Contract-based design: a temporal logics approach

Alessandro Cimatti

cimatti@fbk.eu

Layered Assurance Workshop, 2013
Credits

• Joint work with
  – Stefano Tonetta
  – Michele Dorigatti
  – Pietro Braghieri

• Supported by the European ARTEMIS SafeCer project.
Outline

1. Contract-based design
2. Temporal Logics
3. Contract-based design with temporal logics
4. OCRA: tool support and applications
5. Conclusions and future directions
CONTRACT-BASED DESIGN
Hierarchical decomposition

• System A
• System A decomposed into subsystems B and C
• Subsystem B decomposed into equipments D and E
• Hierarchical decomposition preserves ports
Specifying components with contracts

- A component is immersed in an environment
- Its behaviour is specified by contracts
- Contract: assumptions + guarantees
- Assumptions: what the environment of the component is supposed to do
- Guarantee: what the component shall do
Wish 1: early check of requirements

• Specify components while designing
  – decomposing the specification based on the decomposition of the architecture

• Ensure the correctness of the decomposition
  – Does the contract of A follow from the contracts of B and C?
**Wish**2: component reuse

- Library of trusted components
  - e.g. the SRA of the ESA
- Implementation + contracts
- Pluggable?
  - compare contracts!
Correctness by construction!
Additional advantages

• Follows structured development process
  – E.g. aerospace domain

• Provides formal support
  – Informal to formal ("semantic gap")

• Early automated support
  – No detailed behavioural models required

• Leverages decomposition for scalability
  – No monolithic analysis

• Inherently incremental
  – Local changes do not affect whole decomposition
Ingredients for the formal view

• A formal language to specify contracts
  – Temporal logics
• A framework for correct contract refinement
  – Proof obligations
  – Logical consequence of temporal logic formulae
• A formal language to specify implementation
  – Finite state machines
• Checking implementation
  – Model checking
PROPERTY SPECIFICATION

LANGUAGES
Properties

• **Properties** are expressions in a mathematical logic using symbols of the system description.

• Used to formalize requirements.
  – Often closer to informal than behavioural descriptions

• Each property associated with set of system’s behavior.

• Problems:
  – Specification: define the properties of a system.
  – Verification: check if the system satisfies the properties.
  – Validation: check if we are considering the right properties.
  – Synthesis: construct a system that satisfies the properties.
Properties, traces, and logic

Informal statement 1
Formalized into Property $\phi_1$
Semantics

Informal statement 2
Formalized into Property $\phi_2$
Semantics

A model (trace)

A model of both

Alessandro Cimatti, LAW’13
Linear temporal logic \([\text{Pnu77}]\)

- **Linear models**
  - Traces as sequences of states
- **Built over atomic propositions**
- **Using Boolean connectives**
- **And temporal operators**
LTL examples

• Gp
  – “always p” – invariant

• G(p → Fq)
  – “p is always followed by q” – reaction

• G(p → Xq)
  – “whenever p holds, q is set to true in the next cycle”
  – immediate reaction

• GFp
  – “infinitely many times p” – fairness

• FGp
  – “eventually permanently p”

• G(p → (q U r))
Simple entailment example

• Given the following formulae
  – \( G(\text{request} \rightarrow F(\text{received})) \)
  – \( G(\text{received} \rightarrow F(\text{processed})) \)
  – \( G(\text{processed} \rightarrow X(\text{grant})) \)

• From which we can entail
  – \( G(\text{request} \rightarrow F(\text{grant})) \)
Past operators

• Y p:
  – “in the previous state p”
  – Dual of next operator X

• O p:
  – “in the past once p”
  – Dual of eventually operator F

• H p:
  – “in the past always p”
  – Dual of always operator G

• p S q (p since q)
  – “in a previous state p, and since then q”
  – Dual of until operator U
Regular expressions

• RELTL enriches LTL with regular expressions:
  – Suffix implication \{r\} \rightarrow p means that every finite sequence matching r is followed by a suffix matching p
  – Suffix conjunction \{r\} \diamond \rightarrow p means that there exists one finite sequence matching r followed by a suffix matching p

• Examples
  – {{\neg p}[*];p}[*3] \rightarrow q
  – G({request; busy[*];grant} \rightarrow response)

• Formal basis for PSL hw description language
From finite to infinite

• Use first-order predicates instead of propositions
  – \( G(x \geq a \lor x \leq b) \)
  – \( GF(x = a) \land GF(x=b) \)

• Predicates interpreted according to specific theory \( T \) (from here on, only reals used)

• Next operator to express changes/ transitions:
  – \( G(\text{next}(x) = x + 1) \)
  – \( G(\text{next}(a) – a \leq b) \)
Metric Temporal Logic

