Tools for Controller Synthesis of Timed Systems

Karine Altisen[*] Stavros Tripakis[†]

1 Introduction

Verification is the problem of checking whether the behavior of a *closed* system satisfies a given property. The system is closed in the sense that its behavior is fully specified. *Synthesis* is the problem of finding a way to "close" an *open* system, so that the behavior of the closed system satisfies a given property. The system is open in the sense that its behavior is under-specified: it can be modified (restricted) during the synthesis process. Closing a system usually means coupling it with a *controller* which observes the behavior of the system so far and restricts it by *disabling* or *forcing* some actions. The controller is *state-feedback* when its decisions depend solely on the current state of the system.

In this paper, we present two tools for (state-feedback) controller synthesis of timed systems. Our model is based on timed automata [4] with discrete actions annotated as *controllable* or *uncontrollable*. Controllable actions can be restricted, while uncontrollable cannot. Following the approaches of [23, 7, 3] we associate urgency with actions rather than states. We are interested in controllers ensuring two types of properties, namely, *invariance* or *inevitability*. Invariance means that all behaviors of the closed system remain within a given set of states. Inevitability means that all behaviors of the closed system reach a given set of states. We present our model and define the synthesis problems in Section 2.

The first tool we present is called *SynthKro*. It is a module of the tool suite Kronos [12, 8]. SynthKro is based on the notion of *controllable* states, which are computed using a *backward fixpoint iteration* of special *symbolic predecessor* operators [29, 16, 19, 5]. The tool SynthKro, its algorithms and experimental results are presented in Section 3.

The second tool we present is called *FlySynth*. It is based on an *on-the-fly* synthesis algorithm [25] which works on finite graphs with edges marked controllable or uncontrollable. The algorithm is on-the-fly in the sense that it can find a controller (or say that none exists) without necessarily exploring the entire state space. FlySynth can also be used also for controller synthesis of timed systems, in two ways: either by interpreting the timed automaton model in *discrete time* and using an appropriate abstraction to make the resulting semantic graph finite, or by interpreting the timed automaton in *dense time* and using the *time-abstracting bisimulation* quotient graph [26]. In both cases, we show how timed transitions are interpreted as controllable or uncontrollable. The tool FlySynth, its algorithms and experimental results are presented in Section 4.

Related work Controller synthesis has been studied extensively, both for discrete and dense time systems. One of the oldest discrete-time frameworks is the one of *supervisory control* of *discrete event systems* [22]. A number of tools have been produced by the DES community, such as CTCT [30], TTCT [10, 21], UMDES [27], STCT [31]. Another discrete-time framework is the one of the tool Sigali [20] which is based on the synchronous language Signal and can perform controller synthesis for various objectives including safety, reachability and various combinations of the two. Our main contribution with respect to the above works is the tool FlySynth which performs synthesis on-the-fly.

Controller synthesis in a dense-time context has been studied in [29, 16, 19, 5]. To our knowledge, we provide the first concrete implementation of the ideas introduced in the above works. There are also some differences between our model and the ones used in the works above. First, we use a notion of urgency

[*]BIP, INRIA Rhône-Alpes, 38330 Montbonnot, France. E-mail: Karine.Altisen@inrialpes.fr.
[†]VERIMAG, Centre Equation, 2, ave de Vignate, 38610 Gières, France. E-mail: tripakis@imag.fr. Web: www-verimag.imag.fr.

associated with actions rather than states. Second, we do not assume that a controllable action must infinitely often be taken, as in [19]. Third, we do not assume that the transition relation is given in a *game-theoretic* form, as is done in [5]. It remains to be seen how these assumptions can be handled in practice. Timed controller synthesis is also performed in the tool Circa, used for mission planning [14]. The main differences with our tools are two. First, although the open system in Circa is timed, the controllers are *clockless*, that is, they base their decision only on the discrete part of the system's state. Second, the model used by Circa is a set of guarded actions taking an amount of time in a given interval.

Controller synthesis is of course closely related to game theory, program synthesis and synthesis from logical specifications. Some references to this heritage can be found in [5].

2 Controller synthesis problems

We first present our model, called *controllable timed automata* (CTA). Then we define controllers and state the synthesis problems with respect to invariance or inevitability.

Controllable Timed Automata A CTA is a tuple $\mathsf{A} = (\mathsf{Q}, \mathsf{Q}_0, \mathsf{X}, \mathsf{E}, \mathsf{E}_c, \mathsf{E}_i)$, where:

- Q is a finite set of *discrete states*, $\mathsf{Q}_0 \subseteq \mathsf{Q}$ are *initial* discrete states.
- X is a finite set of *clocks*.
- E is a finite set of *edges*. An edge is a tuple $\mathsf{e} = (\mathsf{q}, \mathsf{q}', \mathsf{a}, \mathsf{g}, X)$, where $\mathsf{q}, \mathsf{q}' \in \mathsf{Q}$ are the source and destination discrete states, a is a *label*, g is the *guard*, $X \subseteq \mathsf{X}$ is the set of clocks to be reset to zero.

  $\mathsf{E}_c \subseteq \mathsf{E}$ is the subset of *controllable* edges. $\mathsf{E} - \mathsf{E}_c$, also denoted E_u, is the subset of *uncontrollable* edges. $\mathsf{E}_i \subseteq \mathsf{E}_u$ is the subset of *urgent* edges (which, once enabled, cannot become disabled by letting time elapse).

As usual, guards are polyhedra represented as conjunctions of atomic constraints of the form $x\#k$, $x - y\#k$, where $x, y \in \mathsf{X}$ are clocks, $\# \in \{<, \leq, =, \geq, >\}$ and $k \in Z$ is an integer constant.

