AGREE: Compositional Reasoning for AADL Models

John Backes, Andrew Gacek, Darren Cofer: Rockwell Collins
Mike Whalen: University of Minnesota
Safe and Secure Systems and Software Symposium
10 June 2014
Outline

• Compositional Reasoning
• AADL
• AGREE Tool
• Examples
• Ongoing Research
Monolithic Reasoning

“Monolith” 2001: a space Odyssey
Monolithic Reasoning

- **Problem**: Verification of high level properties on monolithic models does not scale.
Monolithic Reasoning

- **Problem:** Verification of high level properties on monolithic models does not scale.

Formal Methods Experts

“Monolith” 2001: a space Odyssey
Compositional Reasoning

- **Solution:** Prove localized properties at different levels of the hierarchy in order to prove high level properties.

\[ P_A \land P_B \Rightarrow P_S \]

Formal Methods Experts

“Monolith” 2001: a space Odyssey
Compositional Reasoning

- **Solution:** Prove localized properties at different levels of the hierarchy in order to prove high level properties.

\[ P_C \land P_D \Rightarrow P_A \quad P_E \land P_F \Rightarrow P_B \]

Formal Methods Experts

“Monolith” 2001: a space Odyssey
SUCCESS!
Assume Guarantee Reasoning

• Each component has a **contract** consisting of **assumptions** and **guarantees**
  - **Assumptions**: Constraints over what a component expects to see from its environment
  - **Guarantees**: Constraints over how a component behaves in response to its environment

• The contract of a component **abstracts** the behavior of its **implementation**

• What to solve?
  - Check that system **assumptions imply component assumptions**
  - Check that **component guarantees imply system guarantees**

• How to solve?
  - Use a model checker (JKind/Kind)
Compositional Verification Example

- **Given**
  - Assumptions for system
  - Assumptions/Guarantees for subcomponents (contracts)
- **Prove**
  - System guarantees (requirements)

### Example (to prove)

- $A_S \rightarrow A_A$
- $A_S \land G_A \rightarrow A_B$
- $A_S \land G_A \land G_B \rightarrow A_C$
- $A_S \land G_A \land G_B \land G_C \rightarrow G_S$

- $(A_S)$ Assumption: Input < 10
- $(G_S)$ Guarantee: Output < 50

- $(A_A)$ Assumption: Input < 20
- $(G_A)$ Guarantee: Output < 2*Input

- $(A_B)$ Assumption: Input < 20
- $(G_B)$ Guarantee: Output < Input + 15

- $(A_C)$ Assumption: none
- $(G_C)$ Guarantee: Output = Input1 + Input2
Compositional Verification Example

- **Given**
  - Assumptions for system
  - Assumptions/Guarantees for subcomponents (contracts)
- **Prove**
  - System guarantees (requirements)

---

**Example (to prove)**

- \( A_S \rightarrow A_A \)
- \( A_S \land G_A \rightarrow A_B \)
- \( A_S \land G_A \land G_B \rightarrow A_C \)
- \( A_S \land G_A \land G_B \land G_C \rightarrow G_S \)

---

**Graphical Representation**

- **A**
  - \( (A_A) \) Assumption: Input < 20
  - \( (G_A) \) Guarantee: Output < 2*Input

- **B**
  - \( (A_B) \) Assumption: Input < 20
  - \( (G_B) \) Guarantee: Output < Input + 15

- **C**
  - \( (A_C) \) Assumption: none
  - \( (G_C) \) Guarantee: Output = Input1 + Input2

---

\( (A_S) \) Assumption: Input < 10
\( (G_S) \) Guarantee: Output < 50
Compositional Verification Example

- **Given**
  - Assumptions for system
  - Assumptions/Guarantees for subcomponents (contracts)
- **Prove**
  - System guarantees (requirements)

**Example (to prove)**

- $A_S \rightarrow A_A$
- $A_S \wedge G_A \rightarrow A_B$
- $A_S \wedge G_A \wedge G_B \rightarrow A_C$
- $A_S \wedge G_A \wedge G_B \wedge G_C \rightarrow G_S$

**Assumptions and Guarantees**

- $(A_A)$ Assumption: Input $< 20$
- $(G_A)$ Guarantee: Output $< 2 \times $Input
- $(A_B)$ Assumption: Input $< 10$
- $(G_B)$ Guarantee: Output $< $Input + 15
- $(A_C)$ Assumption: none
- $(G_C)$ Guarantee: Output $= $Input1 + Input2
- $(A_S)$ Assumption: Input $< 10$
- $(G_S)$ Guarantee: Output $< 50$
Compositional Verification Example

- **Given**
  - Assumptions for system
  - Assumptions/Guarantees for subcomponents (contracts)
- **Prove**
  - System guarantees (requirements)

**Example (to prove)**

- $A_S \rightarrow A_A$
- $A_S \land G_A \rightarrow A_B$
- $A_S \land G_A \land G_B \rightarrow A_C$
- $A_S \land G_A \land G_B \land G_C \rightarrow G_S$

**Diagrams and Assumptions/Guarantees**

- **A**
  - $A_A$ Assumption: Input < 20
  - $G_A$ Guarantee: Output < 2*Input

- **B**
  - $A_B$ Assumption: Input < 20
  - $G_B$ Guarantee: Output < Input + 15

- **C**
  - $A_C$ Assumption: none
  - $G_C$ Guarantee: Output = Input1 + Input2

- **System Guarantee**
  - $A_S$ Assumption: Input < 10
  - $G_S$ Guarantee: Output < 50
Compositional Verification Example

- **Given**
  - Assumptions for system
  - Assumptions/Guarantees for subcomponents (contracts)
- **Prove**
  - System guarantees (requirements)

Example (to prove)

- \( A_S \rightarrow A_A \)
- \( A_S \land G_A \rightarrow A_B \)
- \( A_S \land G_A \land G_B \rightarrow A_C \)
- \( A_S \land G_A \land G_B \land G_C \rightarrow G_S \)
Why AADL?

