# CS:5810 Formal Methods in Software Engineering

## Reactive Systems and the Lustre Language[1]

## Part 3

Adrien Champion  Cesare Tinelli

# 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: Assume-Guarantee Paradigm

Setting:

- (Reactive) system is composed of several components
- Every component is provided with its own high-level behavioral specification
- The high-level specification of a component $C[\mathrm{x}, \mathrm{y}]$ with inputs x and outputs y is provided by a *contract*:
  - a set $\mathcal{A}[\mathrm{x}, \mathrm{y}]$ of *assumptions* on $C$'s current input and past I/O behavior
  - a set $\mathcal{G}[\mathrm{x}, \mathrm{y}]$ of *guarantees* on expected behavior, provided assumptions $\mathcal{A}[\mathrm{x}, \mathrm{y}]$ hold

# Assume-Guarantee Reasoning (simplified form)

**Def.** $C$ *respects* its contract $\langle \mathcal{A},\ \mathcal{G} \rangle$ if all of its executions (i.e., traces) satisfy

$$\textbf{always } \mathcal{A} \ \Rightarrow\ \textbf{always } \mathcal{G}$$

# Assume-Guarantee Reasoning (simplified form)

**Def.** $C$ *respects* its contract $\langle \mathcal{A}, \mathcal{G} \rangle$ if all of its executions (i.e., traces) satisfy

$$\textbf{always } \mathcal{A} \Rightarrow \textbf{always } \mathcal{G}$$

**Def.** $C_1[\mathsf{x}_1, \mathsf{y}_1]$ *uses* $C_2[\mathsf{x}_2, \mathsf{y}_2]$ 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 **always** $\mathcal{A}_2[\mathsf{i}, \mathsf{o}]$

# Assume-Guarantee Reasoning (simplified form)

**Def.** $C$ *respects* its contract $\langle \mathcal{A},\ \mathcal{G} \rangle$ if all of its executions (i.e., traces) satisfy

$$\textbf{always}\ \mathcal{A} \ \Rightarrow\ \textbf{always}\ \mathcal{G}$$

**Def.** $C_1[\mathsf{x}_1, \mathsf{y}_1]$ *uses* $C_2[\mathsf{x}_2, \mathsf{y}_2]$ 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 **always** $\mathcal{A}_2[\mathsf{i}, \mathsf{o}]$

**Obs.** If

1. $C_1$ uses $C_2$ safely and
2. $C_2$ respects its own contract $\langle \mathcal{A}_2,\ \mathcal{G}_2 \rangle$

then $C_2$ can be abstracted by $\mathcal{A}_2[\mathsf{i}, \mathsf{o}] \wedge \mathcal{G}_2[\mathsf{i}, \mathsf{o}]$ in $C_1$

# CocoSpec: a Contract Language for Kind 2

An extension of Lustre with contracts

Objectives:

- follow assume-guarantee paradigm
- ease process of writing and reading formal specifications
- enable modular and compositional analysis
- facilitate automatic verification of specs
- improve feedback to user after analysis
- partition information for specification-driven test generation

# Contract-based specification

A contract for a component $C$

- describes declaratively $C$'s behavior under some assumptions
- captures requirements from specification documents

# Contract Example

stopwatch(`toggle`, `reset`: bool) $\rightarrow$ `count`: int

**Assumptions:**

| | |
|---|---|
| reasonable input | $\neg$(`reset` $\wedge$ `toggle`) |

**Guarantees:**

| | | | |
|---|---|---|---|
| output range | `count` $\geq 0$, | initially | 0 or 1 |
| resetting | `reset` | implies | `count` is 0 |
| running | $\neg$`reset` $\wedge$ `on` | implies | `count` increases by 1 |
| stopped | $\neg$`reset` $\wedge$ $\neg$`on` | implies | `count` does not change |

