From control-command synchronous programs to hybrid automata

Analysis and Design of Hybrid Systems 2012, Eindhoven

Olivier Bouissou
Laboratory of Modeling and Analysis of Interacting Systems
Hybrid automata is a powerful model for designing control-command systems:

▶ Discrete switches between places encode the discrete dynamics.
▶ Continuous flow conditions encode the continuous dynamics.

Verification of linear hybrid automata is a well-treated problem with extensions to deal with the non-linear cases:

▶ SpaceEx computes guaranteed over-approximations of the reachable sets.
▶ It allows to verify the safety of such models.
Classical model for embedded systems.

An example.

The heater automata

\[
\begin{align*}
F : \dot{x} &= 9 - \frac{x}{3} \\
I : x &\leq 26 \\
F : \dot{x} &= -\frac{x}{3} \\
I : x &\geq 19
\end{align*}
\]

Running SpaceEx on it
In an industrial context
- control-command systems are designed using high-level tools (Matlab/Simulink) and validated using simulations
- the code is then generated, adding many details to the control-law

Implementation details that impact the behavior
- use of floating-point numbers;
- real-time constraints that impose the code to be synchronous.

Consequence
The execution of an embedded program may greatly differ from the execution of the high-level model.
We introduced in [BM08] the HSIMPLE language.

```plaintext
// X is a discrete variable
// t is the temperature
while (true) do
    sens.t?X;
    if (X>=25)
        a=0;
    if (X=<19)
        a=1;
    act.1!a;
    wait(0.1);
```

A hybrid program consists of:
- standard statements;
- sensor statements;
- actuator statements;
- wait statement;
- differential equations.

\[
f_0(t) = -\frac{t}{3}
\]
\[
f_1(t) = 9 - \frac{t}{3}
\]
Some executions of the program.
Comparing model and implementation.

<table>
<thead>
<tr>
<th>Model</th>
<th>Implementation</th>
</tr>
</thead>
</table>
| **Data type:**  
  only real numbers. | **Data type:**  
  real and floating-point numbers. |
| **Execution model:**  
  while(1)  
  wait for transition  
  execute transition | **Execution model:**  
  while (1)  
  read inputs  
  compute outputs  
  write outputs  
  wait for tic |
| Verification techniques for the *global* system | Verification techniques for the *discrete* system |
We want to apply formal methods on the embedded code.

The first step is to transform a program into an equivalent hybrid automata:
  
  * Automatically build a hybrid automata from a HSIMPLe program.
  * Prove the semantics equivalence.

**Big picture:**

```
// X is a discrete variable
// t is the continuous temperature
while (true) do
  sens.t/X;
  if (t>=26)
    c=0;
  if (t<19)
    c=1;
  act.!c;
  wait(0.1); // delay
```

- HSIMPLe programs
- Sample Hybrid Automata
- Hybrid Automata

\[ P \rightarrow A_P \rightarrow \Phi(A_P) \]
We want to apply formal methods on the embedded code. The first step is to transform a program into an equivalent hybrid automata:
  - Automatically build a hybrid automata from a HSIMPLE program.
  - Prove the semantics equivalence.

**Big picture:**

**HSIMPLE programs**

```
// X is a discrete variable
// t is the continuous temperature
while (true) do
  sens.t/X; // read value
  if (t>=26)
    c=0;
  else
    c=1;
  act.I/e; // acts on actuators
  wait(0.1); // delay
```

**Sample Hybrid Automata**

- **ON**
  - Equation: \( F : x = 9 - \alpha/3 \)
  - Initial condition: \( I : x \leq 25 \)
  - Transition: \( g : x \leq 19 \), \( u : x' = x \)

- **OFF**
  - Equation: \( F : x = -\alpha/3 \)
  - Initial condition: \( I : x \geq 19 \)
  - Transition: \( g : x \leq 19 \), \( u : x' = x \)

**Hybrid Automata**

\[ \Phi(\mathcal{A}_P) \]

\[ \alpha \]

\[ \tau \]
We want to apply formal methods on the embedded code.
The first step is to transform a program into an equivalent hybrid automata:
- Automatically build a hybrid automata from a HSIMPLE program.
- Prove the semantics equivalence.

