Contract-based Compositional Verification of Infinite-State Reactive Systems

Cesare Tinelli

The University of Iowa

Acknowledgments

Collaborators:

Adrien Champion*, Christoph Sticksel*, Arie Gurfinkel, Temesghen Kahsai, Daniel Larraz*, Alain Mebsout*, Mudathir Mohamed, Baoluo Meng, Ruoyu Zhang

(*) Senior Kind 2 developer
Embedded Software

- Used to control the behavior of physical devices
- Typically *reactive*: continually map inputs and internal state to outputs
- Often mission- or safety-critical
- Developed modularly from components
- Development model-based
Model-based Software Development

- Software components modeled formally as computational systems
- Synchronous/asynchronous computational model
- Formal system and components amenable to formal analysis
- Expected behavior specified in terms of safety/liveness properties
- Great progress in last two decades in automating verification
- Compositional reasoning crucial for scalability
This Talk

Experiences in

- designing a contract language on top of a synchronous, dataflow modeling language for embedded software
- leveraging contracts for
  - modular and incremental development
  - compositional model checking

Discussion of

- implementation in the Kind 2 model checker
- a case study with a realistic system
Setting [McMillan, 1999, Bobaru et al., 2008]:

- (Reactive) system is composed of several components
- Every component $C[x, y]$ with inputs $x$ and outputs $y$ has a

  *contract*:
  - a set $A[x, y]$ of *assumptions* on $C$’s current input and past I/O behavior
  - a set $G[x, y]$ of *guarantees* on expected behavior, provided assumptions $A[x, y]$ hold
Assume-Guarantee Reasoning (simplified form)

Def. $C$ respects its contract $\langle A, G \rangle$ if all of its executions satisfy

$$\square A \Rightarrow \square G$$
Def. $C$ respects its contract $\langle A, G \rangle$ if all of its executions satisfy

$$\Box A \Rightarrow \Box G$$

Def. $C_1[x_1, y_1]$ uses $C_2[x_2, y_2]$ if it feeds $C_2$ some input $i$ and reads the corresponding output in $o$

$C_1$ uses $C_2$ safely if $C_1$’s executions satisfy $\Box A_2[i, o]$
**Assume-Guarantee Reasoning (simplified form)**

**Def.** $C$ *respects* its contract $\langle A, G \rangle$ if all of its executions satisfy

\[ \Box A \Rightarrow \Box G \]

**Def.** $C_1[x_1, y_1]$ *uses* $C_2[x_2, y_2]$ if it feeds $C_2$ some input $i$ and reads the corresponding output in $o$

$C_1$ uses $C_2$ *safely* if $C_1$’s executions satisfy $\Box A_2[i, o]$

**Obs.** If

1. $C_1$ uses $C_2$ safely and
2. $C_2$ respects its own contract $\langle A_2, G_2 \rangle$

then $C_2$ can be abstracted by $A_2[i, o] \land G_2[i, o]$ in $C_1$
Modeling Reactive System Components in Lustre

**Lustre**: a synchronous dataflow language [Halbwachs et al., 1992]

**Synchronous:**
all components run in parallel, based on a universal clock

**Dataflow:**
inputs, outputs, variables, constants are all infinite streams of values
**Lustre**: a synchronous dataflow language [Halbwachs et al., 1992]

**Synchronous:**
all components run in parallel, based on a universal clock

**Dataflow:**
inputs, outputs, variables, constants are all infinite streams of values

**Reactive:**
components run forever
at each clock tick, they compute outputs from current inputs and state before the next clock tick
Modeling Reactive System Components in Lustre

Lustre: a synchronous dataflow language [Halbwachs et al., 1992]

Synchronous:
all components run in parallel, based on a universal clock

Dataflow:
inputs, outputs, variables, constants are all infinite streams of values

Reactive:
components run forever
at each clock tick, they compute outputs from current inputs and state before the next clock tick

Declarative:
components defined by set of equations, no statements
A Simple Lustre Component

```lustre
node average (x, y: real) returns (out: real);
let
  out = (x + y) / 2.0;
```

node average (x, y: real) returns (out: real);
let
    out = (x + y) / 2.0 ;
tel

Circuit view:
A simple example

```plaintext
node average (x, y: real) returns (out: real);
let
    out = (x + y) / 2.0 ;
.tel

Mathematical view:

\[ \forall i \in \mathbb{N}, \ out_i = \frac{x_i + y_i}{2} \]
```
A simple example

```plaintext
node average (x, y: real) returns (out: real);
let
    out = (x + y) / 2.0;
tel

Transition system unrolled view:

clock ticks    0    1    2    3    ...
A simple example

node average (x, y: real) returns (out: real);
let
  out = (x + y) / 2.0;
tel

Transition system unrolled view:

clock ticks  0  1  2  3  ...
A simple example

