Trusting Outsourced Components in Flight-Critical Systems

Temesghen Kahsai

NASA Ames / CMU
Joint work with ...
Outsourcing in the aerospace industry
Outsourcing in the aerospace industry

The Boeing 787 Dreamliner’s flight critical, embedded software is built on the WRS ARINC 653 system and is assembled from software components by multiple subcontractors.
Outsourcing in the aerospace industry

Boeing 737 and 747 = 35-50%

Boeing 787 = 70%
The delivery date was pushed back 4 times and was late more than 4 years.

The aft fuselage consisted of 6,000 components, and many of those components failed to conform to Boeing’s specified tolerances, resulting in significant cost and schedule delays.

The first Dreamliner to arrive at the company’s assembly place was missing tens of thousands of parts.
Outsourcing in the aerospace industry
January 2013: 50 Dreamliner was grounded due to issues with the lithium-ion batteries.

On balance with just under 60 aircraft in service, the 787 has had 6 reported mechanical incidents in 2013.

All the individual parts worked in isolation. But, together, under certain circumstances, the parts failed.
“While we can’t completely eliminate failures, the answer lies in system engineering. This involves a process of careful design and architecture … as well as a staged integration of the entire system, and extensive qualification, verification and validation testing.” Prof. S. Eppinger (MIT)
This talk ....

... outsourcing in flight critical software

... virtual integration of outsourced components
Assurance of Flight Critical Systems (FCS)

Aim:

- Develop multidisciplinary V&V tools and techniques that advance safety assurance and certification
- Flight-critical systems: any systems that directly controls the safe conduct of an aircraft’s flight, i.e. air and ground systems

Technical Challenges:

1. Argument-based safety assurance
2. Integrated distributed systems
3. Authority and Autonomy
4. Software intensive systems
5. Assessment environments
Assurance of Flight Critical Systems (FCS)

Aim:

- Develop multidisciplinary V&V tools and techniques that advance safety assurance and certification
- Flight-critical systems: any systems that directly controls the safe conduct of an aircraft’s flight, i.e. air and ground systems

Technical Challenges:

1. Argument-based safety assurance
2. Integrated distributed systems
3. Authority and Autonomy
4. Software intensive systems
5. Assessment environments
Assurance of Flight Critical Systems (FCS)

Topic: “Support for verification of black-box FCS”
Assurance of Flight Critical Systems (FCS)

Topic: “Support for verification of black-box FCS”

Context:

- More and more design and implementation of FCS is contracted out to external companies
- Example: FAA contracts out the implementation of most of the air traffic systems
- Integration of FCS from Commercial Off-The-Shelf (COTS) components
- Current technique is based on black-box testing
- Many of those systems have been first prototyped in-house
- Example: Many FAA systems has been prototyped by MIT Lincoln Lab, NASA etc. (e.g. TCAS, ACAS-X, TSAFE, etc.)
Assurance of Flight Critical Systems (FCS)

Topic: “Support for verification of black-box FCS”

In house prototyping

- System Design Model
- Component Design Model (Simulink/Stateflow)
- Component prototype imp. (C/C++)
- Component prototype imp. (JAVA)
- System Level Properties

Component Outsourced For Implementation
Assurance of Flight Critical Systems (FCS)

Topic: “Support for verification of black-box FCS”

In house prototyping

- System Design Model
  - Component Design Model (Simulink/Stateflow)
  - Component prototype imp. (JAVA)
  - Component prototype imp. (C/C++)
  - System Level Properties

Component Outsourced For Implementation

In house assembling

- Black Box Testing
  - Testing Environment
  - Component Black-box Implementation

Black-box Component Delivery

Wednesday, May 20, 15
Contract-based Compositional verification for outsourced FCS (CoCo)
Outline

- Two stage solution for virtual integration
- 1st stage: contract generation
- 2nd stage: contract compliance
- Flight critical system case studies
Two Stage solution for virtual integration
Two Stage solution for virtual integration

In house prototyping

System Design Model

Component Design Model (Simulink/Stateflow)

Component Design Model (C/C++)

Component prototype imp. (JAVA)

Component prototype imp. (C/C++)

System Level Properties

Component Outsourced For Implementation

In house assembling

Black Box Testing

Testing Environment

C₁

C₂

C₃

Component Black-box Implementation

Black-box Component Delivery

Wednesday, May 20, 15
Two Stage solution for virtual integration

Pre-Delivery Stage

Post-Delivery stage

Component Outsourced For Implementation

Black-box Component Delivery
Two Stage solution for virtual integration

Pre-Delivery Stage

System Design Model

Component Design Model (JAVA)

Component Design Model (Simulink/Stateflow)

Component prototype imp. (C/C++)

System Level Properties

Component Outsourced For Implementation

Post-Delivery stage

Black-box Component Delivery
Two Stage solution for virtual integration

Pre-Delivery Stage

System Design Model → Component Design Model (JAVA) → System Level Properties → Automated Contract Generation

Component Design Model (Simulink/Stateflow) → Component prototype imp. (C/C++)