• $G(p \rightarrow F_{\leq 3} q)$
  – “p si followed by q within 3 time units”

• $G(p \rightarrow G_{\leq 20} q)$
  – “whenever p holds, then q holds in the following 20 time units”

• $G(p \rightarrow (\neg q U_{\geq 5} q))$
  – “p is followed by q but not earlier than 5 time units”
HRELTL: Hybrid RELTL

- $G(\text{der}(x) < 2)$
  - “the derivative of $x$ is always less than 2”
- $G(a \rightarrow \text{der}(x) = 0)$
  - “whenever $a$ holds, the derivative of $x$ is 0”
- $G(a \rightarrow (b \ U \ \text{der}(x) \leq 5))$
  - “Whenever $a$ holds, $b$ remains true until the der $x$ is less or equal to 5”
- $G(\text{speed} > \text{limit} \rightarrow F(\text{warning}))$
Othello

- Human-readable language for HRELTL
- Controlled natural language expressions
- Developed in the EuRailCheck project
  - Funding by the European Railways Agency
- Formalization and validation of ETCS requirements
  - Requirement “The train trip shall issue an emergency brake command, which shall not be revoked until the train has reached standstill and the driver has acknowledged the trip”
    - formalized as
      “always (train_trip implies (emergency_break_command until (der(train_location)=0 and driver_acknowledges_trip)))”
- Validation based on modeling by specification engineers
COMPONENT-BASED DESIGN

Alessandro Cimatti, LAW’13
Component

• A component has
  – A syntactic interface
  – Optionally, an internal structure
  – A behavior
  – An environment
  – Properties
Black-box component interface

• A component interface defines boundary of the interaction between the component and its environment.

• Consists of:
  – Set of input and output **ports** (syntax)
    • Ports represent visible data and events exchanged with environment.
  – Set of **traces** (semantics)
    • Traces represent the behavior, history of events and values on data ports.
A component has an internal structure.

**Architecture view:**
- Subcomponents
- Inter-connections
- Delegations

**State-machine view:**
- Internal state
- Internal transitions
- Language over the ports
Component Implementation

- $I_S$: input ports of component $S$
- $O_S$: output ports of $S$
- $V_S = I_S \cup O_S$: all ports of $S$
- $\text{Tr}(X)$ traces over $X \subseteq V_S$
  - Sequence of assignments to $X$
- State machine $\text{Imp}$ implementation of $S$ iff $L(\text{Imp}) \subseteq \text{Tr}(V_S)$
- $M$ can be associated with $\mu_{\text{Imp}}: \text{Tr}(I_S) \rightarrow 2^{\text{Tr}(O_S)}$ such that $\mu_{\text{Imp}}(\sigma_i) = \{\sigma_0 | \sigma_i \times \sigma_0 \in L(\text{Imp})\}$
  - Input trace mapped to a set of output traces
  - Set used to consider nondeterminism
  - Empty set corresponds to rejected input trace
Component Environment

• State machine $Env$ environment of $S$ iff $L(Env) \subseteq Tr(I_S)$

• Compatibility of implementation with environment (e.g. for reuse)
  – Trace-based (black-box) view:
    • $Imp$ must accept any trace of $Env$
    • $L(Env) \subseteq \{ \sigma | \mu_{Imp}(\sigma) \neq \emptyset \}$
  – State-based (glass-box) view:
    • For any reachable state of $Imp \times Env$, for any input transition of $Env$, there exists a matching transition of $Imp$
    • As in interface theory [AH01]
      – note that $Imp \times Env$ is a closed system
Composite components and connections

• Components are composed to create composite components.
• Different kind of compositions:
  – Synchronous,
  – Asynchronous,
  – Synchronizations:
    • Rendez-vous vs. buffered;
    • Pairwise, multicast, broadcast, multicast with a receiver
• Connections map (general rule of architecture languages):
  – Input ports of the composite component
  – Output ports of the subcomponents
    into
  – Input ports of the subcomponents.
  – Output ports of the composite component
System architecture

• A component is actually a component type.
• A system architecture is an instance of a composite component.
• It defines a tree of component instances.
CONTRACT-BASED DESIGN WITH TEMPORAL LOGICS
Contracts

• Properties of the component and its environment.
  – Can be seen as assertions for component interfaces.
• Contracts used to characterize the correctness of component implementations and environments.
• Typically, properties for model checking have a “fully observable” view of the system internals.
• For components instead:
  – Limited to component interfaces.
  – Structure into assumptions and guarantees.
• Contracts for OO programming are pre-/post-conditions [Meyer, 82].
• For systems, assumptions correspond to pre-conditions, guarantees correspond to post-conditions.
Trace-based contracts