```plaintext
node average (x, y: real) returns (out: real);
let
  out = (x + y) / 2.0;
```

Transition system unrolled view:

```
clock ticks  0  1  2  3  ...  
  4.0 6.0
      4.0+6.0
          2.0
            5.0
  0.0 7.0
      0.0+7.0
          2.0
            3.5
  1.0 1.0
      1.0+1.0
          2.0
            1.0
  7.0 1.0
      7.0+1.0
          2.0
            4.0
```
Combinational programs

- Basic types: `bool`, `int`, `real`

- Constants (i.e., constant streams):
  
  | 2   | 2  | 2  | 2  | 2  | 2  | 2  | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td>...</td>
</tr>
</tbody>
</table>

- Pointwise operators:

  | x   | `x_0` | `x_1` | `x_2` | `x_3` | `x_4` | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td><code>y_0</code></td>
<td><code>y_1</code></td>
<td><code>y_2</code></td>
<td><code>y_3</code></td>
<td><code>y_4</code></td>
<td>...</td>
</tr>
<tr>
<td><code>x + y</code></td>
<td><code>x_0 + y_0</code></td>
<td><code>x_1 + y_1</code></td>
<td><code>x_2 + y_2</code></td>
<td><code>x_3 + y_3</code></td>
<td><code>x_4 + y_4</code></td>
<td>...</td>
</tr>
</tbody>
</table>

- All customary operators are provided
Combinational Components

Conditional expressions

Local variables

```plaintext
code
node max (a, b: real) returns (out: real);
var
c: bool;
let
out = if c then a else b;
c = a >= b;
tel
```
Conditional expressions

Local variables

```
node max (a,b: real) returns (out: real);
var
c: bool;
let
  out = if c then a else b;
c = a >= b;
tel
```

- Equation order does not matter
- Set of equations, not sequence of statements
- Causality is resolved syntactically
Stateful Components

Previous operator \( \text{pre} : \)
\[
\begin{align*}
  (\text{pre} \ x)_0 & \quad \text{undefined} \\
  (\text{pre} \ x)_i = x_{i-1} & \quad \text{for } i > 0
\end{align*}
\]
Stateful Components

Previous operator \( \text{pre} \):
\[
\begin{align*}
(\text{pre } x)_0 &= \text{undefined} \\
(\text{pre } x)_i &= x_{i-1} \quad \text{for } i > 0
\end{align*}
\]

Initialization \( \rightarrow \):
\[
\begin{align*}
(x \rightarrow y)_0 &= x_0 \\
(x \rightarrow y)_i &= y_i \quad \text{for } i > 0
\end{align*}
\]

Examples:

\[
\begin{array}{c|cccccccc}
  x & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \ldots \\
  \text{pre } x & / / & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots \\
  y & y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \ldots \\
  x \rightarrow y & x_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \ldots \\
  2 & 2 & 2 & 2 & 2 & 2 & 2 & \ldots \\
  2 \rightarrow (\text{pre } x) & 2 & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots 
\end{array}
\]
Modularity

Components defined as *nodes* parametrized by inputs

Can have several outputs

Can be understood as macros

```plaintext
node MinMaxSoFar ( X : real ) returns ( Min, Max : real );
let
    Min = X -> if (X < pre Min) then X else pre Min ;
    Max = X -> if (X > pre Max) then X else pre Max ;
tel

