Specification, Synthesis and Implementation of Contract-based Applications via Contract Automata

Davide Basile

Permanent Researcher

ISTI CNR, Pisa

Overview

- behavioural contracts and MSCA;
- software support and an example;
- more details on the synthesis algorithms;
- variability and configurations;
- ongoing and future work.

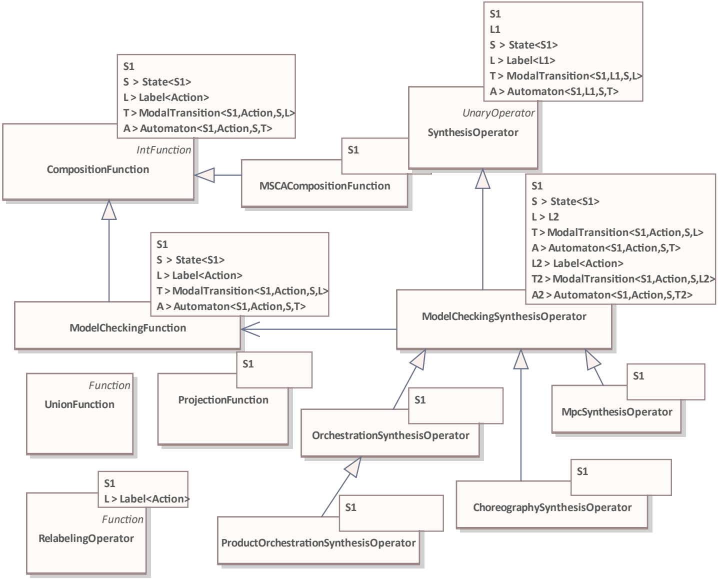
Behavioural Contracts

- Behavioural contracts have been introduced in the literature to model the behaviour of ensembles of services in terms of their interactions;
- they can be used to reason formally about well-behaving properties of ensembles of services, and to build applications that are verified by construction against these properties.
- Behavioural contracts modelled as Finite State Automata are dubbed **contract automata**.
- in contract automata services match their requests and offers between each other to reach an agreement

Modal Service Contract Automata (MSCA)

- MSCA are FSA enhanced with:
  - Partitioned alphabet of actions:
    - offers $!a$ (or $\overline{a}$) (A^o) and requests $?a$ (or a) (A^r)
    - special idle action ($\bullet \notin A^o \cup A^r$)
  - rank : the number of principals in the contract,
  - Transitions partitioned into permitted ($T^\diamond$) and necessary ($T^\Box$),
  - Labels are list of actions and are constrained to be:
    - *offers*: $(\bullet, \bullet, \bullet, !a)$,
    - *requests*: $(\bullet, ?a, \bullet, \bullet)$,
    - *matches*: $(\bullet, ?a, \bullet, !a)$,
  - $size(list) = rank$

Operations of Contract Automata

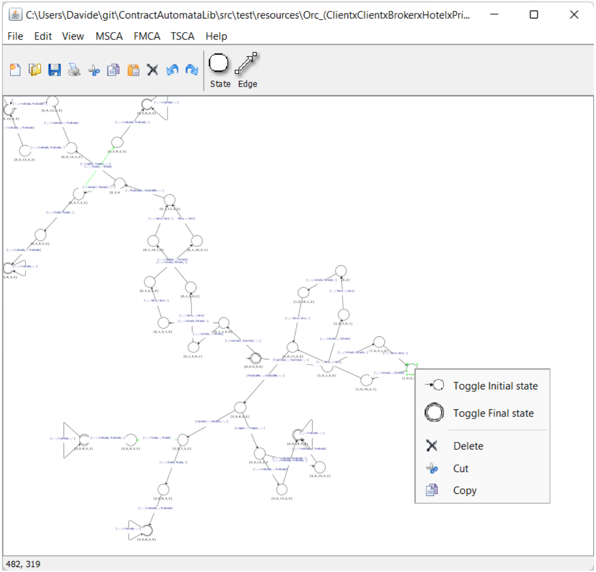


Software Support

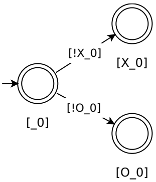
- `CATLib` : library implementing contract automata and their operations;
- `CAT_App` : GUI for designing contract automata;
- `CARE` : Runtime Environment for implementing applications specified via contract automata (work in progress);

https://github.com/contractautomataproject

CAT_App

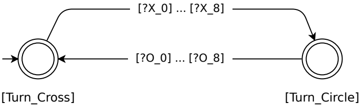

C:\Users\Davide\git\ContractAutomataLib\src\test\resources\Orc_(ClientxClientxBrokerxHotelxPri...
File
Edit
View
MSCA
FMCA
TSCA
Help
State
Edge
Toggle Initial state
Toggle Final state
Delete
Cut
Copy
482, 319