• Assertions used to represent sets of traces over the component ports:
  – \( \Phi(V) \) assertion over variables \( V \)
  – \( \langle \langle \Phi \rangle \rangle \subseteq \text{Tr}(V) \) semantics of \( \Phi \)

• A contract of component \( S \) is a pair \(<A,G>\) of assertions over \( V_S \)
  – \( A \) is the assumption
  – \( G \) is the guarantee

• \( Env \) is a correct environment iff \( L(Env) \subseteq \langle \langle \Phi \rangle \rangle \)

• \( Imp \) is a correct implementation if \( L(Imp) \cap \langle \langle A \rangle \rangle \subseteq \langle \langle G \rangle \rangle \)

Example with Othello assertions:

assume:
always (Pedal_Pos1 iff Pedal_Pos2)
guarantee:
always (Pedal_Pos1 or Pedal_Pos2) implies (time_until(Brake_Line) <= 10));
Refinement: proof obligations

• Given $C=<A,G>$ contract for component
• Given $C_1=<A_1,G_1>$, ..., $<A_n,G_n>$ contracts for subcomponents
• Proof obligations for “$\{ C_i \}$ refines $C$”:
  \[ \{ (A_1 \rightarrow G_1), ..., (A_n \rightarrow G_n) \} \models A \rightarrow G \]

  \[ \{ (A_2 \rightarrow G_2), ..., (A_n \rightarrow G_n) \} \models A \rightarrow A_1 \]

  \[ \ldots \]

  \[ \{ (A_1 \rightarrow G_1), ..., (A_{i-1} \rightarrow G_{i-1}), (A_{i+1} \rightarrow G_{i+1}), ..., (A_n \rightarrow G_n) \} \models A \rightarrow A_i \]

  \[ \ldots \]

  \[ \{ (A_1 \rightarrow G_1), ..., (A_{n-1} \rightarrow G_{n-1}) \} \models A \rightarrow A_n \]
What does it mean?

- Focus on properties of father component
- \( \{(A_1 \rightarrow G_1), \ldots, (A_n \rightarrow G_n)\} \models A \rightarrow G \)
- The contract of the father component \( A \rightarrow G \) must follow from the contracts of the subcomponents

- Alternative view:
  \( \{(A_1 \rightarrow G_1), \ldots, (A_n \rightarrow G_n), A\} \models G \)
What does it mean?

- Focus on i-th subcomponent
- \{ (A_1 \rightarrow G_1), \ldots, (A_{i-1} \rightarrow G_{i-1}),
  (A_{i+1} \rightarrow G_{i+1}), \ldots, (A_n \rightarrow G_n) \} \models A \rightarrow A_i
- The assumptions of the i-th subcomponent must follow from the contracts of the other subcomponents plus the assumptions of the father component
Proof obligations

• PO’s necessary and sufficient for correct contract refinement [CT12]
• Extension to deal with asynchronous composition

• Key issue: diagnostic information!
  – In case of violation, trace
  – Localization by means of proof-based methods
    • unsat core extraction
Weak vs. strong assumptions

• Weak vs. strong assumptions (both important):
  – Weak assumptions
    • Define the context in which the guarantee is ensured
    • As in assume-guarantee reasoning
    • Different assume-guarantee pairs may have inconsistent assumptions (if x>0 then ..., if x<0 then ...)
  – Strong assumptions
    • Define properties that must be satisfied by the environment.
    • Original idea of contract-based design.
    • If not satisfied, the environment can cause a failure (division by zero, out of power, collision).
Assume-guarantee reasoning

• Corresponds to one direction of the contract refinement
• Many works focus on finding the right assumption/guarantee
• E.g. how to break circularity
  \[(G(a \rightarrow b) \land G(b \rightarrow a)) \Rightarrow G(a \land b)\]
• Induction-based mechanisms
  \[(b \land G(a \rightarrow Xb) \land a \land G(b \rightarrow Xa)) \Rightarrow G(a \land b)\]
• Note they are structural ways to prove the property-based refinement
OCRA TOOL SUPPORT
OCRA tool support

- **OCRA**=Othello Contract Refinement Analysis [CDT13]
- Contracts’ assertions specified in Othello.
- Textual representation of the architecture.
- Built on top of nuXmv for infinite-state model checking.
- Integrated with CASE tools:
  - AutoFocus3
    - Developed by Fortiss.
    - For synchronous system architectures.
  - CHESS
    - Developed by Intecs.
    - For SysML and UML modeling.
- One of the few tools supporting contract-based design for embedded systems.
- Publicly available (for non-commercial purposes) at https://es.fbk.eu/tools/ocra
OCRA main features

• Rich component interfaces to specify:
  – Input/output ports
  – Data/Event ports.
  – Including real-time and safety aspects.

