Logical Methods in Computer Science
Volume 22, Issue 1, 2026, pp. 8:1–8:40
https://lmcs.episciences.org/

Submitted Jan. 23, 2025
Published Mar. 05, 2026

FORMAL ANALYSIS OF THE CONTRACT AUTOMATA RUNTIME ENVIRONMENT WITH UPPAAL: MODELLING, VERIFICATION AND TESTING

DAVIDE BASILE

Formal Methods and Tools Lab, ISTI–CNR, Pisa, Italy
e-mail address: davide.basile@isti.cnr.it

Abstract. Recently, a distributed middleware application called contract automata runtime environment (CARE) has been introduced to realise service applications specified using a dialect of finite-state automata. In this paper, we detail the formal modelling, verification and testing of CARE. We provide a formalisation as a network of stochastic timed automata. The model is verified against the desired properties with the tool Uppaal, utilising exhaustive and statistical model checking techniques. Abstract tests are generated from the Uppaal models that are concretised for testing CARE. This research emphasises the advantages of employing formal modelling, verification and testing processes to enhance the dependability of an open-source distributed application. We discuss the methodology used for modelling the application and generating concrete tests from the abstract model, addressing the issues that have been identified and fixed.

1. Introduction

Behavioural contracts [BCZ15] have been introduced to formally describe the interactions among services, to enable to reason formally about well-behaving properties of their composition. Examples of properties are agreement among the parties or reachability of target states.

Contract automata are a dialect of finite state automata used to specify behavioural contracts formally in terms of offers and requests [BDF16]. 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 [BtBD$^+$20, BtBP20], a variation of the synthesis algorithm from supervisory control theory [RW87]. The Contract Automata Runtime Environment (CARE) [BtB23a] provides a middleware to coordinate the services implementing contracts. In CARE, each transition of the orchestration automaton is executed by a series of interactions among the orchestrator and the CARE services, implemented using Java TCP/IP sockets. These interactions may vary according to the specific configuration chosen among those provided by CARE. In [BtB23a] the algorithms implemented in CARE are proved to enforce the adherence of each contract specification to its CARE implementation (basically, the control flow of the application follows the synthesised orchestration automaton).

In this paper, we describe the modelling, verification and testing of the low-level interactions among CARE services and the orchestrator. This aspect is no less important, as witnessed by known cases of algorithms proved to be correct (e.g., the Byzantine distributed

DOI:10.46298/LMCS-22(1:8)2026

consensus [LSP82]) and whose low-level communication implementations were found to have issues, e.g. deadlocks [RCS+22]. We verify several properties, including the absence of deadlocks, absence of undelivered messages, and reachability of target states.

The formal model is a network of stochastic timed automata as accepted by the Uppaal toolbox. Different variants of the formal model are proposed. The model undergoes verification against the desired properties using Uppaal, employing both exhaustive model checking, statistical model checking and model-based testing. This combination allows for thorough analysis of the model's behaviour and ensures scalability when dealing with systems that possess a large state-space.

The benefits of modelling and verifying `CARE` are: (i) increasing the confidence in the reliability of `CARE` due to the formal verification through model checking and model-based testing, (ii) quantitative evaluation of measures of interest of `CARE` obtained through statistical model checking, (iii) improved documentation thanks to the graphical and animatable state machines endowed with precise semantics.

Finally, this paper tackles the challenge of providing a full-fledged model-based development and formal methods approach [GtBvdP20]. The final application has been graphically modelled at an abstract level, formally verified and tested using the formal model. The criteria followed for abstracting away irrelevant details are discussed together with the issues that have been found and fixed thanks to the formal modelling, verification and testing.

All models, formulas, and logs are publicly available in [Basb], together with traceability and model-based testing information connecting the model to the source code.

This work is an extension of the conference paper [Bas24]. The extension includes the addition of more figures and tables, as well as further explanations throughout the paper. The most significant addition concerns the testing phase, which was only briefly mentioned in [Bas24]. Testing is now thoroughly described in Section 7. We cover both the generation of abstract tests from the Uppaal model and the process of concretising these tests in the real application.