CATLib Example Tic-tac-toe




```
//create a list of automata, one for each position, to write either X or O in that position
List<Automaton<String,Action,State<String>,ModalTransition<String, Action, State<String>,
  CALabel>>>  aut = IntStream.range(0, size).mapToObj(i -> {
    State<String> cs_can = new State<>(List.of(new BasicState<>("_" + i, true, true)));
    State<String> cs_cross = new State<>(List.of(new BasicState<>(Cross.cross+"_"+i,false,true)));
    State<String> cs_circle = new State<>(List.of(new BasicState<>(Circle.circle+"_"+i,false,true)));
     return new Automaton<>(Map.of(Cross.cross, cs_cross, Circle.circle, cs_circle)
                .entrySet().stream().map(e -> new ModalTransition<>(cs_can,
                      new CALabel(1, 0, new OfferAction(e.getKey() + "_" + i)),
                      e.getValue(), ModalTransition.Modality.PERMITTED))
               .collect(Collectors.toSet()));}).collect(Collectors.toList());
```

CATLib Example Tic-tac-toe




```
//creating an automaton requiring turns between X and O
State<String> cs_cross = new State<>(List.of(new BasicState<>("TurnCross", true, true)));
State<String> cs_circle = new State<>(List.of(new BasicState<>("TurnCircle", false, true)));
aut.add(new Automaton<>(Stream.concat( //add cross turn and circle turn transitions
    actionsCross.stream().map(ac -> new ModalTransition<>(cs_cross, new CALabel(1, 0, ac),
                             cs_circle, ModalTransition.Modality.PERMITTED)),
    actionsCircle.stream().map(ac -> new ModalTransition<>(cs_circle, new CALabel(1, 0, ac),
                              cs_cross, ModalTransition.Modality.PERMITTED))
    ).collect(Collectors.toSet())));
```

CATLib Example Tic-tac-toe

```
//computing the composition
MSCACompositionFunction<String> mcf = new MSCACompositionFunction<>(aut,
    t -> { Grid m = new Grid(t.getSource().toString());
           return new StrongAgreement().negate().test(t.getLabel()) || m.win() || m.tie();});
return mcf.apply(Integer.MAX_VALUE);

//turning the opponent to uncontrollable and mark winning states
...
//mpc synthesis
MpcSynthesisOperator<String> mso = new MpcSynthesisOperator<>(l->true);
return mso.apply(new Automaton<>(transitions));
```

App.java

```
//application main cycle
currentState = strategy.getInitial(); /* the game starts from the initial state */
while(currentState!=null){ //the forward star is the set of possible next moves in the game
        Set<ModalTransition<String,Action,State<String>,CALabel>> forwardStar =
                                        strategy.getForwardStar(currentState);
        //checking if a winning or tying state is reached, otherwise execute one turn
        if (check(forwardStar)) { currentState=null; } else {
          Symbol turn = (currentState.getState().get(9).getState().equals("TurnCross"))?
                                new Cross() : new Circle();
          if (player.getClass().equals(turn.getClass()))
                        { /* user turn */ currentState = insertPlayer(scan,forwardStar); }
          else  { /* computer turn */ currentState = insertOpponent(forwardStar); }
          System.out.println(new Grid(currentState.toString()));/* printing the grid */ } }
```

https://github.com/contractautomataproject/tictactoe

CATLib Evaluation

Phase	Name
Continuous integration	GitHub Actions
Build	Maven
Testing	JaCoCo, Coveralls, SonarCloud
Unit testing	Mockito
Mutation testing	PITest, Stryker
Analysis	SonarCloud, IntelliJ, CodeQL, SpotBugs, Codiga, Codacy

Table 1: Frameworks and services used for evaluating **CATLib**

Source code		Testing	
Measure	Value	Measure	Value
LOC	2519	Total unit tests	462
Total lines	5152	Total integration tests	105
Statements	947	Total tests	567
Functions	223	Unit tests (LOC)	4565
Classes	49	Integration tests (LOC)	1526
Comment lines	1139	Total tests (LOC)	6091
Comments (%)	31.1	Tests line coverage (%)	100
Lines to cover	1238	Tests branch coverage (%)	100
Conditions to cover	626	Total mutants	795
Cyclomatic complexity	630	Killed mutants	780
Cognitive complexity	287	Timed out mutants	12
		Tests ran	1173
		Tests run per mutation	1.48
		Test suite strength (%)	99.6