Post-Delivery stage

Automated Contract Generation

Component Outsourced For Implementation

Black-box Component Delivery
Two Stage solution for virtual integration

Pre-Delivery Stage

System Design Model

Component Design Model (JAVA)

Component Design Model (Simulink/Stateflow)

Component prototype imp. (C/C++)

System Level Properties

Automated Contract Generation

Component Outsourced For Implementation

Post-Delivery stage

Black-box Component Delivery
Two Stage solution for virtual integration

Pre-Delivery Stage

- System Design Model
  - Component Design Model (JAVA)
  - Component Design Model (Simulink/Stateflow)
  - Component prototype imp. (C/C++)

- Automated Contract Generation

- Component Level Properties

- Component Outsourced For Implementation

Post-Delivery Stage

- Contract - Based Component Testing
  - Component Black-box Implementation

- Contract - Based Integration Testing

- Black-box Component Delivery
Pre-delivery verification stage
Pre-delivery Verification Stage

How to generate formal contracts from models and prototypical code?

1. Define a notion of a component contract
   - system property based
   - allows obtain a higher degree of assurance

2. Design a uniform intermediate modeling formalism
   - to facilitate the integration of different techniques
   - to target heterogeneous in-house system prototypes

3. Develop (semi)-automated techniques to generate contract from models and prototypical code
Notion of a Formal Contract

- Contracts as a method to organize and integrate component-based systems
- Specify precisely the information necessary to reason about a component interactions
- Contracts specify I/O behavior of a component:
  - Define the component guarantees provided that its environment obey certain assumptions.
Notion of a Formal Contract

- Contracts as a method to organize and integrate component-based systems
- Specify precisely the information necessary to reason about a component interactions
- Contracts specify I/O behavior of a component:
  - Define the component guarantees provided that its environment obey certain assumptions.

Different notions of formal contract, e.g.:

- **Othello**: Trace-based contract framework [Tonetta et. al.]
- **AGREE**: Contract language for AADL [Cofer et. al.]
- ACSL, JML, SPARK, etc : Contract in Programming Languages.
Compositional Verification

- Check $P$ on entire system: too complicated (e.g. many states)
- Use system’s natural decomposition into components to break-up the verification task
- Check components in isolation: $M_1 \models P$?
- ... typically a component is designed to satisfy its requirements in specific contexts
Compositional Verification

- Check $P$ on entire system: too complicated (e.g. many states)
- Use system’s natural decomposition into components to break-up the verification task
- Check components in isolation: $M_1 \models P$?
- ... typically a component is designed to satisfy its requirements in specific contexts
Compositional Verification

- Check $P$ on entire system: too complicated (e.g. many states)
- Use system’s natural decomposition into components to break-up the verification task
- Check components in isolation: $M_1 \models P$?
- ... typically a component is designed to satisfy its requirements in specific contexts

- **Assume-Guarantee** reasoning

- *Misra & Chandy 81, Jones 83, Pnueli 84, Pasareanu 01*
Compositional Verification

- Check $P$ on entire system: too complicated (e.g. many states)
- Use system’s natural decomposition into components to break-up the verification task
- Check components in isolation: $M_1 \models P$?
- ... typically a component is designed to satisfy its requirements in specific contexts

- **Assume-Guarantee** reasoning

- *Misra & Chandy 81, Jones 83, Pnueli 84, Pasareanu 01*

  - introduces **assumption** $A$ representing $M_1$’s context
\[ \langle A \rangle M \langle P \rangle \text{ is true if whenever } M \text{ is part of a system that satisfies } A, \text{ then the system must also guarantee } P \]

**Simplest assume-guarantee rule (Asym)**

1. \[ \langle A \rangle M_1 \langle P \rangle \]
2. \[ \langle \text{true} \rangle M_2 \langle A \rangle \]

\[ \frac{\langle \text{true} \rangle M_2 \langle A \rangle}{\langle \text{true} \rangle M_1 \parallel M_2 \langle P \rangle} \]
Compositional Verification

\( \langle A \rangle \ M \ \langle P \rangle \) is true if whenever \( M \) is part of a system that satisfies \( A \), then the system must also guarantee \( P \)

Simplest assume-guarantee rule (Asym)

1. \( \langle A \rangle \ M_1 \ \langle P \rangle \)
2. \( \langle \text{true}\rangle \ M_2 \ \langle A \rangle \)

\[ \langle \text{true}\rangle \ M_1 \parallel M_2 \ \langle P \rangle \]

* Cobleigh et. al “Learning assumption for compositional verification”. TACAS’01
* Emmi et. al “Assume Guarantee Verification for Interface Automata”. FM’08
* Giannakopoulou et. al “Symbolic Learning of component interfaces”. SAS’12
* Howar et. al “Hybrid learning: interface generation through static, dynamic, and symbolic analysis” ISSTA’13.
Compositional Verification

Example of assumptions (*)

- no file “close” before “open”
- access to shared variable “X” must be protected by lock “L”
- (rover executive) whenever thread “T” reads variable “V”, no other thread can read “V” before thread “T” clears it first
- (spacecraft flight phases) a docking maneuver can only be invoked if the launch abort system has previously been jettisoned from the spacecraft

