables and a set $S \setminus O$ of unobservable variables. The outputs of the controller are similarly modeled by a finite set C of variables. All variables $v \in S \cup C$ are assumed to have a finite domain of values denoted $\mathbf{d}(v)$. Given a set of variables V,

$\mathbf{d}(V)$ denotes the Cartesian product of the domains of the variables in V (assuming an order on the variables in V so that this Cartesian product is well defined). Each element $s \in \mathbf{d}(S)$ (resp. $o \in \mathbf{d}(O)$, $c \in \mathbf{d}(C)$), is called a *state* (resp. an *observation*, a *control*). Given a state s, $\mathbf{o}(s)$ denotes the assignment of the variables of O in s, that is the observation associated with s.

We consider a framework involving non-determinism both in the initial state and in the possible effects of the controller outputs. This non-determinism is modeled by two relations: an *initialization* relation I, which contains assignments s corresponding to possible initial states, and a *transition* relation T, which contains triples (s, c, s') such that s' is a possible successor of s when control c is performed.

Last, preconditions can be imposed on controller outputs. They can model either physical limitations, or partial specifications of the controller, to be completed by the synthesis task. They are modeled using a *feasibility* relation F, which contains pairs (s, c) such that control c is feasible in state s. The only assumption made is that a feasible control cannot block the evolution of the system: $\forall s \in \mathbf{d}(S)$, $\forall c \in \mathbf{d}(C)$, $F(s, c) \rightarrow (\exists s' \in \mathbf{d}(S),\ T(s, c, s'))$.

In the models developed, relations I, T, and F will be expressed as sets of constraints, in order to obtain compact and efficient models. All previous elements are gathered in the notion of *control model*.

Definition 1 *A* control model *is a tuple* (S, O, C, I, T, F) *such that* S *is a finite set of finite-domain variables called* state variables*;* $O \subset S$ *is a set of* observable state variables*;* C *is a finite set of finite-domain variables called* control variables*;* $I \subset \mathbf{d}(S)$ *is the* initialization relation*;* $T \subset \mathbf{d}(S) \times \mathbf{d}(C) \times \mathbf{d}(S)$ *is the* transition relation*;* $F \subset \mathbf{d}(S) \times \mathbf{d}(C)$ *is the* feasibility relation*;* $\forall s \in \mathbf{d}(S)$, $\forall c \in \mathbf{d}(C)$, $F(s, c) \rightarrow (\exists s' \in \mathbf{d}(S),\ T(s, c, s'))$.

We then define a decision policy Π for a control model as a memoryless controller, mapping the last observation made $o \in \mathbf{d}(O)$ to a control $c \in \mathbf{d}(C)$. $\Pi(o) = c$ means that the output of the controller is c when observation o is made. A policy may be *partial* in the sense that $\Pi(o)$ can be undefined for some $o \in \mathbf{d}(O)$. Partial policies are useful to define the controller behavior only on the set of reachable states of the system. We are also interested in applicable policies that specify only feasible decisions. These elements are formalized below.

Definition 2 *A* policy *for a control model* (S, O, C, I, T, F) *is a partial function* $\Pi : \mathbf{d}(O) \rightarrow \mathbf{d}(C)$. *The domain of* Π *is* $\mathbf{d}(\Pi) = \{o \in \mathbf{d}(O) \,|\, \Pi(o) \text{ defined}\}$.

Definition 3 *Let* Π *be a policy for a control model* (S, O, C, I, T, F). *A* trajectory *induced by* Π *is a sequence* $[s_0, \ldots s_n]$ *such that (i)* $I(s_0)$ *holds, (ii) for all* $i \in [1..n]$, $\mathbf{o}(s_{i-1}) \in \mathbf{d}(\Pi)$ *and* $T(s_{i-1}, \Pi(\mathbf{o}(s_{i-1})), s_i)$ *hold, and (iii)* $\mathbf{o}(s_n) \notin \mathbf{d}(\Pi)$ *if* $n < +\infty$. *When* $n < +\infty$ *(resp.* $n = +\infty$*), the trajectory is said to be finite (resp. infinite). The set of* reachable states *induced by* Π *is the set of states appearing in at least one trajectory induced by* Π.

Definition 4 *A policy* Π *for a control model* (S, O, C, I, T, F) *is said to be* applicable *if and only if for every reachable state* s *induced by* Π, $(\mathbf{o}(s) \in \mathbf{d}(\Pi)) \rightarrow F(s, \Pi(\mathbf{o}(s)))$.

Policies introduced in Definition 2 are memoryless since all past observations, but the current one, are not considered to determine a controller output. An opposite approach considers belief-state based policies whose outputs at each step may depend on all past observations [3]. An intermediate approach [4] is finite-state controllers. Such controllers maintain an internal state number $q \in [1..N]$, with N a fixed integer. They are defined by mappings $(o, q) \rightarrow (c, q')$ expressing that when the controller makes observation o and is in internal state q, it outputs control c and changes its internal state to q'. Thanks to internal memory $q \in [1..N]$, the controller may record some features (but not necessarily all) concerning past observations, and some problems which do not admit memoryless controllers admit finite-state controllers. As shown in [10], the problem of building a finite-state controller having N internal states can be cast into the problem of building a memoryless controller, by adding new state and control variables. See [10] for more details.