Table 2: Statistics of evaluating **CATLib**: source code and testing

Synthesis of Contract Automata

- adapted from the Supervisory Control Theory for Discrete Event Systems (SCT)
- input: an FSA modelling a plant (with controllable and uncontrollable events)
  - forbidden states, marked states
- output: a controller (i.e. a refinement) that is non-blocking, safe, controllable and maximal,
- the synthesis algorithm prunes transitions in a backward fashion and updates the set of bad states R

Abstract synthesis

- the syntheses algorithms of the mpc, orchestration and choreography differ in the way in which transitions are pruned and states are deemed bad
- it is possible to abstract away such conditions through
  - pruning predicate ϕ_p
  - forbidden predicate ϕ_f

Abstract synthesis

Definition

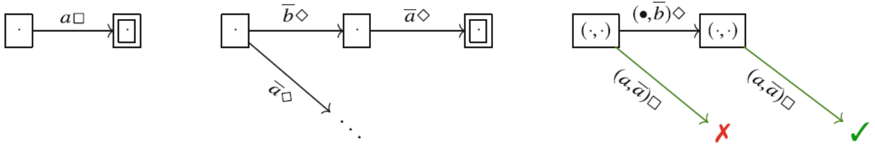
$\mathcal{K}_0 = \mathcal{A}$ and $R_0 = Dangling(\mathcal{K}_0)$.

$$f_{(\phi_p,\phi_f)}(\mathcal{K}_{i-1}, R_{i-1}) = (\mathcal{K}_i, R_i), \text{ with}$$
$$T_{\mathcal{K}_i} = T_{\mathcal{K}_{i-1}} \setminus \{ (\vec{q} \xrightarrow{\vec{a}}) = t \in T_{\mathcal{K}_{i-1}} \mid \phi_p(t, \mathcal{K}_{i-1}, R_{i-1}) = true \}$$
$$R_i = R_{i-1} \cup \{ \vec{q} \mid (\vec{q} \rightarrow) = t \in T^{\square}_{\mathcal{A}},\ \phi_f(t, \mathcal{K}_{i-1}, R_{i-1}) = true \} \cup Dangling(\mathcal{K}_i)$$

- $\phi_p(t, \mathcal{K}_{i-1}, R_{i-1}) = true$: prune transition
- $\phi_f(t, \mathcal{K}_{i-1}, R_{i-1}) = true$: source state is bad

$$\phi_p^{mpc}((\vec{q}, \vec{a}, \vec{q}'), \mathcal{K}, R) = (\vec{q}' \in R) \vee (\vec{q} \text{ is forbidden})$$
$$\phi_f^{mpc}((\vec{q}, \vec{a}, \vec{q}'), \mathcal{K}, R) = (\vec{q}' \in R)$$

Semi-controllability, Mpc vs Orchestration vs Choreography



- necessary $\rightarrow$ semi-controllable: existentially quantified
  - it can be pruned as long as the same request is matched somewhere else

$$\phi_p^{orc}((\vec{q},\vec{a},\vec{q}'),\mathcal{K},R) = (\vec{a} \text{ is a request }) \vee (\vec{q}' \in R)$$

$$\phi_f^{orc}((\vec{q},\vec{a},\vec{q}'),\mathcal{K},R) = \nexists (\vec{q_2} \xrightarrow{\vec{a}_2} \vec{q_2}') \in T_{\mathcal{K}}^{\square} : (\vec{a}_2 \text{ is a match }) \wedge (\vec{q}_2, \vec{q}_2' \notin Dangling(\mathcal{K})) \wedge (\vec{q}_{(i)} = \vec{q_2}_{(i)}) \wedge (\vec{a}_{(i)} = \vec{a_2}_{(i)} = a)$$