(*) C. Pasareanu slides on compositional verification from SSFT 2012
Two Stage solution for virtual integration

Pre-Delivery Stage

System Design Model

- Component Design Model (JAVA)
- Component Design Model (Simulink/Stateflow)
- Component prototype imp. (C/C++)

System Level Properties

Automated Contract Generation

Component Outsourced For Implementation

Post-Delivery stage

Contract - Based Integration Testing

Contract - Based Component Testing

Component Black-box Implementation

Black-box Component Delivery
Two Stage solution for virtual integration

Pre-Delivery Stage

System Design Model

Component Design Model (JAVA)

Component Design Model (Simulink/Stateflow)

Component prototype imp. (C/C++)

System Level Properties

Automated Contract Generation

Component Outsourced For Implementation

Post-Delivery stage

Contract - Based Integration Testing

Contract - Based Component Testing

Component Black-box Implementation

Black-box Component Delivery
Pre-delivery Verification Stage

Our current approach:

- Simulink/Stateflow
- Lustre
- C/C++
- JAVA
- Safety Properties

Uniform intermediate verification language

Different techniques for automated contract generation

Assume/Guarantee contract

Assume/Guarantee contract

CoCoSpec
Pre-delivery Verification Stage

Our current approach:

- Simulink/Stateflow
- Integrated Simulink 2 Lustre Modular Compiler
- LLVM-based languages
- Lustre
- C/C++
- JAVA
- Safety Properties
- Uniform intermediate verification language
- Different techniques for automated contract generation
- Generalized PDR, Concolic Execution, Automata Learning ...
- Assume/Guarantee contract
Pre-delivery Verification Stage

Our current approach:

- Simulink/Stateflow
- Integrated Simulink 2 Lustre Modular Compiler
- LLVM-based languages
- Safety Properties

Uniform intermediate verification language

Different techniques for automated contract generation

Assume/Guarantee contract

CoCoSpec

Wednesday, May 20, 15
Lustre

- **Declarative** and **deterministic** specification language
- Lustre programs = systems of **equational constraints** between input and output streams

```plaintext
node therm_control (actual: real; up, dn: bool )
    returns (heat, cool : bool)
var desired, margin : real;
let
    margin = 1.5;
desired = 21.0 → if dn then (pre desired) − 1.0
    else if up then (pre desired) + 1.0
    else (pre desired);
cool = (actual − desired) > margin;
heat = (actual − desired) < −margin;
```

Wednesday, May 20, 15
A Lustre program models an I/O automaton

Implementing a Lustre program

- Read inputs
- Compute next state and outputs
- Write outputs
- Update state

Repeat at every trigger (external event)
A Lustre program is a collection of nodes: $L = [N_0, N_1, \ldots, N_m]$
Lustre

A Lustre program is a collection of nodes: $L = [N_0, N_1, \ldots, N_m]$

$$N_i = (\mathcal{I}_i, \mathcal{O}_i, \mathcal{L}_i, Init_i, Trans_i)$$

- $\mathcal{I}_i, \mathcal{O}_i, \mathcal{L}_i$: set of input/output/local vars
- $Init_i, Trans_i$: set of formulas for the initial states and transition relation
Lustre

A Lustre program is a collection of nodes: \( L = [N_0, N_1, \ldots, N_m] \)

\[ N_i = (I_i, O_i, L_i, Init_i, Trans_i) \]

- \( I_i, O_i, L_i \) : set of input/output/local vars
- \( Init_i, Trans_i \) : set of formulas for the initial states and transition relation

\[ \bigwedge_{i \in \mathbb{N}} v_i = \rho(s_i) \]

- \( v_i \in O_i \cup L_i \) and \( Vars(s_i) \subseteq I_i \cup O_i \cup L_i \)
- \( s_i \) arbitrary Lustre expression including node calls \( N_j(u_1, \ldots, u_n) \)
- \( \rho \) function maps expression to expression

\[ a \rightarrow b \text{ is projected as } \begin{cases} a \text{ in } Init_i \\ b \text{ in } Trans_i \end{cases} \]
A Lustre program is a collection of nodes: \( L = [N_0, N_1, \ldots, N_m] \)

\[
N_i = (\mathcal{I}_i, \mathcal{O}_i, \mathcal{L}_i, Init_i, Trans_i)
\]

- \( \mathcal{I}_i, \mathcal{O}_i, \mathcal{L}_i \) : set of input/output/local vars
- \( Init_i, Trans_i \) : set of formulas for the initial states and transition relation

\[
\bigwedge_{i \in \mathbb{N}} v_i = \rho(s_i)
\]

- \( v_i \in \mathcal{O}_i \cup \mathcal{L}_i \) and \( Vars(s_i) \subseteq \mathcal{I}_i \cup \mathcal{O}_i \cup \mathcal{L}_i \)
- \( s_i \) arbitrary Lustre expression including node calls \( N_j(u_1, \ldots, u_n) \)
- \( \rho \) function maps expression to expression

\( a \rightarrow b \) is projected as \( \left\{ \begin{array}{l} a \text{ in } Init_i \\ b \text{ in } Trans_i \end{array} \right\} \)

- A safety property \( P \) is any Lustre expression over the main node \( N_0 \)
Our current approach:

- Simulink/Stateflow
- Lustre
- C/C++
- JAVA
- Safety Properties

Uniform intermediate verification language

Different techniques for automated contract generation

Assume/Guarantee contract

Constrained Horn Clauses

Generalized PDR, Concolic Execution, Automata Learning ...

