A Mode-aware Contract Language for Reactive Systems

Adrien Champion      Cesare Tinelli
The University of Iowa
Overview

Introduction to contract-based compositional reasoning and its advantages

Introduction of new specification language aimed at facilitating
  • modular development and
  • compositional reasoning

Discussion of
  • implementation in Kind 2 model checker
  • examples of contract-based specifications
Compositional Reasoning in Kind 2

Based on Assume/Guarantee Paradigm

Every component $C[x, y]$ with inputs $x$ and outputs $y$ has a contract:

- a set $\mathcal{A}[x]$ of *assumptions* on $C$’s environment
- a set $\mathcal{G}[x, y]$ *guarantees* on how $C$ must behave, provided assumptions $\mathcal{A}[x]$ hold

\[^2\text{Formula } \Box \varphi \text{ is true iff } \varphi \text{ is true at all times}\]
Compositional Reasoning in Kind 2

Based on Assume/Guarantee Paradigm

Every component $C[x, y]$ with inputs $x$ and outputs $y$ has a

*contract*:

- a set $A[x]$ of *assumptions* on $C$’s environment
- a set $G[x, y]$ *guarantees* on how $C$ must behave, provided assumptions $A[x]$ hold

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

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

---

$^2$Formula $\Box \varphi$ is true iff $\varphi$ is true at all times
Def. A component $C_1[x_1, y_1]$ uses a component $C_2[x_2, y_2]$ if it feeds $C_2$ some input $a$ and reads the corresponding output in $b$. 

Note: If $C_1$ uses $C_2$ safely and $C_2$ respects its contract, one can assume $\Box G_2[a, b]$ to prove that $C_1$ respects its contract. Effectively, this means that $C_2$ can be abstracted by its contract.
Def. A component $C_1[x_1, y_1]$ uses a component $C_2[x_2, y_2]$ if it feeds $C_2$ some input $a$ and reads the corresponding output in $b$

Let $(A[x_i], G[x_i, y_i])$ be the contract of $C_i$ for $i = 1, 2$
Assume/Guarantee Reasoning (simplified form)

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

Let $(A[x_i], G[x_i, y_i])$ be the contract of $C_i$ for $i = 1, 2$.

Def. $C_1$ uses $C_2$ safely if $C_1$’s executions satisfy $\square A_2[a]$. 
Assume/Guarantee Reasoning (simplified form)

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

Let $(A[x_i], G[x_i, y_i])$ be the contract of $C_i$ for $i = 1, 2$

**Def.** $C_1$ uses $C_2$ *safely* if $C_1$’s executions satisfy $\square A_2[a]$

**Note** If $C_1$ uses $C_2$ safely and $C_2$ respects its contract, one can assume $\square G_2[a, b]$ to prove that $C_1$ respects its contract
Assume/Guarantee Reasoning (simplified form)

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

Let $(A[x_i], G[x_i, y_i])$ be the contract of $C_i$ for $i = 1, 2$

**Def.** $C_1$ uses $C_2$ safely if $C_1$'s executions satisfy $\Box A_2[a]$

**Note** If $C_1$ uses $C_2$ safely and $C_2$ respects its contract, one can assume $\Box G_2[a, b]$ to prove that $C_1$ respects its contract

Effectively, this means that $C_2$ can be abstracted by its contract
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

An extension of Lustre with contracts

Objectives:

• compatibility with the widespread assume / guarantee paradigm
• ease the 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 the specification documents
stopwatch(toggle, reset) → count

Assumptions:
- legit input
- ¬(reset ∧ toggle)

Guarantees:
- output range: count ≥ 0
- resetting: reset implies count is 0
- running: ¬reset ∧ on implies count increases by one
- 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 toggle);

    assume not (reset and toggle);
    guarantee 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
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 component by its contract.
Monolithic Analysis

Monolithic:

- analyze the top level
- considering the whole system

But

- 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

But
Modular Analysis

Modular:

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

But
Modular Analysis

Modular:

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

But
Modular Analysis

Modular:

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

But
Modular Analysis

Modular:

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

But

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

Modular:

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

But
Modular Analysis

Modular:

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

But
Modular Analysis

Modular:

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

But
Modular Analysis

Modular:

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

But

• changing subcomponents *voids old results*
Modular Analysis

Modular:

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

But

- 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 versions are correct)

But

• 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

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

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
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
Compositional and modular:

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

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
Compositional and Modular: Benefits

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 the contract of 1 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 are do not hold
Modes

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

Modes are simply encoded as conditional guarantees
stopwatch(toggle, reset) → count

Assumption:
- legit input ¬(reset ∧ toggle)

Guarantee:
- output range count ≥ 0

Modes: require ensure

- resetting reset count is 0
- running ¬reset ∧ on count increases by one
- stopped ¬reset ∧ ¬on count does not change
CocoSpec represents modes explicitly

A **mode** consists of a *require* \( (\text{req}) \) and an *ensure* \( (\text{ens}) \) 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

$$ A \equiv \bigvee_{m \in M} \text{req}_m $$

$$ G \equiv \bigwedge_{m \in M} (\text{req}_m \Rightarrow \text{ens}_m) $$
stopwatch(toggle, reset) → count

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

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

**Guarantee:**
- output range
  \[ \text{count} \geq 0 \]

**Modes:**

<table>
<thead>
<tr>
<th>Require</th>
<th>Ensure</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>count = 0</td>
</tr>
<tr>
<td>( \neg \text{reset} \land \text{on} )</td>
<td>count increases by one</td>
</tr>
<tr>
<td>( \neg \text{reset} \land \neg \text{on} )</td>
<td>count does not change</td>
</tr>
</tbody>
</table>
Motivation

Detect shortcomings in the specification:

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

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 the 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 ;

    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: bool) ;
(*@contract import stopwatch_spec(toggle, reset) returns (count) ; *)

let ... tel
In contracts, one can

- refer to modes in formulas (with `::<mode_name>`) 
- call contract-free nodes

```plaintext
node count(in: bool) returns (count: int) ;
let
    count = (if in 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
```
CocoSpec Support

CocoSpec is fully supported by Kind 2 model checker

Kind 2:

- multi-engine SMT-based safety checker for Lustre programs
- 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
References
