A Runtime Environment for Contract Automata

Davide Basile and Maurice H. ter Beek

Formal Methods and Tools lab,
ISTI CNR, Pisa

Behavioural Contracts

- Behavioural types: model the behaviour of ensembles of services in terms of their interactions;
  - behavioural contracts, session types.
- Useful for:
  - reasoning formally about well-behaving properties,
  - building applications that are verified by construction against these properties.
- *Contract automata*: behavioural contracts modelled as FSA,
- In contract automata services match their requests and offers between each other to reach an agreement

Motivation

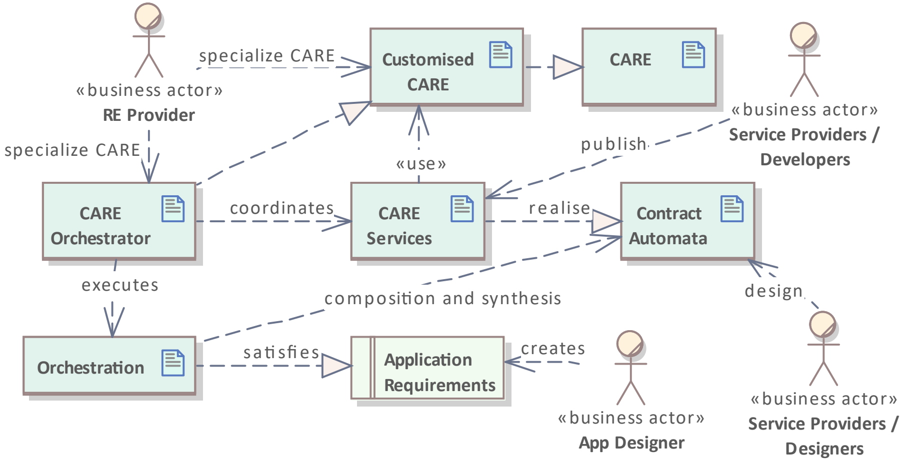
- few studies on how to derive the finalised software from the verified specification as behavioural contracts;
- these behavioural specifications are not yet a feature of standard mainstream programming languages.
- We investigate the connection between a behavioural specification given as contract automata and its implementation.

Contract Automata: Software Support

- `CATLib` : library implementing contract automata and their operations, for *specifying* applications using contract automata;
- `CAT_App` : Graphical front-end for designing contract automata;
- `CARE` : Runtime Environment for *implementing* applications specified via contract automata

`https://github.com/contractautomataproject`

CARE methodology



Operations of Contract Automata

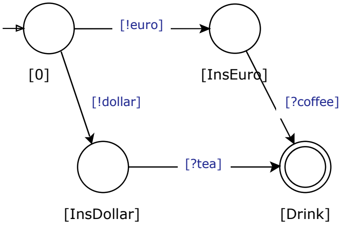
- contract automata are **composed** to model all possible behaviours,
  - composition in agreement: all requests of contracts are matched by corresponding offers,
- if the composition does not satisfy the requirements or it is not in agreement, it is possible to automatically compute (i.e., **synthesise**) a refinement of the composition such that the requirements are satisfied,
- this refined automaton is called *orchestration*,
- the orchestration automaton implicitly assumes the presence of an orchestrator coordinating the services to execute the actions prescribed by the orchestration

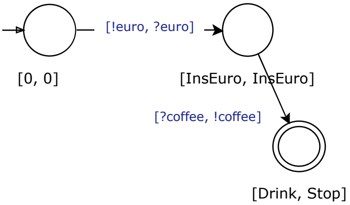
Benefits of CARE

- We show a possible realisation of an orchestration engine,
  - abstracted away in the contract automata theory,
  - but needed for implementing applications specified with contract automata,
- improving our understanding of the relation between a specification with contract automata and its implementation, and the corresponding level of abstraction.
- **Benefits:** formal guarantees, reduction of the software complexity, separation of concerns, modularity.

