A Runtime Environment for Contract Automata

Davide Basile(✉) and Maurice H. ter Beek

Formal Methods and Tools Lab, ISTI–CNR, Pisa, Italy
{davide.basile,maurice.terbeek}@isti.cnr.it

Abstract. Contract automata have been introduced for specifying applications through behavioural contracts and for synthesising their orchestrations as finite state automata. This paper addresses the realisation of applications from contract automata specifications. We present `CARE`, a new runtime environment to coordinate services implementing contracts that guarantees the adherence of the implementation to its contract. We discuss how `CARE` can be adopted to realise contract-based applications, its formal guarantees, and we identify the responsibilities of the involved business actors. Experiments show the benefits of adopting `CARE` with respect to manual implementations.

1 Introduction

From a recent survey in the transport domain [23], it has emerged that the majority of studies on formal methods propose specification languages, models, and their verification, whereas fewer focus on how to derive the finalised software from the verified specification also showing the adherence of the implementation to its specification. The authors of [26] state that these interaction specifications "are not yet a feature of standard mainstream programming languages, so software developers are not able to benefit from them". In this paper, we investigate the connection between a behavioural specification and its implementation, and we provide a possible realisation of those aspects abstracted in a specification.

Contract automata are a dialect of finite state automata used to formally specify the behaviour of service contracts in terms of offers and requests [14]. A composition of contracts is in *agreement* when all requests are matched by corresponding offers of other contracts. A composition can be refined to one in agreement using the orchestration synthesis algorithm [12,13], a variation of the synthesis algorithm from supervisory control theory [30]. Previously, in [10], a library called `CATLib` [11] implementing the operations on contract automata (e.g., composition, synthesis) was presented. A front-end of `CATLib` for graphically editing and operating on contracts is also available [19], called `CAT_App`. The orchestrator is abstracted away in contract automata and until now no examples of concrete implementations were provided in which services implement contract automata specifications.

Whilst `CATLib` and `CAT_App` are used to *specify* applications as contract automata, in this paper we tackle the problem of *implementing* applications that have been specified via contract automata. We introduce `CARE` [17], a newly

M. Chechik et al. (Eds.): FM 2023, LNCS 14000, pp. 550–567, 2023.
https://doi.org/10.1007/978-3-031-27481-7_31

developed software that provides a runtime environment to coordinate the CARE services that implement the contracts of the synthesised orchestration. Thus, CARE advances the state-of-the-art of the research on contract automata and behavioural contracts by detailing how specifications through contract automata can be connected with service-based applications. With CARE, the low-level interactions that are abstracted in contract automata orchestrations are now explicated. We discuss how CARE can promote a separation of concerns among different actors that together cooperate to realise contract-based applications, and among developers and designers of services. The proposed framework is exercised on two examples, showcasing the usage of CARE. Experiments show a neat improvement in terms of decreased complexity of the software when comparing the implementations of the examples exploiting CARE with those that manually implement the low-level interactions among services without relying on CARE.

Related Work. Other approaches to connect implementations with behavioural types (e.g., behavioural contracts, session types) are surveyed in [2,25]. Our approach is closer to [26,34], where behavioural types are expressed as finite state automata of Mungo, called *typestates* [33]. The toolchain of Mungo and StMungo is proposed to implement behavioural types specifications. Similarly to CARE, in Mungo finite state automata are used as behaviour assigned to Java classes (one automaton per class), where transition labels correspond to methods of the classes. A tool similar to Mungo is JaTyC (Java Typestate Checker) [6].

An Eclipse plugin called Diogenes [4] allows to write specifications of services as behavioural contracts using a domain specific language, verify them, and generate skeletal Java programs to be refined using the Java RESTful Web service middleware for contract-oriented computing presented in [8]. Both Diogenes and StMungo generate skeletal Java programs from contract compositions or multiparty session types, respectively, whereas CARE allows to adapt already existing components to realise a new application in a bottom-up approach, fostering adaptability and reusability of services.

CARE adopts a *correct-by-design* approach to implement a specification with formal guarantees. The complementary approach infers a behavioural type from an implementation, where guarantees hold if the typing succeeds. An algorithm to infer a form of behavioural types from programs with assertions is discussed in [35], where programs are written in Mool (Mini object-oriented language), a simple Java-like language incorporating behavioural types. The inference of behavioural types from Go programs is studied in [27]. Go is a language supporting synchronisations on channels inspired by process algebraic formalisms like CSP and CCS [20]. The inference of behavioural types is thus facilitated by the chosen languages, whilst extracting them from unconstrained Java programs is still a challenge [31]. CATLib supports compositions of communicating machines, the formalism of behavioural types used in [27], thus it could be used to suggest amendments to the original Go programs by exploiting its synthesis algorithms.

Finally, the approach proposed by CARE shares aspects with the synthesis/verification of runtime monitors [1,5,32], and is similar to the *automated composition* problem studied in [3,7,21,22], to which CARE and CATLib offer both a runtime engine and tailored novel synthesis algorithms.

Outline. We provide some background on contract automata in Sect. 2. Section 3 details the design of `CARE`. The formal guarantees offered by `CARE` are detailed in Sect. 4, whilst Sect. 5 discusses how `CARE` can be adopted for building applications specified via contract automata and the separation of concerns. Section 6 contains two examples and an evaluation of the benefits of our contribution. Finally, we conclude and discuss future work in Sect. 7.

2 Modal Service Contract Automata

We provide background on contract automata and their synthesis operation.

A Contract Automaton (CA) models either a single service or a multi-party composition of services performing actions. Figure 4 depicts some examples of CA. The number of services of a CA is called its *rank*. When $rank = 1$, the contract is called a *principal* (i.e., a single service). For example, the leftmost and rightmost automata in Fig. 4 are principals, while the automaton in the middle has $rank = 2$. Labels of CA are vectors of atomic elements called *actions*. Actions are either *requests* (prefixed by `?`), *offers* (prefixed by `!`), or *idle* (denoted with a distinguished symbol `-`). Requests and offers belong to the (pairwise disjoint) sets R and O, respectively. The states of CA are vectors of atomic elements called basic states. Labels are restricted to be *requests*, *offers*, or *matches* where, respectively, there is either a single request action, a single offer action, or a single pair of request and offer actions that match, and all other actions are idle. The length of the vectors of states and labels is equal to the rank of the CA.