Semantics A CTA defines an infinite transition system $\mathsf{TS} = (\mathsf{S}, \mathsf{T})$, where:

- S is the set of *states*. A state is a tuple (q, v), where $\mathsf{q} \in \mathsf{Q}$ and $\mathsf{v} : \mathsf{X} \to R_+$ is a function giving for each clock a non-negative real value; v is also called a *valuation*. The valuation assigning zero to all clocks is denoted $\vec{0}$. The set of *initial states* S_0 is the set of all $(\mathsf{q}, \vec{0})$ such that $\mathsf{q} \in \mathsf{Q}_0$.
- T is the set of *transitions*. A transition is of two types: *discrete* or *timed*. A discrete transition is a tuple $(\mathsf{s}, \mathsf{e}, \mathsf{s}')$, also denoted $\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}'$, such that:

  1. $\mathsf{s}, \mathsf{s}' \in \mathsf{S}$, $\mathsf{s} = (\mathsf{q}, \mathsf{v})$, $\mathsf{s}' = (\mathsf{q}', \mathsf{v}')$ and $\mathsf{e} \in \mathsf{E}$, $\mathsf{e} = (\mathsf{q}, \mathsf{q}', \mathsf{a}, \mathsf{g}, X)$,
  2. v satisfies the guard g and v' is obtained by v by setting all clocks in X to zero and keeping the values of the rest of the clocks the same.

  A timed transition is a tuple $(\mathsf{s}, \delta, \mathsf{s}')$, also denoted $\mathsf{s} \xrightarrow{\delta} \mathsf{s}'$, such that:

  1. s, s' are states, $\mathsf{s} = (\mathsf{q}, \mathsf{v})$, $\mathsf{s}' = (\mathsf{q}, \mathsf{v}')$ and $\delta \in R_+$ is a time delay,
  2. $\mathsf{v}' = \mathsf{v} + \delta$ and there exists no urgent edge $\mathsf{e} \in \mathsf{E}_i$, $\mathsf{e} = (\mathsf{q}, \mathsf{q}', \mathsf{a}, \mathsf{g}, X)$, such that v satisfies g and $\mathsf{v} + \delta$ does not satisfy g.

Condition 2 on timed transitions models urgency. It states that once an urgent edge e becomes enabled, it cannot become disabled by letting time pass, that is, e or another edge must be taken before e becomes disabled. Notice that controllable edges are initially *lazy* (non-urgent). This is because it is the role of the controller to determine their urgency: that is, the controller will decide when to let time elapse and when to force a controllable action to occur.

TS represents the *dense-time* semantics of a CTA. We will also associate with a CTA its *discrete-time* semantics, represented by the transition system $\mathsf{TS_{dt}}$. $\mathsf{TS_{dt}}$ is obtained from TS by removing all timed transitions except those with a delay of one time unit. That is, the timed transitions of $\mathsf{TS_{dt}}$ are all timed transitions $\mathsf{s} \xrightarrow{1} \mathsf{s}'$ of TS.

Controllers A *state-feedback controller* (controller for short) is a function $\mathsf{C} : \mathsf{S} \times \mathsf{E_c} \to \{0, 1\}$. Intuitively, if $\mathsf{C}(\mathsf{s}, \mathsf{e}) = 0$ then the controller disables e at state s, otherwise it enables e at s. Note that enabling e does not mean forcing it: in general, the controller is non-deterministic, in the sense that at a given state it may decide to let time pass or force an enabled edge e.

We impose the following conditions on the controllers we consider:

1. For any edge $\mathsf{e} \in \mathsf{E_c}$ and any state s, if e is disabled in s then $\mathsf{C}(\mathsf{s}, \mathsf{e}) = 0$. This means that the controller must not enable an edge at a state where it is disabled.

2. The state space S admits a finite partition into sets $\mathsf{S}_1, ..., \mathsf{S}_k$, such that for all $i = 1, ..., k$, for any $\mathsf{s}, \mathsf{s}' \in \mathsf{S}_i$ and any $e \in \mathsf{E_c}$, $\mathsf{C}(\mathsf{s}, \mathsf{e}) = \mathsf{C}(\mathsf{s}', \mathsf{e})$. This means that the controller has finite memory.

The *closed-loop system* is a transition system $\mathsf{TS_C} = (\mathsf{S}, \mathsf{T_C})$, where $\mathsf{T_C}$ is obtained by restricting T according to the decisions of C. More precisely:

1. $\mathsf{T_C} \subseteq \mathsf{T}$.

2. $\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}'$ is a discrete transition of $\mathsf{T_C}$ iff $\mathsf{C}(\mathsf{s}, \mathsf{e}) = 1$.

3. If $\mathsf{s} \xrightarrow{\delta} \mathsf{s}'$ is a timed transition of $\mathsf{T_C}$, then there exists no controllable edge $\mathsf{e} \in \mathsf{E_c}$, $\mathsf{e} = (\mathsf{q}, \mathsf{q}', \mathsf{a}, \mathsf{g}, X)$, such that $\mathsf{C}(\mathsf{s}, \mathsf{e}) = 1$ and for some $\delta' \leq \delta$, $\mathsf{C}(\mathsf{s} + \delta', \mathsf{e}) = 0$.

Condition 2 says that a discrete transition disabled by the controller cannot be taken. Condition 3 says that all controllable transitions enabled by the controller are assumed to be urgent in the closed loop system. The rationale behind this choice is explained in Example 2.2.

Properties A state s of $\mathsf{TS_C}$ is *reachable* if there is a path in $\mathsf{TS_C}$ from some initial state $\mathsf{s}_0 \in \mathsf{S}_0$ to s. A state s is *live* if there exist $\delta \in R_+$ and $\mathsf{e} \in \mathsf{E_i} \cup \mathsf{E_c}$ such that $\mathsf{s} \xrightarrow{\delta} \xrightarrow{\mathsf{e}} \mathsf{s}'$, for some $\mathsf{s}' \in \mathsf{S}$. $\mathsf{TS_C}$ is called live if all its reachable states are live. An infinite path in $\mathsf{TS_C}$ is called *strongly fair* if it is not the case that an edge $\mathsf{e} \in \mathsf{E_i} \cup \mathsf{E_c}$ is infinitely often enabled in the path, but never taken. An infinite path in $\mathsf{TS_C}$ is called *zeno* if the sum of delays of all timed transitions in the path is bounded. Otherwise the path is *non-zeno*.