Big picture:

Reach(P) ⊆ Reach(A_p) ⊆ Reach(Φ(A_p))
Step 0: preliminaries.

\[
\begin{align*}
\text{HSIMPLE programs} & \quad \rightarrow \\
\text{Sample Hybrid Automata} & \quad \alpha \rightarrow \\
\Phi(\mathcal{A}_P) & \quad \alpha \\
\end{align*}
\]
A hybrid automata is: $\mathcal{A} = (L, V, Lbl, I, F, T)$.

- The formal verification of hybrid automata over-approximates the set of reachable sets.
- A set is reachable if there is a sequence of transition leading to it.

\[
\begin{align*}
(s, l, o, g, u) & \in T \quad \sigma_v \models g \quad (\sigma_v, \sigma_v') \models u \\
\text{DISCRETE} \quad (s, \sigma_v) \xrightarrow{l} (o, \sigma_v')
\end{align*}
\]

\[
\begin{align*}
\exists \tau > 0 \quad \exists \rho : [0, \tau] \rightarrow \Sigma \quad \rho(0) = \sigma_v \\
\rho(\tau) = \sigma_v' \quad \forall t \in [0, \tau], \rho(t) \models I(s), \rho(t), \dot{\rho}(t) \models F(s) \\
\text{CONTINUOUS} \quad (s, \sigma_v) \xrightarrow{\tau} (s, \sigma_v')
\end{align*}
\]
Step 0: preliminaries.

HSIMPLE language.

while (true) do
    sens.t?X;
    if (X >= 26)
        a = 0;
    else
        if (X < 19)
            a = 1;
        act.1!a;
        wait(0.1);
    f_0(t) = -\frac{t}{3}
    f_1(t) = 9 - \frac{t}{3}

State of a hybrid program:

\[ \sigma = \sigma_d \times \sigma_c \times \sigma_a \times c \]

- \( \sigma_d : \text{DVar} \rightarrow \mathbb{F} \) for discrete variables;
- \( \sigma_c : \text{CVar} \rightarrow \mathbb{R} \) for continuous variables;
- \( \sigma_a : \text{AVar} \rightarrow \mathbb{B} \) for shared variables;
- \( c \in \mathbb{B}^m \) for the configuration for the actuators.
A transition system

\[ \sigma'_d = \sigma_d[X \mapsto \sigma_c(y)] \]
\[ (\text{sens}.y?X; P, \langle \sigma, c \rangle) \rightarrow (P, \langle \sigma', c \rangle) \]  

\[ c' = c[i \mapsto \sigma_a(b)] \]
\[ (\text{act}.i!b; P, \langle \sigma, c \rangle) \rightarrow (P, \langle \sigma, c' \rangle) \]

\[ y' = y_\infty(u) \]
\[ \sigma'_c = \sigma_c[y \mapsto y'] \]
\[ (\text{wait} \ u; P, \langle \sigma, c \rangle) \rightarrow (P, \langle \sigma', c \rangle) \]

The term \( y_\infty \) denotes the solution of the ODE:

\[
\begin{cases}
  y' &= f_c(y) \\
  y(0) &= \sigma(y)
\end{cases}
\]
Step 1: sampled hybrid automata.
A Sample Hybrid Automata (SHA) is a hybrid automata plus a sampling period $\tau$ for each location.

In each location, a discrete transition can happen only after $\tau$ seconds.

Different locations may have different sampling periods.

\[
\begin{align*}
\text{ON} & : F : \dot{x} = 9 - \frac{x}{3} \\
& : I : x \leq 26 \\
& : \tau : 0.1 \\
\text{OFF} & : F : \dot{x} = -\frac{x}{3} \\
& : I : x \geq 19 \\
& : \tau : 0.1
\end{align*}
\]

\[
\begin{align*}
x \geq 26 \\
x \leq 19
\end{align*}
\]
Definition

A SHA is a pair $S = (\mathcal{A}, \tau)$ such that:

- $\mathcal{A} = (L, V, Lbl, I, F, T)$ is a hybrid automata;
- $\tau : L \to \mathbb{R}_+^*$ maps locations to sampling periods.
We define a transition system for these automata.

