A Runtime Environment for Contract Automata

Davide Basile and Maurice H. ter Beek

Formal Methods and Tools Lab
ISTI–CNR, Pisa, Italy

Abstract. Realising contract-based applications from formal specifications with formal guarantees requires to show the adherence of a specification, the contract, to its implementation. Contract automata have been introduced for specifying contract-based applications and synthesising their orchestrations as finite state automata. This paper introduces `CARE`, a newly developed library for implementing applications specified via contract automata, providing a runtime environment to coordinate services implementing contracts.

1 Introduction

Contract-based applications enforce a separation of concerns between the specification of the interactions (i.e., the contract) and their implementation. From a recent survey in the transport domain [18] 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. The authors of [22] 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 particular, realising contract-based applications from specifications with formal guarantees requires to show the adherence of an implementation to its specification, the contract.

Contract automata are a dialect of finite state automata used to formalise the behaviour of services in terms of offers and requests [10]. 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 [8,9], a variation of the synthesis algorithm from supervisory control theory [25]. Contract automata abstract from the way in which an orchestration is realised, and until now no examples of concrete implementations were provided. Previously, in [7], a library called `CATLib` [13] 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 [14], called `CAT_App`.

Whilst `CATLib` and `CAT_App` are used to *specify* applications as contract automata, this paper presents `CARE` [11], a newly developed library for *implementing* applications specified via contract automata. `CARE` provides a runtime environment to coordinate the `CARE` services that are implementing the contracts of the synthesised orchestration. Thus, `CARE` is the missing piece between specifications through contract automata and their implementations, to make explicit the low-level interactions realising the prescribed actions.

Related work. Other approaches to connect implementations with behavioural types (e.g., behavioural contracts, session types) are surveyed in [2,21]. Our approach is closer to [22,28], where behavioural types are expressed as finite state automata of `Mungo`, called *typestates* [27]. 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) [24].

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 [6]. 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 [29], 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 [23]. `Go` is a language supporting synchronisations on channels inspired by process algebra formalisms like CSP and CCS [15]. The inference of behavioural types is thus facilitated by the chosen languages, whilst extracting them from unconstrained Java programs is still a challenge [26]. `CATLib` supports compositions of communicating machines, the formalism of behavioural types used in [23], thus it could be used to suggest amendments to the original `Go` programs by exploiting its synthesis algorithms.

Finally, the methodology proposed by `CARE` shares aspects with the synthesis of monitors for runtime enforcement [1,19], and is similar to the *automated composition* problem studied in [3,5,16,17], to which `CARE` and `CATLib` offer both a runtime engine and tailored novel synthesis algorithms.

2 Contract-based Applications with `CARE`

We start by discussing the responsibilities of the business actors involved in the overall realisation of contract-based applications using `CARE`.

Two main elements of `CARE` are the classes `RunnableOrchestratedContract` and `RunnableOrchestration`. This last one is a special service that reads the synthesised orchestration and orchestrates the `RunnableOrchestratedContract` to realise the overall application. Each `RunnableOrchestratedContract` is a service wrapper responsible for pairing the specification of a service (the contract) with its implementation. This service is always listening and forks a parallel process when entering an orchestration. During an orchestration, it receives action commands from the orchestrator or other services, and invokes the corresponding action method. In case the invocation is not allowed by the contract or if the contract is not fulfilled (e.g., by reaching an accepting state), a `ContractViolationException` is raised registering the remote host that violates the contract. This guarantees both the adherence of the implementation to the specification and the accountability in case of a contract violation.

Contract automata abstract from the way in which an orchestration is realised. Crucially, offers and requests of contracts abstract from low-level messages sent between services 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. Some aspects to implement are choices and termination. Currently a "dictatorial" choice (i.e., an internal choice of the orchestrator, external for the services) and a "majoritarian" choice (services vote and the majority wins) are two implemented options. `CARE` also provides default implementations for the low-level message exchanges. Currently, two available options are `CentralisedAction`, where the orchestrator acts as a proxy, and `DistributedAction`, 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 (see Sect. 2). The first actor is the provider of the runtime environment, obtained by specialising `CARE`.

