Formalization of Software Models for Cyber Physical Systems

Sandeep Neema*, Gabor Simko, Tihamer Levendovszky, Joseph Porter, Akshay Agrawal, and Janos Sztipanovits

Institute for Software Integrated Systems
Vanderbilt University

Email: sandeep.neema@vanderbilt.edu

Agenda

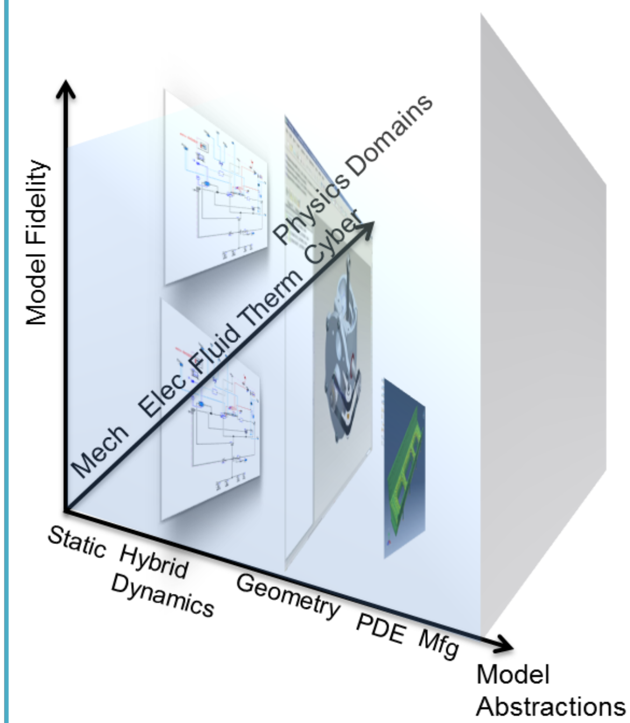
- Context & Motivation
- Background
  - ESMoL Design Toolchain
  - Semantic Backplane
- Formalization of ESMoL
  - Structural Semantics
  - Behavioral Semantics
  - Model Verification
  - Code Verification
- Case Study
- Results & Conclusion

Cyber Physical Systems

CPS Examples

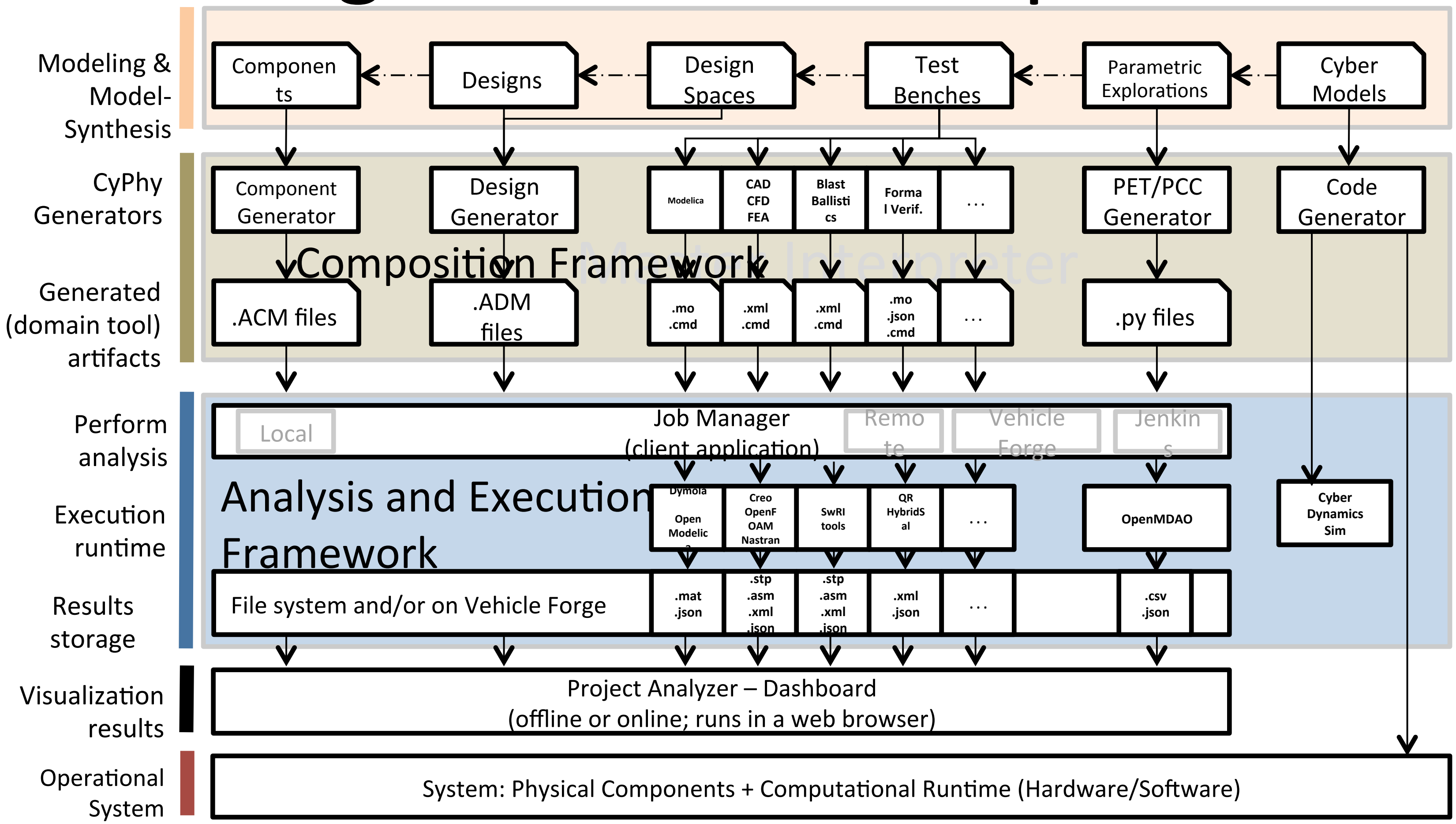
Design

Heterogeneous Domains & Abstractions



- CPS are mechatronic systems, characterized by tight integration between computational, communication, and physical components
- Design of CPS involve heterogeneous domains, involve multiple abstractions, and multiple models with varying fidelities