CoCoSpec
Pre-delivery Verification Stage

Our current approach:

- Simulink/Stateflow
- Lustre
- C/C++
- JAVA
- Safety Properties

Uniform intermediate verification language

Different techniques for automated contract generation

Constrained Horn Clauses

Generalized PDR, Concolic Execution, Automata Learning ...

Assume/Guarantee contract

**CoCoSpec**

Wednesday, May 20, 15
Assume-Guarantee contracts

consist of

- an assumption $\mathcal{A}$: how the component must be used
- a guarantee $\mathcal{G}$: how the component must behave, assuming $\mathcal{A}$
Assume-Guarantee contracts

consist of

- an assumption \( A \): how the component must be used
- a guarantee \( G \): how the component must behave, assuming \( A \)

If \( \langle A, G \rangle \) is a contract for component \( C \), then if \( A \) is always true so is \( G \):

\[
(\Box A) \Rightarrow (\Box G) \quad \text{holds for } C
\]

In practice, usually weakened to

\[
(hist \ A) \Rightarrow G \quad \text{is an invariant of } C
\]
Assume-Guarantee contracts

consist of

- an assumption $\mathcal{A}$: how the component must be used
- a guarantee $\mathcal{G}$: how the component must behave, assuming $\mathcal{A}$

If $\langle \mathcal{A}, \mathcal{G} \rangle$ is a contract for component $C$, then if $\mathcal{A}$ is always true so is $\mathcal{G}$:

$$(\square \mathcal{A}) \Rightarrow (\square \mathcal{G}) \text{ holds for } C$$

In practice, usually weakened to

$$(hist \ \mathcal{A}) \Rightarrow \mathcal{G} \text{ is an invariant of } C$$

If component $C'$ uses $C$, then $\mathcal{A}_{cs}$ ($\mathcal{A}$ at call site) must always be true:

$\mathcal{A}_{cs}$ is an invariant of $C'$
Assume-Guarantee contracts

Improves scalability of the verification of hierarchical systems by abstracting components by their contract.

The analysis is bottom-up:

- *leaves* are analyzed as usual, which can *succeed* or fail.
- for *nodes*, we first abstract the subcomponents, which can *succeed*, or fail.

In case of failure we can restart the analysis after (soundly) refining the abstraction, possibly several times.
Assume-Guarantee contracts

Improves scalability of the verification of hierarchical systems by abstracting components by their contract.

The analysis is bottom-up:

- **leaves** are analyzed as usual, which can **succeed** or **fail**.

- for **nodes**, we first abstract the subcomponents, which can **succeed**, or **fail**.

In case of **failure** we can restart the analysis after (soundly) refining the abstraction, possibly several times.

Implemented in Kind2: a multi engine model checker for Lustre programs

http://kind2-mc.github.io/kind2/
CoCoSpec

- An Assume/Guarantee-based Contract Language on top of Lustre

A CoCoSpec contract is a pair \( \langle A, G \rangle \)

Assumption — how the component must be used:

\[
A \equiv \bigvee (\text{require}_i)
\]

 Guarantee — how the component behaves:

\[
G \equiv \bigwedge (\text{require}_i \Rightarrow \text{ensure}_i)
\]
CoCoSpec

- An Assume/Guarantee-based Contract Language

```plaintext
node component(n1, n2:int; chaos:bool)
    returns (out: bool; corrupted, warning:bool) ;
  --!contract : contr

let
  -- Implementation.
  tel

contract contr(n1, n2:int; chaos:bool)
    returns (out: bool; corrupted, warning:bool) ;
  let
    require (-7 <= n1) and (7 <= n1); -- n1 legal input
    require (-11 <= n2) and (11 <= n2); -- n2 legal input
    ensure (-42 <= out) and (42 <= out); -- out is bounded
  tel
```
CoCoSpec

- An Assume/Guarantee-based Contract Language

```plaintext
node component(n1, n2:int; chaos:bool)
    returns (out: bool; corrupted, warning:bool) ;
--!contract : contr
del

let
    -- Implementation.
let
contract contr(n1, n2:int; chaos:bool)
    returns (out: bool; corrupted, warning:bool) ;
let
    require (-7 <= n1) and (7 <= n1); -- n1 legal input
    require (-11 <= n2) and (11 <= n2); -- n2 legal input
    ensure (-42 <= out) and (42 <= out); -- out is bounded
del
```
CoCoSpec