Control problems Given a control model, several requirements can be imposed on the possible evolutions of the controlled system. The objective can be to find an applicable policy so that all trajectories terminate in a goal state (*goal-oriented control problem*), to find an applicable policy so that some properties are satisfied at each step over an infinite horizon (*safety-oriented control problem*), or to find an applicable policy so that all trajectories both terminate in a goal state and satisfy some properties at each step (*goal and safety-oriented control problem*). These problems are defined below. Other control problems could be considered, such as problems involving liveness properties [6].

Definition 5 *A* goal-oriented control problem *is a pair* $Pb = (M, P_G)$ *with* M *a control model and* $P_G \subset \mathbf{d}(S)$ *a relation called goal relation. A solution to problem* Pb *is an applicable policy* Π *for* M *such that all trajectories* $[s_0, \ldots, s_n]$ *induced by* Π *are finite and verify* $P_G(s_n)$.

A safety-oriented control problem *is a pair* $Pb = (M, P_S)$ *with* M *a control model and* $P_S \subset \mathbf{d}(S)$ *a relation called safety relation. A solution to* Pb *is an applicable policy* Π *for* M *such that all trajectories induced by* Π *are infinite and all states they involve satisfy* P_S.

A goal and safety-oriented control problem *is a triple $Pb = (M, P_G, P_S)$ with M a control model, $P_G \subset \mathbf{d}(S)$ the goal relation, and $P_S \subset \mathbf{d}(S)$ the safety relation. A solution to Pb is an applicable policy Π for M such that all trajectories $[s_0, \ldots, s_n]$ induced by Π are finite, verify $P_G(s_n)$, and all states they involve satisfy P_S.*

Resolution techniques The *Simulate and Branch* (SB) strategy proposed in [10] to solve these problems consists, given a current controller, in exploring the set of reachable states of the system using this controller (simulation phase) and in extending the controller when an uncovered state is encountered (branching phase). Backtracks on branching decisions are also made when an inconsistency is revealed. Other techniques such as recording, conflict-based backjumping, and generic heuristics are also used to improve search efficiency. The SB procedure is implemented in a tool called *Dyncode*, based on the *Gecode* constraint programming library [8].

Methodology The case studies considered thereafter correspond to the synthesis of controllers for modules responsible for handling a subsystem of a satellite. To do this, we first identify control and state variables (C and S), as well as the observable part of the system (variables in $O \subset S$). We then judge whether internal memory for the controller could help in controlling the system, in which cases these internal memory variables are added in S, together with special control variables in C allowing these memory variables to be completely controlled by the controller.

We then identify the "causal" dependencies between the different variables, by building a *dependency graph* in the form given in Figure 1. This figure shows that controller outputs belong to three categories: (1) outputs in direction of the controlled system (commands), responsible for driving the system towards appropriate configurations, (2) outputs in direction of higher-level modules (diagnoses on the current state of the subsystem), and (3) outputs updating the internal memory of the controller, which allow features concerning past observations or past commands to be recorded. In Figure 1, the evolution of the state of the system to be controlled depends on the commands it receives and on the state at the previous step, modeled by a special box labeled with `pre` (relation transition $T(S, C, S')$, expressed here in the equivalent form $T(\texttt{pre}(S), C, S)$). The outputs of the controller depend on the last state of the internal memory and on the last observations performed on the system.

Once qualitative dependencies have been identified, the initialization relation (I), the transition relation (T), the feasibility relation (F), and the safety/goal relation (P_S/P_G) are successively defined, as sets of constraints. In the following, we use this methodology and solver *Dyncode* for two case studies in the space domain.

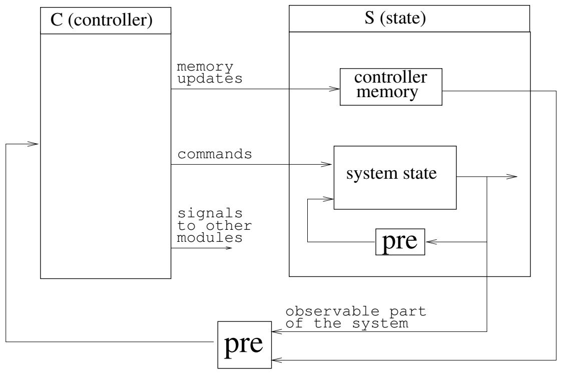

Figure 1. Generic dependency graph

3 Case #1: Digital Signal Processor (DSP)

We consider an Earth observing satellite whose mission is to detect hot points (forest fires, volcanic eruptions...) at the Earth surface. To do this, the satellite is equipped with a wide swath detection instrument and with a component called Digital Signal Processor (DSP), able to analyze images and detect whether they contain hot spots. The DSP is composed of three elements: (1) an analyzer, which analyzes received images, (2) a circuit, which provides current to the analyzer, and (3) a switch, which behaves like a push-button and allows tension to be present in the circuit. The DSP also receives requests to be switched on or off from higher-level modules.

At each step, the controller (or policy) of the DSP must return several outputs: a signal indicating whether a hot spot has been detected, signals indicating whether the DSP is correctly running or not (fault detection), and a command on the switch of the DSP.