CPS Design Toolchains : OpenMETA



- CPS Design toolchains are complex, involve many different models, modeling languages, model transformations, and semantic domains

Formalization of CPS Toolchains

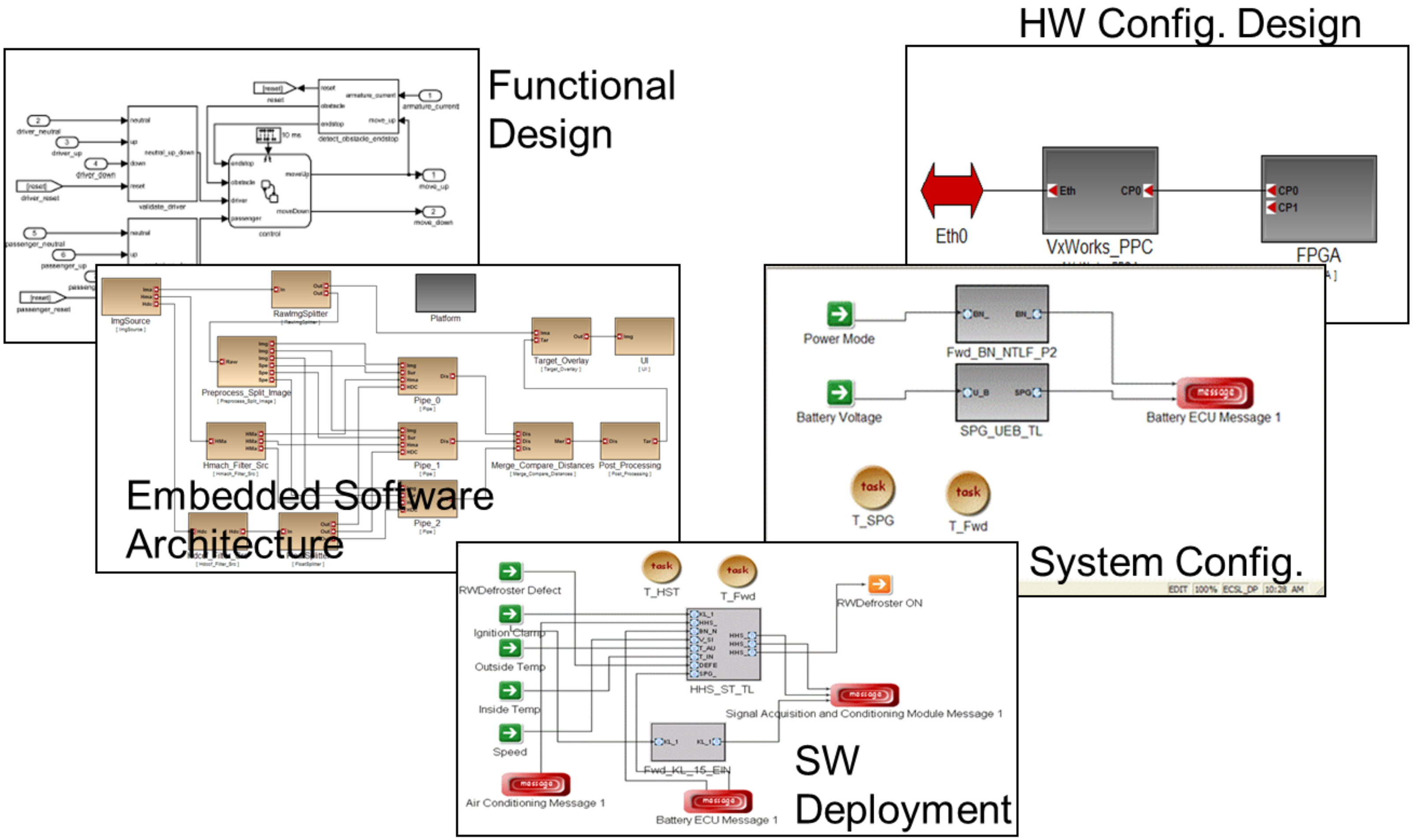
Provable Correctness of CPS depends on many factors:

- Correctness of Modeling Language(s)
- Correctness of Model Transformation Tools
- Correctness of Models
- Correctness of (auto-generated) Software
- Correctness of Runtime