node MinMaxAverageSoFar ( X: real ) returns ( Y: real ) ;
var Min, Max: real ;
let
    Min, Max = MinMax(X) ;
    Y = (Min + Max)/2.0 ;
tel
```
CocoSpec Contract Language

Our extension of Lustre with contracts [Champion et al., 2016a]

Objectives:

• follow assume-guarantee paradigm

• ease process of writing and reading formal specifications

• facilitate automatic verification of specs

• improve feedback to user after analysis

• partition information for specification-driven test generation
Contract-based specification

Contracts over components

- describe their behavior under some assumptions
- correspond to requirements from specification documents
Stopwatch Example

stopwatch(toggle, reset) → count

Assumptions:
reasonable input  ¬(reset ∧ toggle)

Guarantees:
output range  count ≥ 0, initially 0
resetting  reset implies count is 0
running  ¬reset ∧ on implies count increases by 1
stopped  ¬reset ∧ ¬on implies count does not change
node stopwatch(toggle, reset: bool) returns (c: int);
(*@contract
    var on: bool = toggle ->
        (pre on and not toggle) or (not pre on and and toggle) ;

    assume not (reset and toggle) ;
    guarantee c = 0 -> c >= 0 ;

    guarantee reset => c = 0 ;
    guarantee (not reset and on) => c = (1 -> pre c + 1) ;
    guarantee (not reset and not on) => c = (0 -> pre c) ;
*)
let ... tel
Contracts as an Abstraction Mechanism

A component’s contract is usually **simpler** than the component’s definition.

A contract is a **declarative over-approximation** of the component.

Contracts enable **modular** and **compositional** analyses in alternative to a **monolithic** one.

In compositional analyses we **abstract away** the complexity of a subsystem by its contract.
Monolithic Analysis

Monolithic:

- analyze the top level
- considering the whole system

However:

- complete system might be too complex
- changing subcomponents voids old results
- correctness of subcomponents is not addressed
Modular Analysis

Modular:

• analyze all components bottom-up
• **reusing results** from subcomponents
Modular Analysis

Modular:

- analyze all components bottom-up
- reusing results from subcomponents
Modular Analysis

Modular:
- analyze all components bottom-up
- reusing results from subcomponents
Modular Analysis

Modular:
- analyze all components bottom-up
- reusing results from subcomponents
Modular Analysis

Modular:

- analyze all components bottom-up
- reusing results from subcomponents
Modular Analysis

Modular:

- analyze all components bottom-up
- *reusing results* from subcomponents
Modular Analysis

Modular:

- analyze all components bottom-up
- **reusing results** from subcomponents
Modular Analysis

Modular:

- analyze all components bottom-up
- **reusing results** from subcomponents
Modular Analysis

Modular:

- analyze all components bottom-up
- reusing results from subcomponents
Modular Analysis

Modular:

- analyze all components bottom-up
- reusing results from subcomponents

However:

- changing subcomponents voids old results
- complexity can explode as we go up
Compositional Analysis

Compositional:

- analyze the top level
- abstracting subnodes by their contracts
- complexity of the system analyzed is reduced
- changing subcomponents preserves old results as long as new version respects contract
Compositional Analysis

Compositional:

- analyze the top level
- abstracting subnodes by their contracts
- complexity of the system analyzed is reduced
- changing subcomponents preserves old results as long as new version respects contract

However:

- counterexamples might be spurious
- correctness of subcomponents is assumed
Compositional and Modular:

- No abstraction for the leaf components.
- As we move up, we abstract subcomponents.
- In case of failure, we can restart the analysis after refining by removing the abstraction, possibly repeatedly.
- All components are checked.
- Changing subcomponents preserves old results (as long as new versions are correct).
- Results for subcomponents are reused.
- Refining identifies spurious counterexamples.
Compositional and modular:

- no abstraction for the leaf components

In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly.

- all components are checked
- changing subcomponents preserves old results (as long as new versions are correct)
- results for subcomponents are reused
- refining identifies spurious counterexamples
Compositional and modular:

- no abstraction for the leaf components

Compositional and Modular
Compositional and modular:

- no abstraction for the leaf components
Compositional and modular:

- no abstraction for the leaf components
Compositional and modular:

- no abstraction for the leaf components

In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly.

- all components are checked
- changing subcomponents preserves old results (as long as new versions are correct)
- results for subcomponents are reused
- refining identifies spurious counterexamples
Compositional and modular:

- no abstraction for the leaf components
- as we move up, we abstract subcomponents

In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly.

- all components are checked
- changing subcomponents preserves old results (as long as new versions are correct)
- results for subcomponents are reused
- refining identifies spurious counterexamples
Compositional and modular:

- no abstraction for the leaf components
- as we move up, we abstract subcomponents

In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly

- all components are checked
- changing subcomponents preserves old results (as long as new versions are correct)
- results for subcomponents are reused

refining identifies spurious counterexamples
Compositional and modular:

- no abstraction for the leaf components
- as we move up, we abstract subcomponents

In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly.

- all components are checked
