CS:5810 Formal Methods in Software Engineering

A Mode-aware Contract Language for Reactive Systems[1]

Adrien Champion Cesare Tinelli
The University of Iowa

[1] Copyright 20015-17, Adrien Champion and Cesare Tinelli, the University of Iowa. These notes are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holder.

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 $C[\mathbf{x}, \mathbf{y}]$ with inputs $\mathbf{x}$ and outputs $\mathbf{y}$ has a *contract*:

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

[2]Formula $\Box\varphi$ is true iff φ is true at all times

Compositional Reasoning in Kind 2

Based on Assume/Guarantee Paradigm

Every component $C[\mathbf{x}, \mathbf{y}]$ with inputs $\mathbf{x}$ and outputs $\mathbf{y}$ has a *contract*:

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

C *respects* its contract $\langle \mathcal{A},\ \mathcal{G} \rangle$ if all of its executions satisfy[2]

$$\Box\,\mathcal{A} \Rightarrow \Box\,\mathcal{G}$$

[2]Formula $\Box\,\varphi$ is true iff φ is true at all times

Assume/Guarantee Reasoning (simplified form)

Def. A component $C_1[\mathbf{x}_1, \mathbf{y}_1]$ *uses* a component $C_2[\mathbf{x}_2, \mathbf{y}_2]$ if it feeds C_2 some input $\mathbf{a}$ and reads the corresponding output in $\mathbf{b}$

Assume/Guarantee Reasoning (simplified form)

Def. A component $C_1[\mathbf{x}_1, \mathbf{y}_1]$ *uses* a component $C_2[\mathbf{x}_2, \mathbf{y}_2]$ if it feeds C_2 some input $\mathbf{a}$ and reads the corresponding output in $\mathbf{b}$

Let $(\mathcal{A}[\mathbf{x}_i], \mathcal{G}[\mathbf{x}_i, \mathbf{y}_i])$ be the contract of C_i for $i = 1, 2$

Assume/Guarantee Reasoning (simplified form)

Def. A component $C_1[\mathsf{x}_1, \mathsf{y}_1]$ *uses* a component $C_2[\mathsf{x}_2, \mathsf{y}_2]$ if it feeds C_2 some input $\mathbf{a}$ and reads the corresponding output in $\mathbf{b}$

Let $(\mathcal{A}[\mathsf{x}_i], \mathcal{G}[\mathsf{x}_i, \mathsf{y}_i])$ be the contract of C_i for $i = 1, 2$

Def. C_1 uses C_2 *safely* if C_1's executions satisfy $\Box\, \mathcal{A}_2[\mathbf{a}]$

Assume/Guarantee Reasoning (simplified form)

Def. A component $C_1[\mathbf{x}_1, \mathbf{y}_1]$ *uses* a component $C_2[\mathbf{x}_2, \mathbf{y}_2]$ if it feeds C_2 some input $\mathbf{a}$ and reads the corresponding output in $\mathbf{b}$

Let $(\mathcal{A}[\mathbf{x}_i], \mathcal{G}[\mathbf{x}_i, \mathbf{y}_i])$ be the contract of C_i for $i = 1, 2$

Def. C_1 uses C_2 *safely* if C_1's executions satisfy $\Box\, \mathcal{A}_2[\mathbf{a}]$

Note If C_1 uses C_2 safely and C_2 respects its contract, one can assume $\Box\, \mathcal{G}_2[\mathbf{a}, \mathbf{b}]$ to prove that C_1 respects its contract

Assume/Guarantee Reasoning (simplified form)

Def. A component $C_1[\mathbf{x}_1, \mathbf{y}_1]$ *uses* a component $C_2[\mathbf{x}_2, \mathbf{y}_2]$ if it feeds C_2 some input $\mathbf{a}$ and reads the corresponding output in $\mathbf{b}$

Let $(\mathcal{A}[\mathbf{x}_i], \mathcal{G}[\mathbf{x}_i, \mathbf{y}_i])$ be the contract of C_i for $i = 1, 2$

Def. C_1 uses C_2 *safely* if C_1's executions satisfy $\Box\, \mathcal{A}_2[\mathbf{a}]$

Note If C_1 uses C_2 safely and C_2 respects its contract, one can assume $\Box\, \mathcal{G}_2[\mathbf{a}, \mathbf{b}]$ to prove that C_1 respects its contract

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

Modularity in Lustre

Components defined as *nodes* parametrized by inputs

Can have several outputs

Can be understood as macros

```
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

Contract Example

stopwatch(`toggle`, `reset`) $\rightarrow$ `count`

Assumptions:

- legit input $\neg$(`reset` $\wedge$ `toggle`)

Guarantees:

- output range `count` ≥ 0
- resetting `reset` implies `count` is 0
- running $\neg$`reset` $\wedge$ `on` implies `count` increases by one
- stopped $\neg$`reset` $\wedge$ $\neg$`on` implies `count` does not change

Contract Example