For example, the label `[!euro, ?euro]` is a match where the request action `?euro` is matched by the offer action `!euro`. Note the difference between a request label (e.g., `[?coffee, -]`) and a request action (e.g., `?coffee`). A transition may also be called a request, offer, or match according to its label.

The goal of each service is to reach an accepting (*final*) state such that all its request (and possibly offer) actions are matched. In [12], CA were equipped with *modalities*, i.e., *necessary* ($\Box$) and *permitted* ($\Diamond$) transitions, respectively. Permitted transitions are controllable, whereas necessary transitions can be uncontrollable or semi-controllable. The resulting formalism is called *Modal Service Contract Automata* (MSCA). In the following definition, given a vector $\vec{a}$, its ith element is denoted by $\vec{a}_{(i)}$.

Definition 1 (MSCA). *Given a finite set of basic states $\mathcal{Q} = \{q_1, q_2, \ldots\}$, an MSCA $\mathcal{A}$ of rank $= n$ is a tuple $\langle Q, \vec{q_0}, A^r, A^o, T, F\rangle$, with set of states $Q = Q_1 \times \ldots \times Q_n \subseteq \mathcal{Q}^n$, initial state $\vec{q_0} \in Q$, set of requests $A^r \subseteq \mathsf{R}$, set of offers $A^o \subseteq \mathsf{O}$, set of final states $F \subseteq Q$, set of transitions $T \subseteq Q \times A \times Q$, where $A \subseteq (A^r \cup A^o \cup \{\texttt{-}\})^n$, partitioned into* permitted *transitions $T^{\Diamond}$ and* necessary *transitions $T^{\Box}$, such that: (i) given $t = (\vec{q}, \vec{a}, \vec{q}') \in T$, $\vec{a}$ is either a request, or an offer, or a match; and (ii) $\forall i \in 1 \ldots n$, $\vec{a}_{(i)} = \texttt{-}$ implies $\vec{q}_{(i)} = \vec{q}'_{(i)}$.*

Composition of services is rendered through the composition of their MSCA models by means of the *composition operator* $\otimes$, which is a variant of a synchronous product. This operator basically interleaves or matches the transitions

of the component MSCA, but whenever two component MSCA are enabled to execute their respective request/offer, then the match is forced to happen. Moreover, a match involving a necessary transition of an operand is itself necessary. The rank of the composed MSCA is the sum of the ranks of its operands. The vectors of states and actions of the composed MSCA are built from the vectors of states and actions of the component MSCA, respectively. Typically, in a composition of MSCA various properties are analysed. We are especially interested in *agreement*. In a contract that is in agreement, all requests are matched, i.e., transitions are only labelled with offers or matches.

We recall the specification of the abstract synthesis algorithm of CA from [13]. The synthesis of a controller, an orchestration, and a choreography of CA are all different special cases of this abstract synthesis algorithm, formalised in [13] and implemented in `CATLib` [10]. This algorithm is a fixed-point computation where at each iteration the set of transitions of the automaton is refined (using pruning predicate ϕ_p) and a set of forbidden states R is computed (using forbidden predicate ϕ_f). The synthesis is parametric with respect to these two predicates, which provide information on when a transition has to be pruned from the synthesised automaton and when a state has to be deemed forbidden, respectively. We refer to MSCA as the set of (MS)CA, where the set of states is denoted by Q and the set of transitions by T (with $T^{\Box}$ denoting the set of necessary transitions). For an automaton $\mathcal{A}$, the predicate $Dangling(\mathcal{A})$ contains those states that are not reachable from the initial state or that cannot reach any final state.

Definition 2 (abstract synthesis [13]). *Let $\mathcal{A}$ be an MSCA, $\mathcal{K}_0 = \mathcal{A}$, and $R_0 = Dangling(\mathcal{K}_0)$. Given two predicates $\phi_p, \phi_f : T \times MSCA \times Q \to \mathbb{B}$, let the* abstract synthesis function $f_{(\phi_p,\phi_f)} : MSCA \times 2^Q \to MSCA \times 2^Q$ *be defined as:*

$$
\begin{aligned}
&f_{(\phi_p,\phi_f)}(\mathcal{K}_{i-1}, R_{i-1}) = (\mathcal{K}_i, R_i), \text{ with}\\
&T_{\mathcal{K}_i} = T_{\mathcal{K}_{i-1}} - \{\, t \in T_{\mathcal{K}_{i-1}} \mid \phi_p(t, \mathcal{K}_{i-1}, R_{i-1}) = true \,\}\\
&R_i = R_{i-1} \cup \{\, \vec{q} \mid (\vec{q} \to) = t \in T^{\Box}_{\mathcal{A}},\ \phi_f(t, \mathcal{K}_{i-1}, R_{i-1}) = true \,\} \cup Dangling(\mathcal{K}_i)
\end{aligned}
$$

Subsequently, the abstract controller is defined as the least fixed point of $f_{(\phi_p,\phi_f)}$ (cf. [13, Theorem 5.2]). The synthesised orchestration guarantees the reachability of final states, the agreement property (i.e., all requests are matched) and that all reachable necessary requests are not pruned (i.e., controllability).

Tooling. CA and their functionalities are implemented in a software artefact, called Contract Automata Library (`CATLib`), whose development is active [11]. This software artefact is a by-product of our scientific research on behavioural contracts and implements results that have previously been formally specified in several publications (cf., e.g., [12–14]). Scalability features offered by `CATLib` include a bounded on-the-fly state-space generation optimised with pruning of redundant transitions and parallel streams computations. The software is open source [11], it has been developed using principles of model-based software engineering [10] and it has been extensively validated using various testing and analysis tools to increase the confidence on the reliability of the library [11].