• An Assume/Guarantee-based Contract Language

```plaintext
node component(n1, n2:int; chaos:bool)
    returns (out: bool; corrupted, warning:bool) ;
--!contract : contr ;

let
    -- Implementation.

contract contr(n1, n2:int; chaos:bool)
    returns (out: bool; corrupted, warning:bool) ;
let
    require (-7 <= n1) and (7 <= n1); -- n1 legal input
    require (-11 <= n2) and (11 <= n2); -- n2 legal input
    ensure (-42 <= out) and (42 <= out); -- out is bounded
```
CoCoSpec

- An Assume/Guarantee-based Contract Language

```plaintext
node component(on, off: bool) returns (active: bool);
--!contract : nop;
--!contract : inhibited;
let
    -- Implementation.
.tel
contract inhibited(on, off: bool) returns (active: bool);
var
    act_raise: bool ; last_act_raise: int;
let
    active_raise = false -> active and not pre active;
    last_act_raise = 0 -> if pre active_raise then 1
    else 1 + pre last_act_raise;
.require last_act_raise <= n;
.ensure active;
.tel
```
CoCoSpec

- An Assume/Guarantee-based Contract Language

```plaintext
node component(on, off: bool) returns (active: bool) ;
--!contract : nop    ;
--!contract : inhibited ;
let
  -- Implementation.
tel

contract inhibited(on, off: bool) returns (active: bool) ;
var
  act_raise: bool ; last_act_raise: int ;
let
  active_raise = false -> active and not pre active ;
  last_act_raise = 0  -> if pre active_raise then 1
                    else 1 + pre last_act_raise ;
require last_act_raise <= n ;
ensure active ;
tel
```
Our current approach:

- Simulink/Stateflow
- Lustre
- C/C++
- JAVA
- Safety Properties

Uniform intermediate verification language

Different techniques for automated contract generation

Assume/Guarantee contract

CoCoSpec

Pre-delivery Verification Stage

Generalized PDR, Concolic Execution, Automata Learning...
Pre-delivery Verification Stage

Our current approach:

- **Simulink/Stateflow**
- **Lustre**
- **C/C++**
- **JAVA**
- **Safety Properties**

Uniform intermediate verification language

Different techniques for automated contract generation

Assume/Guarantee contract

**CoCoSpec**
Zustre

A verification engine and CoCoSpec generator for Lustre program

Lustre + safety property

Modular compiler

Lustre2Horn

Horn clauses

SPACER

Z3

Unsafe (CEX) Safe

Zustre

A verification engine and CoCoSpec generator for Lustre program

Lustre + safety property

Lustre2Horn

Horn clauses

SPACER

Z3

CoCoSpec

Modular compiler

Generalized Property-based Reachability

Unsafe (CEX) × Safe

modular invariants

NB. Currently it only infers guarantees

Constrained Horn Clause
Constrained Horn Clause

- A fragment of First Order Logic.
- A uniform way to represent transition systems for verification.

\[ F : \text{set of function symbols} \]
\[ P : \text{set of predicate symbols} \]
\[ V : \text{set of variables} \]

Constrained Horn Clause (CHC) is a formula:

\[ \forall V \cdot (\phi \land p_1[X_1] \land \cdots \land p_n[X_n] \rightarrow h[X]), \text{ for } n \geq 0 \]

- \( \phi \): constraint over \( F \cup V \) with respect to some background theory e.g. arithmetic, arrays, SMT
- \( X_i, X \subseteq V \): (possibly empty) vectors of variables
- \( p_1, \ldots, p_n, h \): n-ary predicates
- \( p_i[X_i] \): application \( p(t_1, \ldots, t_n) \) of an \( n \)-ary predicate symbol
node therm_control (actual: real; up, dn: bool)
  returns (heat, cool: bool)
var desired, margin: real;
let
  margin = 1.5;
  desired = 21.0 → if dn then (pre desired) – 1.0
                  else if up then (pre desired) + 1.0
                  else (pre desired);
  cool = (actual – desired) > margin;
  heat = (actual – desired) < –margin;
end
Example 1

[ \text{node} \ \text{therm\_control} ( \text{actual: real; up, dn: bool}) ]

\begin{align*}
\text{returns} & \quad (\text{heat, cool: bool}) \\
\text{var} & \quad \text{desired, margin: real}; \\
\text{let} & \quad \text{margin} = 1.5; \\
& \quad \text{desired} = 21.0 \rightarrow \text{if} \ \text{dn} \ \text{then} \ (\text{pre} \ \text{desired}) - 1.0 \\
& \quad \quad \quad \text{else if} \ \text{up} \ \text{then} \ (\text{pre} \ \text{desired}) + 1.0 \\
& \quad \quad \quad \text{else} \ (\text{pre} \ \text{desired}); \\
& \quad \text{cool} = (\text{actual} - \text{desired}) > \text{margin}; \\
& \quad \text{heat} = (\text{actual} - \text{desired}) < -\text{margin}; \\
\text{tel} &
\end{align*}

\[
[ \text{margin} = 1.5 \\
\wedge \text{desired} = 21.0 \\
\wedge \text{cool} = \text{actual} - \text{desired} > \text{margin} \\
\wedge \text{heat} = \ldots ] \Rightarrow TC_{\text{init}} (\text{actual, up, dn, heat, cool, desired})
\]