# Contract Example in Kind 2

```
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 (0 <= c and c <= 1) -> 0 <= c ;

  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

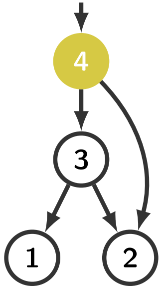
# Monolithic Analysis

Monolithic:

- analyze the top level
- considering the whole system

However:

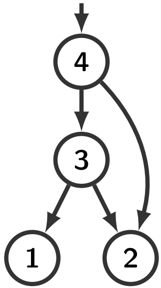
- complete system might be too complex
- changing subcomponents voids old results
- correctness of subcomponents is not addressed



# Modular Analysis

Modular:

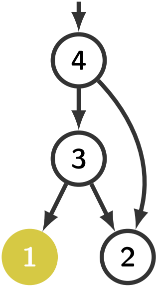
- analyze all components bottom-up
- reusing results from subcomponents



# Modular Analysis

Modular:

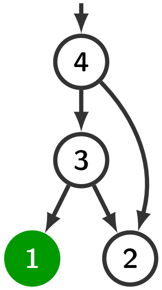
- analyze all components bottom-up
- reusing results from subcomponents



# Modular Analysis

Modular:

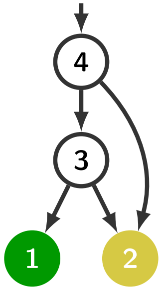
- analyze all components bottom-up
- reusing results from subcomponents



# Modular Analysis

Modular:

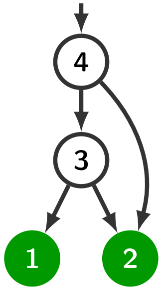
- analyze all components bottom-up
- reusing results from subcomponents



# Modular Analysis

Modular:

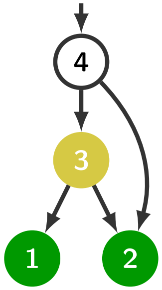
- analyze all components bottom-up
- reusing results from subcomponents



# Modular Analysis

Modular:

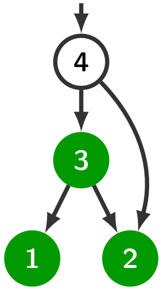
- analyze all components bottom-up
- reusing results from subcomponents



# Modular Analysis

Modular:

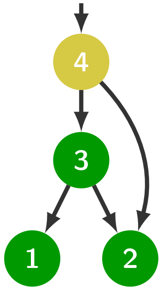
- analyze all components bottom-up
- reusing results from subcomponents



# Modular Analysis

Modular:

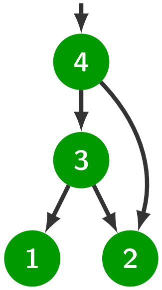
- analyze all components bottom-up
- reusing results from subcomponents



# Modular Analysis

Modular:

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



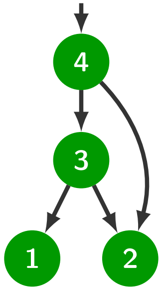
# Modular Analysis

Modular:

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

However:

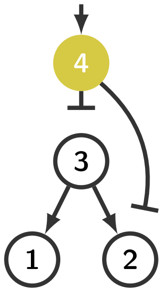
- changing subcomponents voids old results
- complexity can explode as we go up



# Compositional Analysis

Compositional:

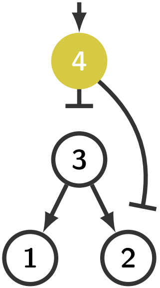
- analyze the top level
- abstracting subnodes by their contracts
- complexity of the system analyzed is reduced
- changing subcomponents preserves old results as long as new version respects contract



# 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

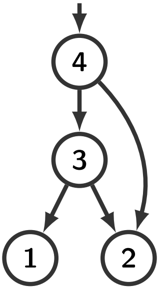



However:

- counterexamples might be spurious
- correctness of subcomponents is assumed

# Compositional and Modular

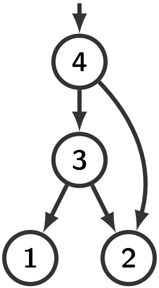
Compositional and modular:



# Compositional and Modular

Compositional and modular:

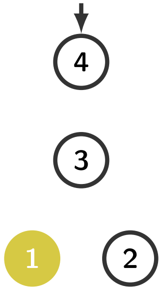
- no abstraction for the leaf components



# Compositional and Modular

Compositional and modular:

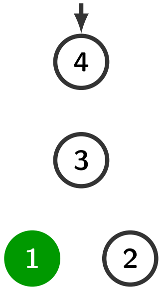
- no abstraction for the leaf components



# Compositional and Modular

Compositional and modular:

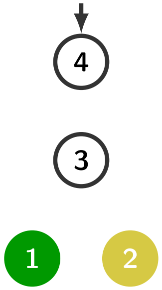
- no abstraction for the leaf components



# Compositional and Modular

Compositional and modular:

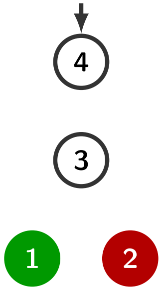
- no abstraction for the leaf components



# Compositional and Modular

Compositional and modular:

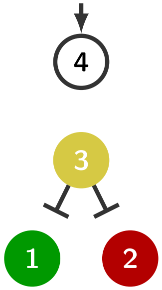
- no abstraction for the leaf components



# Compositional and Modular

Compositional and modular:

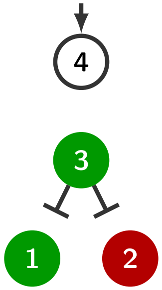
- no abstraction for the leaf components
- as we move up, we abstract subcomponents



# Compositional and Modular

Compositional and modular:

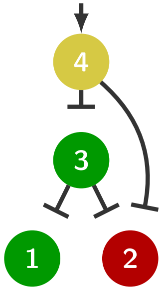
- no abstraction for the leaf components
- as we move up, we abstract subcomponents



# Compositional and Modular

Compositional and modular:

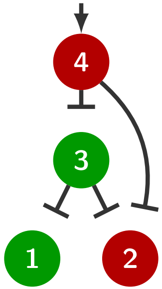
- no abstraction for the leaf components
- as we move up, we abstract subcomponents



# Compositional and Modular

Compositional and modular:

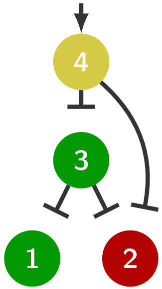
- no abstraction for the leaf components
- as we move up, we abstract subcomponents



# Compositional and Modular

Compositional and modular:

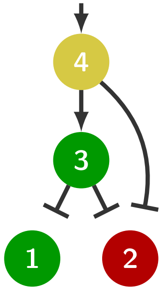
- no abstraction for the leaf components
- as we move up, we abstract subcomponents
  In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly



# Compositional and Modular

Compositional and modular:

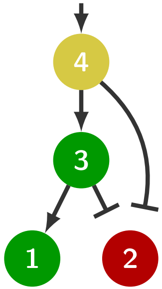
- no abstraction for the leaf components
- as we move up, we abstract subcomponents
  In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly



# Compositional and Modular

Compositional and modular:

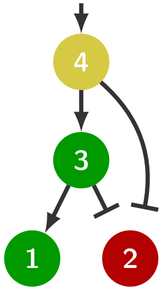
- no abstraction for the leaf components
- as we move up, we abstract subcomponents
  In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly



# Compositional and Modular

Compositional and modular:

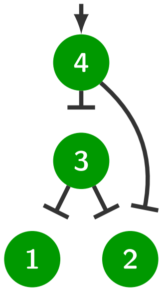
- no abstraction for the leaf components
- as we move up, we abstract subcomponents
  In case of failure we can restart the analysis after refining by removing the abstraction, possibly repeatedly
- all components are checked
- changing subcomponents preserves old results (as long as new versions are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

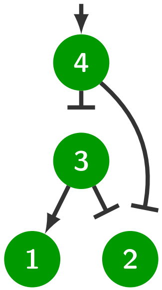
If all components are valid, with refinement:

- 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

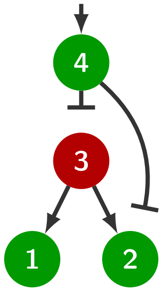
If we had to refine component 1 to prove 3 correct, that's probably because 1's contract is too weak



# 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 do not hold



# Modes

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

when/if this is the case, do that

# Modes: Example

stopwatch(`toggle`, `reset`: bool) $\rightarrow$ `count`: int

Assumption:

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

Guarantee:

- output range `count` $\geq 0$, initially 0 or 1

| Modes: | require | ensure |
|---|---|---|
| • resetting | `reset` | `count` is 0 |
| • running | $\neg$`reset` $\wedge$ `on` | `count` increases by 1 |
| • stopped | $\neg$`reset` $\wedge$ $\neg$`on` | `count` does not change |

# 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 . . .*

. . . because modes are simply encoded as conditional guarantees

# Solution

*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 `req` $\Rightarrow$ `ens`

**Effect:** Separation between

- global behavior (guarantees) and
- transient behavior (modes)

# Modes in a 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

$$\begin{aligned} \mathcal{A} &\equiv \bigvee_{m \in M} \mathtt{req}_m \\ \mathcal{G} &\equiv \bigwedge_{m \in M} (\mathtt{req}_m \Rightarrow \mathtt{ens}_m) \end{aligned}$$

**Note:** $\mathtt{req}_m$'s need not be mutually exclusive

# Modes: Example

stopwatch(toggle, reset) $\rightarrow$ count

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

Assumption:

- reasonable input $\neg(\texttt{reset} \wedge \texttt{toggle})$

Guarantee:

- output range $\texttt{count} \geq 0$, initially 0 or 1

| Modes: | require | ensure |
|---|---|---|
| • resetting | reset | $\texttt{count} = 0$ |
| • running | $\neg\texttt{reset} \wedge \texttt{on}$ | count increases by 1 |
| • stopped | $\neg\texttt{reset} \wedge \neg\texttt{on}$ | count does not change |

# Modes: Advantages

*Detect shortcomings in the specification:*

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

# Modes: Advantages

*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

# Contracts for Lustre

Kind 2's input language extends Lustre with contracts

A Kind 2 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 (0 <= c and c <= 1) -> 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
```

# Additional Features

In contracts, one can

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

```
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
```

# Modes in Kind 2: Advantages

Defensive check:

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

Check performed on the spec, independently of the implementation

# Modes in Kind 2: 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 in Kind 2: Advantages

Mode reachability:

- modes provide a finite abstraction of component
  (abstract state at time $i$ = 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 in Kind 2: Advantages

Mode reachability:

- modes provide a finite abstraction of component
  (abstract state at time $i$ = 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 in Kind 2: Advantages

Test generation:

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

# Conclusion

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