Agenda

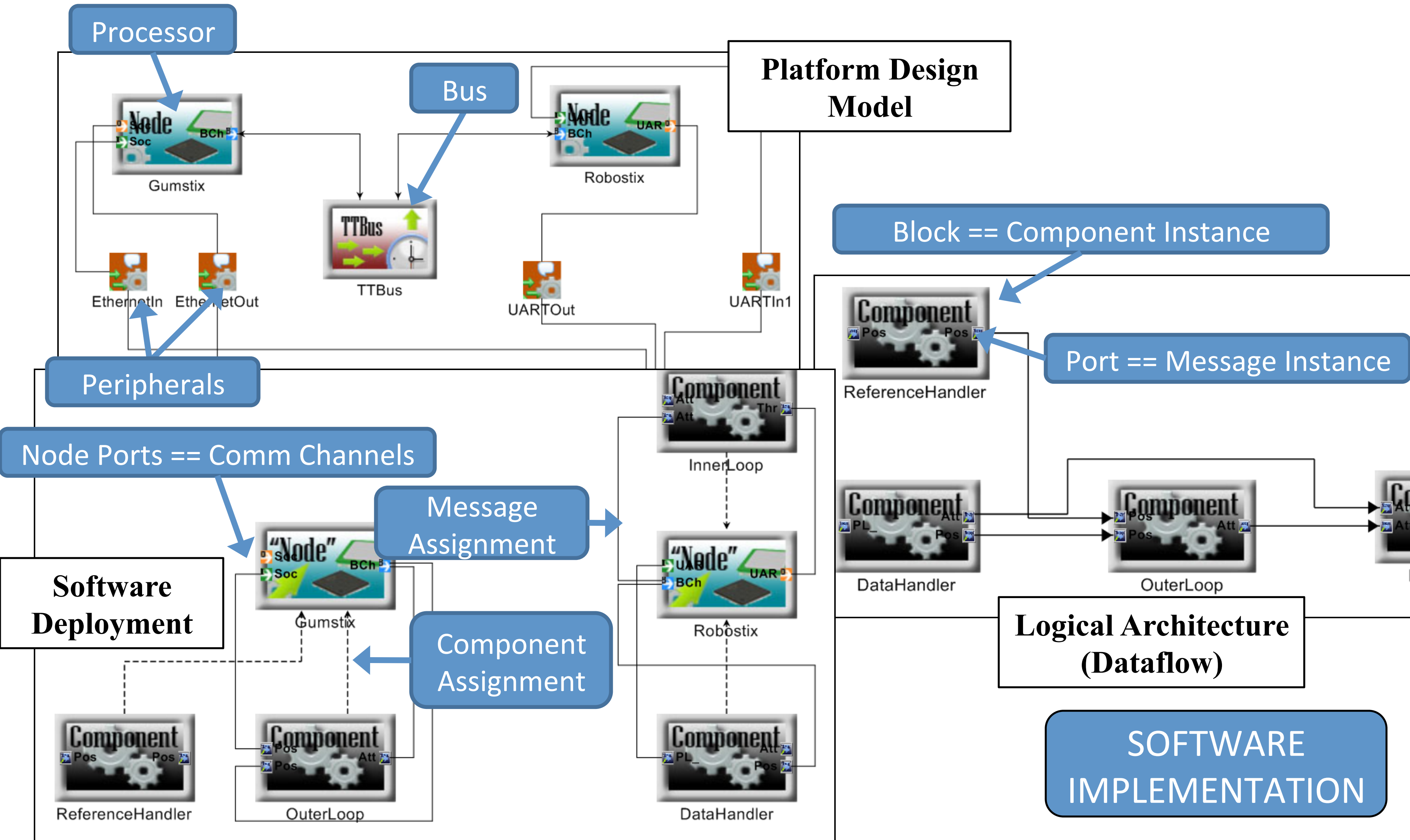
- Context & Motivation
- Background
  - ESMoL Design Toolchain
  - Semantic Backplane
- Formalization of ESMoL
  - Structural Semantics
  - Behavioral Semantics
  - Model Verification
  - Code Verification
- Case Study
- Results & Conclusion

Embedded Systems Modeling (ESMoL) Toolchain



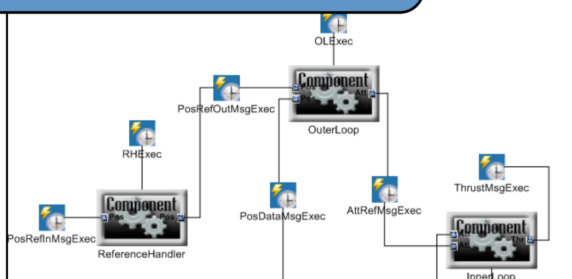
- ESMoL is a toolchain for design, simulation, analysis and synthesis of controllers

ESMoL Example: Quadrotor Software



ESMoL Use case: Simulation with TrueTime

SOFTWARE IMPLEMENTATION



Component Timing Parameters:

- TTSchedule – start times
- ExecPeriod
- WC Duration

OLExec

Attributes	Preferences	Properties
TTSchedule	0.015	
ExecPeriod	20ms	
WCDuration	1.5ms	

SOFTWARE ANALYSIS

Schedule Spec

Resolution 5us

Proc RS 4MHz 0s 0s
Comp InnerLoop =50Hz 1ms
Comp DataHandling =50Hz 1ms
Comp ADC =50Hz 1us
Comp SerialIn =50Hz 1ms
Comp SerialOut =50Hz 1ms
Msg DataHandling.sensor_data 8B RS/ADC RS/DataHandling
Msg DataHandling.pos_ref 8B RS/SerialIn RS/DataHandling
Msg InnerLoop.thrust_commands 8B RS/InnerLoop RS/SerialOut
Msg LocalOrder 1B RS/DataHandling RS/InnerLoop

Proc GS 100MHz 0s 0s
Comp OuterLoop =50Hz 1ms

Bus TT_I2C 100kb 1ms
Msg OuterLoop.ang_ref 8B GS/OuterLoop RS/InnerLoop
Msg DataHandling.pos_msg 8B RS/DataHandling GS/OuterLoop

Calculated Schedule

Hyperperiod 20 ms

GS/OuterLoop_0 9.495

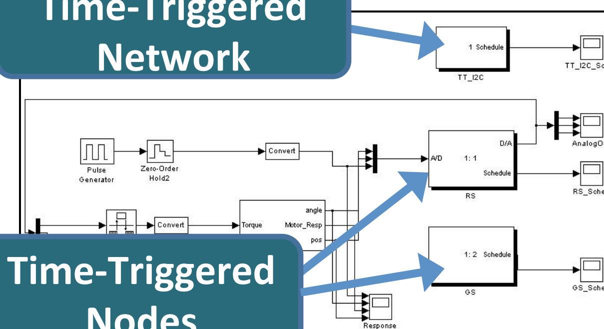
RS/SerialIn_0 7.75
RS/ADC_0 8.995
RS/InnerLoop_0 9.495
RS/DataHandling_0 10.495
RS/SerialOut_0 11.865