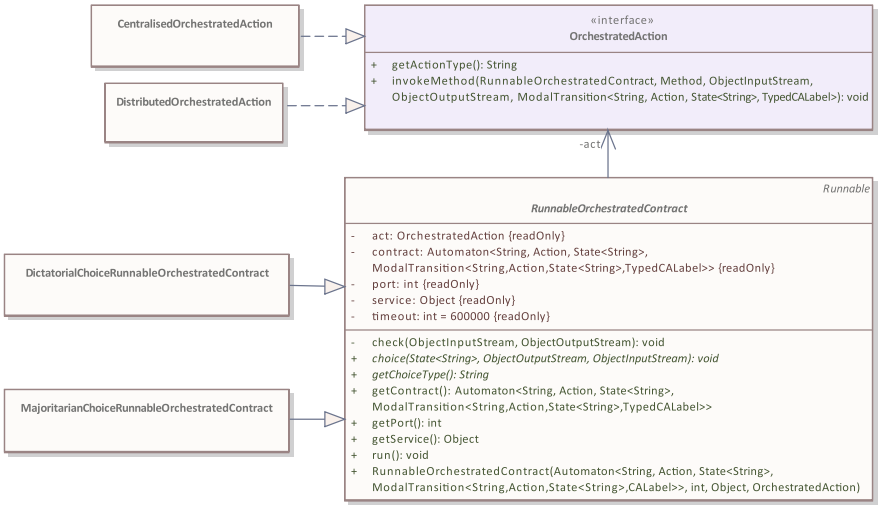


Fig. 1. The class diagram for the orchestrated services; the methods of the derived classes are visible in their super-class/interface as abstract methods (in italic)

3 `CARE` Design

We start by discussing the design of `CARE`. This software is organised into classes for the orchestrated services (cf. Fig. 1) and classes for the orchestrator.

In Fig. 1, `RunnableOrchestratedContract` is an abstract class that implements an executable wrapper responsible for the composition of the specification of a service (instance variable `contract` storing a contract automaton) with its implementation (instance variable `service` implementing the service). `RunnableOrchestratedContract` 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 (by means of the instance variable `act` of type `OrchestratedAction`).

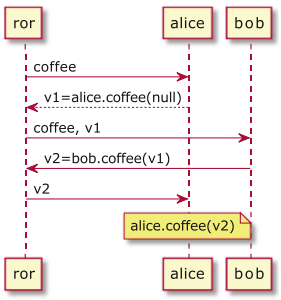
The realisation of an orchestration is abstracted away in contract automata. Crucially, offers and requests of contracts are an abstraction of low-level messages sent between the services and the orchestrator to realise them. `CARE` exploits the abstractions provided by Java to allow its specialisation according to different implementation choices, using abstractions of object-oriented design, as showed in Fig. 1. Two aspects to implement are choices and termination (through the abstract method `choice`). `CARE` is equipped with default implementations, but can be extended (by implementing the relative interfaces and abstract methods) to include other options, other than the default ones. Currently, a so-called 'dictatorial' choice (i.e., an internal choice of the orchestrator, external for the services) and a so-called 'majoritarian' choice (services vote and the majority wins) are two implemented options. `MajoritarianChoiceRunnableOrchestratedContract` and `DictatorialChoiceRunnableOrchestratedContract` are the two classes

specialising `RunnableOrchestratedContract` according to how the choice is handled and implementing the abstract methods. `CARE` also provides default implementations for the low-level message exchanges. Currently, the two available options are the 'centralised' action, where the orchestrator acts as a proxy, and the 'distributed' action, where two services matching their actions directly interact with each other once the orchestrator has made them aware of a matching partner and its address/port. Accordingly, each `RunnableOrchestratedContract` has an `OrchestratedAction` (instance variable `act`) used to implement the corresponding actions that can be either distributed or centralised according to the current implementation.

The abstract class `RunnableOrchestration` (which is not displayed in Fig. 1) implements a special service that reads the synthesised orchestration (stored in the instance variable `contract`) and orchestrates the `RunnableOrchestratedContract` to realise the overall application. Similarly to the case of the orchestrated contract, also the orchestrator is specialised according to either a dictatorial or a majoritarian implementation of the abstract method `choice`. Moreover, an `OrchestratorAction` instance variable is used to implement each action of the orchestration, either centralised or distributed, thus matching the corresponding actions of the orchestrated services.

Finally, the class `ContractViolationException` implements an exception raised in case an invocation of the orchestrator is not allowed by the orchestrated contract or if that contract is not fulfilled. When thrown, the exception stores the remote host that violates the contract. This guarantees the accountability in case of a contract violation. Each label of a contract automaton is extended using `CARE` with the information on the types of parameters and returned values from the corresponding method implementing the corresponding action. These typed labels are implemented into the class `TypedCALabel`, extending a `CALabel` of `CATLib`. This class also overrides the matching between requests and offers to also take into account their types: the returned value of the request must be of a super-class of the parameter class of the offer and vice versa. This guarantees that no `ClassCastException` will ever be raised when invoking the actions. Note that the signature of each action declared by the interface is not fixed, so other types can be used (e.g., `JSon` values).

We briefly detail the centralised implementation of a match label in `CARE`. We will use the match `[?coffee,!coffee]` from the example in Sect. 6, in which `Alice` is requesting a `coffee` and `Bob` is offering a `coffee`. The method `coffee` of `Alice` is invoked twice: firstly, passing no argument, it generates an `Integer` value (e.g., the amount of sugar) that is passed (by the orchestrator `ror`) as argument to the method `coffee` of `Bob`, which in turn produces a `String` value that is eventually passed as argument to the method `coffee` of `Alice`, thus fulfilling the `coffee` request.



4 Formal Guarantees

We now discuss the formal guarantees of correctness and the adherence of the implementation to the specification brought by the usage of `CARE`. To begin with, to guarantee that an orchestration is *correct-by-design*, the contract automata operators of composition and orchestration synthesis are used, exploiting the theoretical results on contract automata (cf. Sect. 2). More concretely, these operations are performed in the constructor of a `RunnableOrchestration` using `CATLib`. As discussed in Sect. 2, the synthesised orchestration ensures properties such as absence of deadlocks, matching of all requests of contracts with corresponding offers of other contracts, and reachability of final states.

After a well-behaving orchestration has been synthesised, it is important to ensure that the low-level implementations of the distributed services interacting with each other will adhere to the operations prescribed by the orchestration synthesised from their contracts. This task is addressed by using Algorithm 1 and Algorithm 2, both implemented in `CARE`, reproduced below in pseudo-code.