\[
[ \text{margin} = 1.5 \\
\wedge \text{desired}' = \text{ite} (\text{dn} \ (\text{desired} - 1.0) \ (\text{ite} \ldots)) \\
\wedge \text{cool} = \text{actual} - \text{desired}' > \text{margin} \\
\wedge \text{heat} = \ldots ] \Rightarrow TC_{\text{trans}} (\text{actual, up, dn, heat, cool, desired, desired'})
\]

\[
TC_{\text{init}} (\text{actual, up, dn, heat, cool, desired}) \Rightarrow \text{Loop}(\text{actual, up, dn, heat, cool, desired})
\]

\[
\text{Loop}(\text{actual}', \text{up}', \text{dn}', \text{heat}', \text{cool}', \text{desired}') \\
\wedge TC_{\text{trans}} (\text{actual, up, dn, heat, cool, desired, desired'}) \\
\Rightarrow \text{Loop}(\text{actual, up, dn, heat, cool, desired'})
\]
Zustre

A verification engine and CoCoSpec generator for Lustre program

Lustre + safety property

\[ \text{Lustre2Horn} \]

\[ \text{Horn clauses} \]

\[ \text{SPACER} \]

\[ \text{Z3} \]

- \[ N(I, O, S, S') = \bigwedge \varphi \] : an invariant for a node \( N \)
- Modular invariants: invariants for each node

Unsafe (CEX)

Safe

modular invariants
node Sofar( X : bool ) returns ( Y : bool );
let
  Y = (true -> pre Y) and X;
tel

-- assignment other

node Store( Delta : int ) returns ( Total : int );
var Prev : int;
let
  Prev = 0 -> pre Total;
  Total = if Delta < 0 and Prev > 0 then Prev+Delta
           else if Delta > 0 and Prev < 10 then Prev+Delta
           else Prev;
tel

node top( Delta : int ) returns ( OK : bool );
var Total : int;
  S: bool;
    -- Delta_const : int;
let
  -- Delta_const = Delta -> pre Delta_const;
  Total = Store( Delta );

S = Sofar( -1 <= Delta and Delta <= 1 );
OK = S => 0 <= Total and Total <= 20;
-- IPROPERTY : OK=true;
-- IMAIN:true;
tel
From Horn Clauses to CoCoSpec

node Sofar( X : bool ) returns ( Y : bool );
let
  Y = (true -> pre Y) and X;
let

-- assignment other
node Store( Delta : int ) returns ( Total : int );
var Prev : int;
let
  Prev = 0 -> pre Total;
  Total = if Delta < 0 and Prev > 0 then Prev+Delta
         else if Delta > 0 and Prev < 10 then Prev+Delta
         else Prev;
let

node top( Delta : int ) returns ( OK : bool );
var Total : int;
  S: bool;
  -- Delta_const : int;
let
  -- Delta_const = Delta -> pre Delta_const;
  Total = Store( Delta );

S = Sofar( -1 <= Delta and Delta <= 1 );
OK = S => 0 <= Total and Total <= 20;
--iPROPERTY : OK=true;
--iMAIN: true;

let
From Horn Clauses to CoCoSpec

node Sofar( X : bool ) returns ( Y : bool );
let
  Y = (true -> pre Y) and X;
end

-- assignment other
node Store( Delta : int ) returns ( Total : int );
var Prev : int;
let
  Prev = 0 -> pre Total;
  Total = if Delta < 0 and Prev > 0 then Prev+Delta
    else if Delta > 0 and Prev < 10 then Prev+Delta
    else Prev;
end

node top( Delta : int ) returns ( OK : bool );
var Total : int;
  S: bool;
    -- Delta_const : int;
let
  -- Delta_const = Delta -> pre Delta_const;
  Total = Store( Delta );
  S = Sofar( -1 <= Delta and Delta <= 1 );
  OK = S => 0 <= Total and Total <= 20;
--iPROPERTY : OK=true;
--iMAIN: true;
end
From Horn Clauses to CoCoSpec

NB. We use Z3 tactics to simplify (manipulate) formulas
Pre-delivery Verification Stage

Our current approach:

- Simulink/Stateflow
- Lustre
- C/C++
- JAVA
- Safety Properties

Uniform intermediate verification language

Different techniques for automated contract generation

Assume/Guarantee contract

Constrained Horn Clauses

Generalized PDR, Concolic Execution, Automata Learning...

CoCoSpec
Pre-delivery Verification Stage

Our current approach:

- Simulink/Stateflow
- Lustre
- C/C++
- JAVA
- Safety Properties

Unified intermediate verification language

Different techniques for automated contract generation

Assume/Guarantee contract

CoCoSpec
Integrated Analysis Framework

MATLAB R2013a

>> coco('.../test/gac/properties/property_3_test.mdl')
(Info) [geneode] Welcome to the CoCo -- Contract generation and verification of Simulink models
MATLAB Sim2PreludeLustre is free software: you can redistribute it
and/or modify it under the terms of the GNU General Public License
as published by the Free Software Foundation, either version 3 of
the License, or (at your option) any later version.
MATLAB Sim2PreludeLustre is distributed in the hope that it will be
useful, BUT WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.
You should have received a copy of the GNU General Public License.