Only the **Continuous** rule differs.

\[
(s, l, o, g, u) \in T \quad \sigma_v \models g \quad (\sigma_v, \sigma'_v) \models u
\]

\[
(s, \sigma_v) \xrightarrow{\tau} (o, \sigma'_v)
\]

\[
\rho(0) = \sigma_v \quad \rho(\tau(s)) = \sigma'_v \quad \forall t \in [0, \tau(s)], \rho(t), \dot{\rho}(t) \models F(s)
\]

\[
(s, \sigma_v) \xrightarrow{\tau} (s, \sigma'_v)
\]
Step 2: from SHA to hybrid automata.

HSIMPLE programs

// X is a discrete variable
// t is the continuous temperature
while (true) do
    sensit t ; // read value
    if (t > 26) then
        c := 0;
    else
        c := 1;
    end
    act ! c ; // acts on actuators
    wait (0.1); // delay

P

→

Sample Hybrid Automata

ON
F: x = 9 - x/3
l : x ≤ 25

OFF
F: x = - x/3
l : x ≥ 19

Hybrid Automata

ON
F: x = 9 - x/3
l : x ≤ 25

OFF
F: x = - x/3
l : x ≥ 19

Φ(Λ_P)
α → τ

α →
Compilation from SHA to HA.

From a SHA \((\mathcal{A}, \tau)\), we build the HA \(\mathcal{A}'\) by:

1. adding an extra variable \(t\) for time with flow \(\dot{t} = 1\) in each location;
2. replacing all invariants by \(t \leq \tau(s)\);
3. extending the guards with \(t = \tau(s)\);
4. inserting new transitions.

\[
\begin{align*}
F : \dot{x} &= 9 - x/3 \\
I : x &\leq 26 \\
\tau : 0.1 \\
\text{ON} &\quad \text{on} \\
\text{OFF} : \dot{x} &= -x/3 \\
I : x &\geq 19 \\
\tau : 0.1 \\
\end{align*}
\]
Compilation from SHA to HA.

From a SHA \((\mathcal{A}, \tau)\), we build the HA \(\mathcal{A}'\) by:

1. adding an extra variable \(t\) for time with flow \(\dot{t} = 1\) in each location;

\[
\begin{align*}
\dot{x} &= 9 - x/3 \\
\dot{t} &= 1 \\
I : \ x &\leq 26 \\
\tau : \ 0.1
\end{align*}
\]

\[
\begin{align*}
\dot{x} &= -x/3 \\
\dot{t} &= 1 \\
I : \ x &\geq 19 \\
\tau : \ 0.1
\end{align*}
\]
Compilation from SHA to HA.

From a SHA \((A, \tau)\), we build the HA \(A'\) by:

1. adding an extra variable \(t\) for time with flow \(\dot{t} = 1\) in each location;
2. replacing all invariants by \(t \leq \tau(s)\);

\[
\begin{align*}
F & : \begin{cases} 
\dot{x} = 9 - x/3 \\
\dot{t} = 1 
\end{cases} \\
I & : t \leq 0.1 \\
\tau & : 0.1 \\
\end{align*}
\]

\[
\begin{align*}
F & : \begin{cases} 
\dot{x} = -x/3 \\
\dot{t} = 1 
\end{cases} \\
I & : t \leq 0.1 \\
\tau & : 0.1 \\
\end{align*}
\]
Compilation from SHA to HA.

From a SHA \((A, \tau)\), we build the HA \(A'\) by:

1. adding an extra variable \(t\) for time with flow \(\dot{t} = 1\) in each location;
2. replacing all invariants by \(t \leq \tau(s)\);
3. extending the guards with \(t = \tau(s)\);

\[
\begin{align*}
\dot{x} & = 9 - x/3 \\
\dot{t} & = 1 \\
I & : t \leq 0.1 \\
\tau & : 0.1 \\
ON & \rightarrow OFF
\end{align*}
\]

\[
\begin{align*}
\dot{x} & = -x/3 \\
\dot{t} & = 1 \\
I & : t \leq 0.1 \\
\tau & : 0.1 \\
OFF & \rightarrow ON
\end{align*}
\]

\[
\begin{align*}
x & \geq 26 \\
t & = 0.1
\end{align*}
\]

\[
\begin{align*}
x & \leq 19 \\
t & = 0.1
\end{align*}
\]
Compilation from SHA to HA.