Faults may occur on each element of the DSP. If the analyzer is not faulty and if it receives current, then the detection of hot spots is correct. If the analyzer is faulty, then it detects neither the presence nor the absence of hot spots. If the circuit is faulty, then the analyzer receives no current and does not run. If the switch is faulty (switch blocked in the on or off position), then the tension in the circuit does not necessarily reflect the tension command applied via the switch.

The safety property to be satisfied is that, at each step, if no fault is present, the DSP must return a correct analysis if it has been switched on, and no analysis if it has been switched off.

3.1 Control model and control problem

State variables (S) In order to model this problem, we use the following state variables:

$\texttt{switchOn} \in \{0,1\}$: switch on request on the DSP from a higher-level module (impulse command);

$\texttt{switchOff} \in \{0,1\}$: switch off request on the DSP from a higher-level module (impulse command);

$\texttt{switched} \in \{0,1\}$: last request received from a higher-level module (internal memory); value 0 if the last request is to switch the DSP off, value 1 otherwise;

$\texttt{tension} \in \{0,1\}$: presence of tension in the circuit;

$\texttt{current} \in \{0,1\}$: presence of current in the circuit;

$\texttt{analyzIm} \in \{\texttt{NULL}, \texttt{NORMAL}, \texttt{HOTSPOT}\}$: analysis performed by the analyzer (value HOTSPOT if a hot spot is detected, value NORMAL if no hot spot is detected, value NULL otherwise);

$\texttt{faultSwitch} \in \{0,1\}$: fault on the switch;

$\texttt{faultCircuit} \in \{0,1\}$: fault on the circuit;

$\texttt{faultAnalyzer} \in \{0,1\}$: fault on the analyzer;

$\texttt{receivIm} \in \{\texttt{NORMAL}, \texttt{HOTSPOT}\}$: content of the image received (value HOTSPOT if it contains a hot spot, value NORMAL if it does not).

Variables switchOn, switchOff, switched, tension, current, and analyzIm are observable (variables in $O \subset S$). Other variables are not: faults are not directly detected and the content of received images is not directly known.

Control variables (C) The outputs of the controller correspond to updates of the controller internal memory, to commands sent to the system, and to signals giving the status of the DSP and hot spot detections:

$\texttt{switch} \in \{0,1\}$: command to the internal memory available via state variable switched;

$\texttt{cmdTension} \in \{0,1\}$: command in tension applied via the switch of the DSP;

$\texttt{running} \in \{0,1\}$: DSP running status, sent to the higher-level module (value 1 if the DSP is switched on and runs correctly, value 0 otherwise;

$\texttt{failing} \in \{0,1\}$: DSP failing status, sent to the higher-level module (value 1 if it does not run correctly, value 0 otherwise);

$\texttt{hotSpot} \in \{0,1\}$: detection of a hot spot.

Dependency graph The second modeling step consists in defining the *dependency graph* (Figure 2), which gives a graphical view of the way the previous variables influence each other. This graph looks like representations used in the SCADE graphical language [7]. The differences are that the controller part is unknown initially, and that the dependence between the input and the output of a box is relational (not necessarily functional), that is an input can induce several outputs. In Figure 2, special boxes labeled with pre allow the value of a state variable at a previous step to be accessed. For instance, Figure 2 expresses that the tension at the next step depends on the tension at the previous step, on the command in tension applied via the switch, and on the presence of a fault on the switch.

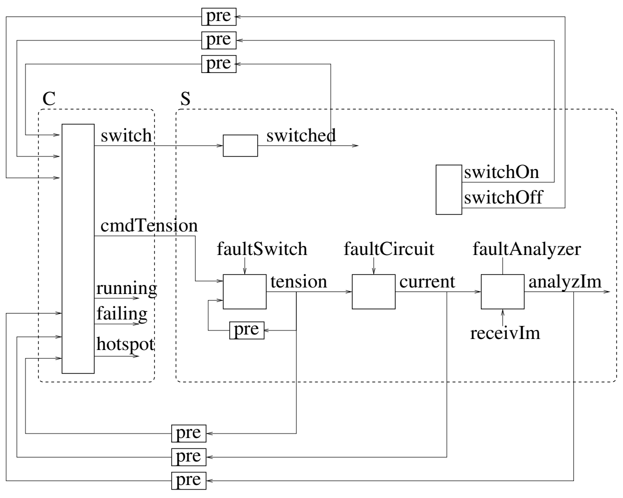

Figure 2. DSP control problem: dependency graph

Initialization relation (I) The initial values of the different state variables are constrained as follows:

$$\neg(\texttt{switchOn} \wedge \texttt{switchOff}) \tag{1}$$

$$\neg\texttt{switched},\ \neg\texttt{tension},\ \neg\texttt{current},\ \texttt{analyzIm} = \texttt{NULL} \tag{2}$$

These constraints mean that initially, signals switchOn and switchOff cannot be true simultaneously and the DSP is off (no tension, no current, no image analyzed).