(Info) [geneode] Generating Lustre code from Simulink model: .../test/gac/properties/property_3_test.mdl
(Info) [geneode] Internal representation building
(Info) [geneode] Printing original dataflow model
(Info) [geneode] Flattening of virtual SubSystems
(Info) [geneode] Printing flattened dataflow model
(Info) [geneode] Internal representation browsing for implicit data type conversions detection
(Info) [geneode] Printing flattened-type-converted dataflow model
(Info) [geneode] Code printing
(Warning) [writemode] A Terminator block have been found. No code will be generated for it:
property_3_test/Terminator
(Info) [geneode] End of code generation
(Info) [geneode] Cleaning temporary files
(Info) [traceability] Traceability data generated in file: ../test/gac/properties/src_property_3_test/property_3_test.trace
(Info) [Generation result] Lustre code generated in file: ../test/gac/properties/src_property_3_test/property_3_test.lus
(Info) [Safety] Running Lustre

lustre = 
/Users/test/Documents/GitHub/lustre/src/
(Info) [Lustre property checking] Lustre result for property node [property_3_test_observer]: SAFE

>>
Specify safety properties using synchronous observers
Specify safety properties using synchronous observers
Specify safety properties using synchronous observers
Integrated Analysis Framework
Integrated Analysis Framework
Pre-delivery Verification Stage

Our current approach:

- Simulink/Stateflow
- Sim2Lus
- Lustre
- C/C++
- JAVA
- Safety Properties
- Uniform intermediate verification language
- Different techniques for automated contract generation
- Assume/Guarantee contract
- CoCoSpec

CoCoSpec

Generalized PDR, Concolic Execution, Automata Learning ...

Constrained Horn Clauses

Zustre

Wednesday, May 20, 15
Pre-delivery Verification Stage

Our current approach:

- Simulink/Stateflow
- Sim2Lus
- Uniform intermediate verification language
- Different techniques for automated contract generation
- Assume/Guarantee contract
- Lustre
- C/C++
- JAVA
- Safety Properties
- CoCoSpec
- Constrained Horn Clauses
- Generalized PDR, Concolic Execution, Automata Learning...
A framework for verifying LLVM-based programs

SeaHorn

Program + Safety properties

Automated Analysis

Correct

Incorrect

NB. (i) Current version targets C programs (ii) and does not generate CoCoSpec

A. Gurfinkel, T. Kahsai, J. Navas, :“Algorithmic Logic-based verification”. In ACM-SIGLOG, April 2015.


SeaHorn

A framework for verifying LLVM-based programs

- Program + Safety properties
- SeaHorn
- SAFE + Certificate
- UNSAFE + CEX

NB. (i) Current version targets C programs
(ii) and does not generate CoCoSpec

A. Gurfinkel, T. Kahsai, J. Navas, :“Algorithmic Logic-based verification”. In ACM-SIGLOG, April 2015.
A framework for verifying LLVM-based programs

SeaHorn

Program + Safety properties

SAFE + Certificate

UNSAFE + CEX

NB. (i) Current version targets C programs (ii) and does not generate CoCoSpec


Post-delivery verification stage
Two Stage solution for virtual integration

Pre-Delivery Stage

System Design Model

Component Design Model (JAVA)

Component Design Model (Simulink/Stateflow)

Component prototype imp. (C/C++)

System Level Properties

Automated Contract Generation

Component Outsourced For Implementation

Post-Delivery stage

Contract - Based Component Testing

Component Black-box Implementation

Contract - Based Integration Testing

Black-box Component Delivery
Two Stage solution for virtual integration

Pre-Delivery Stage

System Design Model

Component Design Model (JAVA)

Component Design Model (Simulink/Stateflow)

Component prototype imp. (C/C++)

System Level Properties

Automated Contract Generation

Component Outsourced For Implementation

Post-Delivery stage

Contract - Based Integration Testing

Contract - Based Component Testing

Component Black-box Implementation

Black-box Component Delivery

Wednesday, May 20, 15
Post-delivery Verification Stage

Component
Black-box
Implementation

? =

CoCoSpec

Contract-based test generation

- Test generation via Bounded Model Checking
- **Coverage** and mutation oriented
- **TestEAS**: test execution and analysis system
Components are represented as transition systems:

- $s$ is the vector of state variables of the system
- $I(s_0)$ is the init predicate, true if $s_0$ is initial
- $T(s_i, s_{i+1})$ is the transition predicate, true if $s_{i+1}$ is a successor of $s_i$

Given a test objective $O(s)$, we can query an SMT solver for a trace of $k$ states leading to it:

$$I(s_0) \land T(s_0, s_1) \land \cdots \land T(s_{k-2}, s_{k-1}) \land O(s_{k-1})$$
Test generation via BMC

Components are represented as transition systems:
- $s$ is the vector of state variables of the system
- $I(s_0)$ is the init predicate, true if $s_0$ is initial
- $T(s_i, s_{i+1})$ is the transition predicate, true if $s_{i+1}$ is a successor of $s_i$