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
```

Algorithm 1 illustrates the main operations performed during an orchestration. The algorithm requires that a correct and non-empty orchestration has been synthesised. This requirement is necessary to ensure that no exceptions will be thrown at runtime. Initially, the orchestrator connects to the services (their ports and addresses are stored during instantiation). The current state of the execution is set to the initial state. Subsequently, a loop is executed in continuation. Inside the loop, one of the transitions is selected from the set of outgoing transitions (i.e., the forward star) of the current state, using the implementation of the abstract method `choice`. Here, if there is a deadlock (no outgoing transitions and the current state is not final), an exception is thrown. After that, if the current state is final, but there are also outgoing transitions, then the

choice can be to stop or to continue; otherwise, if the state is not final, then the choice can only be to select one of the outgoing transitions. If the selected transition of the orchestration does not satisfy agreement (i.e., its label is a request), then an exception is thrown. Otherwise, the action of the selected transition is executed using the implementation of the abstract method `doAction` and the current state is updated to the target state of the transition. As discussed in Sect. 2, if the orchestration automaton has been synthesised using the contract automata synthesis, then this formally guarantees that the described exceptions will never be thrown by the orchestrator.

Algorithm 2 summarises the execution of an orchestrated service following its contract. The service is multi-threaded and spawns a new thread each time a new request of connection is received. The algorithm depicts the operations performed by a spawned thread. Similarly to the orchestration, there is an initialisation of the socket, and the current state is set to the initial state of the contract. After that, a continuous loop is executed. Firstly, an action is received from the orchestrator. If the choice is of terminating and the contract is in a final state, then the service terminates successfully; otherwise, if the state is not final, an exception is thrown. If the orchestrator requires to make a choice, then the implementation of the abstract method `choice` is called to perform a choice (possibly interacting with the orchestrator). Otherwise, the orchestrator is requiring to perform an action. In this case, the prescribed action is selected from the outgoing transitions of the current state of the service contract. If there is no such action, then a contract violation exception is thrown since the orchestrator is requiring to perform an operation not prescribed by the contract. Otherwise, the method of the service that is paired with the corresponding action of the contract is invoked. These steps ensure that the low-level implementation of the actions of the services are correctly executed according to the actions prescribed by the orchestration synthesised from the composition of contracts. Finally, the current state is set to the target state of the contract and the loop is repeated. Similarly to the orchestration case, if the orchestrator is executing a correctly synthesised orchestration, then the services will never throw any such exception. Indeed, this would be a contradiction to the formal results discussed in Sect. 2.

Interaction Correctness. As stated above, the execution of an *action* or a *choice* is abstracted in `CARE`. Two implementations are currently available for both actions and choices, and the framework is extensible. We now summarise the formal verification of the TCP/IP sockets interactions performed by the available implementations of actions and choices. This provides a complementary verification of the aspects that are abstracted in the above algorithms. The implementation of `CARE` has been formally modelled in Uppaal as a network of timed automata. Figure 2 depicts the automaton for the `RunnableOrchestration`. Due to lack of space, the automaton for the `RunnableOrchestratedContract` and traceability information linking the model to the source code are available from [9]. Both the synthesised orchestration (which is assumed to have been synthesised correctly) and other details specified in Algorithm 1 and Algorithm 2 are abstracted away in the formal model.

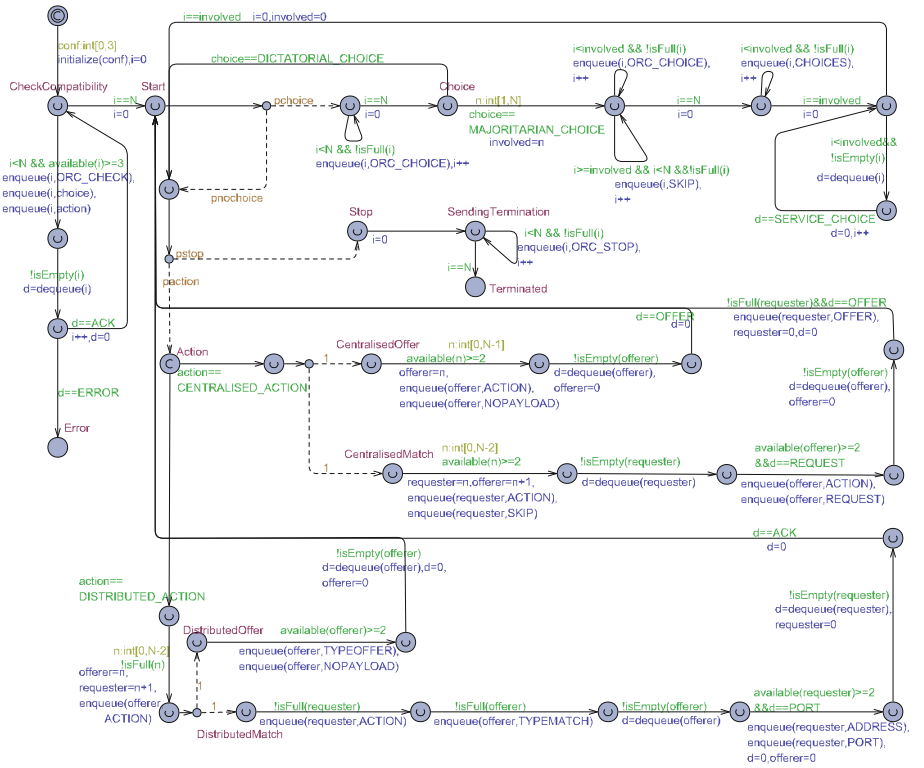


Fig. 2. The `RunnableOrchestration` UPPAAL model

The behaviour according to the given configuration of action and choice is modelled inside each automaton. Global declarations include the number of services N, the size of the buffers, two variables `action` and `choice` storing the corresponding configuration for all automata, and the communication buffers. Java TCP/IP sockets communications are asynchronous with FIFO buffers. In the model, arrays are used to encode these buffers that are only modified with functions for enqueueing and dequeuing messages. Each party communicates with the partner using two buffers (one for sending and one for receiving). Both automata declare a method `enqueue` for sending a message to the partner. Similarly, both automata have a method `dequeue` for consuming messages from their respective buffers. According to the semantics of Java TCP/IP sockets, a transition having a send in its effect will check in its guard whether there is enough space left in the buffer of the partner by calling either the method `available` (returning the space left) or `isFull`. Moreover, before reading it is always checked whether the buffer is not empty with the method `!isEmpty`. When the buffer is empty, the automaton blocks until a message is received. The locations of the model are *urgent* (denoted with `U`) to guarantee that when the appropriate message is received it will eventually be consumed (i.e., there is no starvation).

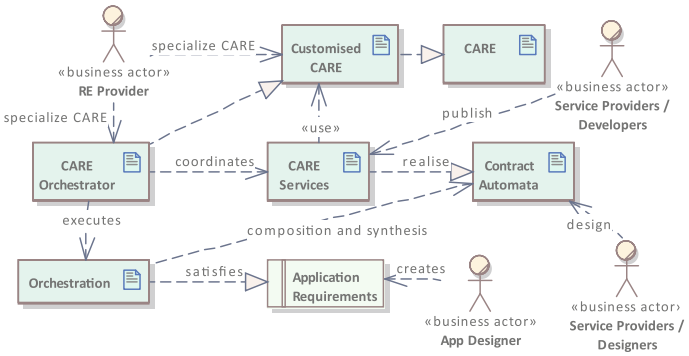


Fig. 3. The `CARE` business actors developing contract-based applications