Transition relation (T) Transition relation T must define, for every "box" B in Figure 2, the possible outputs of B given the inputs of B. For instance, it must specify the possible values of state variable analyzIm given the values of receivIm, current, and faultAnalyzer. Transition relation T is expressed by the following set of constraints:

switchOn and switchOff :

$$\neg(\texttt{switchOn} \wedge \texttt{switchOff}) \tag{3}$$

switched :

$$\texttt{switched} = \texttt{switch} \tag{4}$$

tension :

$$\neg\texttt{faultSwitch} \rightarrow (\texttt{tension} = \texttt{cmdTension}) \tag{5}$$

$$(\texttt{pre}(\texttt{tension}) \wedge \texttt{cmdTension}) \rightarrow \texttt{tension} \tag{6}$$

$$(\neg\texttt{pre}(\texttt{tension}) \wedge \neg\texttt{cmdTension}) \rightarrow \neg\texttt{tension} \tag{7}$$

current :

$$\texttt{current} \equiv (\texttt{tension} \wedge \neg\texttt{faultCircuit}) \tag{8}$$

$\mathtt{analyzIm}$:

$$\neg\mathtt{current} \rightarrow (\mathtt{analyzIm} = \mathtt{NULL}) \quad (9)$$

$$\mathtt{faultAnalyzer} \rightarrow (\mathtt{analyzIm} = \mathtt{NULL}) \quad (10)$$

$$(\mathtt{current} \wedge \neg\mathtt{faultAnalyzer}) \rightarrow (\mathtt{analyzIm} = \mathtt{receivIm}) \quad (11)$$

Constraint 3 expresses that `switchOn` and `switchOff` cannot be received simultaneously (kind of correlation between `switchOn` and `switchOff`). Constraint 4 expresses that state variable `switched` is directly controlled by command `switch`. State variable `switched` acts as an internal memory allowing the controller to record whether the last switch command received is a switch on or a switch off. Constraint 5 expresses that if there is not fault on the switch, then the tension in the circuit equals the command in tension applied via the switch. Constraint 6 (resp. 7) expresses that if there was tension (resp. no tension) at the previous step and a tension command (resp. no command) on the switch, then there is still tension (resp. no tension) at the next step. In other cases, all values of state variable `tension` are possible. Constraint 8 expresses that there is current if and only if there is tension and no fault in the circuit. Last, constraints 9 to 11 specify the evolution of `analyzIm`.

Feasibility of controller outputs (F) In this context, the following feasibility constraints can be imposed. They act as a partial initial specification of the controller, imposing or forbidding some outputs in some given states:

$\mathtt{switch}$:

$$\mathtt{pre(switchOn)} \rightarrow \mathtt{switch} \quad (12)$$

$$\mathtt{pre(switchOff)} \rightarrow \neg\mathtt{switch} \quad (13)$$

$$(\neg\mathtt{pre(switchOn)} \wedge \neg\mathtt{pre(switchOff)}) \rightarrow (\mathtt{switch} = \mathtt{pre(switched)}) \quad (14)$$

$\mathtt{cmdTension}$:

`no feasibility constraint`

$\mathtt{failing}$ and $\mathtt{running}$:

$$\mathtt{running} \equiv (\mathtt{pre(switched)} \wedge \mathtt{pre(tension)} \wedge \mathtt{pre(current)} \wedge (\mathtt{pre(analyzIm)} = \mathtt{pre(receivIm)})) \quad (15)$$

$$\mathtt{failing} \equiv ((\mathtt{pre(tension)} \neq \mathtt{pre(switched)}) \vee (\mathtt{pre(current)} \neq \mathtt{pre(tension)}) \vee (\mathtt{pre(switched)} \wedge (\mathtt{pre(analyzIm)} = \mathtt{NULL})) \quad (16)$$

$$\neg(\mathtt{running} \wedge \mathtt{failing}) \quad (17)$$

$$\mathtt{pre(switched)} \rightarrow (\mathtt{running} \vee \mathtt{failing}) \quad (18)$$

$$(\mathtt{pre(switched)} \wedge \neg\mathtt{pre(faultSwitch)} \wedge \neg\mathtt{pre(faultCircuit)} \wedge \neg\mathtt{pre(faultAnalyzer)}) \rightarrow \mathtt{running} \quad (19)$$

$$\mathtt{failing} \rightarrow (\mathtt{pre(faultSwitch)} \vee \mathtt{pre(faultCircuit)} \vee \mathtt{pre(faultAnalyzer)}) \quad (20)$$

$\mathtt{hotSpot}$:

$$\mathtt{hotSpot} \equiv (\mathtt{running} \wedge \mathtt{pre(analyzIm)} = \mathtt{HOTSPOT}) \quad (21)$$

$$(\mathtt{pre(receivIm)} = \mathtt{HOTSPOT} \wedge \mathtt{running}) \rightarrow \mathtt{hotSpot} \quad (22)$$

$$\mathtt{hotSpot} \rightarrow (\mathtt{pre(receivIm)} = \mathtt{HOTSPOT}) \quad (23)$$

Constraints 12 to 14 together provide a functional definition of output `switch`. Constraints 15 and 16 give properties to be satisfied at each step by `running` and `failing`. Constraints 17 and 18 specify that `running` and `failing` cannot be true simultaneously and that if the DSP is switched on, they are not both false. Constraint 19 expresses that if the DSP should be on and no fault is present, then `running` must be true. Constraint 20 expresses that if `failing` equals true, then there necessarily exists a fault in the DSP.