- We have been very successful at applying formal methods to software components produced in model-based development environments
- What about systems?
  - Leverage this knowledge and experience to improve the system design process
- Issues
  - Modeling language and tools
  - Different models of computation
  - Will it scale?
- AADL is a good fit and provides sufficiently formal notation
  - Architecture Analysis and Design Language
  - OSATE: open source, Eclipse-based tool (SEI)
Architecture Analysis and Design Language (AADL)

AADL = SAE AS5506 standard
Target: Embedded, real-time, distributed systems
Describes both hardware and software
Extensible syntax
Common tools: Formal Methods Workbench

- OSATE
- Trusted Build
- seL4
- eChronos
- AADL
- Architecture Translation
- Resolute
- Assurance Case
- AGRE
- Behavioral Analysis
- Lute
- Structural Analysis
- Architecture Analysis
- Kind/JKind

Assumption: Input < 20
Guarantee: Output < 2*Input

Assumption: Input < 20
Guarantee: Output < Input + 15

Assumption: none
Guarantee: Output = Input1 + Input2

Assumption: Input < 10
Guarantee: Output < 50
AGREE Language

- Expressions consist of constraints over **booleans**, **integers**, and **reals**
  - AADL types are interpreted as one of the three.
- Constraints can reason about **past states** of expressions

```plaintext
assume "target speed is positive" : Target_Speed.val >= 0.0;

guarantee "actual speed is less than constant target speed" :
  const_tar_speed => (Actual_Speed.val <= Target_Speed.val);
```
AADL Annex

- AGREE syntax is incorporated as an annex of AADL

```plaintext
system Car
  features
    Target_Speed: in data port Types::speed.speed_impl;
    Actual_Speed: out data port Types::speed.speed_impl;

  annex agree {**
    const MAX_ACCEL : real = 2.0;
    assume "target speed is positive" : Target_Speed.val >= 0.0;
    assume "reasonable target speed" : Target_Speed.val < 150.0;
    eq const_tar_speed : bool = Agree_Nodes.H(Target_Speed.val = prev(Target_Speed.val,0.0));
    guarantee "actual speed is less than constant target speed":
      const_tar_speed => (Actual_Speed.val <= Target_Speed.val);
    guarantee "acceleration is limited" :
      Agree_Nodes.abs(Actual_Speed.val - prev(Actual_Speed.val, 0.0)) < MAX_ACCEL;
  **};
end Car;
```
AADL Annex

- AGREE syntax is incorporated as an annex of AADL

```plaintext
system Car
features
  Target_Speed: in data port Types::speed.speed_impl;
  Actual_Speed: out data port Types::speed.speed_impl;

annex agree {**
  const MAX_ACCEL : real = 2.0;
  assume "target speed is positive" : Target_Speed.val >= 0.0;
  assume "reasonable target speed" : Target_Speed.val < 150.0;
  eq const_tar_speed : bool =
    Agree_Nodes.H(Target_Speed.val = prev(Target_Speed.val,0.0));
  guarantee "actual speed is less than constant target speed" :
    const_tar_speed => (Actual_Speed.val <= Target_Speed.val);
  guarantee "acceleration is limited" :
    Agree_Nodes.abs(Actual_Speed.val - prev(Actual_Speed.val, 0.0)) < MAX_ACCEL;
**};
end Car;
```
AGREE Features

- Most features provided by Xtext
  - Syntax highlighting
  - Code refactoring features
  - Variable linking
  - Type Checking

- Counterexample Views
  - Export to spreadsheet
  - Linking from results to implementation
  - Counterexample generalization (blame)
DEMO: Medical Device
Current Research
Loosening Synchrony

- **Problem:** How to analyze systems with different clocks
- It is challenging to reason about systems with different clock domains compositionally
AADL Connection Semantics

- 4 different connection types in AADL with different semantics
  - **Data Port**: Just a wire with data on it
  - **Event Port**: Like data ports with destination queues
  - **Data Access**: Shared memory
  - **Bus Access**: Multiple components reading/writing

- Challenges with modeling each type in AGREE
  - **Data Port**: How to handle fan-in?
  - **Event Port**: Queues don’t scale well and are hard to reason about compositionally
  - **Data Access**: Similar issue with handling fan-in
AADL Connection Semantics

• Possible solutions to each modeling challenges
  – Restrict the types of models that we analyze
    • E.g., don’t allow models to have multiple fan-in for a single data port
  – Break from the AADL semantics for AGREE
    • E.g., handle event ports like RPC calls rather than queues
Conclusions

• We have been successful with modeling and analyzing synchronous examples
  – In the case of the medical device, AGREE was much more capable of verifying the model
• Ongoing work is focusing on loosening synchrony
• We are adding more support for different AADL constructs
Thank You!

- Download AGREE on github!
  - [http://github.com/smaccm](http://github.com/smaccm) (or just google “git smaccm”)