The absence of deadlocks was verified by model checking the CTL formula `A[](not deadlock || (ror.Terminated && (forall(i:id_t) ROC(i).Terminated)))`, in which `ror` is the orchestrator and `ROC(i)` is a runnable orchestrated contract identified with index i. Moreover, the absence of orphan messages was verified by model checking `A[]((ror.Terminated && (forall (i:id_t) ROC(i).Terminated)) imply allEmpty())`, in which the predicate `allEmpty()` is satisfied when the buffers are empty. Finally, `A[](ror.Stop imply A<>(ror.Terminated && (forall (i:id_t) ROC(i).Terminated) && allEmpty()))` was used to verify that whenever a choice to stop is made, eventually all services and the orchestrator will terminate their execution.

5 Building Applications with `CARE`

We now discuss how `CARE` can be adopted to develop applications with contract automata. The diagram in Fig. 3 depicts the responsibilities of the business actors involved in the overall realisation of contract-based applications using `CARE`. The first actor is the provider of the runtime environment (`RE Provider` in Fig. 3). This actor customises `CARE` and its classes according to specific needs, possibly introducing new different options for choices and actions implementing the abstract methods provided by `CARE` (described in Sect. 3), and delivers to the other actors a customised version of `CARE`, which also comprehends an orchestrator. Note that this customisation is not necessary, but is a further possibility allowed by the `CARE` software design.

The second kind of actors are the service providers, who publish their contracts, implemented by remote (non-disclosed) Java classes, and use a `RunnableOrchestratedContract` to make their contract publicly accessible using `CARE`, while hiding implementation details. Service providers may choose among different realisations of their `RunnableOrchestratedContract`, provided by the first actor above. Notably, implementing each atomic action of a service and designing the interaction behaviour through contract automata are two different concerns. The designer (cf. Fig. 3) specifying interactions as a contract is not required to be an expert in the underlying implementation technology (e.g., Java sockets), while the developer implementing actions and selecting the `CARE`

Table 1. The roles and responsibilities of the business actors involved in developing applications specified via contract automata.

Role	Responsibility
Runtime Environment Provider	Customisation of `CARE`, implementation of abstract methods if needed
Service Providers/Designers	Design contract automata and publish them
Service Providers/Developers	Implement the actions prescribed by contracts, select one of the available configurations of the runtime environment
App Designer	Design the requirements of the application, discover contracts, select one of the available configurations of the runtime environment

configuration is not required to be skilled in contract automata theory. The specification and implementation of a service can thus be seamlessly integrated using the facilities provided by `CARE`. This integration using `CARE` is depicted with a *realize* arrow from the services to the contracts. Most importantly, when implementing the service, the developer does not need to worry about the underlying low-level interactions between services, potential deadlocks and other communication issues. This error-prone implementation activity is already resolved by `CARE`, as discussed in Sect. 4. This separation of concerns also solves the problem of "muddling the main program logic with auxiliary logic related to error handling" (i.e., handling the Java communication exceptions) [24].

The third actor is the application designer (`App Designer` in Fig. 3). This is a user of both the second and the first actor. The designer is responsible for specifying the *requirements* of the application, and to find a suitable set of remote services whose synthesised orchestration satisfies the desired requirements. Once the contracts are discovered, the orchestration enforcing the requirements is automatically synthesised as a new contract. This is depicted by an arrow from `Orchestration` to `Contract Automata` in Fig. 3. The application designer exploits `CARE`, choosing a specific implementation of `RunnableOrchestration` and `RunnableOrchestratedContract`, passing as arguments the addresses of the services, as well as the synthesised orchestration. Formal results from contract automata theory [12–15] (cf. Sect. 2) guarantee that no `ContractViolationException` will ever be raised at runtime (cf. Sect. 4). Finally, note that one individual could take the roles of more actors if needed (e.g., covering both roles of developer and designer, designing a global requirement, implementing a new choice, and publishing a target contract). The proposed separation of concerns is logical. The roles and responsibilities of the various business actors described in this section are summarised in Table 1.

6 Examples and Evaluation

We discuss the usage of `CARE` using two examples. Their source code, video tutorials, and evaluation data are available from [18].

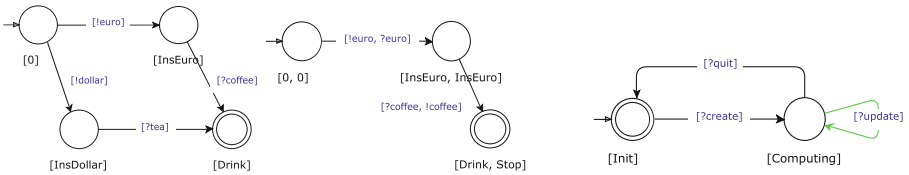


Fig. 4. From left to right, the contract of `Alice`, the orchestration of `Alice` and `Bob` enforcing the given requirement, and the contract of the `Client`

Alice and Bob. This is a basic yet illustrative example. In this example, the requirement `req` of the application, designed by the `App Designer`, is an automaton specifying that an action `coffee` is observed after an action `euro`. In this example, the `RE Provider` will simply provide `CARE` as it is, without further providing customised implementations of the abstract methods.

We now move to the `Service Provider/Designers` actors. Consider Fig. 4 (the automata have been constructed using `CAT_App`). The leftmost automaton is the contract of `Alice` and specifies that `Alice` offers either a `!euro` or a `!dollar` to her partner. Then `Alice` requires `?coffee` or `?tea`, depending on which offer has been accepted. Such a contract can be interpreted as describing the interaction pattern of `Alice`, whilst abstracting away from the actual implementation of each action. To declare the signature of each contract action, `CARE` uses Java Interfaces, as shown below.