Note: when a system is live, this means that all its fair non-zeno paths from an initial state will not get stalled at some discrete state unless there exists an explicit self-loop enabled forever from this state.

Suppose $\mathsf{S_{safe}}$ and $\mathsf{S_{target}}$ are subsets of S. An *invariance* property is of the form $\Box(\mathsf{S_{safe}})$ and an *inevitability* property is of the form $\Diamond(\mathsf{S_{target}})$. We say that the closed-loop system $\mathsf{TS_C}$ satisfies $\Box(\mathsf{S_{safe}})$ if for every reachable state s in $\mathsf{TS_C}$, $\mathsf{s} \in \mathsf{S_{safe}}$. We say that $\mathsf{TS_C}$ satisfies $\Diamond(\mathsf{S_{target}})$ if all its strongly fair non-zeno paths starting at an initial state reach a state $\mathsf{s} \in \mathsf{S_{target}}$.

Definition 2.1 (Controller synthesis invariance problem) *Given a CTA A and an invariance property ϕ, find a controller C such that the closed-loop system is live and satisfies ϕ.*

Definition 2.2 (Controller synthesis inevitability problem) *Given a CTA A and an inevitability property ϕ, find a controller C such that the closed-loop system satisfies ϕ.*

Note that we require that the controller is live only for invariance properties. Liveness implies that discrete transitions occur infinitely often, which guarantees non-zenoness in certain cases (see Assumption 2.1). On the other hand, we do not loose in generality: if we want to express the fact that the system is allowed to "stop" in a discrete state without producing any more discrete events, we can model this by adding an uncontrollable *self-loop* edge on that state (with trivial guard true). The reason we do not require a live controller for inevitability is because the latter is already a "liveness" property in some sense. Also, not requiring liveness of the controller makes the inevitability synthesis procedure much simpler.

Example 2.1 *This is an example for invariance. Consider the CTA shown in Figure 1: c_1 and c_2 are controllable, u_i is uncontrollable and inevitable, u is uncontrollable. We want to synthesize a live controller for $\Box(\neg \mathsf{q}_3)$. The least restrictive controller for this problem restricts the guard of c_1 to* false *and the guard of c_2 to $x \leq 2$. The reasons are: the states $\mathsf{s} = (\mathsf{q}_1, \mathsf{v})$ are safe but not live so c_1 is disabled; the controller also has to forbid the system to reach q_3 by taking u. This is done by disabling a part of the guard of c_2 which becomes urgent before 2 in the closed-loop system.*

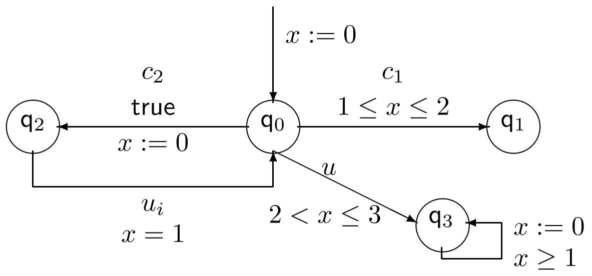

Figure 1: A controller synthesis problem for invariance.

Example 2.2 *As mentioned above, all controllable edges are interpreted as urgent in the closed loop system. The rationale behind this choice is explained by the example shown in Figure 2. Assume we want to synthesize a controller for $\Diamond(p)$ and that both edges labeled c_1 and c_2 are controllable. Suppose the controller was allowed to "mark" controllable edges as urgent or not, and a solution which marks as urgent as few edges as possible was preferable. Then, there is no unique best solution in this example. Indeed, the controller must mark at least one of c_1, c_2 as urgent (if both are lazy, then time can elapse indefinitely at q_0 and this is a strongly fair path since c_1, c_2 are disabled after one time unit).*

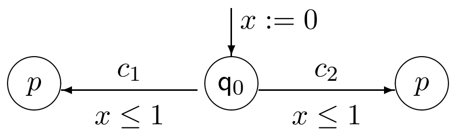

Figure 2: A controller synthesis problem for inevitability.

Assumption 2.1 *One of the problems in timed synthesis is the issue of zenoness. Despite of the liveness and fairness requirements, this issue arises in both cases of invariance and inevitability. In the invariance case, the controller may be zeno: it may forbid time to advance by forcing controllable actions to occur in zero or smaller and smaller amounts of time. The problem is dual in the inevitability case, where the system may be zeno and forbid a controller to reach the target. In our tools, we opted for simplicity and did not directly incorporate methods to ensure that the controllers synthesized are non-zeno. However, this can be guaranteed for so-called* strongly non-zeno *systems, where an infinite number of discrete transitions takes an unbounded amount of time. A sufficient condition for strong non-zenoness is the existence, for every loop in the CTA, of a clock x which is reset and bounded from below (e.g., $x \geq 1$) at some points in the loop. In the rest of the paper, we will assume that the CTA modeling the open system is strongly non-zeno. Indeed, this has been true in all case studies we treated with SynthKro and FlySynth.*

Notation In the rest of the paper, we use the following notation. Given $\mathsf{s} \in \mathsf{S}$, $\mathsf{e} \in \mathsf{E}$, we write $\mathsf{s} \xrightarrow{\mathsf{e}}$ if there exists $\mathsf{s}' \in \mathsf{S}$ such that $\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}'$. Otherwise, we write $\mathsf{s} \overset{\mathsf{e}}{\not\to}$. Similarly, for $\delta \in R_+$, we write $\mathsf{s} \xrightarrow{\delta}$ or $\mathsf{s} \overset{\delta}{\not\to}$. If $P \subseteq \mathsf{S}$ is a set of states, we write $\mathsf{s} \xrightarrow{\mathsf{e}} P$ (resp., $\mathsf{s} \xrightarrow{\delta} P$) if there exists $\mathsf{s}' \in P$ such that $\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}'$ (resp., $\mathsf{s} \xrightarrow{\delta} \mathsf{s}'$). We also use such notation embedded in quantified formulas, with the natural meaning. For example, when we write $\exists \mathsf{e}, \mathsf{s} \xrightarrow{\mathsf{e}} P$, we mean $\exists \mathsf{e}, \exists \mathsf{s}' \in P, \mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}'$. When we write $\forall \mathsf{e}, \mathsf{s} \xrightarrow{\mathsf{e}} P$, we mean $\forall \mathsf{e}, \forall \mathsf{s}', \mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}' \implies \mathsf{s}' \in P$.