```
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
```

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 component by its contract

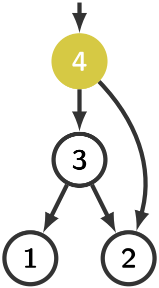
Monolithic Analysis

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

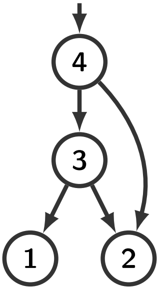


Modular Analysis

Modular:

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

But

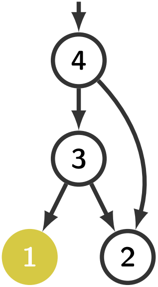


Modular Analysis

Modular:

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

But

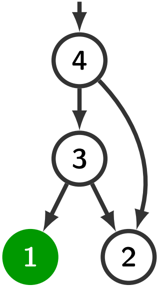


Modular Analysis

Modular:

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

But

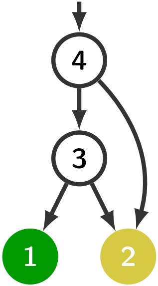


Modular Analysis

Modular:

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

But

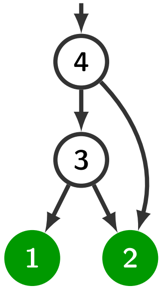


Modular Analysis

Modular:

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

But

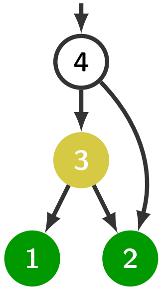


Modular Analysis

Modular:

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

But

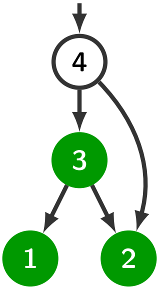


Modular Analysis

Modular:

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

But

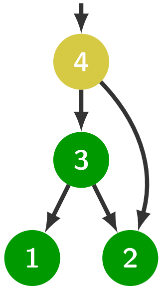


Modular Analysis

Modular:

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

But



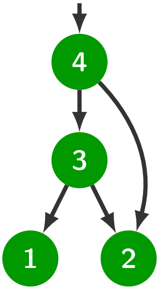
Modular Analysis

Modular:

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

But

- changing subcomponents voids old results



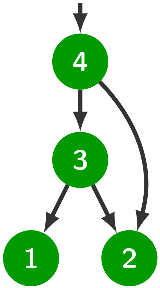
Modular Analysis

Modular:

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

But

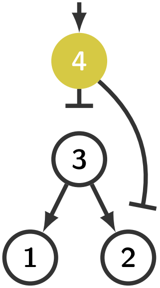
- changing subcomponents voids old results
- complexity can explode as we go up



Compositional Analysis

Compositional:

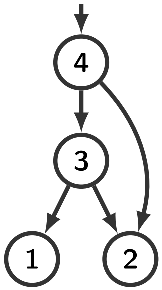
- analyze the top level
- abstracting subnodes by their contracts
- complexity of the system analyzed is reduced
- changing subcomponents preserves old results
  (as long as new versions are correct)



But

- counterexamples might be spurious
- correctness of subcomponents is assumed

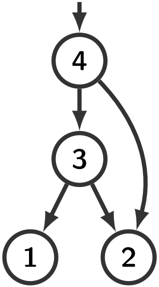
Compositional and modular:



Compositional and Modular

Compositional and modular:

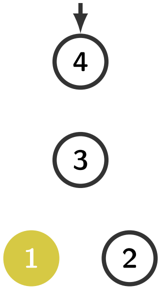
- no abstraction for the leaf components



Compositional and Modular

Compositional and modular:

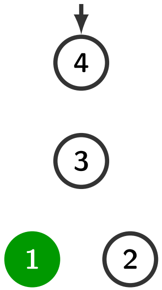
- no abstraction for the leaf components



Compositional and Modular

Compositional and modular:

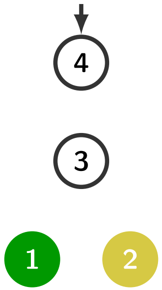
- no abstraction for the leaf components



Compositional and Modular

Compositional and modular:

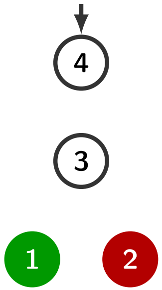
- no abstraction for the leaf components



Compositional and Modular

Compositional and modular:

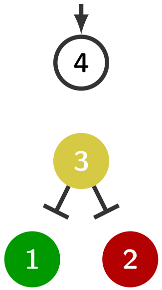
- no abstraction for the leaf components



Compositional and Modular

Compositional and modular:

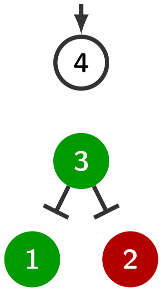
- no abstraction for the leaf components
- as we move up, we abstract subcomponents



Compositional and Modular

Compositional and modular:

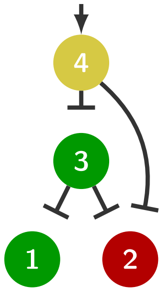
- no abstraction for the leaf components
- as we move up, we abstract subcomponents



Compositional and Modular

Compositional and modular:

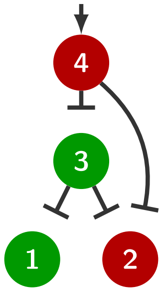
- no abstraction for the leaf components
- as we move up, we abstract subcomponents



Compositional and Modular

Compositional and modular:

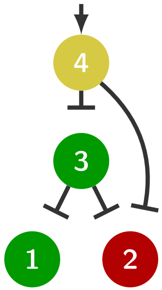
- no abstraction for the leaf components
- as we move up, we abstract subcomponents



Compositional and Modular

Compositional and modular:

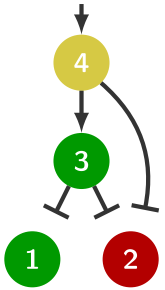
- no abstraction for the leaf components