```
public interface AliceInterface {
  Integer coffee(String arg); Integer tea(String arg);
  Integer euro(String arg); Integer dollar(String arg); }
```

In the interpretation of contracts provided by `CARE`, each contract action is implemented by a method of an interface, whose names are in correspondence. The implementation will be developed by the actor `Service Provider/Developer`. By implementing the corresponding interface it is possible to pair the interaction logic described in Fig. 4 (left) with an actual implementation, as shown below.

```
RunnableOrchestratedContract alice = new DictatorialChoiceRunnableOrchestratedContract(ca,
  8080,new Alice(),new CentralisedOrchestratedAction());
```

The parameter `ca` contains the leftmost contract in Fig. 4. The class `Alice` implements `AliceInterface`. This implementation is paired with the corresponding contract: the service listens to port 8080 and the chosen implementation of the low level interactions is `CentralisedOrchestratedAction`. Notably, `RunnableOrchestratedContract` will take care of the low-level communications, abstracted away in `Alice.java`. In `AliceInterface`, each action requires an argument (of type `String`) and returns a value (of type `Integer`). During initialisation, each label of the contract is extended with the information on the types of parameters and returned values from the interface, by instantiating a `TypedCALabel`. The contract of `Bob` is dual to the one of `Alice` (i.e., all requests

are turned to offers). The class `RunnableOrchestration` can be instantiated as shown below.