3 The tool SynthKro

In this section we present the tool SynthKro. SynthKro is a module of the Kronos tool suite [12, 8]. It solves the two controller synthesis problems defined in Section 2 by using a *backward fixpoint* algorithm. In what follows, we present this algorithm, then the tool and finally some experimental results.

3.1 The backward fixpoint synthesis algorithm

The algorithm used in SynthKro works in two phases. In the first phase, the algorithm computes a subset of states, $\mathsf{S}_c \subseteq \mathsf{S}$, called the *controllable states.* In the second phase, the controller is computed based on S_c. S_c is computed by a fixpoint iteration which uses a *controllable predecessor operator*, specialized for the cases of invariance (Definition 2.1) or inevitability (Definition 2.2). We present the two cases below and then show how to extract a controller from the set of controllable states. The definitions that follow (Equations (1)-(5)) refer to the open transition system TS (e.g., when we write $\mathsf{s} \xrightarrow{\mathsf{e}}$ we mean there exists a discrete transition e from s in T).

3.1.1 Computing controllable states for invariance

We first define the following predicates on a state s and a set of states P:

$$\Diamond_d(\mathsf{s}) \quad = \quad \exists \delta \in R_+, \exists \mathsf{e} \in \mathsf{E_i}, \mathsf{s} \xrightarrow{\delta}\xrightarrow{\mathsf{e}} \tag{1}$$

$$\Box_u(\mathsf{s}, P) \quad = \quad \forall u \in \mathsf{E_u}, \mathsf{s} \xrightarrow{u} P \tag{2}$$

$\Diamond_d(\mathsf{s})$ says that s is live, i.e., time cannot elapse indefinitely from s, that is, an urgent transition will eventually be taken in every strongly fair paths from s[1]. $\Box_u(\mathsf{s}, P)$ says that any uncontrollable edge from s leads to P.

We now define the operator $\mathsf{pre}^{\mathsf{c}}_{\Box}(P)$, where P is a set of states:

$$\begin{aligned} \mathsf{pre}^{\mathsf{c}}_{\Box}(P) = \{\mathsf{s} \mid & \Diamond_d(\mathsf{s}) \quad \wedge \\ & \Box_u(\mathsf{s}, P) \quad \wedge \\ & \forall \delta \in R_+, (\mathsf{s} \xrightarrow{\delta} \neg P) \implies (\exists \delta' < \delta, \exists c \in \mathsf{E_c}, \mathsf{s} \xrightarrow{\delta'}\xrightarrow{c} P)\} \end{aligned} \tag{3}$$

$\mathsf{pre}^{\mathsf{c}}_{\Box}(P)$ includes all states s such that: (1) s is live, (2) any uncontrollable edge from s leads to P and (3) if the system moves out of P after δ time units, then there exists a controllable edge which can be forced before δ and keep the system in P.

Then, we compute the following sequence:

$$\begin{aligned} \mathsf{S}^0_{\Box} &= \mathsf{S_{safe}} \\ \mathsf{S}^{i+1}_{\Box} &= \mathsf{S}^i_{\Box} \cap \mathsf{pre}^{\mathsf{c}}_{\Box}(\mathsf{S}^i_{\Box}) \end{aligned}$$

It can be shown [2] that the above sequence converges in a finite number of steps to a set $\mathsf{S}_{\Box}$, which is the greatest fixpoint of the equation $P = \mathsf{S_{safe}} \cap \mathsf{pre}^{\mathsf{c}}_{\Box}(P)$. $\mathsf{S}_{\Box}$ is the greatest set of controllable states S_c in the case of invariance.

3.1.2 Computing controllable states for inevitability

We first define the following predicate on a state s and a set of states P:

$$\Diamond_u(\mathsf{s}, P) \quad = \quad \Diamond_d(\mathsf{s}) \ \wedge \ \forall \delta, \forall u \in \mathsf{E_u}, \mathsf{s} \xrightarrow{\delta}\xrightarrow{u} P \tag{4}$$

$\Diamond_u(\mathsf{s}, P)$ states that an uncontrollable edge will eventually lead from s to P, and no uncontrollable edge can lead out of P meanwhile.

[1]This is true even in the case where an urgent transition has a guard which is not upper-bounded, because of the requirement of strong fairness.

We now define the operator $\mathsf{pre}^{\mathsf{c}}_{\Diamond}(P)$, where P is a set of states:

$$\begin{aligned}\mathsf{pre}^{\mathsf{c}}_{\Diamond}(P) = \{s \mid & \exists c \in \mathsf{E_c}, \mathsf{s} \xrightarrow{c} P \wedge \Box_u(\mathsf{s}, P) \quad \vee \\ & \Diamond_u(\mathsf{s}, P) \quad \vee \\ & \exists \delta, \mathsf{s} \xrightarrow{\delta} P \wedge \forall \delta' \leq \delta, \forall u \in \mathsf{E_u}, \mathsf{s} \xrightarrow{\delta'} \xrightarrow{u} P\}\end{aligned} \tag{5}$$

$\mathsf{pre}^{\mathsf{c}}_{\Diamond}(P)$ includes all states s such that: either (1) the system can be led immediately to P with a controllable action and any uncontrollable action will also lead to P, or (2) the system will be led to P by uncontrollable urgent actions, or (3) the system will go to P after δ time units, and no uncontrollable action can lead out of P before that.