• Contracts in **temporal logics**.

• Temporal formulas used to characterize set of traces over the ports of components.
OCRA language

COMPONENT system
...

COMPONENT A
...

COMPONENT B
...
Component interface

COMPONENT system
  INTERFACE
    INPUT PORT x: continuous;
    OUTPUT PORT a: boolean;

  REFINEMENT
    ...

COMPONENT A
  ...

COMPONENT B
  ...

Alessandro Cimatti, LAW’13
Othello contracts

COMPONENT simple system
  INTERFACE
    INPUT PORT x: continuous;
    OUTPUT PORT v: boolean;

    CONTRACT v_correct
      assume: always x>=0;
      guarantee: always (x=0 implies v);

  RENFEINEMENT
    ...

COMPONENT A
  ...

COMPONENT B
  ...
COMPONENT simple system

INTERFACE
  INPUT PORT x: continuous;
  OUTPUT PORT v: boolean;

  CONTRACT v_correct
    assume: always x>=0;
    guarantee: always (x=0 implies v);

REFINEMENT
  SUB a: A;
  SUB b: B;

  CONNECTION a.x := x;
  CONNECTION b.yi := a.v;
  CONNECTION v:= b.vo;

...
COMPONENT simple system

INTERFACE

INPUT PORT x: continuous;
OUTPUT PORT v: boolean;

CONTRACT v_correct
  assume: always x>=0;
  guarantee: always (x=0 implies v);

REFINEMENT

SUB a: A;
SUB b: B;

CONNECTION a.x := x;
CONNECTION b.vi := a.v;
CONNECTION v:= b.vo;

CONTRACT v_correct RefinedBy a.v_correct, b.pass;
Complete example

System

A

B

v_correct

pass

v_correct

v

x

v

x

x

v

vi

vo
The complete example

COMPONENT simple system
INTERFACE
INPUT PORT x: continuous;
OUTPUT PORT v: boolean;

CONTRACT v_correct
assume: always (x>=0);
guarantee: always (x=0 implies v);

REFINEMENT
SUB a: A;
SUB b: B;

CONNECTION a.x := x;
CONNECTION b.vi := a.v;
CONNECTION v := b.vo;

CONTRACT v_correct RefinedBy a.v_correct, b.pass;

COMPONENT A
INTERFACE
INPUT PORT x: continuous;
OUTPUT PORT v: boolean;

CONTRACT v_correct
assume: always (x>=0);
guarantee: always (x=0 implies v);

COMPONENT B
INTERFACE
INPUT PORT vi: boolean;
OUTPUT PORT vo: boolean;

CONTRACT pass
assume: true;
guarantee: always (vi implies vo);
OCRA: a parameterized view

• The flow implemented in OCRA is parametric
  – Multiple logics are supported
  – Traces can be discrete or hybrid

• Required functionalities
  – Logical entailment
OCRA hybrid aspects

• Port types are either
  – Finite types: “boolean”, enumeratives, ...
  – Infinite types: “real”, “integer”, ...
  – “continuous”: real-value ports constrained to evolve continuously in time
  – “event”: boolean-value ports assigned only on discrete transitions

• Atomic formulas may be:
  – Boolean variables.
  – Equalities.
  – Arithmetic predicates over integer, real, and continuous terms.

• Temporal operators: as in LTL, and HRELTL
Functionalities

• Syntax checking
• Refinement checking
  – Generate all po’s and run entailment checkers
• Consistency checking
  – Are there inconsistent assumptions/guarantees
• Checking implementation
  – Does component implementation satisfy contracts?
• Checking receptiveness
  – Does component implementation able to react to every input of the environment?
Contract refinement results

• For every component, for every refined contract, check refinement.