Safety properties (P_S) The control problem defined is a safety-oriented control problem (M, P_S), with P_S (the safety relation) defined by the following set of constraints:

$$(\neg\mathtt{faultSwitch} \wedge \neg\mathtt{faultCircuit} \wedge \neg\mathtt{faultAnalyzer} \wedge \mathtt{switched}) \rightarrow (\mathtt{analyzIm} = \mathtt{receivIm}) \quad (24)$$

$$(\neg\mathtt{switched} \wedge \neg\mathtt{faultSwitch}) \rightarrow (\mathtt{analyzIm} = \mathtt{NULL}) \quad (25)$$

Constraint 24 expresses that if the DSP is on and no fault appears, then the analysis performed by the DSP must be correct. Constraint 25 expresses that if the DSP should be switched off and there is no fault on the switch, then no image is analyzed.

3.2 Controller synthesized

We ran our experiments on a Xeon processor 2GHz, 1GB RAM. Using variables and constraints defined previously, algorithm SB (Simulate and Branch) implemented in Dyncode synthesizes a solution controller instantaneously, in $120ms$. The controller produced is memoryless ($N = 0$) and induces 30 reachable observations (hence 30 mappings of the form $o \rightarrow c$). When written in a compact form as a Binary Decision Diagram (BDD [5]), it uses only 59 BDD nodes. On the solution found, it can be noticed that the controller always chooses `cmdTension = switch` (even if it could have been, this information was not given in the initial model).

4 Case #2: connexions control problem

The second case study considered concerns the control of connexions between an observation instrument and the mass memory of a satellite. As shown in Figure 3, the observation instrument is composed of a set of lines of detectors. Each detector line can be connected to some compressors (COMs), responsible for compressing data produced by detector lines. Each compressor can then be connected to a set of memory banks in the mass memory of the satellite. In order to be able to write data in the mass memory, a detector line must be connected to a COM which is itself connected to a memory bank. The objective is to build a controller allowing every detector line to write data in the mass memory, even in presence of a certain number of faulty COMs or in presence of a certain number of unavailable (full) memory banks. In other words, the controller to be synthesized must be able both to configure connexions initially and to reconfigure them in case of changes in equipment availability.

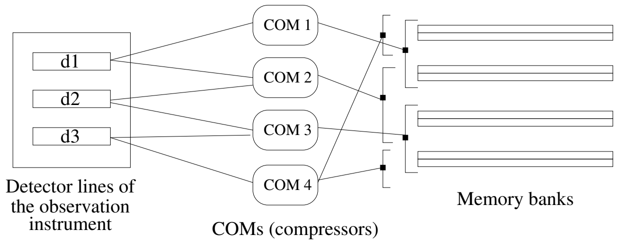

Figure 3. Connexions control problem

Data describing a specific physical architecture and failure cases considered can be defined as follows:

`ND` : number of detector lines in the instrument;

`NC` : number of COMs (memory compressors);

`NM` : number of memory banks;

`ConnectedDC` : detectors/COMs connexion matrix (`ConnectedDC[d,c]` = 1 if there exists a connection between detector line *d* and COM *c*, 0 otherwise);

`ConnectedCM` : COMs/memory banks connexion matrix (`ConnectedCM[c,m]` = 1 if there exists a connection between COM *c* and memory line *m*, 0 otherwise);

`MaxComFail` : maximum number of COMs failures;

`MaxMemUnavail` : maximum number of memory banks unavailable.

In the following, *d* (resp. *c*, *m*) denotes an arbitrary element in $[1..ND]$ (resp. $[1..NC]$, $[1..NM]$).

4.1 Control model and control problem

State variables (S) The state of the subsystem considered can be described using the following state variables:

$\texttt{dout[d]} \in [\texttt{0..NC}]$: index of the COM to which detector line *d* is connected (value 0 if the detector line is not connected to any COM);

$\texttt{cout[c]} \in [\texttt{0..NM}]$: index of the memory bank to which COM *c* is connected (value 0 if the COM is not connected to any memory bank);

$\texttt{cfail[c]} \in \{0,1\}$: presence of a failure on COM *c*;

$\texttt{mavail[m]} \in \{0,1\}$: availability of memory bank *m* (value 1 if data can be written in *m*, 0 otherwise);

$\texttt{newconf} \in \{0,1\}$: presence of a new configuration for the system considered (new fault on a COM or change in the availability status of a memory bank);

$\texttt{cwriting[c]} \in \{0,1\}$: capacity of COM *c* to write data in the mass memory in the current configuration;

$\texttt{dwriting[d]} \in \{0,1\}$: capacity of detector line *d* to write in memory in the current configuration;

$\texttt{running} \in \{0,1\}$: capacity for all detector lines to write data in the mass memory in the current configuration (valid behavior of the whole system).

We define the set of observable variables as $O = \{\texttt{dout[d]} \mid d \in [1..ND]\} \cup \{\texttt{cout[c]} \mid c \in [1..NC]\} \cup \{\texttt{cfail[c]} \mid c \in [1..NC]\} \cup \{\texttt{mavail[m]} \mid m \in [1..NM]\}$ (first four sets of variables). Note that the other variables could be considered as observable, since constraints imposed thereafter induce a functional dependency between these other variables and variables in O.

Control variables (C) Two kinds of commands can be used to configure/reconfigure connexions:

$\texttt{cmddout}[d] \in [-1..NC]$: connexion command on the output of detector line *d* (value -1 if no new connexion command, value 0 for a command of non-connexion, value $c \geq 1$ for connexion to COM *c*);

$\texttt{cmdcout}[c] \in [-1..NM]$: connexion command on the output of COM *c* (value -1 if no new connexion command, value 0 for a command of non-connexion, value $m \geq 1$ for connexion to memory bank *m*).