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 services 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*: $(\bullet, ?a, \bullet, !a)$,
  - $size(list) = rank$





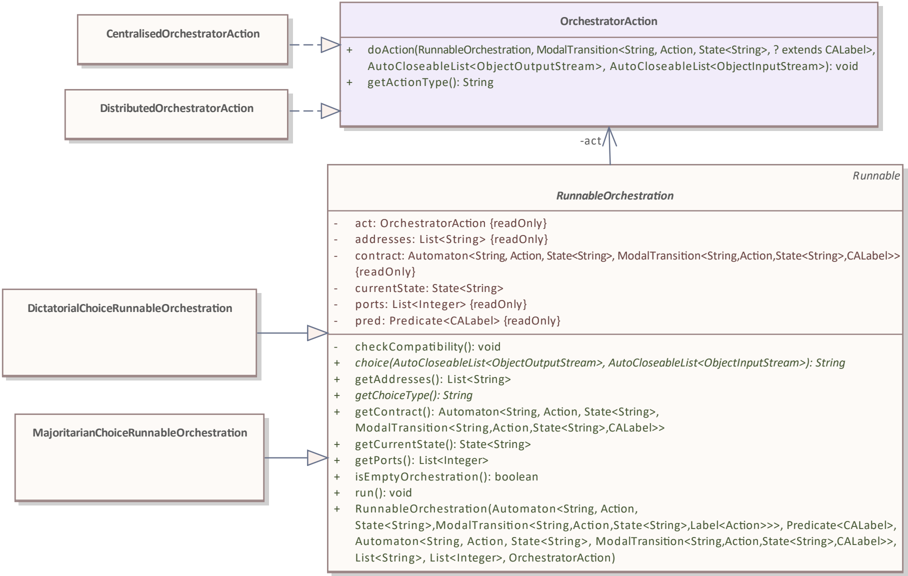
Contract Automata Runtime Environment

- the software is organised into:
  - the classes for the orchestrator and
  - the classes for the orchestrated services;
- offers and requests of contracts are an abstraction of low-level messages sent between the services and the orchestrator to realise them.

Runnable Orchestration, Runnable Orchestrated

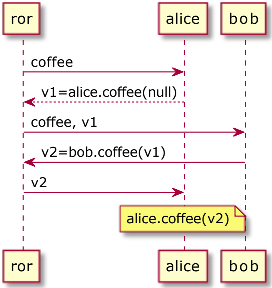
- `RunnableOrchestration`:
  - abstract class implementing an *orchestrator*,
  - it reads the synthesised orchestration and orchestrates the `RunnableOrchestratedContract` to realise the overall application.
- `RunnableOrchestratedContract`:
  - abstract class implementing an executable wrapper responsible for the *composition of the specification of a service with its implementation*.
  - It implements a service that is always listening and spawns a parallel process when entering an orchestration.
  - During an orchestration, it receives action commands from the orchestrator or from other services, and it invokes the corresponding action method.

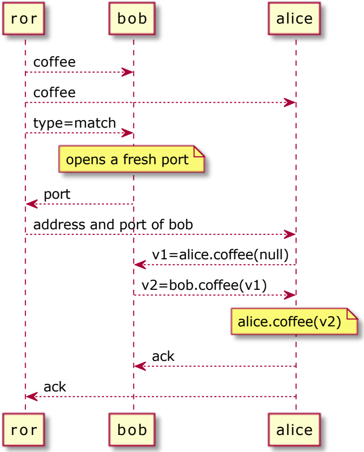
Two aspects to implement: action and choice



CARE - match implementation

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





Formal Guarantees : Adherence to the Contract

Algorithm 1 Orchestration

```
Require: non-empty orchestration automaton
Ensure: no exception is thrown
  init Sockets                          ▷ connect to the services
  cs ← initialState                     ▷ current state
  while true do
      fws ← forwardStar(cs)
      if empty(fws) & notFinal(cs) then
          throw Exception
      end if
      choice ← choice()                 ▷ interact with services
      if choice == stop & final(cs) then
          return
      end if
      tr ← select(fws,choice)
      if tr not in agreement then
          throw Exception
      end if
      doAction(tr)                      ▷ interact with services
      cs ← targetState(tr)
  end while
```