- changing subcomponents preserves old results (as long as new versions are correct)
- results for subcomponents are reused
- refining identifies spurious counterexamples
Compositional and modular:

- no abstraction for the leaf components
- as we move up, we abstract subcomponents

In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly.

- all components are checked
- changing subcomponents preserves old results (as long as new versions are correct)
- results for subcomponents are reused
- refining identifies spurious counterexamples
Compositional and modular:

- no abstraction for the leaf components
- as we move up, we abstract subcomponents

In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly
Compositional and modular:

- no abstraction for the leaf components
- as we move up, we abstract subcomponents

In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly.
Compositional and modular:

- no abstraction for the leaf components
- as we move up, we abstract subcomponents

In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly
Compositional and modular:

- no abstraction for the leaf components
- as we move up, we abstract subcomponents
  In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly
- all components are checked
- changing subcomponents preserves old results (as long as new versions are correct)
- results for subcomponents are reused
- refining identifies spurious counterexamples
If all components are valid, *without refinement*:  

- the system as a whole is correct  
- changing a component by a different, *correct* one does not impact the correctness of the whole system
If all components are valid, with refinement:

- the system as a whole is correct
- but the contracts are not good enough for a compositional analysis to succeed

Refinement gives hints as to why
If we had to refine component 1 to prove 3 correct, that’s probably because 1’s contract is **too weak**
If after refining all sub-components we still cannot prove 3 correct, that’s because

- the assumptions of 3 are **too weak**, and/or
- the guarantees of 3 **do not hold**
Modes

Often, specifications are *contextual (mode-based)*:

when/if this is the case, do that
stopwatch\((\text{toggle}, \text{reset})\) \rightarrow \text{count}

Assumption:
- reasonable input \(\neg(\text{reset} \land \text{toggle})\)

Guarantee:
- output range \(\text{count} \geq 0\), initially 0

Modes:

<table>
<thead>
<tr>
<th>Modes</th>
<th>require</th>
<th>ensure</th>
</tr>
</thead>
<tbody>
<tr>
<td>resetting</td>
<td>\text{reset}</td>
<td>\text{count is }0</td>
</tr>
<tr>
<td>running</td>
<td>\neg \text{reset} \land \text{on}</td>
<td>\text{count increases by }1</td>
</tr>
<tr>
<td>stopped</td>
<td>\neg \text{reset} \land \neg \text{on}</td>
<td>\text{count does not change}</td>
</tr>
</tbody>
</table>
Often, specifications are *contextual (mode-based)*: when/if this is the case, do that

*Assume-Guarantee contracts do not adequately capture this sort of specifications*. . .

. . . because modes are simply encoded as conditional guarantees
Represent modes explicitly in the contract

A mode consists of a require (req) and an ensure (ens) clause

• expresses a transient behavior

• corresponds to a guarantee $\text{req} \Rightarrow \text{ens}$

$\Rightarrow$ separation between global behavior (guarantees)
and transient behavior (modes)
A set of modes $M$ can be added to a contract

Its semantics is an assume-guarantee pair $\langle A, G \rangle$ with

\[
A \equiv \bigvee_{m \in M} \text{req}_m \\
G \equiv \bigwedge_{m \in M} (\text{req}_m \implies \text{ens}_m)
\]

Note: $\text{req}_m$’s need not be mutually exclusive
Modes: Example

\[\text{stopwatch}(\text{toggle}, \text{reset}) \rightarrow \text{count}\]

\[
\text{var on: bool = toggle} \rightarrow (\text{pre on and not toggle}) \quad \text{or} \quad (\text{not pre on and toggle});
\]

Assumption:
- reasonable input \( \neg (\text{reset} \land \text{toggle}) \)

Guarantee:
- output range \( \text{count} \geq 0, \text{initially 0} \)

Modes:

<table>
<thead>
<tr>
<th>Mode</th>
<th>require</th>
<th>ensure</th>
</tr>
</thead>
<tbody>
<tr>
<td>resetting</td>
<td>\text{reset}</td>
<td>\text{count} = 0</td>
</tr>
<tr>
<td>running</td>
<td>\neg \text{reset} \land \text{on}</td>
<td>\text{count} \text{ increases by 1}</td>
</tr>
<tr>
<td>stopped</td>
<td>\neg \text{reset} \land \neg \text{on}</td>
<td>\text{count} \text{ does not change}</td>
</tr>
</tbody>
</table>
Detect shortcomings in the specification:

• do the modes cover all situations the assumptions allow?
• enables specification-checking before model-checking
Detect shortcomings in the specification:

- do the modes cover all situations the assumptions allow?
- enables specification-checking before model-checking

Produce better feedback for counterexamples:

- indicate which modes are active at each step
- provide a mode-based abstraction of the concrete values
- abstraction is in terms of user-specified behaviors
A CocoSpec contract is

- a set of assumptions,
- a set of guarantees, and
- a set of modes

Can contain *internal* variables

It can use *specification* nodes

Can be *inlined* in a node or *stand-alone*

Stand-alone contracts can be *imported* and *instantiated*
contract stopwatch_spec(tgl, rst: bool) returns (c: int) ;
let
    var on: bool = tgl -> (pre on and not tgl) or (not pre on and tgl) ;

assume not (rst and tgl) ;
guarantee c = 0 -> c >= 0 ;

mode resetting ( 
    require rst ; ensure c = 0 ; ) ;
mode running ( 
    require not rst and on ; ensure c = (1 -> pre c + 1) ; ) ;
mode stopped ( 
    require not rst and not on ; ensure c = (0 -> pre c) ; ) ;
tel

node stopwatch(toggle, reset: bool) returns (count: int) ;
(*@contract import stopwatch_spec(toggle, reset) returns (count) ; *)
let ... tel
In contracts, one can

- refer to modes in formulas (with ::\langle mode_name \rangle)
- call contract-free nodes