- as we move up, we abstract subcomponents
  In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly



Compositional and Modular

Compositional and modular:

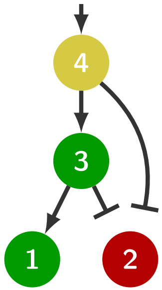
- no abstraction for the leaf components
- as we move up, we abstract subcomponents
  In case of failure we can restart the analysis
  after refining by removing the abstraction,
  possibly repeatedly



Compositional and Modular

Compositional and modular:

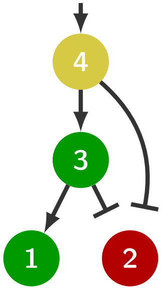
- no abstraction for the leaf components
- as we move up, we abstract subcomponents
  In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly



Compositional and Modular

Compositional and modular:

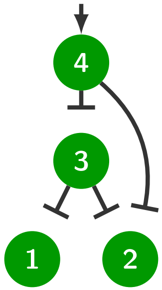
- no abstraction for the leaf components
- as we move up, we abstract subcomponents
  In case of failure we can restart the analysis
  after refining by removing the abstraction,
  possibly repeatedly
- all components are checked
- changing subcomponents preserves old results
  (as long as new versions are correct)
- results for subcomponents are reused
- refining identifies spurious counterexamples



Compositional and Modular: Benefits

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

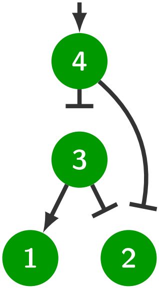
If all components are valid, with refinement:

- 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

Compositional and Modular: Benefits

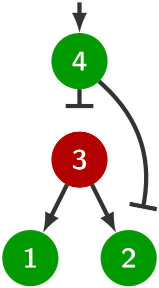
If we had to refine component 1 to prove 3 correct, that's probably because the contract of 1 is too weak



Compositional and Modular: Benefits

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

Modes: Example

stopwatch(`toggle`, `reset`) $\rightarrow$ `count`

Assumption:

- legit input — $\neg$(`reset` $\wedge$ `toggle`)

Guarantee:

- output range — `count` ≥ 0

Modes:	require	ensure
• resetting	`reset`	`count` is 0
• running	$\neg$`reset` $\wedge$ `on`	`count` increases by one
• stopped	$\neg$`reset` $\wedge$ $\neg$`on`	`count` does not change

Modes in CocoSpec

CocoSpec represents modes explicitly

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

- expresses a transient behavior
- corresponds to a guarantee `req` $\Rightarrow$ `ens`

$\Rightarrow$ separation between global behavior (guarantees)
and transient behavior (modes)

Modes in Contract

A set of modes M can be added to a contract

Its semantics is an assume / guarantee pair $\langle \mathcal{A}, \mathcal{G} \rangle$ with

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

$$\mathcal{G} \equiv \bigwedge_{m \in M} (\texttt{req}_m \Rightarrow \texttt{ens}_m)$$

Modes: Example

stopwatch(`toggle`, `reset`) $\rightarrow$ `count`

```
var on: bool = toggle -> (pre on and not toggle) or (not pre on and
 toggle) ;
```

Assumption:

- legit input — $\neg($`reset` $\wedge$ `toggle`$)$

Guarantee:

- output range — `count` ≥ 0

Modes:	require	ensure
• resetting	`reset`	`count` $= 0$
• running	$\neg$`reset` $\wedge$ `on`	`count` increases by one
• stopped	$\neg$`reset` $\wedge$ $\neg$`on`	`count` does not change

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

CocoSpec Contracts

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

Stand-alone Contract with Modes

```
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
```

Additional Features

In contracts, one can

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

```
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

[1] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. CoCoSpec: A Mode-Aware Contract Language for Reactive Systems. In Proceedings of the 14th International Conference on Software Engineering and Formal Methods (SEFM 2016), Vienna, Austria, 2016. Springer

[2] Kind 2 User Documentation.