```
RunnableOrchestration ror = new DictatorialChoiceRunnableOrchestration(req,new Agreement(),
  Arrays.asList(alice.getContract(),bob.getContract()),Arrays.asList(null,null),
    Arrays.asList(alice.getPort(),bob.getPort()),new CentralisedOrchestratorAction());
```

`DictatorialChoiceRunnableOrchestration` provides an implementation of the branch/termination selection where the orchestrator autonomously selects a branch. It is instantiated by passing as parameters the requirement `req` to be enforced, the predicate on interactions among contracts (i.e., the property of agreement), the list of contracts to compose, addresses and ports of the `RunnableOrchestratedContract` of `Alice` and `Bob`, and an object of class `CentralisedOrchestratorAction` implementing an `OrchestratorAction`. In this example, services are run locally on the same host as the orchestrator. During instantiation, the contracts passed as arguments will be composed to synthesise their safe orchestration in agreement.

In this example, the contract of `Bob` is in agreement with that of `Alice` (each request is matched by a corresponding offer). The orchestration `orc` is the central automaton in Fig. 4. After `ror` has been instantiated, its method `isEmptyOrchestration()` is used to check if an agreement among contracts exists, i.e., if the synthesised orchestration is non-empty. During instantiation, `RunnableOrchestration` also interacts with all services (using Java TCP sockets) to ensure that all share the same configuration, which in this case is a dictatorial choice with centralised action. If this is not the case, an exception is thrown. Upon successful instantiation, `ror` can be executed to realise the application modelled through the requirement `req` using the two contracts above.

Finally, we remark that it suffices to change the requirement to automatically adapt the services to generate a new application. In this example, if `req` were changed to also allow a `tea` in case of payment with `dollar`, then `Alice` and `Bob` could be adapted to fulfill this new requirement automatically.

Composition Service. Computing a composition of contract automata can be a costly operation. For a front-end running on a standard laptop (e.g., `CAT_App`), a desirable feature could be to delegate such costly computations to a remote service, hosted on a powerful machine. This example showcases a service built with `CARE` that computes a composition of contract automata. The service receives the operand automata together with other scalability options (e.g., a bound, invariants) from a client service. The client service interacts through the console with a user who indicates which automata to compose and the other options. `CATLib` features on-the-fly bounded composition. When extending the bound of a previously computed composition, the previously generated states of the composition are not recomputed. The newly generated states are limited to those that exceeded the previous bound.

The client contract is the rightmost automaton in Fig. 4, whilst the service contract is dual (all requests are turned to offers). The client contract can perform a necessary request `update` from state `Computing`. This guarantees that in a non-empty orchestration, the necessary request of the client is matched by a

corresponding offer. If such a request were not necessary, a non-empty orchestration could also be obtained when the client is composed with a service that does not offer the `update` action, but only actions `create` and `quit`.

From state `Init`, the client can either terminate or perform a `create` request. During the execution of this method the user interacts at console and types the needed input. The payload returned by the request method is submitted by the runtime support to the service executing the matching offer. The offer implementation takes as parameter the payload and returns the composed automaton (which can be bounded to a specific depth), which is sent back to the requester. In the implementation of the `update` request, the client sends an incremented bound to the service, which proceeds to compute the composition with the extended bound and sends it to the client. The request `quit` is used as a signal for resetting both the computed composition and the bound.

There are two choices: in state `Init`, the orchestration can terminate or an action `create` can be executed. In state `Computing`, two possible actions can be performed. The `MajoritarianChoiceRunnableOrchestratedContract` method `select` is overridden by each service, to implement the specific choices to be made. The composition service always replies with an empty answer. This means that all choices are external to the service, the service does not indicate which choice has to be made. The client service implements both choices as internal. The user of the client service will interact at console with the client service, and will indicate which choice has to be made. More details can be found in [18].

Evaluation. We now measure the advantages brought by adopting `CARE`. To do so, we compare two different implementations for the two examples. These two implementations of each example perform the exact same operations as described above. Both implementations exploit the operations of composition and synthesis of contract automata provided by `CATLib`. However, only one of them uses `CARE` (as described above) whereas the other manually implements the prescribed interactions between the services and the orchestrator, without using any of the facilities provided by `CARE`. In this way, it is possible to isolate and measure the benefits brought solely by using `CARE`. These two implementations per example are open source and available for inspection from [18], where they are located in two separate packages.

The comparison was performed using measures of code complexity as provided by `SonarCloud` [16], an online service well integrated with `GitHub` that performs, among others, continuous inspection of code quality and static analysis of code to detect bugs, and reports on code complexity. We in particular used the code complexity reports feature of `SonarCloud`. We used three different measures of complexity to showcase the benefits of using `CARE`. The first measure is the total amount of lines of code (thus excluding, among others, the lines of comments and white spaces). We also adopted cyclomatic complexity [29] and cognitive complexity [16]. Cyclomatic complexity measures the number of independent paths in the software and it is a measure of code testability (this number is close to the number of branches to cover in the program). Cognitive complexity

Table 2. Different measures of complexity of the examples from Sect. 6 implemented either with or 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

measures how difficult the control flow is to understand. This measure is roughly a counter incremented each time a control flow structure is encountered (e.g., `if` and `for`) and it is incremented commensurated with the level of nesting of control flow structures (e.g., a first-level structure triggers an increment of 1, a second-level structure triggers an increment of 2, and so on) [16].

The results are reported in Table 2. To compare these quantities, we use the relative percent difference (*rpd*): $\frac{|\texttt{withCARE}-\texttt{withoutCARE}|}{max(\texttt{withCARE},\texttt{withoutCARE})} \times 100$. This measures the change of complexity when using `CARE` with respect to the reference value (i.e., `withoutCARE`). The advantage of using `CARE` is clear, as it drastically reduces the complexity of the software. Indeed, when using `CARE` (for the "Alice and Bob" and "Composition Service" examples, respectively) the *rpd* are: for the lines of code 80.31% and 67.33%, for the cyclomatic complexity 88.06% and 72.90%, and for the cognitive complexity 95.18% and 73.93%. These results are not surprising: the experiments underline the complexity of the operations performed by `CARE` and its key role in developing applications specified via contract automata. Indeed, for both examples the complexity of implementing the low-level communications is the dominant factor if compared to the interaction logic. This is more prominent for the "Alice and Bob" example, which in fact has greater *rpd* values. The burden of implementing these low-level communications among the services and the orchestrator is still on the developer side when not using `CARE`. We also remark how implementing the low-level communications is an error-prone activity that is completely delegated to the runtime support if one uses `CARE`, thus improving the confidence in the correctness of the final application.

Scalability. The above experiments only measured the complexity of the software developed with or without `CARE`. Another important aspect is the possibility of scaling to larger automata. `CARE` is a runtime environment and does not face any scalability issue typical of static analysers (e.g., state-space explosion). On the other hand, the synthesis of a safe orchestration of contracts is computed using `CATLib`, which may face scalability challenges when dealing with large compositions. In Sect. 2, the scalability features offered by `CATLib` are reported. The performance of `CATLib` has been previously measured in [10]. Concerning the formal verification of `CARE` discussed in Sect. 4, we recall that the orchestration automaton is abstracted away in the Uppaal model. Thus, the formal model of `CARE` is verified for any orchestration of any size.