From a SHA ($A, \tau$), we build the HA $A'$ by:

1. adding an extra variable $t$ for time with flow $\dot{t} = 1$ in each location;
2. replacing all invariants by $t \leq \tau(s)$;
3. extending the guards with $t = \tau(s)$;
4. inserting new transitions.

\[
\begin{align*}
F : & \begin{cases}
\dot{x} = 9 - x/3 \\
\dot{t} = 1
\end{cases} \\
I : & t \leq 0.1
\end{align*}
\]

\[
\begin{align*}
F : & \begin{cases}
\dot{x} = -x/3 \\
\dot{t} = 1
\end{cases} \\
I : & t \leq 0.1
\end{align*}
\]

\[
\begin{align*}
F : & \begin{cases}
\dot{x} = 9 - x/3 \\
\dot{t} = 1
\end{cases} \\
I : & t \leq 0.1
\end{align*}
\]

\[
\begin{align*}
F : & \begin{cases}
\dot{x} = -x/3 \\
\dot{t} = 1
\end{cases} \\
I : & t \leq 0.1
\end{align*}
\]

\[
\begin{align*}
F : & \begin{cases}
\dot{x} = 9 - x/3 \\
\dot{t} = 1
\end{cases} \\
I : & t \leq 0.1
\end{align*}
\]

\[
\begin{align*}
F : & \begin{cases}
\dot{x} = -x/3 \\
\dot{t} = 1
\end{cases} \\
I : & t \leq 0.1
\end{align*}
\]
Compilation from SHA to HA.

Formally.

**Definition**

For all SHA $A = (A, \tau)$, we define $\phi(A)$ by

$$
\phi(A) = (L, V', Lbl', I', F', T')
$$

- $V' = V \cup \{t\}$ where $t$ is a fresh variable representing the time;
- $Lbl' = Lbl \cup \{\tau\}$ where $\tau$ is a fresh label;
- $\forall s \in L, \ I'(s) = t \leq \tau(s)$ and $F'(s) = F(s) \land t = 1$;
- $\forall (s, l, o, g, u) \in T, (s, l, o, g \land t = \tau(s), u \land t' = \tau(o)) \in T'$;
- $\forall s \in L, (s, \tau, s, I(s) \land t = \tau(l), t' = 0) \in T'$.

*This transformation preserves the semantics:* the reachable sets are the same (more on that later).
Step 3: from HSIMPLE to SHA.

HSIMPLE programs

```
// X is a discrete variable
// t is the continuous temperature
while (true) do
    sens.t/X; // read value
    if (t>20)
        c:=0;
    if (t<19)
        c:=1;
    act.1!; // acts on actuators
    wait(0.1); // delay
```

Sample Hybrid Automata

```
ON
F: \dot{x} = 9-x/3
I: x \leq 25
\begin{align*}
g: x \leq 19 \\
u: u' = x
\end{align*}
OFF
F: \dot{x} = -x/3
I: x \geq 19
\begin{align*}
g: x \leq 19 \\
u: u' = x
\end{align*}
```

Hybrid Automata

```
ON
F: \dot{x} = 9-x/3
I: x \leq 25
\begin{align*}
g: x \leq 19 \\
u: u' = x
\end{align*}
OFF
F: \dot{x} = -x/3
I: x \geq 19
\begin{align*}
g: x \leq 19 \\
u: u' = x
\end{align*}
```

\[ P \xrightarrow{\alpha} A_P \xrightarrow{\alpha} \Phi(A_P) \]
We consider hybrid programs of the form:

```plaintext
init(X); // Initialization of discrete variables
while (true) do
    sens.t1?X1; sens.t2?X2;... // Sensing the input values
    treat(X, c);
    act.1!c1; act.2!c2;... // Computing the output values
    wait(u);
    \{f_c | c ∈ \mathbb{B}^m\} // Waiting for next sampling
```

- Each actuator configuration $c ∈ \mathbb{B}^m$ represents a discrete state.
- There is a jump from $c = (0, 1)$ to $c' = (1, 0)$ iff the statements:
  - act.1!1;
  - act.2!0;

are executed.
Detecting when statements are executed.