Then, we compute the following sequence:

$$\begin{aligned}\mathsf{S}^0_{\Diamond} &= \mathsf{S_{target}} \\ \mathsf{S}^{i+1}_{\Diamond} &= \mathsf{S}^i_{\Diamond} \cup \mathsf{pre}^{\mathsf{c}}_{\Diamond}(\mathsf{S}^i_{\Diamond})\end{aligned}$$

It can be shown [2] that the above sequence converges in a finite number of steps to a set $\mathsf{S}_{\Diamond}$, which is the least fixpoint of the equation $P = \mathsf{S_{target}} \cup \mathsf{pre}^{\mathsf{c}}_{\Diamond}(P)$. $\mathsf{S}_{\Diamond}$ is the greatest set of controllable states S_c in the case of inevitability.

3.1.3 Extracting the controller from the set of controllable states

Assuming the set S_c produced by the first phase is non-empty and contains the initial states (otherwise, no controller exists), the controller C is defined by the following procedure:

```
for each $\mathsf{q} \in \mathsf{Q}$
  for each $\mathsf{e} \in \mathsf{E_c}, \mathsf{e} = (\mathsf{q}, \mathsf{q}', \_, \_, \_)$
    let $\mathsf{S}' = \mathsf{S}^{\mathsf{q}}_c \cap \mathsf{pre}(\mathsf{e}, \mathsf{S}^{\mathsf{q}'}_c)$
    set $\mathsf{C}(\mathsf{S}', \mathsf{e}) = 1$
    set $\mathsf{C}(\mathsf{S} - \mathsf{S}', \mathsf{e}) = 0$
```

where $\mathsf{S}^{\mathsf{q}}_c$ is the set of all states $\mathsf{s} \in \mathsf{S}_c$ such that $\mathsf{s} = (\mathsf{q}, \_)$ and $\mathsf{pre}(\mathsf{e}, P) = \{\mathsf{s} \mid \exists \delta \in R_+, \mathsf{s} \xrightarrow{\delta} \xrightarrow{\mathsf{e}} P\}$. That is, for each discrete state q and controllable edge e, C disables e at all states, except those which are controllable and lead to a new controllable state after taking e.

3.2 The tool SynthKro

The fixpoint algorithms have been implemented in the Kronos module SynthKro (see figure 3). SynthKro takes as inputs a `.tg` file describing the CTA and a `.tctl` file specifying the property. SynthKro outputs the set of controllable states in the `.eval` file and the closed-loop system (represented as a CTA) in the `.contr` file. SynthKro is publicly available as part of the Kronos distribution[2].

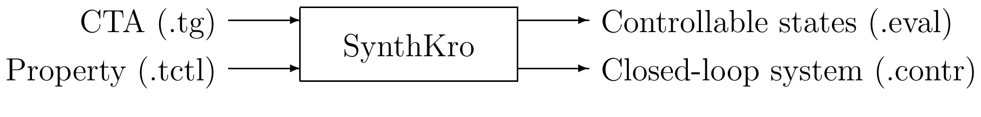

Figure 3: SynthKro tool chain.

We briefly discuss the implementation of the controllable predecessor operators $\mathsf{pre}^{\mathsf{c}}_{\Box}$ and $\mathsf{pre}^{\mathsf{c}}_{\Diamond}$. These operators are significantly harder to implement than usual predecessor (or successor) operators in model checking tools such as Kronos or Uppaal [17]. The difficulty comes from the fact that $\mathsf{pre}^{\mathsf{c}}_{\Box}$ and $\mathsf{pre}^{\mathsf{c}}_{\Diamond}$ use multiple alternations of quantifiers. In a first implementation, we used complementation to eliminate

[2]downloadable from www-verimag.imag.fr/TEMPORISE/kronos/

the alternations, but this is expensive, since complementations do not preserve convexity and cannot be implemented using the *DBM* data structure [13]. More efficient ways to implement $\mathsf{pre}^{\mathsf{c}}_{\Box}$ and $\mathsf{pre}^{\mathsf{c}}_{\Diamond}$ have been developed in [1] and are currently used in SynthKro. They are based on a direct way of eliminating quantifiers, without complementation. Some examples of how this is done can be found in [24].

3.3 Experimental results

We report performance results on three case studies treated with SynthKro. We first considered a robotic arm case study [18]. The arm is programmed to take objects from a conveyor belt, store them in a shelf and eventually put them into a basket. The system is controlled by five processes sharing a CPU, namely a trajectory controller, a lifter and a putter, a sensor reader and a motion planner (for refining trajectories). This latter process is optional: it is run only as long as it does not cause the other processes to miss their deadlines. We modeled the system as a CTA and used SynthKro to compute a controller which ensures mutual exclusion (two processes do not use the CPU at the same time) and that no deadlines are missed (both properties can be expressed as invariance). The results are shown in Table 1.

The second study considers scheduling three processes sharing three resources [18]. The first process is periodic with an initial jitter; the other two processes are sporadic with minimum inter-arrival times. The first and second processes are mandatory. The goal is to compute a controller that allocates the mutually exclusive resources so that the deadlines of the processes are not missed. In the first experiment, we modeled only the two mandatory processes; in the second experiment we added the optional process; in the last experiment (where SynthKro failed to synthesize a controller due to lack of memory) we added another mandatory periodic process.

The last case study was taken from [6]: it represents a small part of the communication mode of a GSM terminal. The goal for the controller is to treat the signals from the environment without missing any of them. Again, this property can be modeled as invariance.

Case study	CTA size			Performance	
	states	trans.	clocks	memory	time
robotic arm	349	1132	5	115MB	3h20m
2 processes	17	32	4	3MB	4s
3 processes	46	126	6	37MB	10m
4 processes	80	540	8	∞	∞
GSM mobile terminal	165	454	4	140MB	2m45s

Table 1: Performance results of SynthKro.

4 The tool FlySynth

In this section we present the tool FlySynth. FlySynth is an on-the-fly synthesis tool for finite graphs. In what follows, we first briefly discuss the ideas behind on-the-fly synthesis [25], then we present the tool and explain how it is used for synthesis of timed systems. We finally present some experimental results.

