Generating Sound, Resource-aware Embedded-code from Hybrid Systems Models

Madhukar Anand, Jesung Kim, Sebastian Fischmeister, and Insup Lee
University of Pennsylvania

Mar 16, 2006
Automotive Software Workshop, San Diego
Motivation

- **Embedded System** – Digital programs interacting with analog environment.
  - Examples: Motion Controller, Temperature Control, etc.
- With more sophisticated tasks, there is a need for a sound discipline in writing software
- Benefits of model-based design
  - Detecting errors in early stages
  - Powerful analysis and formal guarantees
  - Automatic code generation
Hybrid Systems Review

- Hybrid systems models, which allow mixing state-machine based discrete control with differential equation based continuous dynamics, are appropriate for design and analysis of such systems.

  - Continuous dynamics: Differential Equations
    - Differential, Algebraic equations: evolution of a variable over time
      - $x' = 1$ : constant increase ($x = t$)
      - $x' = x$: exponential increase ($x = e^t$)

  - Discrete control: Finite State Machine
    - Transition: switching of equations
      - $x' = 1$ ($x \geq 20$) $x' = -1$
Tools for Hybrid Systems

- The Hybrid Toolbox
  - is a Matlab/Simulink toolbox for modeling and simulating hybrid dynamical systems

- HYTECH
  - is an automatic verification tool for hybrid systems from UC Berkeley.

- CHARON
  - modular specification of interacting hybrid systems based on the notions of hierarchy of agent and modes developed at Penn.
Ensuring Correctness in Embedded Software through Model-based Design

- There exist appropriate models for embedded systems design
  - Hybrid Systems, Timed Automata, etc.
- We also have tools that support automatic code generation from such models
  - Charon, Simulink, etc.
- However, a number of concerns w.r.t correctness still remain
  - What guarantees can we provide in the implementation with respect to the model?
  - How do we ensure that optimal use of resources in this process?
**Guiding Example: Trailing Vehicle**

- **State Transition Diagram**
  - **State 1 (q0)**: $x_1 = v_1$, $v_1 = u$
  - **State 2 (q1)**: $x_2 = v_2$, $v_2 = 0$
  - **State 3 (q2)**: $x_2 = v_2$, $v_2 = 1$
  - **State 4 (q3)**: $x_2 = v_2$, $v_2 = -1$

- **Constraints**
  - $G_1: x_1 - x_2 \geq d_{\text{max}}$
  - $G_2: x_1 - x_2 \leq d_{\text{min}}$
  - $G_3: x_1 - x_2 \in (d_{\text{min}}, d_{\text{max}})$
  - $G_4: x_1 - x_2 \in (d_{\text{min}}, d_{\text{max}})$
  - $G_5: x_1 - x_2 \geq d_{\text{max}}$
  - $G_6: x_1 - x_2 \leq d_{\text{min}}$
Switching Discrepancies - I

- Faulty transitions in the code is a transition that has occurred in the code, but there is no corresponding transition in the model.

- Example: Consider the trailing vehicle with $d_{\text{min}} = 0.1$, $d_{\text{max}} = 0.5$, communication delay = 0.03s.

<table>
<thead>
<tr>
<th>$t$</th>
<th>$x_1(A_1)$</th>
<th>$x_1(A_2)$</th>
<th>$x_2(A_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.06</td>
<td>0.3072</td>
<td>0.3072</td>
<td>0.2018</td>
</tr>
<tr>
<td>0.10</td>
<td>0.3172</td>
<td>0.3072</td>
<td>0.2018</td>
</tr>
<tr>
<td>0.12</td>
<td>0.3172</td>
<td>0.3072</td>
<td>0.2072</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Faulty Transition to $q_3$ (deceleration)
A missed transition is a transition that must be taken to satisfy the invariant, but is not taken by the code.

Example: Consider the trailing vehicle with $d_{\text{min}} = 0.25$, $d_{\text{max}} = 0.5$, communication delay = 0.03s,

<table>
<thead>
<tr>
<th>$t$</th>
<th>$x_1(A_1)$</th>
<th>$x_2(A_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.15</td>
<td>0.48</td>
<td>0.0</td>
</tr>
<tr>
<td>0.25</td>
<td>0.98</td>
<td>0.0</td>
</tr>
<tr>
<td>0.30</td>
<td>0.98</td>
<td>0.6862</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Missed Transition to $q_1$
Code generation framework

CHARON Model

```java
agent()
{
}

mode()
{
}
```

ISA: MIPS

CPU Speed: 384Mhz

Memory: 32MB

Platform Description

<table>
<thead>
<tr>
<th>Constraint Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) Switching Semantics</td>
</tr>
<tr>
<td>b) Resource Constraints</td>
</tr>
</tbody>
</table>

Code Generator

Executable Code

| 010101 |
| 010010 |
| 101010 |
| 101010 |
Correctness Criteria

- **(Faithful Implementation)** We define that the code is a faithful implementation of the model if there exists a constant error bound on the variables in the code and the discrete states are equivalent modulo the timing of execution in the code. [ISORC’05]
  - To ensure this condition, we will have ensure that faulty transitions are addressed
  - However, this condition is hard to implement in a system with communication delays and multiple frequencies of update.
  - Relax the faithful implementation criteria: States should be equivalent within the maximum skew and communication delay.
    - *(Relative Faithful Implementation)*
Preventing Faulty Transitions [HSCC’04]