TT_I2C/OuterLoop.ang_ref_0 9.175
TT_I2C/DataHandling.pos_msg_0 10.815

GENERATION & EXECUTION

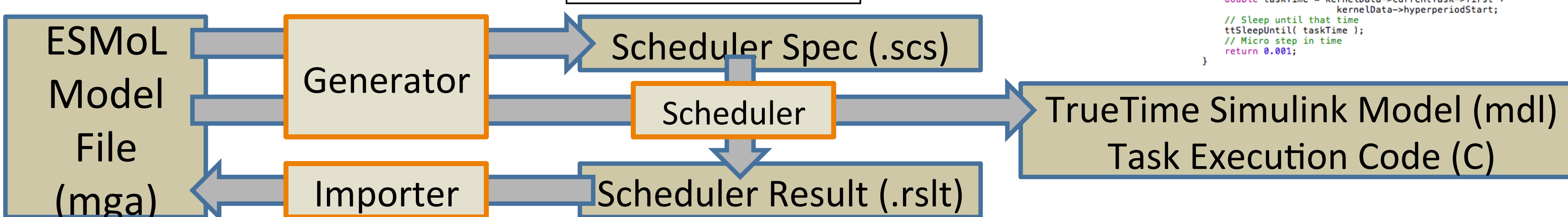
Time-Triggered Network

Time-Triggered Nodes



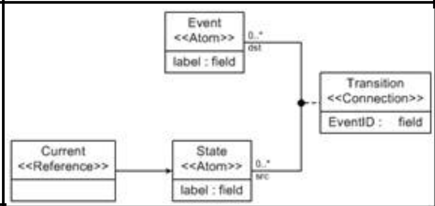
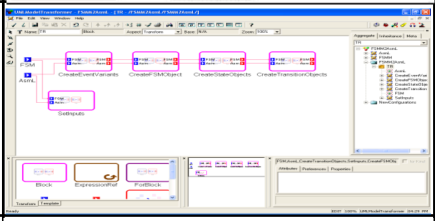
Generated Task Execution Code

```
    // ... in the start of a hyperperiod ...
    if ( seg == 1 ) {
        // Determine start of current hyperperiod
        kernelData->hyperperiodStart = ttCurrentTime();
    }
    // Otherwise we should schedule a task
    else {
        // We are woken up, now schedule the task
        ttCreateJob( kernelData->currentTask->second.c_str() );
        ... ext task
        ... ask++;
        ... end of hyperperiod
        ... rentTask == kernelData->tasks.end() ) {
        ... k list pointer
        ... rentTask = kernelData->tasks.begin();
        ... hyperperiod count
        ... rperiod++;
        ... t of here
        ... ;
        }
    }
    // Determine time of the next task to be executed
    double taskTime = kernelData->currentTask->first +
                      kernelData->hyperperiodStart;
    // Sleep until that time
    ttSleepUntil( taskTime );
    // Micro step in time
    return 0.001;
}
```



Semantic Backplane Specification Layers

VANDERBILT
SCHOOL OF ENGINEERING

Functions	(Meta)Models	Languages	Tools	Roles
Metamodeling		MetaGME	• GME • MetaGME2 Formula	• DSML specification • Constraint Checking • Metaprogrammable Tools • Bridge to other MBDT
Transformation Modeling		UMTL (Python, C++)	• GReAT • UDM • BON2	• Transf. specification • Compiling to transformations • Graph matching – based operations
Formal Metamodeling	domain DFA { primitive Event ::= (lbl: Integer). primitive State ::= (lbl: Integer). [Closed(src, trg, dst)] primitive Transition ::= (src: State, [Closed(st)] primitive Current ::= (st: State).	Formula (MSR)	• Model Visualizer • Trace Gen.	• Formal specification • Metamodel checking • DSML composition • Evolving structures • Model generation • Model validation
Formal Transformation Modeling	transform Step<fire: in1.Event> from DFA out1.State(x) :- in1.State(x). out1.Event(x) :- in1.Event(x). out1.Transition(s, e, sp) :- in1.Trans out1.Current(sp) :- in1.Current(s), ir out1.Current(s) :- in1.Current(s), fai }		• Semantic Anchoring	• Semantics for complex DSMLs • Composition of Semantics • Cross-domain invariants

Agenda

- Context & Motivation
- Background
  - ESMoL Design Toolchain
  - Semantic Backplane
- Formalization of ESMoL
  - Structural Semantics
  - Behavioral Semantics
  - Model Verification
  - Code Verification
- Case Study
- Results & Conclusion

ISIS
VANDERBILT
SCHOOL OF ENGINEERING

Formalization of Structural Semantics

$L = \langle Y, R_Y, C, ([\ ]_i)_{i \in J} \rangle$

$D(Y,C) = \{ r \in R_Y \mid r \models C \}$

$[\ ]: R_Y \mapsto R_{Y'}$

Key Concept: DSML syntax is understood as a constraint system that identifies behaviorally meaningful models. ***Structural semantics provides mathematical formalism for interpreting models as well-formed structures.***

Y: set of concepts,
R_Y: set of possible model realizations
C: set of constraints over R_Y
$D(Y,C)$: domain of well-formed models
[ ]: interpretations

Structural Semantics defines modeling domains using Algebraic Data Types and First-Order Logic with Fixpoints. Semantics is specified by Constraint Logic Programming.