4.1 On-the-fly synthesis for finite graphs

FlySynth takes as input a *controllable graph* (CG), that is, a tuple $G = (V, v_0, E)$ where V is a finite set of vertices, $v_0 \in V$ is the initial vertex and E is a finite set of edges[3]. $E_{\mathsf{c}} \subseteq E$ is the set of controllable edges. $E_{\mathsf{u}} = E - E_{\mathsf{c}}$ is the set of uncontrollable edges. $E_{\mathsf{i}} \subseteq E_{\mathsf{u}}$ is the set of urgent (uncontrollable) edges. A controller of a CG is a function $C : V \times E_{\mathsf{c}} \to \{\mathsf{0}, \mathsf{1}\}$. Since an edge has a unique source vertex, we can simplify C by writing $C : E_{\mathsf{c}} \to \{\mathsf{0}, \mathsf{1}\}$. The closed loop system is the subgraph of G where all edges e such that $C(e) = \mathsf{0}$ are removed. FlySynth synthesizes a controller which ensures either liveness and invariance

[3]For simplicity of presentation, we assume that $E \subseteq V \times V$. In fact, FlySynth accepts graphs with labeled edges, and there can be more than one edge between two vertices.

(all vertices of the closed loop graph are in a given set of vertices and every vertex has a successor) or inevitability (all paths of the closed loop graph reach a given set of target vertices).

FlySynth uses the on-the-fly synthesis algorithms described in [25]. The algorithm performs a forward exploration of the vertex space by using marking techniques. It is a depth first search adapted to (un)controllable label semantics: all uncontrollable transitions and at least one controllable or inevitable edge is explored, departing from each visited vertex. For the invariance property, if a bad vertex (w.r.t. the property) is visited, a back-track procedure is applied in order for the controller to avoid reaching this vertex; and if a good vertex is reached, then the exploration stops. Conversely, for the reachability property, whenever a bad vertex is reached, the exploration stops; whereas when a good vertex is visited, a back-track procedure is launched in order to exhibit the paths to target vertices.

The exploration procedure of the algorithms follows the logical scheme below:

$$\mathsf{visit}(v) = \text{not BAD}(v) \wedge \big(\text{GOOD}(v) \vee \forall (v, v') \in E_{\mathsf{u}}.\mathsf{visit}(v') \wedge \exists (v, v') \in E_{\mathsf{c}} \cup E_{\mathsf{i}}.\mathsf{visit}(v')\big).$$

The first call to visit is done by $\mathsf{visit}(v_0)$. If $\mathsf{visit}(v_0)$ is true then a controller exists otherwise none exists. The meaning of GOOD and BAD, as well as the way vertices are marked differ depending on the property. For invariance property, GOOD(v) means that v has been visited twice: a safe cycle has been found from v, i.e. the exploration stops. The predicate BAD(v) means that v is unsafe, i.e. either it is in the set of unsafe vertices or it may lead to one of them. For inevitability property, GOOD(v) means that v eventually leads to target vertices. BAD(v) becomes true when v is visited for the second time: a cycle has been found that could never lead to target vertices.

The controller is computed as follows. In the case of invariance the controller disables all edges except edges (v, v') such that none of v or v' are marked BAD. In the case of inevitability the controller disables all edges except edges (v, v') such that both v and v' are marked GOOD. When no controller has been found by the algorithm this means that none exists. In that case a diagnostic is provided under the form of a counter-example: this is the subgraph of the CG generated by the BAD vertices.

On-the-fly Exploration FlySynth synthesizes a controller on-the-fly, that is, it partially explores the vertex space and stops as soon as a controller is found. Thus, it does not always explore the entire graph. For example, if an uncontrollable edge from the initial vertex leads to an unsafe vertex, then no controller exists: if this edge is explored first, then no other edge will be explored. Also, if a controllable edge leads to a safe part of the graph, then no other controllable edges need to be explored from this vertex. The fact that FlySynth works on-the-fly implies that the synthesized controllers are not *least restrictive*. Informally, a controller is least restrictive if it disables as few controllable actions as possible. Optionally, the user can request FlySynth to synthesize a least restrictive controller. In such a case, FlySynth will explore all controllable edges from a vertex and disable only the ones that need to be disabled.

The worst case complexity of the algorithm is linear (in time and memory used) in the CG size. In fact, the algorithm worst case scenario visits every reachable vertices twice (once to explore and once to back-track).

4.2 Using FlySynth for timed synthesis

FlySynth can be used for solving the synthesis problems of Definitions 2.1 and 2.2 in two ways, depending on whether we associate with the open system a discrete or dense time semantics. We present the two possibilities in what follows.

4.2.1 Using FlySynth for discrete time synthesis

In this case, the CG G is obtained from the discrete time transition system $\mathsf{TS}_{\mathsf{dt}}$. As $\mathsf{TS}_{\mathsf{dt}}$ is generally infinite, an abstraction is applied to make it finite: all states s of $\mathsf{TS}_{\mathsf{dt}}$ where time can progress forever without any new transition becoming enabled or disabled are removed.

All controllable (resp. uncontrollable, urgent) discrete transitions of $\mathsf{TS}_{\mathsf{dt}}$ become controllable (resp. uncontrollable, urgent) edges of G. A timed transition $\mathsf{s} \xrightarrow{1} \mathsf{s}'$ of $\mathsf{TS}_{\mathsf{dt}}$ is treated as follows: if there exists $c \in \mathsf{E}_{\mathsf{c}}$ such that $\mathsf{s} \xrightarrow{c}$ then $\mathsf{s} \xrightarrow{1} \mathsf{s}'$ is taken to be controllable in G, otherwise, it is uncontrollable and urgent.

4.2.2 Using FlySynth for dense time synthesis

In this case, the CG G is obtained by taking the quotient graph of the dense time transition system TS with respect to the greatest time abstracting bisimulation (TAB) [26]. A TAB is an equivalence on states which preserves discrete transitions (if two states are equivalent then they can perform the same discrete transitions and lead to equivalent states) and abstracts timed transitions (if two states are equivalent then if one can perform a timed transition δ then the other can perform a timed transition δ' and go to an equivalent state). The quotient of a CTA A with respect to the TAB can be computed using the module of Kronos *Minim*. An initial partition can be given to distinguish, for instance, safe from unsafe states.