Given a test objective $O(s)$, we can query an SMT solver for a trace of $k$ states leading to it:

$$I(s_0) \land T(s_0, s_1) \land \cdots \land T(s_{k-2}, s_{k-1}) \land O(s_{k-1})$$

- Coverage-oriented: the set of test cases are generated to realize some coverage criterion on the source file, e.g. (O)MC/DC.
- Mutation-based: alter the syntax of the source code and generate test cases failing on (killing) the mutants.
Post-delivery Integration testing

- A. Cimatti et al: “A property-based proof system for contract based design”. In SEAA 2012.
Post-delivery integration testing

- pre-delivery:
  - contract-based test generation for all components,
Post-delivery integration testing

- pre-delivery:
  - contract-based test generation for all components,
  - compile complex components without their subcomponents,
Post-delivery integration testing

- **pre-delivery:**
  - contract-based test generation for all components,
  - compile complex components without their subcomponents,

- **post-delivery:**
  - unit testing of the binaries,

---

Diagram:
- Prototype: A, B, C
- Integration harness (compiled)
- Unit testing: A, B
- Test

---

Wednesday, May 20, 15
Post-delivery integration testing

- **pre-delivery:**
  - contract-based test generation for all components,
  - compile complex components without their subcomponents,

- **post-delivery:**
  - unit testing of the binaries,
  - integration testing using the compiled prototype component.

---

**Diagram:**

- **Prototype:**
  - C
  - A
  - B

- **Integration Harness (Compiled):**
  - C

- **Unit Testing:**
  - A
  - B

- **Integration Testing:**
  - A
  - B

---

Wednesday, May 20, 15
Post-delivery integration testing

- pre-delivery:
  - contract-based test generation for all components,
  - compile complex components without their subcomponents,

- post-delivery:
  - unit testing of the binaries,
  - integration testing using the compiled prototype component.
The TCM – a twin-engine tube and wings configuration aircraft simulation, scaled up from the Generic Transport Model (GTM).
Transport Class Model (TCM)

The TCM – a twin-engine tube and wings configuration aircraft simulation, scaled up from the Generic Transport Model (GTM).

UAV-sized (wingspan ~6ft) version of a plane with geometry similar to a transport -class aircraft

Intended as an experimental platform for controls and health management system
Transport Class Model (TCM)

The TCM – a twin-engine tube and wings configuration aircraft simulation, scaled up from the Generic Transport Model (GTM).

UAV-sized (wingspan ~6ft) version of a plane with geometry similar to a transport-class aircraft

Intended as an experimental platform for controls and health management system

Simulink simulator for the avionics (transport delay), actuators, engines, landing gear, aero, sensors (including noise) ...
TCM Autopilot
TCM Autopilot

Kahsai et. al. “Verifying the safety of a flight critical software”. FM’15.

- Safety verification via model checking
- Manual decomposition of ‘hard’ safety properties
NextGen Air-Traffic Control

- **NextGen.** New national airspace system in the US.
- **Air-Traffic Control.** Separation assurance: resolution of potential future conflicts between aircrafts.
- **Loss of Separation.** Two airplanes come closer than a specified safe distance (horizontally or vertically)
NextGen Air-Traffic Control

- Air-traffic control. Provides separation assurance by resolving potential future conflicts between aircraft
- Loss of separation. Airplanes come closer than a specified safe distance (horizontally and vertically)
NextGen Air-Traffic Control

NextGen component. 3-20 min time horizon
Java prototype developed at NASA Ames
- 2,500 classes, 150kloc (w/ ACES)
- 150 classes, 65kloc (w/o ACES)
- (+ NASA Worldwind, etc.)

AutoResolver
3-20 minutes prior to LoS
automated separation assurance implemented in the ground system

TSAFE
0-3 minutes prior to LoS
Collision avoidance implemented in the aircraft

TCAS
<1 minute prior to LoS

Pilot Visual Avoidance
Last resort
Summary
This talk ...

... outsourcing in flight critical software

... virtual integration of outsourced components
Two Stage solution for virtual integration

Pre-Delivery Stage

- System Design Model
  - Component Design Model (JAVA)
  - Component Design Model (Simulink/Stateflow)
  - Component prototype imp. (C/C++)

- System Level Properties

- Automated Contract Generation

- Component Outsourced For Implementation

Post-Delivery stage

- Contract - Based Integration Testing

- Contract - Based Component Testing

- Black-box Component Delivery

Wednesday, May 20, 15
Simulink/Stateflow

Lustre

C/C++

JAVA

Safety Properties

Zustre

SeaHorn

Constrained Horn Clauses

Generalized PDR, Concolic Execution, Automata Learning ...

Psycho

CoCoSpec

Kind-2

Component

Black-box

Implementation

Contract-based
testing

Wednesday, May 20, 15
Thank you
Temesghen Kahsai
Research Scientist @ RSE (Code TI) NASA Ames / CMU
email: temesghen.kahsaiazene@nasa.gov
web (work): http://ti.arc.nasa.gov/profile/tkahsaia/
web (personal): http://www.lememta.info/