The second kind of actors are the service providers, which publish their contracts, implemented by remote (non-disclosed) Java classes, and use a `RunnableOrchestratedContract` to make their contract publicly accessible by using `CARE`, whilst 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 specifying interactions as contract is not required to be an expert in the underlying implementation technology (e.g., Java sockets), whilst the developer implementing actions and selecting the `CARE` configuration is not required to be trained in the theory of contract automata. The specification and implementation of a service can thus be seamlessly integrated using the facilities provided by `CARE`. Most importantly, when implementing the service, the developer need not worry about the underlying low-level interactions between services, potential deadlocks and other communications issues. This error-prone implementation activity is already resolved by `CARE`. This separation also solves the problem of "muddling the main program logic with auxiliary logic related to error handling" (i.e., handling the Java communication exceptions) [20].

The third actor is the designer of the application. 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. The application designer exploits `CARE`, choosing a specific implementation of `RunnableOrchestratedContract`,

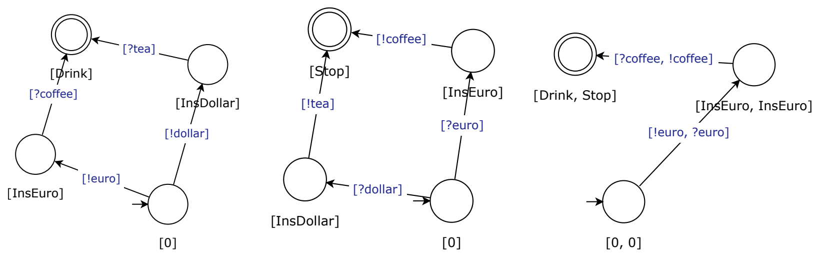


Fig. 1. From left to right, the contracts of `Alice` and `Bob`, and their orchestration enforcing the given requirement.

passing as arguments the addresses of the services, as well as the synthesised orchestration. Formal results from contract automata theory guarantee that no `ContractViolationException` will ever be raised at runtime. Finally, note that one individual could take the roles of more actors if needed (e.g., designing a global requirement, implementing a new choice and publishing a target contract).

Usage. We discuss the usage of `CARE` using a simple yet illustrative example (its source code and a video tutorial are available from [12]). `CATLib` allows to either import an automaton describing the requirement, or to implement it. The requirement `req` specifies that an action `coffee` is observed after an action `euro`.

Consider now Fig. 1 (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.

```java
public interface AliceInterface {
    public Integer coffee(String arg); public Integer tea(String arg);
    public Integer euro(String arg); public 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. By implementing the corresponding interface it is possible to pair the interaction logic described in Fig. 1 (left) with an actual implementation, as shown below.

```java
1  MSCA ca = new DataConverter().importMSCA(dir+"Alice.data");
2  RunnableOrchestratedContract alice = new DictatorialChoiceRunnableOrchestratedContract(ca,8080,
3      new Alice(),new CentralisedOrchestratedAction());
```

In line 1, the leftmost contract in Fig. 1 is imported (`Alice.data`) in one of the formats supported by `CATLib`. The class `Alice` implements `AliceInterface`. This implementation is paired with the corresponding contract in lines 2–3: the service listens to port 8080 and `CentralisedOrchestratedAction` is the chosen implementation of the low level interactions (see below). 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, as provided by the class `TypedCALabel` extending a `CALabel`. 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). The contract of `Bob` is the central automaton in Fig. 1.

The class `RunnableOrchestration` can be instantiated as follows:

```java
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());
```

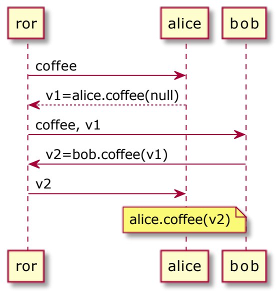
`DictatorialChoiceRunnableOrchestration` provides an implementation of the branch/termination selection where the orchestrator autonomously selects a branch. It is instantiated by passing as parameters the requirement `req` to be enforced, the predicate on interactions among contracts (i.e., the property 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. The constructor of `RunnableOrchestration` uses `CATLib` functionalities to synthetise the safe orchestration with the following instructions:

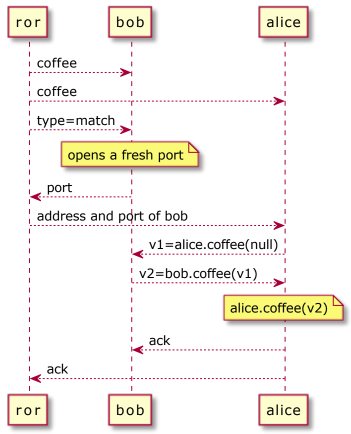
```java
1 MSCA comp = new CompositionFunction(contracts).apply(pred.negate(),bound);
2 MSCA orc = new OrchestrationSynthesisOperator(pred,req).apply(comp);
```

Line 1 computes the composition of the retrieved `contracts`. The other two arguments are optimisations to reduce the size of the composition. The parameter `pred` is the agreement property. Transitions violating agreement (i.e., satisfying `pred.negate()`) are not explored when computing the composition, and `bound` is the bound to the depth of the computed automaton. In line 2, the synthesis of the orchestration enforcing agreement (parameter `pred`) and the requirement (parameter `req`) is applied. 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 rightmost automaton in Fig. 1. 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 example is a dictatorial choice with centralised action. If this is not the case an exception is thrown. Upon successful instantiation, `ror` can be executed.

We discuss the possible implementations of a match action in `CARE`, e.g. `[?coffee,!coffee]`, in which `Alice` is requesting a `coffee` and `Bob` is offering a `coffee`. The sequence of service invocations of both the `CentralisedAction` and `DistributedAction` implementations of `CARE` are displayed below.

In both implementations, 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 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.





In the `DistributedAction` implementation, the orchestrator communicates the chosen action to both matching services, and the offerer is also notified of a match (rather than an offer) action, so that it can open a fresh port to interact with the requester, which is notified of the address and port of the offerer. Upon successful termination of the interactions, each `RunnableOrchestrated-Contract` updates its contract status, and the orchestrator proceeds with the next invocation according to the overall orchestration contract, whose status is also updated. Of course, other implementations are possible. `CARE` allows to tailor the runtime environment to specific needs.

3 Conclusion

We have presented `CARE`, the first library for the realisation of applications from contract automata specifications. The benefits of the methodology introduced by `CARE` are listed. First, the separation of concerns between different actors, providers and users that together cooperate to realise contract-based applications, with a customisable environment provided by `CARE`. Moreover, a separation of concerns between formal methods experts specifying the expected behaviour using automata, and developers implementing the actions while abstracting away from low-level communications provided by `CARE`. Furthermore, the automatic synthesis of the orchestration automaton satisfying the given requirements and carrying formal guarantees on properties such as absence of deadlocks, reachability of final states, absence of `ContractViolationException`. Finally, modularity of an application composed by different services that are reusable in different applications and that can be adapted to satisfy different requirements through the synthesis of a well-behaving orchestration.

Future work. `CATLib` already implements the synthesis of choreographies [9], 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).

References

1. Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: Comparing controlled system synthesis and suppression enforcement. Int. J. Softw. Tools Technol. Transf. **23**(4), 601–614 (2021). https://doi.org/10.1007/s10009-021-00624-0
2. Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida, N.: 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. Discret. 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.) Proceedings of the 36th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'16). LNCS, vol. 9688, pp. 52–61. Springer (2016). https://doi.org/10.1007/978-3-319-39570-8_4