- The transitions will take place when the act statements are executed.
- We must define when this is the case.

**Definition**

For each act statement, we define the function $F_j : \Sigma_c \times \mathbb{B}^m \rightarrow \mathbb{B}$ such that $F_j(\sigma_c, c) = 1$ if the statement is executed with input $\sigma_c$ and in the configuration $c$.

```plaintext
while (true) do
    sens.t? X;
    if (X >= 26)
        c = 0;
    if (X <= 19)
        c = 1;
    if (c == 0)
        act.1!0;
    else
        act.1!1;
    wait(0.1);
```

$F_1(\sigma_c) = \sigma_c(t) \geq 26$

$F_2(\sigma_c) = \sigma_c(t) \leq 19$
We consider a program with 2 actuators, so four states.

- We thus have 4 functions:
  - $F_1$ that detects when actuator 1 is turned on (act .1!1)
  - $F_2$ that detects when actuator 2 is turned on (act .2!1)
  - $F_3$ that detects when actuator 1 is turned off (act .1!0)
  - $F_4$ that detects when actuator 2 is turned off (act .2!0)
Abstraction of the detection functions.

- It is often the case that the function $F_i$ are not computable.
- We can use interval analysis to compute an over-approximation of it.
- In such a case, we replace the conditions $F_i(\sigma_c)$ by their abstract version.
- More details are in the paper.
Step 4: equivalence theorem.

Reach(P) ⊆ Reach(/tcp) ⊆ Reach(Φ/tcp)
What we have so far.

- A model for embedded programs in an extension of imperative programs.
  - Explicit statements for sensors and actuators.
  - Description of the environment using differential equations.

- Hybrid automata as a higher level model for hybrid systems.
  - Discrete transitions are modeled by state changes.
  - Abstract away the notions of floating point numbers and sampling periods.

- A translation from HSIMPLE to SHA and a translation from SHA to HA.

**Question**
Are these translations correct?
Let $P$ be a hybrid program and $A_P$ be the constructed SHA.

**Remember**

$P$ and $A_P$ have the same set of continuous variables but $P$ also has discrete variables.

- $\text{Reach}(P)$ is the set of states $\sigma = (\sigma_c, \sigma_d, \sigma_a)$ within the transitive closure of $\rightarrow$.
- $\text{Reach}(A_P)$ is the set of states $\sigma = (\sigma_c, q)$ within the transitive closure of $\rightarrow_\tau$.

**Theorem 1.**

$$\Pi_c(\text{Reach}(P)) \subseteq \Pi_c(\text{Reach}(A_P))$$
Correctness theorem 2.

Correctness of the translation from SHA to HA.

Let $\mathcal{A}$ be a SHA and $\Phi(\mathcal{A})$ be the constructed HA.

Remember

$\mathcal{A}$ can only make temporal evolution of duration $\tau(l)$ in each location $l$.

Consequence: reachable sets will only be equal at sampling times.

Theorem 2.

$$\text{Reach}(\mathcal{A}) = \downarrow \text{Reach}(\Phi(\mathcal{A}))$$

$$= \left\{ (s, \sigma_v) \mid \exists (s', \sigma'_{v}) \in \text{Reach}_{\mathcal{A}'} : \sigma'_{v}(t) = \tau(s) \text{ and } \forall v \in V, \sigma'_{v}(v) = \sigma_{v}(v) \right\}$$
We presented a new model for embedded programs which is very close to existing programming languages.

We presented a complete translation from HSIMPLE to hybrid automata, using an intermediate formalism.

We showed that the resulting automata can be used to prove the absence of errors in the embedded program.

**Perspectives**

- Extend this transformation to programs with *numerical* actuators.
- Develop better abstraction techniques to construct automata that are better suited for verification tools.
- Limit the number of discrete states using abstract interpretation techniques (as in [BJ10]).