Formal Verification of Simulink/Stateflow Diagrams Using Theorem Prover

Naijun Zhan

State Key Lab. of Computer Science
Institute of Software, Chinese Academy of Sciences

Joint work with Liang Zou, Shuling Wang, Martin Fränzle and Shengchao Qin

AADL standards meeting, Orlando, July. 7-10, 2014
Outline

1. Background
2. Introduction to Simulink, HCSP, and HHL
3. Translating Simulink Diagrams into HCSP
4. Implementation
5. Case Study
Simulink/Stateflow

- Simulink is an environment for the model-based analysis and design of signal-processing systems.
- Stateflow is an integrated toolbox of Simulink by adding facilities for modeling and simulating reactive systems by means of hierarchical statecharts, extending Simulink’s scope to event-driven and hybrid forms of embedded control.
- Simulink/Stateflow have become a de-facto standard in the embedded systems industry.
- Due to the inherent incompleteness of system validation, complementing simulation by formal verification would be desirable.

**A prerequisite for formal verification is to have a formal semantics of Simulink/Stateflow**

- In this talk, we mainly focus on formal verification of Simulink diagrams, formal verification of Simulink/Stateflow can be seen [Zou et al 2014].
Related work

- Operational semantics of Simulink by Bouissou et al [Bouissou 2012]
- Denotational semantics and operational semantics for Simulink/Stateflow by Hamon et al [Hamon 2004, Hamon 2005, Hamon 07]
- Translation from discrete-time Simulink models to Lustre by Tripakis [Tripakis 2003, Tripakis 2004, Tripakis 2005]
- Translation from discrete-time Simulink diagrams to Circus by Cavalcanti et al [Cavalcanti 2005]
- Translation from a subset of Simulink into input language of model checker NuSMV by Meenakshi et al [Meenakshi 2006]
- Translation from Simulink models (with restricted continuous blocks) to a real-time specification language Timed Interval Calculus (TIC) by Dong et al [Dong 2009]
This talk addresses

- How to encode **Simulink** diagrams into **Hybrid CSP** with a full automatic tool **Simulink2HCSP**

- How to utilize a **HHL Prover** to verify the translated HCSP models

- Demonstrate our approach on a combined scenario originating from **CTCS-3**
Outline

1. Background

2. Introduction to Simulink, HCSP, and HHL

3. Translating Simulink Diagrams into HCSP

4. Implementation

5. Case Study
Features of Simulink

1. A software package for modeling, simulating, and analyzing dynamical systems
2. Supports linear and nonlinear systems, modeled in continuous time, sampled time, or their combination
3. Provides a GUI for building models as block diagrams, using click-and-drag mouse operations
4. Simulink models are hierarchical

An Example of Simulink

- A Simulink diagram consists of a set of blocks, subsystems and wires;
- Blocks and subsystems cooperate by wires between;
- User-defined parameters of a block to alter its functions.
Hybrid CSP

Related work

Hybrid CSP (HCSP) due to He [He 1994], Zhou et al [Zhou et al, 1995], is an extension of CSP by introducing differential equation to describe continuous evolution and three kinds of interruptions to model the interaction between continuous evolution and discrete jumps.

Hybird CSP (cont’d)

Syntax of HCSP

\[ P ::= \text{skip} \mid x := e \mid \text{wait } d \mid \text{ch}?x \mid \text{ch}!e \mid P; Q \mid B \rightarrow P \mid P \sqcup Q \mid P^* \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \triangleright_d Q \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \triangleright \bigwedge_{i \in I}(\text{ch}_i^* \rightarrow Q_i) \]

\[ S ::= P \mid S \parallel S \]

Here \( ch, ch_i \in \Sigma, \) \( ch_i^* \) stands for a communication event, i.e., either \( \text{ch}i?x \) or \( \text{ch}i!e, \) \( x, s \in V, \) \( B \) and \( e \) are Boolean and arithmetic expressions, \( d \) is a non-negative real constant, \( P, Q, Q_i \) are sequential processes, and \( S \) stands for a system, i.e., an HCSP process.
Hybrid Hoare Logic