Dependency graph Figure 4 gives the dependencies between the previous state and control variables.

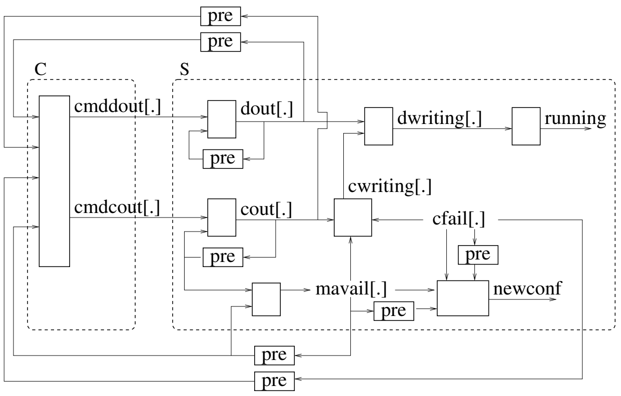

Figure 4. Connexions control: dependency graph

Initialization relation (I)

$$\texttt{newconf} \wedge \neg\texttt{running} \tag{26}$$

$$\forall \texttt{d} \in [1..\texttt{ND}],\ \texttt{dout[d]} = 0 \wedge \neg\texttt{dwriting[d]} \tag{27}$$

$$\forall \texttt{c} \in [1..\texttt{NC}],\ \texttt{cout[c]} = 0 \wedge \neg\texttt{cwriting[c]} \wedge \neg\texttt{cfail[c]} \tag{28}$$

$$\forall \texttt{m} \in [1..\texttt{NM}],\ \texttt{mavail[m]} \tag{29}$$

Transition relation (T) The constraints expressing the dependence between inputs and outputs of boxes in the state part of Figure 4 are defined below:

$$
\begin{aligned}
&\mathtt{dout}[.] : \forall \mathtt{d} \in [1..\mathtt{ND}] \\
&\qquad (\mathtt{cmddout[d]} \neq -1) \rightarrow (\mathtt{dout[d]} = \mathtt{cmddout[d]}) && (30)\\
&\qquad (\mathtt{cmddout[d]} = -1) \rightarrow (\mathtt{dout[d]} = \mathtt{pre(dout[d])}) && (31)\\
&\mathtt{cout}[.] : \forall \mathtt{c} \in [1..\mathtt{NC}] \\
&\qquad (\mathtt{cmdcout[c]} \neq -1) \rightarrow (\mathtt{cout[c]} = \mathtt{cmdcout[c]}) && (32)\\
&\qquad (\mathtt{cmdcout[c]} = -1) \rightarrow (\mathtt{cout[c]} = \mathtt{pre(cout[c])}) && (33)\\
&\mathtt{cwriting}[.] : \forall \mathtt{c} \in [1..\mathtt{NC}] \\
&\qquad \mathtt{cwriting[c]} \equiv (\neg \mathtt{cfail[c]} \wedge \mathtt{mavail[cout[c]]}) && (34)\\
&\qquad (\text{with convention } \mathtt{mavail}[0] = \textit{false}) \\
&\mathtt{dwriting}[.] : \forall \mathtt{d} \in [1..\mathtt{ND}] \\
&\qquad \mathtt{dwriting[d]} \equiv \mathtt{cwriting[dout[d]]} && (35)\\
&\qquad (\text{with convention } \mathtt{cwriting}[0] = \textit{false}) \\
&\mathtt{newconf} : \\
&\qquad \mathtt{newconf} \equiv ((\exists \mathtt{c} \in [1..\mathtt{NC}],\ \mathtt{cfail[c]} \neq \mathtt{pre(cfail[c])}) \\
&\qquad\qquad \vee (\exists \mathtt{m} \in [1..\mathtt{NM}],\ \mathtt{mavail[m]} \neq \mathtt{pre(mavail[m])})) && (36)\\
&\mathtt{cfail}[.] : \\
&\qquad \forall \mathtt{c} \in [1..\mathtt{NC}],\ \mathtt{pre(cfail[c])} \rightarrow \mathtt{cfail[c]} && (37)\\
&\qquad \mathtt{card}\{\mathtt{c} \in [1..\mathtt{NC}] \mid \mathtt{cfail[c]}\} \leq \mathtt{MaxComFail} && (38)\\
&\mathtt{mavail}[.] : \\
&\qquad \forall \mathtt{m} \in [1..\mathtt{NM}],\ (\mathtt{pre(mavail[m])} \wedge (\forall \mathtt{c} \in [1..\mathtt{NC}], \\
&\qquad\qquad \mathtt{pre(cout[c])} \neq \mathtt{m})) \rightarrow \mathtt{mavail[m]} && (39)\\
&\qquad \mathtt{card}\{\mathtt{m} \in [1..\mathtt{NM}] \mid \neg \mathtt{mavail[m]}\} \leq \mathtt{MaxMemUnavail} && (40)\\
&\mathtt{running} : \\
&\qquad \mathtt{running} \leftrightarrow (\forall \mathtt{d} \in [1..\mathtt{ND}],\ \mathtt{dwriting[d]}) && (41)
\end{aligned}
$$

Constraints 30 to 33 express that connexion commands succeed. Constraint 34 specifies that a COM can write in the mass memory if and only if it is not faulty and connected to an available memory bank. Constraint 35 specifies that a detector line can write is the mass memory if and only if it is connected to a COM that can write in the mass memory. Constraint 36 expresses that a new configuration appears in case of change in the failure status of a COM or in the availability status of a memory bank. A COM failure is permanent (constraint 37). The number of faulty COMs is less than `MaxComFail` (constraint 38). If a memory bank m is available at a given step and no COM is writing in m, then m is still available at the next step (constraint 39). Also, the number of memory banks unavailable is less than threshold `MaxMemUnavail` (constraint 40). Last, the system is running if and only if every detector line can write in the mass memory of the satellite (constraint 41).