On a side note, the single-responsibility principle [28] advocates to assign a single responsibility to each class. By interpreting this principle over behavioural contracts, we conclude that a contract automaton assigned to a single class (e.g., the rightmost automaton in Fig. 4) should not exhibit a large behaviour.

7 Conclusion

We have presented the first runtime environment for contract automata, called `CARE`. Our proposal advances the state-of-the-art of the research on contract automata by showing a possible realisation of an orchestration engine, abstracted away in the contract automata theory, but needed for implementing applications specified with contract automata, and guaranteeing that the implementation of service-based applications respect their specification. This contribution improves our understanding of the relation between a specification with contract automata and its implementation, and the corresponding level of abstraction.

With `CARE`, it is possible to promote a separation of concerns between formal methods experts specifying the expected behaviour using automata on one side, and developers (not required to be experts in formal methods) implementing the actions on the other. Furthermore, an application built using `CARE` is based on rigorous theoretical results from the contract automata theory, guaranteeing properties such as absence of deadlocks and absence of orphan messages, reachability of final states, and absence of `ContractViolationException`. Moreover, `CARE` promotes modularity of applications composed by different services that are reusable in different applications and that can be adapted to satisfy different requirements through the synthesis of well-behaving orchestrations. Experiments showed the improvement in terms of decreased software complexity when using `CARE` instead of manually implementing the low-level interactions among services implementing the operations prescribed by their contracts.

Future Work. `CATLib` already implements the synthesis of choreographies [13], which `CARE` will support in the future. Although `CARE` has been developed in the framework of contract automata, we plan to investigate the integration of this technology with other behavioural types languages and tools (e.g., typestates).

Acknowledgment. Funded by MUR PRIN 2017FTXR7S project IT MaTTerS (Methods and Tools for Trustworthy Smart Systems) and PRIN 2020TL3X8X project T-LADIES (Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems).

References

1. Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: Comparing controlled system synthesis and suppression enforcement. Int. J. Softw. Tools Technol. Transfer **23**(4), 601–614 (2021). https://doi.org/10.1007/s10009-021-00624-0
2. Ancona, D., et al.: Behavioral types in programming languages. Found. Trends Program. Lang. **3**(2–3), 95–230 (2016). https://doi.org/10.1561/2500000031

3. Atampore, F., Dingel, J., Rudie, K.: A controller synthesis framework for automated service composition. Discrete Event Dyn. Syst. **29**(3), 297–365 (2019). https://doi.org/10.1007/s10626-019-00282-0
4. Atzei, N., Bartoletti, M.: Developing honest Java programs with Diogenes. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 52–61. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_4
5. Azzopardi, S., Piterman, N., Schneider, G.: Incorporating monitors in reactive synthesis without paying the price. In: Hou, Z., Ganesh, V. (eds.) ATVA 2021. LNCS, vol. 12971, pp. 337–353. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88885-5_22
6. Bacchiani, L., Bravetti, M., Giunti, M., Mota, J., Ravara, A.: A Java typestate checker supporting inheritance. Sci. Comput. Program. **221**, 102844 (2022). https://doi.org/10.1016/j.scico.2022.102844
7. Barati, M., St-Denis, R.: Behavior composition meets supervisory control. In: Proceedings of the 2015 International Conference on Systems, Man, and Cybernetics (SMC), pp. 115–120. IEEE (2015). https://doi.org/10.1109/SMC.2015.33
8. Bartoletti, M., Cimoli, T., Murgia, M., Podda, A.S., Pompianu, L.: A contract-oriented middleware. In: Braga, C., Ölveczky, P.C. (eds.) FACS 2015. LNCS, vol. 9539, pp. 86–104. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28934-2_5
9. Basile, D.: Uppaal Models of the Contract Automata Runtime Environment. https://github.com/contractautomataproject/CARE/tree/master/src/spec/uppaal
10. Basile, D., ter Beek, M.H.: A clean and efficient implementation of choreography synthesis for behavioural contracts. In: Damiani, F., Dardha, O. (eds.) COORDINATION 2021. LNCS, vol. 12717, pp. 225–238. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78142-2_14
11. Basile, D., ter Beek, M.H.: Contract automata library. Sci. Comput. Program. **221** (2022). https://doi.org/10.1016/j.scico.2022.102841. https://github.com/contractautomataproject/ContractAutomataLib
12. Basile, D., et al.: Controller synthesis of service contracts with variability. Sci. Comput. Program. **187**, 102344 (2020). https://doi.org/10.1016/j.scico.2019.102344
13. Basile, D., ter Beek, M.H., Pugliese, R.: Synthesis of orchestrations and choreographies: bridging the gap between supervisory control and coordination of services. Log. Methods Comput. Sci. **16**(2), 9:1–9:29 (2020). https://doi.org/10.23638/LMCS-16(2:9)2020
14. Basile, D., Degano, P., Ferrari, G.L.: Automata for specifying and orchestrating service contracts. Log. Methods Comput. Sci. **12**(4), 6:1–6:51 (2016). https://doi.org/10.2168/LMCS-12(4:6)2016
15. Basile, D., Degano, P., Ferrari, G.L., Tuosto, E.: Relating two automata-based models of orchestration and choreography. J. Log. Algebraic Methods Program. **85**(3), 425–446 (2016). https://doi.org/10.1016/j.jlamp.2015.09.011
16. Campbell, G.A.: Cognitive complexity: an overview and evaluation. In: Proceedings of the 2018 International Conference on Technical Debt (TechDebt), pp. 57–58. ACM (2018). https://doi.org/10.1145/3194164.3194186
17. Contract Automata Runtime Environment (CARE) v1.0.0. https://github.com/contractautomataproject/CARE/
18. CARE Examples and Evaluation. Including video tutorials for reproducing the examples. https://github.com/contractautomataproject/CARE_Examples/
19. CAT_App. https://github.com/contractautomataproject/ContractAutomataApp

20. Dilley, N., Lange, J.: An empirical study of messaging passing concurrency in Go projects. In: Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 377–387. IEEE (2019). https://doi.org/10.1109/SANER.2019.8668036
21. Farhat, H.: Web service composition via supervisory control theory. IEEE Access **6**, 59779–59789 (2018). https://doi.org/10.1109/ACCESS.2018.2874564
22. Felli, P., Yadav, N., Sardina, S.: Supervisory control for behavior composition. IEEE Trans. Autom. Control **62**(2), 986–991 (2017). https://doi.org/10.1109/TAC.2016.2570748
23. Ferrari, A., ter Beek, M.H.: Formal methods in railways: a systematic mapping study. ACM Comput. Surv. **55**(4), 69:1–69:37 (2023). https://doi.org/10.1145/3520480
24. Francalanza, A., Mezzina, C.A., Tuosto, E.: Towards choreographic-based monitoring. In: Ulidowski, I., Lanese, I., Schultz, U.P., Ferreira, C. (eds.) RC 2020. LNCS, vol. 12070, pp. 128–150. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-47361-7_6
25. Gay, S., Ravara, A. (eds.): Behavioural Types: From Theory to Tools. River (2017). https://doi.org/10.13052/rp-9788793519817
26. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo: a session type toolchain for Java. Sci. Comput. Program. **155**, 52–75 (2018). https://doi.org/10.1016/j.scico.2017.10.006
27. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in Go using behavioural types. In: Proceedings of the 40th International Conference on Software Engineering (ICSE), pp. 1137–1148. ACM (2018). https://doi.org/10.1145/3180155.3180157
28. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR (2003)
29. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **2**(4), 308–320 (1976). https://doi.org/10.1109/TSE.1976.233837
30. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control. Optim. **25**(1), 206–230 (1987). https://doi.org/10.1137/0325013
31. Rubbens, R., Lathouwers, S., Huisman, M.: Modular transformation of Java exceptions modulo errors. In: Lluch Lafuente, A., Mavridou, A. (eds.) FMICS 2021. LNCS, vol. 12863, pp. 67–84. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85248-1_5
32. Sánchez, C., et al.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019). https://doi.org/10.1007/s10703-019-00337-w
33. Strom, R.E., Yemini, S.: Typestate: a programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. **12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929
34. Trindade, A., Mota, J., Ravara, A.: Typestates to automata and back: a tool. In: Lange, J., Mavridou, A., Safina, L., Scalas, A. (eds.) Proceedings of the 13th Interaction and Concurrency Experience (ICE). EPTCS, vol. 324, pp. 25–42 (2020). https://doi.org/10.4204/EPTCS.324.4
35. Vasconcelos, C., Ravara, A.: From object-oriented code with assertions to behavioural types. In: Proceedings of the 32nd Symposium on Applied Computing (SAC), pp. 1492–1497. ACM (2017). https://doi.org/10.1145/3019612.3019733