- **Hybrid Hoare Logic (HHL)** was first proposed in [Zhou et al 2010], which is an extension of **Hoare logic** to hybrid systems.
- The assertion logic of HHL consists of two parts: the First Order Logic (**FOL**) and Duration Calculus (**DC**).
- A **Hoare triple** for a sequential process $P$ is of the form \{Pre\}$P$\{Post; HF\}, where $Pre, Post$ represent **pre-/post-condition**, and $HF$ **history formula**.
- The proof system of HHL consists of the following three parts:
  - axioms and inference rules for **FOL**
  - axioms and inference rules for **DC**, and
  - axioms and inference rules for **the constructs of HCSP**
- A theorem prover based on **Isabelle/HOL** has been implemented.
Outline

1. Background
2. Introduction to Simulink, HCSP, and HHL
3. Translating Simulink Diagrams into HCSP
4. Implementation
5. Case Study
Sketch of the translation

1. Customize Hybrid CSP patterns for each type of Simulink blocks.
2. Partition a diagram into continuous and discrete parts.
3. Translate the resulted continuous and discrete parts, respectively.
4. Translate subsystems.
5. Translate user options.
# HCSP Patterns for Simulink Blocks

## Pattern for continuous blocks

\[
\mathcal{PC}(\text{init, ps}) \triangleq out := \text{init}; P^*
\]

\[
P \triangleq \langle F_1(out, out, in, ps) = 0 & B_1(in, ps) \rangle \trianglerighteq \bigwedge_{i \in I}(io_i \rightarrow \text{skip});
\]

\[
\ldots;
\]

\[
\langle F_m(out, out, in, ps) = 0 & B_m(in, ps) \rangle \trianglerighteq \bigwedge_{i \in I}(io_i \rightarrow \text{skip})
\]

## Pattern for discrete blocks

\[
\mathcal{PD}(\text{init, ps, n}) \triangleq out := \text{init}; P^*
\]

\[
P \triangleq cin?in; P_{\text{comp}}; cout!out; \text{wait st}
\]

\[
P_{\text{comp}} \triangleq B_1(in, ps) \rightarrow P_{\text{comp}_1}(in, out, ps);
\]

\[
\ldots;
\]

\[
B_m(in, ps) \rightarrow P_{\text{comp}_m}(in, out, ps)
\]
Computing Inherited Sample Times

Sample Times of Blocks and Wires

- Sample times of blocks are set by user, which define how frequently the computation is taken.
- Sample time of a wire is equal to the sample time of its source block.
- Sample time ranges over non-negative reals, it is set to $-1$ when it is inherited.

Computing inherited sample times

1. For every block of a diagram with inherited sample times, if all sample times of its input wires are not inherited, set its sample time to the \text{GCD} of these sample times.

2. Repeat step 1 until all blocks have been done.
Algorithm 1 Computing inherited sample times

Require: A diagram $diag$
Ensure: Calculate all determined sample times in $diag$

1: for (flag ← true; flag; skip) do
2:     flag ← false;
3:     for all $b$ in $diag$ do
4:         if $\neg known(b) \land allKnown(b.srcBlocks())$ then
5:             $b.st \leftarrow GCD(b.srcBlocks())$;
6:             flag ← true;
7:         end if
8:     end for
9: end for
Partition a Diagram into Continuous and Discrete SCCs

Strategy

- Wires between continuous blocks are modelled as shared variables, and the two connected continuous blocks are put into one SCC;
- Wires between a continuous block and a discrete block are modelled as a channel, and the two blocks are put into two different SCCs;
- Wires between discrete blocks are modelled either as shared variables, if these blocks are centralized, therefore put into one SCC, or as coordinators if these blocks are distributed, thus any two of them cannot be put into one SCC.
Algorithm 2 Separating diagrams

Require: A diagram \textit{diag} \\
Ensure: Return a partition \textit{partition} of the diagram \\
1: \textbf{new} \textit{partition} \\
2: \textbf{for all} \textit{block} in \textit{diag} \textbf{do} \\
3: \hspace{1em} \textbf{if} \ !\textit{visited}(\textit{block}) \textbf{then} \\
4: \hspace{2em} \textbf{new} \textit{scc}; \textit{scc}.\text{add}(\textit{block}); \textit{setVisited}(\textit{block}); \\
5: \hspace{2em} \textbf{new} \textit{bs}; \textit{bs}.\text{add}(\textit{block}); \\
6: \hspace{2em} \textbf{while} \ !\textit{bs}.\text{empty}() \textbf{do} \\
7: \hspace{3em} \textit{b} \leftarrow \textit{bs}.\text{top}(); \\
8: \hspace{3em} \textbf{for all} \textit{cb} in \textit{b}.\text{getConBlocks}() \textbf{do} \\
9: \hspace{4em} \textbf{if} \ \textit{isShared}(\textit{b}, \textit{cb}) \wedge !\textit{scc}.\text{contains}(\textit{cb}) \textbf{then} \\
10: \hspace{5em} \textit{bs}.\text{add}(\textit{cb}); \textit{scc}.\text{add}(\textit{cb}); \textit{setVisited}(\textit{cb}); \\
11: \hspace{4em} \textbf{end if} \\
12: \hspace{4em} \textbf{end for} \\
13: \hspace{2em} \textit{bs}.\text{remove}(); \\
14: \hspace{2em} \textbf{end while} \\
15: \hspace{2em} \textit{partition}.\text{add}(\textit{scc}); \\
16: \hspace{1em} \textbf{end if} \\
17: \textbf{end for}
Translating Continuous Diagrams