Synthesis of Choreographies

- the interactions with the orchestrator are implicit
- principals can safely interact synchronously without orchestrator if a specific condition is met
  - *principals perform their offers/outputs independently of the other principals in the composition*
- for each reachable pair of states $\vec{q}_1$, $\vec{q}_2$

$$\forall \vec{a} \text{ match action } . \ (\vec{q}_1 \xrightarrow{\vec{a}} \wedge \, snd(\vec{a}) = i \wedge \vec{q}_{1(i)} = \vec{q}_{2(i)}) \text{ implies } \vec{q}_2 \xrightarrow{\vec{a}}.$$

$$\phi^{cor}_p((\vec{q}, \vec{a}, \vec{q}'), \mathcal{K}, R) = (\vec{a} \text{ is a request or an offer }) \vee (\vec{q}' \in R) \vee (\exists\, \vec{q}_2 \in Q_{\mathcal{K}} \ : \ (snd(\vec{a}) = i) \wedge (\vec{q}_{(i)} = \vec{q}_{2(i)}) \wedge (\vec{q}_2 \xrightarrow{\vec{a}} \notin T_{\mathcal{K}}))$$

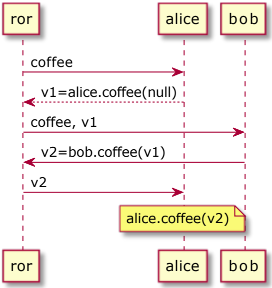
$$\phi^{cor}_f((\vec{q}, \vec{a}, \vec{q}'), \mathcal{K}, R) = \nexists (\vec{q} \xrightarrow{\vec{a}_2} \vec{q}_2') \in T^{\square}_{\mathcal{K}} \ : \ (\vec{a}_2 \text{ is a match }) \wedge (\vec{q}, \vec{q}_2' \notin Dangling(\mathcal{K})) \wedge (\vec{a}_{(i)} = \vec{a}_{2(i)} = \overline{a})$$

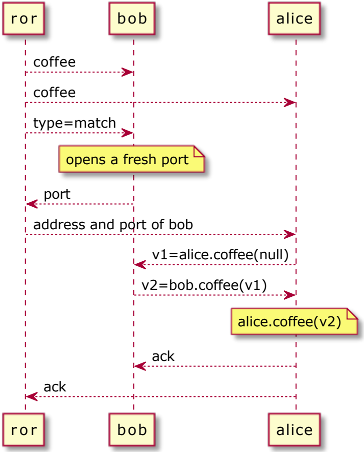
Product Lines

- FMCA $= (\mathcal{A}, \varphi)$
  - family of service applications
  - atoms = features = actions (as in MTS with variability)
- $\mathcal{A}_1 \otimes \mathcal{A}_2 \Rightarrow \varphi_{\mathcal{A}_1} \wedge \varphi_{\mathcal{A}_2}$
- Products: truth assignments satisfying φ
  - *required features* (T): must be available in $\mathcal{K}$
  - *forbidden features* (F): must not be available in $\mathcal{K}$
- orchestration of a *product* p of $\mathcal{A} \Rightarrow \mathcal{K}_{\mathcal{A}_p}$
  - further constraints on required and forbidden features of p
- orchestration of a *product line* $\mathcal{A} \Rightarrow \mathcal{K}_{\mathcal{A}}$
  - no need to generate all products!
  - union of *canonical* products

CARE - match implementation

```
[?coffee,!coffee]
Alice:  Integer coffee(String arg)
Bob:  String coffee(Integer arg)
```





Other arguments not covered...

- weak agreement, MILP implementation of the algorithms (FORTE16);
- relation with Propositional Contract Logic and Intuitionistic Linear Logic with Mix (LMCS16);
- correspondence between CA and CFSM (supported by the software), asynchronous systems, open systems (JLAMP16);
- partial order of controllers (LMCS20);
- real-time contract automata (ISSE20);
- distributed multi-agents systems (submitted to ISOLA22)

Ongoing and Future work

- more applications of `CATLib`, `CARE`
  - that will open further research goals
- support for choreographies of product lines is currently missing
- `CARE` does not support neither choreographies nor product-lines
- many others enhancements to do

  `https://github.com/orgs/contractautomataproject/projects/1`

Bibliography

- Basile, D. and ter Beek, M.H., 2022. Contract automata library. To appear in: Science of Computer Programming (Original Software Publication).
- Pugliese, R., Ter Beek, M.H. and Basile, D., 2020. Synthesis of orchestrations and choreographies: bridging the gap between supervisory control and coordination of services. Logical Methods in Computer Science, 16.
- Basile, D., ter Beek, M.H., Degano, P., Legay, A., Ferrari, G.L., Gnesi, S. and Di Giandomenico, F., 2020. Controller synthesis of service contracts with variability. Science of Computer Programming, 187, p.102344.
- Basile, D. and ter Beek, M.H., 2022. A Runtime Environment for Contract Automata. arXiv preprint arXiv:2203.14122.