Use of structural semantics:

- Conformance testing: $x \in D$
- Non-emptiness checking: $D(Y,C) \neq \{nil\}$
- DSML composing: $D_1 * D_2 | D_1 + D_2 | D' \ includes \ D$
- Model finding: $S = \{s \in D | s | = P\}$
- Transforming: $m' = T(m); m' \in X; m \in Y$

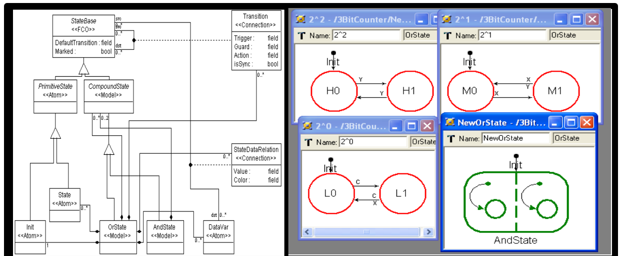
Jackson & Sz. ‘2007
Jackson, Schulte, Sz. ‘2008
Jackson & Sz. ‘2009

Microsoft Research Tool: FORMULA

- Fragment of LP is equivalent to full first-order logic
- Provide semantic domain for model transformations.

Explicit Methods for Specifying Behavioral Semantics 1/2

VANDERBILT SCHOOL OF ENGINEERING

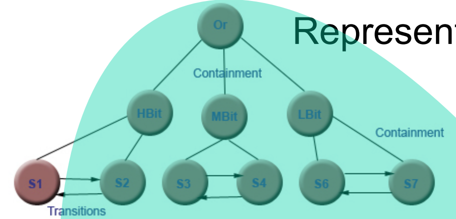


$$D(Y,C)=\{r\in R_Y \mid r \models C\}$$

$$[\;]: R_Y \mapsto R_{Y'}$$

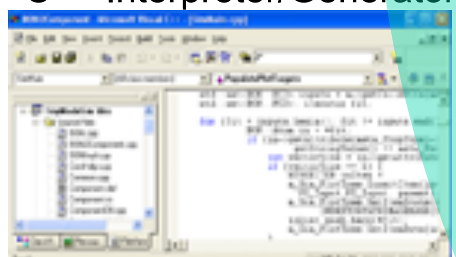
$$D(Y',C')=\{r\in R_{Y'} \mid r \models C'\}$$
$$[\;]: R_{Y'} \mapsto R_{Y''}$$

Representation as AST

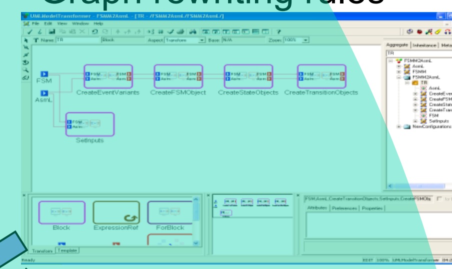


Explicit

C++ Interpreter/Generator



Graph rewriting rules



Executable Model (Simulators)