• For every proof obligation, check its validity:
  – [OK] if valid
  – [BOUND OK] if no counterexample found up to k
  – [FAIL] if found counterexample
Application in the Forever project

• OCRA integrated within CHESS design environment

• Various applications developed by industrial project partners
  – Thales-Alenia Space engineers
SenseSpacecraftRate Example
The complete example

COMPONENT SenseSpacecraftSpeed system
INTERFACE
INPUT PORT speed: continuous;
OUTPUT PORT sensed_speed: real;
OUTPUT PORT sensed_speed_is_present: boolean;

CONTRACT sense
assume: TRUE;
guarantee:
always ((sensed_speed - speed <= error) and
(sensed_speed - speed >= -error) and
sensed_speed_is_present);

REFINEMENT
SUB sensor1: SpeedSensor;
SUB sensor2: SpeedSensor;
SUB monitor1: MonitorPresence;
SUB monitor2: MonitorPresence;
SUB selector: Selector;

CONNECTION sensor1.speed := speed;
CONNECTION sensor2.speed := speed;
CONNECTION monitor1.input_is_present := sensor1.sensed_speed_is_present;
CONNECTION monitor2.input_is_present := sensor2.sensed_speed_is_present;
CONNECTION monitor1.enabled := (selector.current_use=1);
CONNECTION monitor2.enabled := (selector.current_use=2);
CONNECTION selector.input1 := sensor1.sensed_speed;
CONNECTION selector.input2 := sensor2.sensed_speed;
CONNECTION selector.switch_current_use := monitor1.absence_alarm or monitor2.absence_alarm;
CONNECTION selector.switch_current_use := monitor1.absence_alarm or monitor2.absence_alarm;
CONNECTION selector.switch_current_use := monitor1.absence_alarm or monitor2.absence_alarm;

CONTRACT sense
assume: TRUE;
guarantee: always ((sensed_speed - speed <= error) and
(sensed_speed - speed >= -error) and
sensed_speed_is_present);

COMPONENT MonitorPresence
INTERFACE
INPUT PORT input_is_present: boolean;
INPUT PORT enabled: boolean;
OUTPUT PORT absence_alarm: event;

CONTRACT monitor
assume: true;
guarantee: always (absence_alarm iff (enabled and fall(input_is_present)));}

COMPONENT Selector
INTERFACE
INPUT PORT input1: real;
INPUT PORT input1_is_present: boolean;
INPUT PORT input2: real;
INPUT PORT input2_is_present: boolean;
INPUT PORT switch_current_use: boolean;
OUTPUT PORT output: real;
OUTPUT PORT output_is_present: boolean;

CONTRACT select
assume: true;
guarantee:
always ((current_use=1 implies
(output=input1 and output_is_present=input1_is_present)) and
(current_use=2 implies
(output=input2 and output_is_present=input2_is_present)));
Plugin for AutoFocus (fortiss)
Plugin for AutoFocus
Related work

• Basic concepts on contract-based design for embedded systems:

• META program and AGREE tool by D. Cofer and colleagues.
  – Also on system architecture with temporal logics
  – Focus on discrete-time, and assume-guarantee reasoning (weak assumptions).
Summary

• Contract-based design powerful
  – For stepwise refinement
  – Compositional reasoning
• Temporal logic is suitable for component contracts
• Contract framework parameterized by logic
• SMT-based model checking used to reason with expressive properties
• OCRA tool support
Model-based safety assessment

• Safety assessment
  – Analyze behaviour of system under faults
  – Artifacts: Fault Trees, FMEA tables

• Model-based Safety Assessment
  – Extend nominal model with faults
    • Symbolic fault injection
  – Analyze extended model

• Key issues
  – Requires behavioural models, comes too late
  – Requires model of faults, requires human intervention
  – No use of hierarchical decomposition, flat fault trees
Hierarchical safety assessment

• Extend contract-based design for safety assessment
• Automatic fault injection based on failure to fulfill contract
  – Environment violates assumptions
    • Power always available
  – Component violates assumptions
    • Braking action delivered
Hierarchical Safety Assessment

Alessandro Cimatti, LAW’13

Fault Injection

Safety Assessment

Failures Dependences
The Wheel Braking System [ARP10]
FT for the WBS

- Automatically generated (cfr [ARP10])

Unannunciated loss of all wheel braking

- Loss of all wheel braking
- Loss of annunciation capability

Alternate Brake System does not operate
- Normal Brake System does not operate
- Emergency Brake System does not operate

- Loss of Green Hydraulic Supply
- Loss of Normal Brake System Hydraulic component
- Loss of BSCU ability to command braking

- BSCU fault causes loss of braking commands
- Loss of aircraft Electrical Power to BSCU

BSCU1 failure causes loss of braking commands
- BSCU Validity Monitors incorrectly report dual failures

BSCU2 failure causes loss of braking commands
- Switch failed “stuck” in intermediate position

Alessandro Cimatti, LAW’13
Future directions

• Application level
  – Usability
  – Advanced NLP techniques?
  – Diagnostic information

• Technological level
  – Improve back ends for hybrid cases
    • The nuXmv tool: NuSMV + MathSAT
  – Devise synthesis techniques
    • Strongest contract such that...
    • Region of assignment to parameters such that...