Algorithm 2 Service Thread

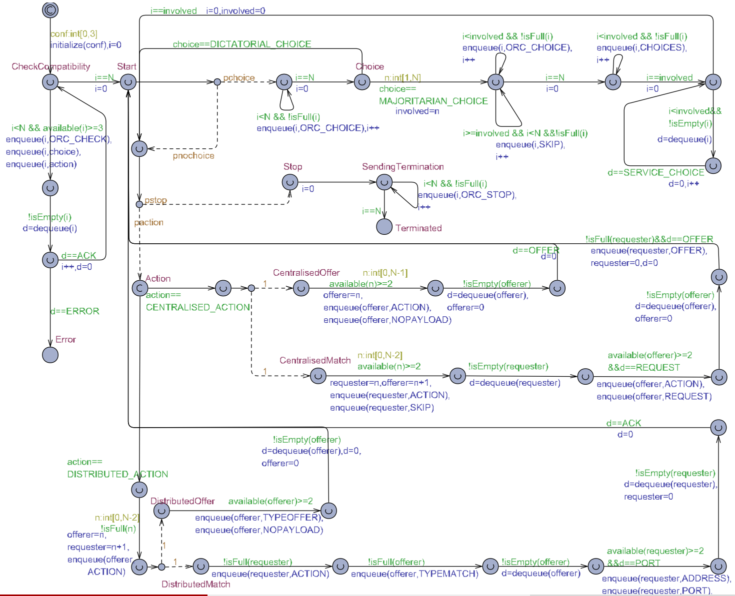
```
Require: connected to the orchestrator
  init Socket                           ▷ set socket timeout
  cs ← initialState                     ▷ current state
  while true do
      act ← receive(socket)
      if stop(act) then
          if final(cs) then
              return
          else throw ContractViolationException
          end if
      end if
      if choice(act) then
          performChoice()               ▷ interact with or-
          continue                      ▷ chestration
      end if
      tr ← select(forwardStar(cs),act)
      if no valid action then
          throw ContractViolationException
      else
          invokeMethod(tr)
      end if
      cs ← targetState(tr)
  end while
```

if the orchestration is correctly synthesised
no contract exception will ever be raised

Formal Guarantees : Uppaal model

- formal model covering aspects that are abstracted in the algorithms:
  - centralised/distributed action
  - majoritarian/dictatorial choice
- TCP/IP sockets modelled through blocking FIFO buffers
- Difference between contract automata and `Uppaal` models:
  - *contract automata* are high level models of service applications that can be implemented using `CARE`,
  - `CARE` is a distributed application, suitable to be verified with model checking,
  - `CARE` itself is a software that has been formally modelled and verified using `Uppaal`.

Formal Guarantees : Uppaal model



Formal Model : Traceability

https://github.com/contractautomataproject/CARE/tree/master/src/spec/uppaal

Model Checking

- `A[](not deadlock || (ror.Terminated && (forall(i:id_t) ROC(i).Terminated))`
  - absence of deadlocks
- `A[]((ror.Terminated && (forall (i:id_t) ROC(i).Terminated)) imply allEmpty())`
  - absence of orphan messages
- `A[](ror.Stop imply A<>(ror.Terminated && (forall (i:id_t) ROC(i).Terminated) && allEmpty()))`
  - whenever a choice to stop is made, eventually all services and the orchestrator will terminate their execution.

CARE evaluation

- two examples: Alice and Bob, Composition service
- two implementations: one with CARE, one without using CARE

		LOC	Cyclomatic Complexity	Cognitive Complexity
Alice and Bob	**without** CARE	777	134	166
	with CARE	153	16	8
Composition Service	**without** CARE	854	155	211
	with CARE	279	42	55

Conclusion, Future work

- We have presented the first runtime environment for contract automata, called `CARE`.
- **Future work** support for choreographies and product lines is formalised in contract automata and implemented in `CATLib`, but not supported by `CARE`
- Model-based testing
  - `Uppaal` tests generation,
  - generation of tests at the level of contract automata.
- Other enhancements to do

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

Other references

- Basile, D. and ter Beek, M.H., 2022. Contract automata library. 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.