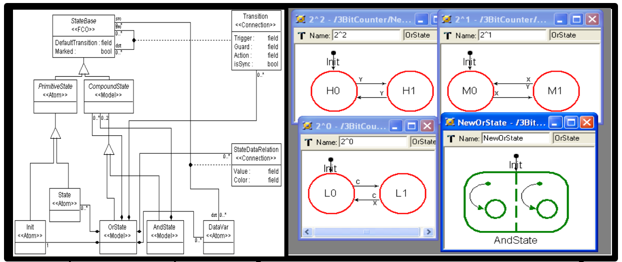
Executable Code

Executable Specification

Heterogeneous math domain; Operational semantics

Reasonable tool support; Easy to understand

Explicit Methods for Specifying Behavioral Semantics 2/2



$$D(Y,C)=\{r\in R_Y \mid r \models C\}$$

$$[\;]: R_Y \mapsto R_{Y'}$$

$$D(Y',C')=\{r\in R_{Y'} \mid r \models C'\}$$

$$[\;]: R_{Y'} \mapsto R_{Y''}$$

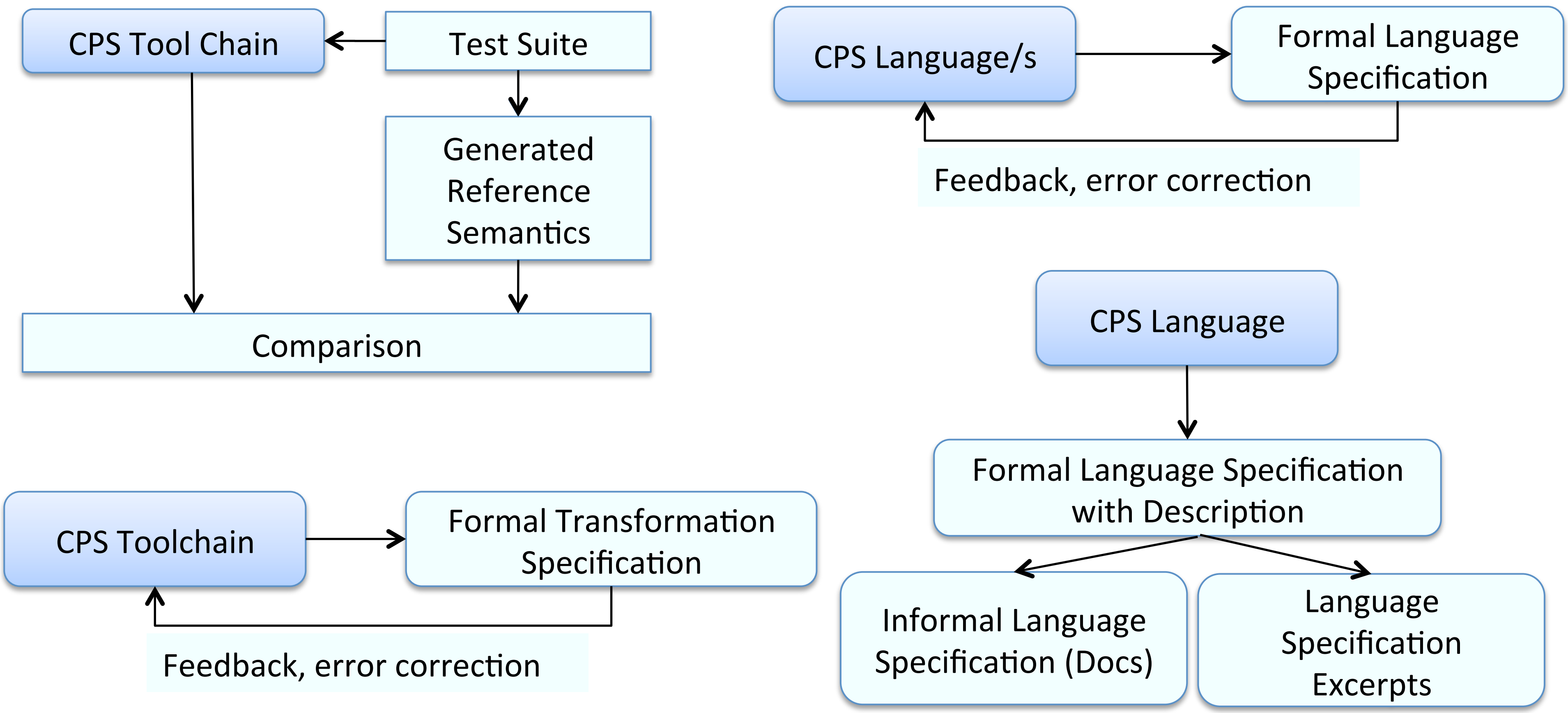
Single math framework
Unified approach

```
domain AcausalBG_elements
{
  primitive Sf ::= (id: String).
  primitive Se ::= (id: String).
  primitive R  ::= (id: String).
  //…
primitive TF ::= (id: String).
  primitive GY ::= (id: String).
  primitive ZeroJunction ::= (id: String).
  primitive OneJunction  ::= (id: String).
  Source   ::= Sf + Se.
  //..
}
```

```
transform BG_DenotationalSemantics
  from in1::AcausalBG
  to out1::DAEquations
{
  Eq(e_a,p_x) :- x is Se, Src(a,x).
  Eq(f_a,p_x) :- x is Sf, Src(a,x).
  Eq(e_a, Mul(p_x,f_a)) :- x is R, Dst(a,x).
  DiffEq(e_a, Mul(Inv(p_x),f_a)) :-
    x is C, Dst(a,x).
//…
}
```

```
domain DAEquations
{
  primitive Variable ::=
    (name: String, id: String).
  primitive Param ::= (id: String).
  primitive Neg ::= (Term).
  primitive Inv ::= (Term).
  //..
  Term ::= Variable + Param + Neg + Inv + Mul +
Sum.
  primitive Eq ::= (Variable, Term).
  primitive DiffEq ::= (Variable, Term).
  primitive SumZero ::= (Sum).
  Equation ::= Eq + DiffEq + SumZero.
}
```

Semantic Backplane Use Cases



Agenda

- Context & Motivation
- Background
  - ESMoL Design Toolchain
  - Semantic Backplane
- Formalization of ESMoL
  - Structural Semantics
  - Behavioral Semantics
  - Model Verification
  - Code Verification
- Case Study
- Results & Conclusion

Formalization of ESMoL

- Structural semantics of Stateflow sub-language of ESMoL
- Behavioral semantics of Stateflow sub-language of ESMoL
- Formal verification of Stateflow models using NuSMV
- Formal verification of code-generated from ESMoL-Stateflow

ESMoL - Structural Semantics

Terms

```
State ::= new (name:String,
               decomposition:{OR,AND},
               entryAction:String,
               duringAction:String,
               exitAction:String,
               order:Integer).
```

```
StateContainment ::= fun (State =>
                          State + Subsystem).
```

```
Transition ::= new (name:String,
                    src:Transition_Source,
                    dst:Transition_Destination,
                    guard: String,
                    condition_action:String,
                    transition_action:String,
                    order:Integer).
```

Structural validity Rules

```
valid_transition(T) :- T is Transition,
   StateContainment(T.src,P),
   StateContainment(T.dst,P).

valid_transition(T) :- T is Transition,
   StateContainment(T.src,P1),
   StateContainment(T.dst,P2),
   StateContainment(P1,P2).

valid_transition(T) :- T is Transition,
   StateContainment(T.src,P1),
   StateContainment(T.dst,P2),
   StateContainment(P2,P1).

invalid_Transition(T) :- T is Transition, no
   valid_transition(T).
```

MAAB Structural validity Rules

```
contains_at_least_two_substates(X) :-
   StateContainment(Y,X),
   StateContainment(Z,X), Y != Z.

Invalid_db_0137 :- X is State,
X.decomposition=OR, no
contains_at_least_two_substates(X).
```

ESMoL – Behavioral Semantics

- Translational Semantics: Mapping of the modeling language to a formal domain that has pre-defined operational semantics
- ESMoL Stateflow → Mathworks Stateflow (operational semantics: Hamon and Rushby, 2007)

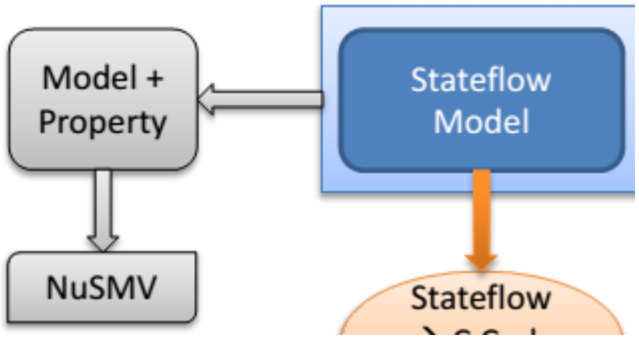
State mapping

```
map(S, Stateflow.state(name, comp,
    entryAction, duringAction, exitAction,
    onAction, order))
:- S is Signalflow.State,
    name = S.name,
    map(S.entryAction,entryAction),
    map(S.duringAction,duringAction),
    map(S.exitAction, exitAction),
    onAction = null,
    map(S.decomposition,comp),
    order = S.order.
```