The TAB-quotient graph is finite and contains two types of edges: edges corresponding to discrete transitions of TS and edges corresponding to timed transitions. The latter are labeled by the special label ϵ: for instance, $(\mathsf{q}, x < 1) \xrightarrow{\epsilon} (\mathsf{q}, x \geq 1)$ denotes the fact that all states in the equivalence class $(\mathsf{q}, x < 1)$ are bisimilar and by letting time pass they lead to another equivalence class, $(\mathsf{q}, x \geq 1)$.

All controllable (resp. uncontrollable, urgent) discrete edges of the TAB-quotient $\mathsf{TS}_{\mathsf{dt}}$ become controllable (resp. uncontrollable, urgent) edges of G. An edge $P \xrightarrow{\epsilon} Q$ of the TAB-quotient is treated as follows: if there exists $c \in \mathsf{E_c}$ such that $P \xrightarrow{c}$ then $P \xrightarrow{\epsilon} Q$ is taken to be controllable in G, otherwise, it is uncontrollable and urgent.

We should note that in this case, synthesis is not entirely on-the-fly, since the TAB-quotient must be generated first. Since no good on-the-fly techniques exist for generating bisimulation quotients[4] the TAB-quotient must be entirely generated before FlySynth can be applied. Timed-automata model-checking tools often use another graph, called the *simulation graph* for verification. The simulation graph is generated in a forward on-the-fly manner. One may wonder, then, why the simulation graph cannot be used for synthesis. The reason is explained in the following example.

$$\begin{tikzcd}
 & {} \arrow[d, "x := 0"] & \\
\mathsf{q}_1 & \mathsf{q}_0 \arrow[l, "u", "x \geq 1"'] \arrow[r, "c"] & \mathsf{q}_2
\end{tikzcd}$$

A controllable timed automaton

$$\begin{tikzcd}
 & {} \arrow[d] & \\
(\mathsf{q}_1, x \geq 1) & (\mathsf{q}_0, \mathsf{true}) \arrow[l, "u"'] \arrow[r, "c"] & (\mathsf{q}_2, \mathsf{true})
\end{tikzcd}$$

Its simulation graph

$$\begin{tikzcd}
 & {} \arrow[d] & \\
 & (\mathsf{q}_0, x < 1) \arrow[dd, "\epsilon"] \arrow[dr, "c"] & \\
 & & (\mathsf{q}_2, \mathsf{true}) \\
(\mathsf{q}_1, x \geq 1) & (\mathsf{q}_0, x \geq 1) \arrow[l, "u"'] \arrow[ur, "c"'] &
\end{tikzcd}$$

Its bisimulation graph

Figure 4: A CTA and its simulation and bisimulation graphs.

Example 4.1 *We give an example to show why the simulation graph cannot be used for on-the-fly synthesis with FlySynth. The example is shown in Figure 4. The figure shows a CTA, its simulation graph and its*

[4]Algorithms are usually based on a Paige-Tarjan technique of "splitting" equivalence classes and propagating splittings backwards.

bisimulation graph. Edges labeled c are controllable and the ones labeled u are uncontrollable. Suppose we are interested in synthesizing a controller which avoids q_1 *(that is, which ensures the invariance property* $\Box(\mathsf{q}_0 \vee \mathsf{q}_2)$*. As one can see, the simulation graph does not distinguish between states* $(\mathsf{q}_0, x < 1)$ *and states* $(\mathsf{q}_0, x \geq 1)$*. Consequently, there is an uncontrollable edge from* $(\mathsf{q}_0, \mathsf{true})$ *to* $(\mathsf{q}_1, x \geq 1)$ *in the simulation graph. Looking at the latter as a purely discrete graph, since the node* $(\mathsf{q}_1, x \geq 1)$ *violates the invariance, the initial node* $(\mathsf{q}_0, \mathsf{true})$ *is also uncontrollable, which means the algorithm finds no solution. This problem is avoided in the bisimulation graph, where the algorithm finds the controllable subgraph* $(\mathsf{q}_0, x < 1) \xrightarrow{c} (\mathsf{q}_2, \mathsf{true})$*. This corresponds to a controller that forces the controllable transition c at* q_0*, before x reaches 1 time unit.*

4.3 The tool FlySynth

FlySynth takes as input a controllable graph, a list of its vertices and an option specifying the property (invariance or inevitability). It outputs a controller which ensures the desired property. The controller is presented as a list of *decision points*. A decision point is a vertex of the CG and a list of the disabled controllable edges. In case no controller that satisfies the property exists, the tool provides a diagnostic. The diagnostic is given as a counter-example graph which shows why the system cannot be controlled.

Since the controllable graph is a low level system description, FlySynth is connected to a number of tools using more high level specification languages (see Figure 5). Three connections are currently available: from Kronos (via the module Minim), from IF [9] and from Prometheus [15].

FlySynth can be made available by contacting the authors.

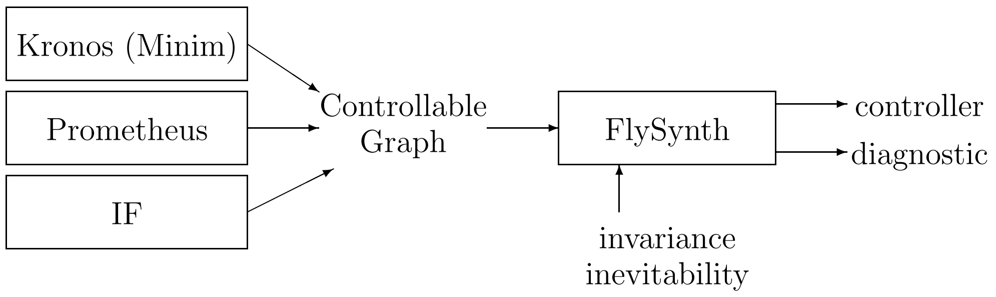

Figure 5: FlySynth tool chain.