Feasibility relation (F)

$$
\begin{aligned}
&\mathtt{pre(running)} \rightarrow (\forall \mathtt{d} \in [1..\mathtt{ND}],\ \mathtt{cmddout[d]} = -1) && (42)\\
&\mathtt{pre(running)} \rightarrow (\forall \mathtt{c} \in [1..\mathtt{NC}],\ \mathtt{cmdcout[c]} = -1) && (43)\\
&\forall \mathtt{c} \in [1..\mathtt{NC}],\ \mathtt{card}\{\mathtt{d} \in [1..\mathtt{ND}] \mid \mathtt{cmddout[d]} = \mathtt{c}\} \leq 1 && (44)\\
&\forall \mathtt{m} \in [1..\mathtt{NM}],\ \mathtt{card}\{\mathtt{c} \in [1..\mathtt{NC}] \mid \mathtt{cmdcout[c]} = \mathtt{m}\} \leq 1 && (45)\\
&\forall \mathtt{d} \in [1..\mathtt{ND}],\ \forall \mathtt{c} \in [1..\mathtt{NC}],\ (\mathtt{cmddout[d]} = \mathtt{c}) \\
&\qquad \rightarrow (\mathtt{ConnectedDC[d,c]} \wedge \neg \mathtt{pre(cfail[c])}) && (46)\\
&\forall \mathtt{c} \in [1..\mathtt{NC}],\ \forall \mathtt{m} \in [1..\mathtt{NM}],\ (\mathtt{cmdcout[c]} = \mathtt{m}) \\
&\qquad \rightarrow (\mathtt{ConnectedCM[c,m]} \wedge \mathtt{pre(mavail[m])}) && (47)\\
&\forall \mathtt{d} \in [1..\mathtt{ND}] \\
&\quad \mathtt{cmddout[d]} \neq \mathtt{0} && (48)\\
&\quad (\mathtt{cmddout[d]} = -1) \rightarrow (\forall \mathtt{c} \in [1..\mathtt{NC}], \\
&\qquad ((\mathtt{pre(dout[d])} = \mathtt{c}) \rightarrow \neg \mathtt{pre(cfail[c])})) && (49)\\
&\forall \mathtt{c} \in [1..\mathtt{NC}] \\
&\quad \forall \mathtt{d} \in [1..\mathtt{ND}],\ (\mathtt{cmddout[d]} = \mathtt{c}) \rightarrow (\mathtt{cmdcout[c]} \neq \mathtt{0} \\
&\qquad \wedge ((\mathtt{cmdcout[c]} = -1) \rightarrow (\mathtt{pre(cout[c])} \neq \mathtt{0}))) && (50)\\
&\quad (\mathtt{pre(cfail[c])} \wedge \mathtt{pre(cout[c])} \neq \mathtt{0}) \rightarrow (\mathtt{cmdcout[c]} = \mathtt{0}) && (51)\\
&\quad (\mathtt{pre(cfail[c])} \wedge \mathtt{pre(cout[c])} = \mathtt{0}) \rightarrow (\mathtt{cmdcout[c]} = -1) && (52)\\
&\quad (\mathtt{pre(cout[c])} \neq \mathtt{0} \wedge \mathtt{cmdcout[c]} = -1) \rightarrow \mathtt{pre(cwriting[c])} && (53)
\end{aligned}
$$

Constraints 42 and 43 impose that no reconfiguration must be made when the system is running correctly. Constraints 44 and 45 express that it is forbidden to produce connexion commands inducing several uses of the same resource. Constraints 46 and 47 express that connexions must be consistent with physical links available in the architecture and that connexions must not be made towards unavailable equipments. Constraints 48 and 49 respectively forbid a connexion command for a detector line to be null and restrict cases where it can equal -1. Constraint 50 enforces that a COM used by a detector line must be connected to the memory. Constraints 51 and 52 ensure that a faulty COM is not connected to the memory at the next step. Last, constraint 53 restricts cases in which the connexion command sent to a COM can equal -1.