Transition mapping

```
map(T, Stateflow.transition(src, dst, event,
    guard, conditionAction, transitionAction,
    order))
    :- T is Signalflow.Transition,
map(T.src,m_src),
map(T.dst,m_dst),
map(T.trigger,m_event),
map(T.guard, guard),
map(T.conditionAction,conditionAction),
map(T.transitionAction,transitionAction),
order = T.order.
```

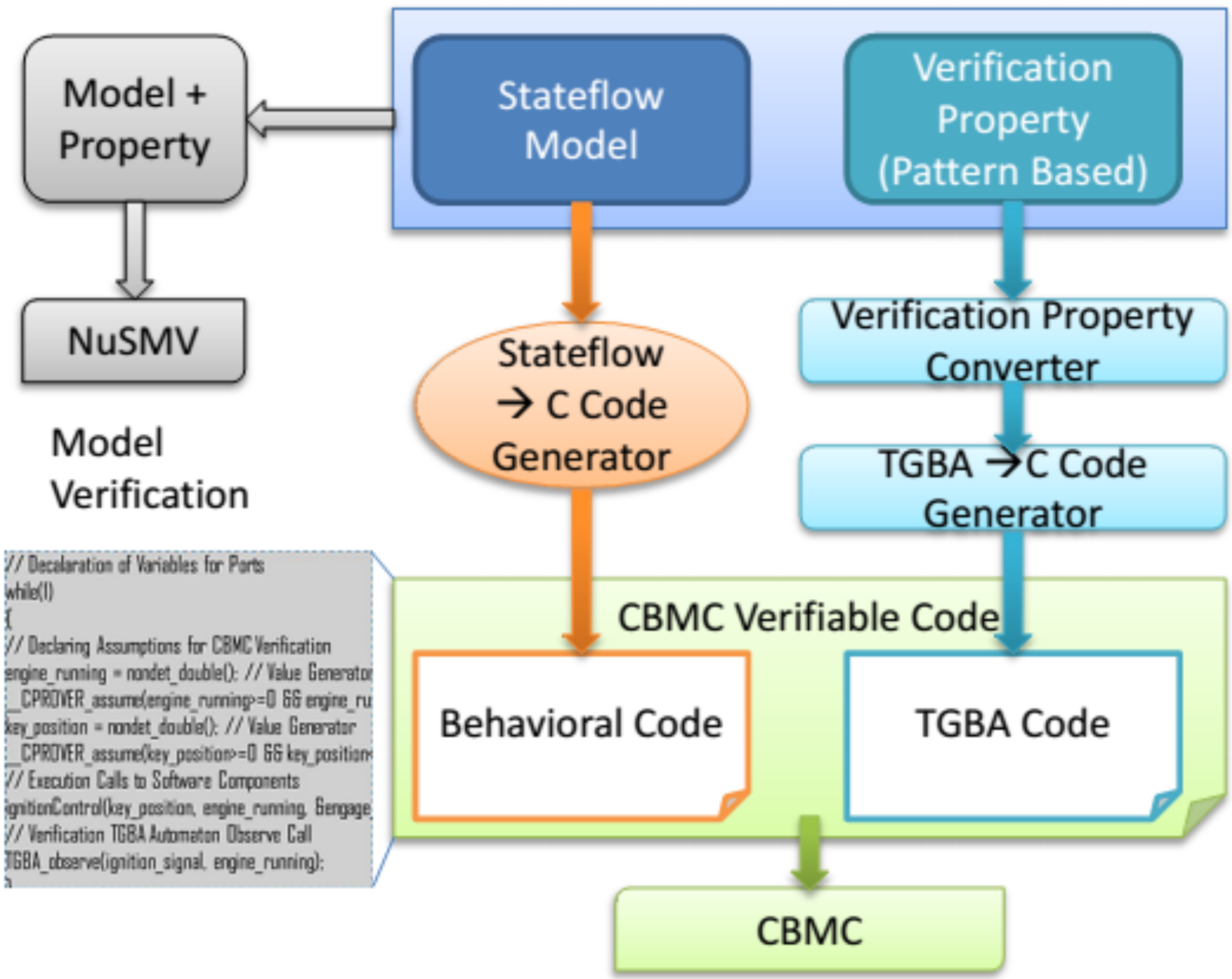
ESMoL - Model Verification



- Leverages automated translation from Stateflow to NuSMV reported in publication by Miller, Whalen, and Cofer.

```
G ((key position>1 & engine running<1) ->
 X (Ignition_Logic.engage_starter=1) )
```

ESMoL - Code Verification

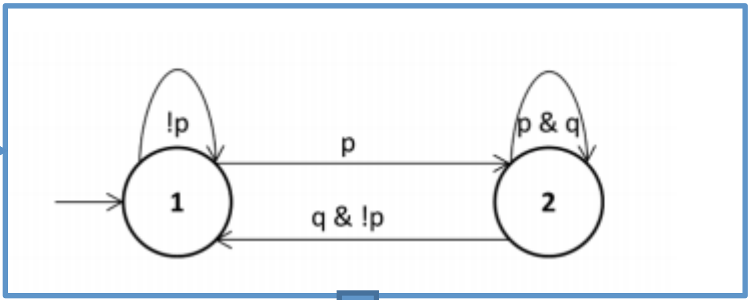


Property Specification Templates

Occurrence Pattern	**Meaning**	**Scope Pattern**	**Meaning**
Existence(P)	'P' holds true	Globally	Defined *occurrence pattern* must be true always
Immediate Response(P & S)	if 'P' occurs at some time-step then 'S' occurs in the next time-step after 'P'	Before R	Defined *occurrence pattern* must be true before occurrence of event 'R'
Response(P & S)	if 'P' occurs at some time-step then 'S' occurs in the future after 'P'	After Q	Defined *occurrence pattern* must be true after occurrence of event 'Q'
Precedence(P & S)	'S' must have already occurred before 'P' occurs at some time-step	Between Q and R	Defined *occurrence pattern* must be true between occurrences of events 'Q' and 'R', in that order. Uses strong until operator (**U**).
		After Q Until R	Analogous to *Between Q and R* but uses weak until operator (**W**).