```plaintext
node count(b: bool) returns (count: int);
let
    count = (if b then 1 else 0) + (0 -> pre count);
tel

contract stopwatch_spec(tgl, rst: bool) returns (c: int);
let
    ...
    mode running (...);
    mode stopped (...);
    ...
    guarantee not (::running and ::stopped);
    guarantee count(::resetting) > 0 => c < count(true);
tel
```
Defensive check:

- modes **must** cover all reachable states
- **may** be declared as mutually exclusive

Check performed on the spec, **independently of the implementation**
Modes: Advantages

Defensive check:

• **modes** must cover **all reachable states**
• **may** be declared as **mutually exclusive**

Check performed on the spec, **independently of the implementation**

Mode references:

• can refer to a mode directly as a propositional var
• can write more **robust / trustworthy spec**
• can express guarantees **about the spec easily**
Modes: Advantages

Mode reachability:

- modes provide a finite abstraction of component (abstract state at time $i = \text{set of modes active at time } i$)
- can explore graph of connected modes
- from the initial state (BMC style)
- to compare with user’s understanding
Modes: Advantages

Mode reachability:

- modes provide a finite abstraction of component
  (abstract state at time $i = \text{set of modes active at time } i$)
- can explore graph of connected modes
- from the initial state (BMC style)
- to compare with user’s understanding

Abstraction for counterexample (cex) traces:

- cex traces feature concrete values and can be hard to read
- we can annotate states with active modes
- therefore abstracting the states using user-provided information
Modes: Advantages

Test generation:

- can generate **witnesses** for abstract executions
- thus obtaining **specification-based, implementation-agnostic** test cases from the model
CocoSpec Support

CocoSpec is fully supported by Kind 2 model checker

Kind 2 [Champion et al., 2016b]:

- multi-engine SMT-based safety checker for Lustre models
- competitive with state-of-the-art checkers for infinite-state systems
- engines run concurrently and cooperatively
- can run modular / compositional, mode-aware analysis
- implements all the features discussed so far
- used at Rockwell Collins, GE, Peugeot, …
Case Study: Transport Class Model (TCM)

System developed by NASA Langley in Simulink [Brat et al., 2015]

Generic model of a mid-size, twin-engine transport aircraft [Hueschen, 2011]

System requirements elicited from Federal Avionic Regulations
We formalized in Lustre TCM’s mode logic + autopilot controllers
[Champion et al., 2016a]

- looks arbitrarily far in the past
- non-linear arithmetic expressions
Case Study: Transport Class Model (TCM)

We formalized in Lustre TCM’s mode logic + autopilot controllers [Champion et al., 2016a]

- looks arbitrarily far in the past
- non-linear arithmetic expressions

Hi-level architecture:
TCM formalization in CoCoSpec+Lustre and analysis with Kind 2

- Guessed contracts for subcomponents mostly by trial and error (auto-active model checking?)
- Mode-related feedback invaluable for us, not aviation experts, to specify TCM
- Additional contracts added to abstract non-linear arithmetic expressions
- Monolithic analysis unsuccessful after several hours
- Modular and compositional analysis successful on the whole subsystem (including non-linear exprs) in under 2 minutes
Mode-based Assume-Guarantee Contracts:

- more scalable verification thanks to compositional reasoning
- bring contract language closer to specification documents
- improve user feedback (blame assignment, abstract cex traces)
- raise trust in specification, improve maintainability, ...
- enable specification-based test generation
http://kind.cs.uiowa.edu/

Thanks!
Automated assume-guarantee reasoning by abstraction refinement.

Verifying the safety of a flight-critical system.

CoCoSpec: A mode-aware contract language for reactive systems.