- Exploit non-determinism in guard conditions
  - A transition can (but need not immediately) be taken when the guard is enabled
- Assume a maximum error due to discretization
  - Our focus: errors due to different sampling rates of communicating models
- “Tighten” the guard to prevent transition errors even in the presence of the given maximum error
  - The trace of discrete transitions is equivalent to that of the model while errors in valuation are bounded
Instrumentation Illustrated

check transition conditions

γ maximum error

false-enabled transition condition

guard of the model

instrumented guard of the code
Switching Discrepancies-I (Recap)

- Example: Consider the trailing vehicle with $d_{\text{min}} = 0.1$, $d_{\text{max}} = 0.5$, communication delay = 0.03s, Sampling frequencies are (0.1s, 0.06s)

<table>
<thead>
<tr>
<th>$t$</th>
<th>$x_1(A_1)$</th>
<th>$x_1(A_2)$</th>
<th>$x_2(A_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.06</td>
<td>0.3072</td>
<td>0.3072</td>
<td>0.2018</td>
</tr>
<tr>
<td>0.10</td>
<td>0.3172</td>
<td>0.3072</td>
<td>0.2018</td>
</tr>
<tr>
<td>0.12</td>
<td>0.3172</td>
<td>0.3072</td>
<td>0.2072</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Faulty Transition to $q_3$ (deceleration)
Example of Instrumentation

- Here, if the time interval under consideration is [0.05,1], then, \((x_1-x_2)'\) will also lie in [0.05,1].
  - Communication delay is 0.03, and the maximum roundoff, truncation error is 0.001
  - The guard \(x_1-x_2=0.1\) becomes \(x_1-x_2 \leq (0.1-0.001-0.1-(0.02+0.03))\) i.e., \(x_1-x_2 \leq 0.049\)
  - Therefore, there will be no faulty transitions at \(t=0.12\)
Missed Transitions

- A *faithful* implementation of the code must not only consider faulty transitions but also the possibility of *missed* transitions. A missed transition is a transition that must be taken to satisfy the invariant, but is not taken by the code.

- Missed transitions occur due to
  - Insufficient sampling
  - Scheduling,
  - Communication delays
Sufficient Criteria for preventing Missed Transitions

- **Theorem (Informal):** Consider a distributed system of agents with an eager policy on transitions. If the guards are instrumented and the guard and the invariant overlap by at least $2h_j$ where $h_j$ is the frequency of evaluation of the guard then, there will be no missed transitions.
Switching Discrepancies -II (Recap)

- Example: Consider the trailing vehicle with \(d_{\text{min}} = 0.25\), \(d_{\text{max}} = 0.5\), communication delay = 0.03s, sampling period is (0.25s,0.15s)

<table>
<thead>
<tr>
<th>(t)</th>
<th>(x_1(A_1))</th>
<th>(x_2(A_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.15</td>
<td>0.48</td>
<td>0.0</td>
</tr>
<tr>
<td>0.25</td>
<td>0.98</td>
<td>0.0</td>
</tr>
<tr>
<td>0.30</td>
<td>0.98</td>
<td>0.6862</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Missed Transition to \(q_1\)
The guard being tested after instrumentation is 
\((x_1 - x_2)\)' in [0.3, 0.45].

Now, if the dynamics of the problem is such that,
\[ 0.45 < (x_1 - x_2)' < 0.5. \]

- If it grows as fast as 0.5, then, the guard will be true in (0.16, 0.41)
- If it grows as 0.45, then, the guard will be true in (0.167, 0.45)
- The intersection of these is 0.243 < 2(0.15). [Does not meet the sufficient condition]
While it is important to ensure the semantics of the model, we also need to see if they can be supported on the platforms.

- Choose the sampling times such that
  - No switching discrepancies
  - Ensure that these can be supported with resources available.
    - Use only minimum amount of resources to achieve this.
Algorithm for optimal sampling periods

- **Smallest-K**
  - While \( \sum_{M(j)=N}^{M} \frac{W_j}{k \rho_j} > \alpha \land |RESOURCE-CHK(p)| = false \)
    - \( k := k + 1 \)

- **Select-periods-node** returns smallest \( k \) given the maximum and minimum amount of utilization supported.

- **Select-periods** then searches for a \( k \) from the minimum utilization such that there no switching discrepancies.
Implementation

- Code generator translates CHARON models into C++ code
  - Each object of CHARON is mapped to a C++ class
- Generated code is compiled by the target compiler along with additional code
  - Run-time scheduler: determines ordering of primitives
  - API interface routines: associates variables to devices
Conclusions

- We have presented a framework for automatic code-generation from hybrid system models for embedded systems
  - In particular, handling switching discrepancies
    - Faulty transitions – Instrumentation
    - Missed transitions – Sufficient condition
  - We choose a sampling periods such that,
    - Avoid switching discrepancies
    - Optimal usage of resources to achieve this.
- We are currently working on completing the implementation and in future, we plan to address the problem of dynamicity in resources of different agents.