Model-based Code Generation – CHARON Case Study

Jesung Kim
in collaboration with
Rajeev Alur, Yerang Hur, Franjo Ivancic,
Insup Lee, and Oleg Sokolsky

University of Pennsylvania
Contents

• Motivation
• CHARON overview
• Example: Sony dog
• Code generation procedure
• Soundness of generated code
• Preventing switching errors
• Conclusion
Motivation

- Formal specification of hybrid systems
  - Subject to formal verification
- Automatic generation of the code
  - Elimination of coding errors
- Formulation of the differences between the model and the generated code
  - Bounded difference desired
- Case study in a robotic platform (AIBO)
  - Code generation for fairly complicated systems
CHARON Framework

- Language for formal specification of hybrid systems
  - Analog variable
  - Differential/algebraic equation
  - Discrete state transition
- Hierarchical specification
  - Architectural hierarchy
    - agent: communicating entity
  - Behavioral hierarchy
    - mode: hierarchical state machine with continuous dynamics
- Simulation
- Model checking
- Code generation
- Run-time verification
Example: Four Legged Robot

- Control objective
  - \( v = c \)

- High-level control laws
  \[
  \begin{align*}
  \dot{x} &= -v \\
  x &\geq -\text{stride} / 2 \\
  \dot{y} &= -kv \\
  x &\leq \text{stride} / 2 \\
  \dot{y} &= kv
  \end{align*}
  \]

- Low-level control laws
  \[
  j_1 = \arctan(x/y) - \arccos\left(\frac{x^2 + y^2 + L_1^2 - L_2^2}{2L_1\sqrt{x^2 + y^2}}\right)
  \]
  \[
  j_2 = \arccos\left(\frac{x^2 + y^2 + L_1^2 - L_2^2}{2L_1L_2}\right)
  \]

CHARON Code Generator

- CHARON code generator translates CHARON models into C++ code
  - Each object of CHARON models is translated into a C++ structure
- Generated C++ code is compiled by the target compiler along with additional code
  - Run-time scheduler: invokes active components periodically
  - API interface routines: associates variables with devices
Translation of CHARON models

- Analog variable
  - C++ class: read and write to variables can be mapped to system API
- Differential equation
  - Euler’s method: \( x' = 10 \rightarrow x += 10 \times h \) (h: step size)
  - Runge-Kutta method
- Algebraic equation
  - Assignment statement executed in a data dependency order
    - \( x == 2y; y == 2z \rightarrow y = 2z; x = 2y; \)
- Invariant
  - Assertion statement for runtime verification
  - Instrumented for safer check
- Discrete transition
  - If-then statement
    - urgent transition policy
  - “Instrumented” if-then statement
    - not-so-urgent transition policy
- Mode
  - C++ class: collection of variables, equations, transitions, and reference to submodes
- Agent
  - C++ class: interleave execution of each step of subagents
Soundness of Generated Code

• Code may behave differently from the model:
  – Numerical errors (numerical integration)
  – Floating-point errors (fixed precision arithmetic)
  – Switching errors
    • Missed switching: enabled transition is missed due to discrete testing of switching conditions [LCTES 2003]
    • Invalid switching: disabled transition is evaluated as enabled due to different update frequencies of shared variables [HSCC 2004]

  Properties held in the model may not be guaranteed in the code even if automatically generated!

• How to check / prevent switching errors?
  – Exploit non-determinism of the model
    • Check the model if it gives sufficient room for the code to take a switch
  – Design the switching policy
    • Take switching conservatively to prevent an invalid switch

Switch Miss

\[ d(x) = \begin{cases} 1 & \text{if } x \leq 25 \\ -1 & \text{if } x \geq -25 \end{cases} \]

position of the tail

\[ \begin{array}{c}
\text{d(x) = 1} \\
\text{x == 25} \\
\text{x <= 25}
\end{array} \quad \begin{array}{c}
\text{d(x) = -1} \\
\text{x >= -25} \\
\text{x == -25}
\end{array} \quad \begin{array}{c}
\text{d(x) = 1} \\
\text{x <= 25}
\end{array} \]

invariant violation
Analysis of Switch Miss

- Given a step size $\Delta$ and a CHARON model, check whether for all pairs of invariants $I$ and guards $G$ of switch ($= \Delta$-lookahead agent)

  $\text{Post}_\Phi (I \setminus G, \Delta) \subseteq I$

- If so, the generated code will produce a valid execution trace
  - The code performs a switch immediately when the guard is true

- Otherwise, even for the generated code we cannot guarantee a valid execution trace.
Invalid Switch

- Variables $x_1, x_2, \ldots$ are updated by different steps $\Delta_1, \Delta_2, \ldots$
- Code evaluates switching conditions $f(x_1, x_2, \ldots)$ by referencing $x_1(t_1), x_2(t_2), \ldots$
  - $(x_1(t_1), x_2(t_2), \ldots)$ may not on the trajectory of $(x_1, x_2, \ldots)$ unless $t_1 == t_2 == \ldots$

Diagram:
- $x_2$ vs $x_1$
- Guard of the model
- Instrumented guard of the code
- Maximum error
- False-enabled transition condition
Preventing Invalid Switch through Guard Instrumentation

- Exploit non-determinism in the guard conditions
  - Transition can (but need not immediately) be taken when the guard is enabled

- Compute a maximum error due to asynchrony
  - \( \max(|x'| \Delta) \)
    - assumption: independent dynamics and rectangular guard sets

- “Tighten” the guards such that transitions are not falsely enabled at the presence of the maximum error
  - Trace of discrete states is equivalent to that of the model
Example

\[ x := -150 \]
\[ z := 0 \]
\[ \dot{x} = -100z + 200 \]
\[ \dot{z} = 1 \]
\[ z > 3 \]
\[ \dot{x} = -1 \]
\[ \dot{z} = 1 \]

\[ y := 0 \]
\[ \dot{y} = 1 \]
\[ x \geq 50 + \gamma, y > 2 \]
\[ \dot{y} = 0 \]

\[ \gamma = (\dot{x}(t) \cdot \max(h))_{\text{max}} \]
\[ = (-100t + 200) \cdot 0.002_{\text{max}} \]
\[ = 0.4 \]

Transition Policy

- lead to future invariant violation
  - model checking
- sound behavior performing better
- sound behavior
- inconsistent behavior due to noisy values
  - instrumentation

*guard*

*invariant*

*trajectory*

*event detection*
Design Flow of our Framework

- Communicating Hybrid Automata
  - Continuous time domain
- Discretized Communicating Hybrid Automata
  - Single discrete time domain
- Instrumented Communicating Hybrid Automata
  - Each automaton having its own discrete time domain
  - Guard “instrumented” to prevent errors due to different time domains
- Code
  - Machine executable
CHARON code generator automates translation of complicated hybrid systems specification into modular C++ code.

Each C++ module can be mapped to a periodic task of RTOS of the target system to approximate continuous update.

Even automatically generated code is not semantically equivalent to the original model:
- Numerical errors, floating-point errors, switching errors…

Switching errors due to different update rates of shared variables can be prevented through instrumentation of the switching conditions.