LTL → TGBA → Verification Cond.

LTL Formulae



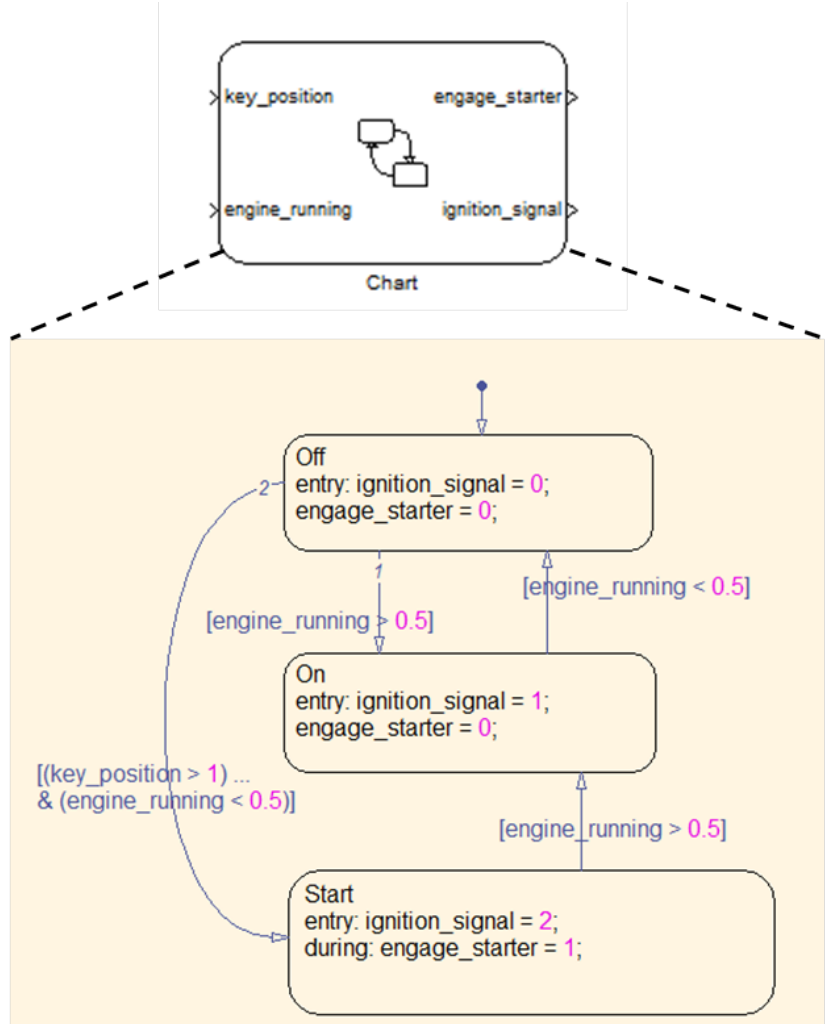

```
1: if (state==1 && p) { state = 2; }
2: else if (state==1 && !p) { state = 1; }
3: else if (state==2 && p && q) {state = 2;}
4: else if (state==2 && !p && q)
        {state = 1;}
5: else { assert(0); }
```

Agenda

- Context & Motivation
- Background
  - ESMoL Design Toolchain
  - Semantic Backplane
- Formalization of ESMoL
  - Structural Semantics
  - Behavioral Semantics
  - Model Verification
  - Code Verification
- Case Study
- Results & Conclusion

Ignition Controller - Model

- Function
  - Control Ignition lights on display
  - Actuate engine starter based on ignition key and engine state
- Signals
  - key_pos
  - engine_running
  - ignition_signal
  - engine_starter
- Textual Requirements
  - When the ignition key is turned on, while engine is not running the starter must engage to actuate the engine and disengage once the engine is running
  - Ignition light on dashboard must reflect the status of engine correctly



Ignition Controller - Properties

Property	Description	Occurrence Pattern	Scope Pattern
1	the engine should be already running before the ignition light reflects that the engine is running	Precedes(P & S) – S precedes P S: (engine_running > 0.5) P: (ignition_signal == 1.00)	Globally
2	if the ignition key is turned on when the engine is not running then the starter should get engaged so as to start the Engine	Immediate Response(P & S) – S occurs next after P S: (engage_starter == 1.00) P: (key_position > 1.00 && engine_running < 1.00)	Globally
3	always whenever the ignition key is turned off while the starter is on then in the next time step the starter should get Disengaged	Immediate Response(P & S) - S occurs next after P S: (engage_starter < 1.00) P: (key_position < 1.00 && engage_starter > 0.00)	Globally

Ignition Controller - Results

- CBMC bound set to 30
- Property 1 is not violated
- Property 2 & 3 are violated
- Results consistent with NuSMV
- Counter-examples analogous to those generated by NuSMV
- → Code generator is correct with respect to the checked properties

Conclusions

- Verification of CPS is paramount
- Formal methods need to be applied holistically to model-based CPS design tool chains
- Scalability of verification methods is a huge barrier to widespread adoption – that need to be addressed by pragmatic approaches
- Presented an example CPS toolchain with application of formal methods to multiple aspects of the toolchain