1. A continuous diagram is initialised by initialising all of its blocks independently.

2. Continuous evolution of a continuous diagram behaves as the product of all continuous evolutions of the blocks in the diagram, as signals produced by different blocks are independent. Hence, the continuous evolution can be represented by Cartesian product.

3. Communications of a continuous diagram are always enabled in its HCSP pattern, but they are represented as communication interrupts.
Algorithm 3 Translating continuous diagrams

Require: A continuous diagram \textit{diag}

Ensure: Return an HCSP process \textit{proc}

\begin{enumerate}
\item \textit{init}.\texttt{setEmpty}(); \textit{comms}.\texttt{setEmpty}()
\item \textbf{for all} \textit{block} \textbf{in} \textit{diag} \textbf{do}
\item \textit{init} $\leftarrow$ \textit{init} $\triangleright$ \texttt{getInit}($\textit{block}$)
\item \textit{comms} $\leftarrow$ \textit{comms} $\cup$ \texttt{getComms}($\textit{block}$)
\item \textbf{end for}
\item \textit{procR}.\texttt{setEmpty}()
\item \textit{newDiffs} $\leftarrow$ Cartesian($\textit{diag}.\texttt{getDiffs}()$)
\item \textbf{for all}($d q, b$) \textbf{in} \textit{newDiffs} \textbf{do}
\item \textit{procR} $\leftarrow$ \textit{procR} $\triangleright \langle d q \& b \rangle$ $\triangleright \big[ i o \in \textit{comms} i o \rightarrow \text{skip} \big]$ 
\item \textbf{end for}
\item \textit{proc} $\leftarrow$ \textit{init} $\triangleright$ ($\textit{procR}$)*
\end{enumerate}
Translating Discrete Diagrams

1. Find out a causal relation among these discrete blocks in the discrete diagram.

2. A discrete diagram behaves as a sequential composition of the behaviours of all of its blocks by the causal relation.

3. The sample time of a discrete diagram is the GCD of the sample times of its blocks.

4. Because a discrete diagram can only communicate with continuous diagrams and a continuous diagram is always ready for communications, so all inputs and outputs of a discrete diagram can be sequentially composed before or after computation of the diagram, respectively.
Algorithm 4 Translating discrete diagrams

Require: A discrete diagram $diag$
Ensure: Return an HCSP process $proc$

1: $odiag \leftarrow order(diag);$ init.setEmpty();
2: $cin.setEmpty();$ cout.setEmpty();
3: for all block in $odiag$ do
4: \quad $init \leftarrow init \; \bowtie \; getInit(block)$
5: \quad $cin \leftarrow cin \; \bowtie \; getCin(block)$
6: \quad $cout \leftarrow cout \; \bowtie \; getCout(block)$
7: end for
8: $bc \leftarrow GCD(odiag.getst())$
9: $procR.setEmpty()$
10: for all block in $odiag$ do
11: \quad $procR \leftarrow (procR \; \bowtie \; (block.st \mid t \rightarrow block.comp))$
12: end for
13: $procR \leftarrow cin \; \bowtie \; procR \; \bowtie \; cout$
14: $procR \leftarrow procR \; \bowtie \; (temp := t) \; \bowtie \; (t = 1 \& t < temp + bc)$
15: $proc \leftarrow init \; \bowtie \; (t := 0) \; \bowtie \; (procR)^*$
Translating Normal Subsystems

1. A normal subsystem does not offer anything except for hierarchy.
2. We flatten the subsystem directly by connecting the in-ports and out-ports attached to it to the corresponding in-ports and out-ports attached to the blocks inside it.
Translating Triggered Subsystems

Triggered subsystems

- A **triggered subsystem** additionally contains a triggered block, and there is a corresponding input triggering signal targeting at the subsystem. The sample times of all the other input signals are equal to the one of the triggering signal.