4.4 Experimental results

We report performance results on two case studies treated with FlySynth. Both case studies are of non-trivial size. In fact, Kronos fails to produce the syntactic product of the component automata involved in the two studies, therefore, we could not run neither SynthKro nor Minim. Instead, we used IF to generate the discrete-time transition system of the examples, which is then used by FlySynth for synthesis.

The first case study models a mine pump [11]. Captors measure environmental variables such as water level or CH4 ratio. Depending on those variables, the pump must be activated or stopped within given delays. We used FlySynth to compute a controller ensuring the delays are respected (modeled as an invariance property). Table 2 shows experimental results: the 1st column shows the number of reachable states for the whole system. The 2nd and 3rd columns show the explored state space and the size of the controllers. The 4th and 5th columns show the same information in the case where the tool was asked to synthesize the least restrictive controller.

The second case study is inspired from Case study number 7 of the VHS project [28]. The study models a piece of a chemical plant: the goal is to fill buffers with raw materials in order to have a continuous production at the exit of the plant. Between entrance and exit, materials react in different buffers. Here, we used FlySynth to synthesize a controller ensuring that the production is never stopped at the exit of the plant (modeled as an invariance property).

From Table 2 one can see the advantages of on-the-fly synthesis, by comparing the sizes of the entire reachable state space versus the explored state space.

Problem	CG	Controller		l.r. Controller		Performances	
	r.states	expl.st.	C size	expl.st.	C size	time	memory
mine pump	280 801	52 765	24 274	196 987	58 427	18s	39 MB
chemical plant	917 445	543 662	395 854	829 248	647 530	1m18s	132MB

Table 2: Performances for FlySynth

5 Conclusions

We presented the tools SynthKro and FlySynth for controller synthesis of timed systems with respect to invariance or inevitability properties. SynthKro uses a backward fixpoint algorithm using symbolic predecessor operators whereas FlySynth applies a forward on-the-fly algorithm on a finite-graph quotient of the system.

Regarding perspectives, one is to develop synthesis techniques for other properties than invariance or inevitability, in particular, combinations of the two, such as "reach the target states while avoiding unsafe states". Dealing with state explosion is obviously a critical issue, as in many formal techniques. Finally, introducing timed controller synthesis techniques into application domains such as robotics, where currently mainly discrete event frameworks are used, is another promising direction.

References

[1] K. Altisen. Génération automatique d'ordonnancements pour systèmes temporisés. Technical report, Mémoire de DEA, Ensimag, Grenoble, 1998. In french.

[2] K. Altisen. *Application de la synthèse de contrôleur à l'ordonnancement de systèmes temps-réel.* PhD thesis, Institut National Polytechnique de Grenoble, December 2001. In French.

[3] K. Altisen, G. Goessler, A. Pnueli, J. Sifakis, and S. Tripakis. A framework for scheduler synthesis. In *IEEE Real-Time Systems Symposium, RTSS'99*, 1999.

[4] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[5] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, 1995.

[6] V. Bertin, J. Pulou, M. Poize, and J. Sifakis. Towards validated real-time software. In *Proc. 12th Euromicro Conference on Real-Time Systems*, 2000.

[7] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality*, LNCS 1536, 1998.

[8] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: a model-checking tool for real-time systems. In *Proc. of the 10th Conference on Computer-Aided Verification, CAV'98.* LNCS 1427, Springer-Verlag, 1998.

[9] M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: a validation environment for timed asynchronous systems. In *Proc. CAV'00*, 2000.

[10] B.A. Brandin and W.M. Wonham. Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 39(2), 1994.

[11] A. Burns and A. Wellings. *Real Time Systems and Programming Language.* Addison Wesley, 1996.

[12] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[13] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, pages 197–212. Springer–Verlag, 1989.

[14] R.P. Goldman, D.J. Musliner, and M.J. Pelican. Exploiting implicit representations in timed automaton verification for controller synthesis. In *Hybrid Systems: Computation and Control*, 2002.

[15] G. Gößler. PROMETHEUS — a compositional modeling tool for real-time systems. In *Proc. Workshop RT-TOOLS 2001*, 2001.

[16] G. Hoffmann and H. Wong Toi. Symbolic synthesis of supervisory controllers. In *American Control Conference*, 1992.

[17] K. Larsen, P. Petterson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1/2), October 1997.

[18] L. Lusini and M. Vicario. Static analysis and dynamic steering of time dependent systems using petri nets. Technical Report 28.98, University of Florence, 1998.

[19] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS '95*, 1995.

[20] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, 2000.

[21] C. Meder. The TTCT tool manual., 1997.

[22] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, January 1989.

[23] J. Sifakis and S. Yovine. Compositional specification of timed systems. In *13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96*, pages 347–359, Grenoble, France, February 1996. Lecture Notes in Computer Science 1046, Spinger-Verlag.

[24] S. Tripakis. *The formal analysis of timed systems in practice*. PhD thesis, Université Joseph Fourrier de Grenoble, 1998.

[25] S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and timed systems. In *World Congress on Formal Methods, FM'99*, 1999.

[26] S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, January 2001.

[27] DES group University of Michigan. Umdes software library. Available at: www.eecs.umich.edu/umdes/projects.html#lib.

[28] Verification of hybrid systems – VHS project. http://www-verimag.imag.fr/VHS/.

[29] H. Wong-Toi and D.L. Dill. Synthesizing processes and schedulers from temporal specifications. In *Computer-Aided Verification, CAV*, LNCS 531, 1990.

[30] W.M. Wonham. Notes on control and discrete event systems. Department of Electrical and Computer Engineering, University of Toronto, 1999.

[31] Z.H. Zhang and W.M. Wonham. STCT: An efficient algorithm for supervisory control design. In *Symposium on Supervisory Control of Discrete Event Systems (SCODES2001)*, 2001.