5. Barati, M., St-Denis, R.: Behavior composition meets supervisory control. In: Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC'15). pp. 115–120. IEEE (2015). https://doi.org/10.1109/SMC.2015.33
6. Bartoletti, M., Cimoli, T., Murgia, M., Podda, A.S., Pompianu, L.: A contract-oriented middleware. In: Braga, C., Ölveczky, P.C. (eds.) Proceedings of the 12th International Conference on Formal Aspects of Component Software (FACS'15). LNCS, vol. 9539, pp. 86–104. Springer (2015). https://doi.org/10.1007/978-3-319-28934-2_5
7. Basile, D., ter Beek, M.H.: A clean and efficient implementation of choreography synthesis for behavioural contracts. In: Damiani, F., Dardha, O. (eds.) Proceedings of the 23rd IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION'21). LNCS, vol. 12717, pp. 225–238. Springer (2021). https://doi.org/10.1007/978-3-030-78142-2_14
8. Basile, D., ter Beek, M.H., Degano, P., Legay, A., Ferrari, G.L., Gnesi, S., Di Giandomenico, F.: Controller synthesis of service contracts with variability. Sci. Comput. Program. **187** (2020). https://doi.org/10.1016/j.scico.2019.102344
9. 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
10. Basile, D., Degano, P., Ferrari, G.L.: Automata for Specifying and Orchestrating Service Contracts. Log. Meth. Comp. Sci. **12**(4), 6:1–6:51 (2016). https://doi.org/10.2168/LMCS-12(4:6)2016
11. https://github.com/ContractAutomataProject/CARE/tree/60032cdd1a8c70667c66273ada7e95f3a42eb8b7, February 2022
12. https://github.com/ContractAutomataProject/CARE_Example/tree/ee938f97549d02f34a202585be9a3dacf3c3403e, including a video tutorial reproducing the example, February 2022
13. https://github.com/ContractAutomataProject/ContractAutomataLib
14. https://github.com/ContractAutomataProject/ContractAutomataApp
15. Dilley, N., Lange, J.: An empirical study of messaging passing concurrency in Go projects. In: Proceedings of the 26th IEEE International Conference on Software

Analysis, Evolution and Reengineering (SANER'19). pp. 377–387. IEEE (2019). https://doi.org/10.1109/SANER.2019.8668036

16. Farhat, H.: Web service composition via supervisory control theory. IEEE Access **6**, 59779–59789 (2018). https://doi.org/10.1109/ACCESS.2018.2874564
17. 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
18. Ferrari, A., ter Beek, M.H.: Formal methods in railways: a systematic mapping study. CoRR **abs/2107.05413** (2021), https://arxiv.org/abs/2107.05413
19. Francalanza, A.: A theory of monitors. Inf. Comput. **281**, 104704 (2021). https://doi.org/10.1016/j.ic.2021.104704
20. Francalanza, A., Mezzina, C.A., Tuosto, E.: Towards choreographic-based monitoring. In: Ulidowski, I., Lanese, I., Schultz, U.P., Ferreira, C. (eds.) Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405, LNCS, vol. 12070, pp. 128–150. Springer (2020). https://doi.org/10.1007/978-3-030-47361-7_6
21. Gay, S., Ravara, A. (eds.): Behavioural Types: from Theory to Tools. River (2017). https://doi.org/10.13052/rp-9788793519817
22. 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
23. 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 ACM/IEEE International Conference on Software Engineering (ICSE'18). pp. 1137–1148. ACM (2018). https://doi.org/10.1145/3180155.3180157
24. Mota, J., Giunti, M., Ravara, A.: Java typestate checker. In: Damiani, F., Dardha, O. (eds.) Proceedings of the 23rd IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION'21). LNCS, vol. 12717, pp. 121–133. Springer (2021). https://doi.org/10.1007/978-3-030-78142-2_8
25. 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
26. Rubbens, R., Lathouwers, S., Huisman, M.: Modular transformation of Java exceptions modulo errors. In: Lluch Lafuente, A., Mavridou, A. (eds.) Proceedings of the 26th International Conference on Formal Methods for Industrial Critical Systems (FMICS'21). LNCS, vol. 12863, pp. 67–84. Springer (2021). https://doi.org/10.1007/978-3-030-85248-1_5
27. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng. **12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929
28. 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'20). EPTCS, vol. 324, pp. 25–42 (2020). https://doi.org/10.4204/EPTCS.324.4
29. Vasconcelos, C., Ravara, A.: From object-oriented code with assertions to behavioural types. In: Proceedings of the 32nd ACM Symposium on Applied Computing (SAC'17). pp. 1492–1497. ACM (2017). https://doi.org/10.1145/3019612.3019733