- Three types of triggering events: **rising**, **falling** and **changing** of the sign of the signal.

Translating Triggered Subsystems

1. Flatten the subsystem except for the triggering signal and the triggered block, and translate it as a discrete diagram, but without sample time.

2. So, the whole subsystem is $procR \leftarrow tri? \circ cin \circ procR \circ cout$

3. Accordingly, the outside block is updated as follows:
Translating Triggered Subsystems (cont’d)

Updated Continuous Pattern

\[
\langle F_1(\dot{out}, out) = 0 & B_1 \land \neg B_{tri} \rangle \triangleright \cdots ;
\]
\[
\cdots
\]
\[
\langle F_m(\dot{out}, out) = 0 & B_m \land \neg B_{tri} \rangle \triangleright \cdots ;
\]
\[
B_{tri} \rightarrow tri!;
\]
\[
\langle F_1(\dot{out}, out) = 0 & B_1 \land B_{tri} \rangle \triangleright \cdots ;
\]
\[
\cdots
\]
\[
\langle F_m(\dot{out}, out) = 0 & B_m \land B_{tri} \rangle \triangleright \cdots
\]

Updated Discrete Pattern

We update discrete pattern by revising \( P_{comp} \) as follows.

\[
osig := out_{tri}; P_{comp}; B_{tri} \rightarrow tri!
\]
Translating Enabled Subsystems

An enabled subsystem is activated during the time interval when its enable signal is above 0.

1. Since all blocks of an enabled subsystem are activated by the enable signal, an enabled subsystem will become a normal subsystem if we distribute enable condition to all blocks in it.

2. For every continuous block, we add $en > 0$ as a conjunction with the domains of all its differential equations, meanwhile, add an extra differential equation $\langle \dot{out} = 0 \& en \leq 0 \rangle$ to the block.

3. For every discrete block, we update its computation $P_{comp}$ to $en > 0 \rightarrow P_{comp}$. 
Translating User Options

- Options in Separating the Diagram
- Options in Abstraction (by rewriting semantics of blocks)

More can be referred to the paper [Emsoft13].
Outline

1. Background
2. Introduction to Simulink, HCSP, and HHL
3. Translating Simulink Diagrams into HCSP
4. Implementation
5. Case Study
Verification Architecture for Simulink Models

Simulink models → Translator → Annotated HCSP models

Refined assertions and goal → HHL specifications → Verification conditions

HHL prover

Interactive prover → Automated prover

Figure: Verification Architecture for Simulink Models
Translator

Two Steps for Implementing the Translator

1. Take a Simulink model as input, and build an object for the model in Java.

2. Generate an annotated HCSP model as output, which is written in the input syntax of HHL prover.

The Outputs

The generated HCSP model consists of four files for variable definitions, process definitions, assertion definitions, and a goal to be proved, respectively.
Outline

1. Background
   - Background

2. Introduction to Simulink, HCSP, and HHL
   - Introduction to Simulink, HCSP, and HHL

3. Translating Simulink Diagrams into HCSP
   - Translating Simulink Diagrams into HCSP

4. Implementation
   - Implementation

5. Case Study
   - Case Study

N. Zhan (SKLCS)
An Introduction for the Protocol

- Mode transition from FS to CO at \( x_2 \).
- Level transition from CTCS-2 to CTCS-3 at \( x_2 \).
- Specification requirements for the scenario in CTCS-3.
The Top-level View of Simulink Model

Figure: The Top-level View of Simulink Model
The Simulation Result

Figure: The Result of Simulation
The Theorem Proving Result

- Four files (varDef, procDef, assertDef and goal) are generated by our tool Simulink2HCSP, which contain 488 lines of code in all.
- During the translation, several simulation settings have been abstracted away by our tool.
- We have proved in HHL Prover that the train will never pass the position $x_2$. 
Conclusion

We present an automatic translation from Simulink models to HCSP processes.

The resulting HCSP processes are the input files for HHL prover, and can be verified directly by the prover.

We demonstrate our approach by considering a case study on a combined scenario of CTCS-3.

Such an approach has been extend to formalize Simulink/Stateflow models [Zou et al. 2014].

On-going and future Work

Extending our approach to AADL is on-going.

We will apply our approach to more practical hybrid systems, especially the other scenarios of CTCS-3 and their combinations.
Thanks for your attention!