Safety properties (P_S)

$$
\begin{aligned}
&\forall \mathtt{c} \in [1..\mathtt{NC}],\ \mathtt{card}\{\mathtt{d} \in [1..\mathtt{ND}] \mid \mathtt{dout[d]} = \mathtt{c}\} \leq 1 && (54)\\
&\forall \mathtt{m} \in [1..\mathtt{NM}],\ \mathtt{card}\{\mathtt{c} \in [1..\mathtt{NC}] \mid \mathtt{cout[c]} = \mathtt{m}\} \leq 1 && (55)\\
&\forall \mathtt{c} \in [1..\mathtt{NC}],\ (\forall \mathtt{d} \in [1..\mathtt{ND}],\ \mathtt{dout[d]} \neq \mathtt{c}) \rightarrow (\mathtt{cout[c]} = \mathtt{0}) && (56)\\
&\neg \mathtt{newconf} \rightarrow \mathtt{running} && (57)
\end{aligned}
$$

Constraint 54 specifies that two detector lines cannot be connected to the same COM. Constraint 55 is similar. Constraint 56 imposes that an unused COM must not be connected to any memory bank. The main safety constraint to be satisfied is constraint 57: it specifies that at any step, if no new configuration appears, then the whole system must be running correctly.

4.2 Controller synthesized

We consider the instance shown in Figure 3 ($\mathtt{ND} = 3$, $\mathtt{NC} = 4$, $\mathtt{NM} = 8$, and matrices `ConnectedDC` and `ConnectedCM` specified as in Figure 3). Table 1 gives the results obtained with Dyncode for `MaxComFail` = 1 (at most one COM failure) and for different values of `MaxMemUnavail`. These results show that the time necessary to build a controller, which is paid only offline, highly depends on MaxMemUnavail. This result is due to a combinatorial explosion of the number of possible scenarios when MaxMemUnavail increases. Controller synthesis can therefore handle situations involving a certain number of failures. A possible way to overcome this limitation could be to use the controllers produced offline for reactive reconfiguration in "standard failure cases", and to use a dedicated online algorithm in case of numerous failures. In terms of memory size, the controllers produced are quite compact (third column).

MaxMemUnavail	CPU time (s)	Controller size (nb BDD nodes)	NbObs
0	0.04	336	9
1	0.30	1696	276
2	4.22	3751	2699
3	48.20	7519	16079
4	$\geq$ 3h	-	-

Table 1. Results for the connexions control problem (NbObs: number of observations corresponding to reachable states induced by the controller)

5 Related work and perspectives

This paper presented a methodology and the resolution of two case studies for controller synthesis in the context of autonomous systems.

In terms of kinds of requirements imposed, the framework used is able to handle safety properties and goal reachability properties. Some existing controller synthesis frameworks can deal with more general properties. For example, Anzu and RATSY [9, 11] can deal with a subpart of LTL called *Generalize Reactive(1)* design. It should however be stressed that Anzu and RATSY only consider completely observable systems, whereas the approach we use can deal with partial observability. Also, compared to UPPAL-TiGA [2], the approach proposed is more expressive on some points, but not yet appropriate to efficiently handle temporal aspects. Time could be modeled by adding temporal variables/clocks in the set S of state variables, but the synthesis algorithm used in Dyncode would then probably be less efficient than dedicated techniques on timed-automata, such as the use of the *region automata*.

Compared to existing controller synthesis frameworks, the main contribution is actually the constraint-based nature of the approach. Constraints are useful first to ease the modeling task, and second for efficiency reasons. For example, dedicated constraint programming algorithms available in Gecode (and in most constraint programming engines) have been useful to efficiently handle cardinality constraint in the second case study.

Last, in pratice, even if controller synthesis is claimed to produce controllers valid by construction, there can still be errors in the initial specification. The modeler could therefore be interested in re-checking the controllers produced. Simulation techniques allowing the modeler to play the control could be used, as well as techniques allowing a structured (and easier to read) controller to be returned. In another direction, when no solution controller is found, it would be useful to identify an inconsistent kernel of the model, that is a small set of constraints inducing inconsistency.

Acknowledgment

This work has been performed in the context of the French CNES-ONERA-LAAS AGATA project (Autonomy Generic Architecture: Tests and Applications; see http://www.agata.fr), whose aim is to develop techniques allowing space system autonomy to be improved.

References

[1] A. Arnold, A. Vincent, and I. Walukiewicz, 'Games for synthesis of controllers with partial observation', *Theoretical Computer Science*, **303**(1), 7–34, (2003).

[2] G. Behrmann, A. Cougnard, A. David, E. Fleury, K.G. Larsen, and D. Lime. UPPAL-TiGA. Available from http://www.cs.aau.dk/~adavid/tiga.

[3] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso, 'Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking', in *Proc. of IJCAI-01*, pp. 473–478, (2001).

[4] B. Bonet, H. Palacios, and H. Geffner, 'Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners', in *Proc. of ICAPS-09*, (2009).

[5] R. Bryant, 'Graph-based algorithms for boolean function manipulation', *IEEE Transactions on Computers*, **C-35**(8), 677–691, (1986).

[6] E. Emerson, 'Temporal and Modal Logic', in *Handbook of Theoretical Computer Science, Volume B:Formal Models and Semantics*, 995–1072, (1990).

[7] Esterel Technologies. SCADE. http://www.esterel-technologies.com/.

[8] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from http://www.gecode.org.

[9] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, 'Anzu: A Tool for Property Synthesis', in *Proc. of Computer Aided Verification (CAV)*, pp. 258–262, (2007).

[10] C. Pralet, G. Verfaillie, M. Lemaître, and G. Infantes, 'Constraint-Based Controller Synthesis in Non-Deterministic and Partially Observable Domains', in *Proc. of ECAI-10* (2010).

[11] Ratsy Team. RATSY: Requirement Analysis Tool with Synthesis. Available from http://rat.fbk.eu/ratsy/.