Summarising, the core contributions of this paper are: (i) the bottom-up formal modelling of `CARE`, an already established, open-source distributed middleware, modelled as a network of stochastic timed automata suitable for the Uppaal toolbox; (ii) the verification of desired properties of the `CARE` formal model through the application of both exhaustive and statistical model checking techniques within Uppaal; (iii) the establishment of a direct connection between the abstract formal model and the actual source code of `CARE` using traceability and model-based testing. Abstract tests are generated from the Uppaal models and concretised as JUnit tests for the `CARE` implementation. This connection helps to validate the chosen level of abstraction.

Outline We start by discussing the related work in Section 2. In Section 3, we provide the background on contract automata, their tool support, Uppaal, and the contract automata runtime environment. The methodology used for modelling and analysing `CARE` is described in Section 4. Section 5 contains the description of the models, whilst the verification is described in Section 6. Model-based testing is addressed in Section 7. Finally, the conclusion and future work are presented in Section 8.

2. Related work

Several applications of Uppaal to various case studies are available in the literature, including land transport [BtBL20], maritime transport [SVW20], medical systems [LRN+22], and autonomous agents path planning [GJP+22]. These case studies, along with the present paper, adopt a model-based approach, wherein partial representations of the applications are created using models.

A recent survey conducted on formal methods [FtB22] reveals that numerous publications lack a direct correlation between the concrete implementation and its abstract model. This holds especially when the systems are only envisioned, yet to be realised, such as for example the ERTMS L3 railway signalling system [BtBC18, BtBFL22]. This is also the case for industrial systems whose implementations are non-disclosed [HMG+23]. In all these cases, determining the accuracy of the model in relation to the actual system and assessing the appropriateness of the chosen level of abstraction becomes challenging. Furthermore, measuring the influence of the formal modelling and verification phase on the analyzed system poses difficulties.

Indeed, a recent survey on formal methods in the transport domain [FtB22] identifies the need "to lower the degree of abstraction, by applying formal methods and tools to development phases that are closer to software development". This is also confirmed by a recent survey on formal methods among 130 high-profile experts [GtBvdP20], where nearly half of them believe that the most likely future users of formal methods are "a small number of skilled experts" as long as "skills like modelling, specification, abstraction are involved".

In contrast to the previously mentioned literature, in this paper we present a bottom-up formal analysis of an established open-source system that has already been developed [BtB23a]. The availability of the source code further enables us to establish a connection between the abstract formal model and the actual source code. This capability facilitates the precise identification of specific aspects of the real system that have been abstracted in the formal model. Additionally, it allows us to validate the appropriateness of the chosen level of abstraction.

Recently, Yggdrasil [HLM+08], the offline model-based testing functionality of Uppaal, has been used in [KLN+15] for testing an industrial real-time automotive turn indicator system, and in [HL23] for testing EIP-1559 Ethereum smart contracts. The usage of the Uppaal suite, and more specifically Yggdrasil, in three industrial case studies (i.e., medical devices, an automotive system and a pump manufacturing) is reviewed in [LLN18]. Many of the above applications of model-based testing using Uppaal concern industrial, non-disclosed case studies. Therefore, only the models and abstract test cases are reported, whilst concrete test cases and source code have not been made public. In comparison, CARE is an open-source application. The concrete tests generated from the Uppaal models discussed in this paper, as well as the system under testing, are available at [Basb]. To the best of our knowledge, there are no other non-trivial open-source applications for which their Uppaal formal model is openly accessible and directly connected to the source code through traceability and model-based testing. This contribution assists in linking formal methods, particularly Uppaal, to the software development process.

The contract automata approach is closer to [KDPG18], where behavioural types are expressed as finite state automata of Mungo, called typestates [SY86, AD24]. Similarly to CARE, in Mungo finite state automata are used as behaviour assigned to Java classes (one automaton per class), with transition labels corresponding to methods of the classes. A tool to translate typestates into automata was presented in [TMR20]. CATApp is a graphical front-end tool for designing contract automata [Basa]. A tool similar to Mungo is JaTyC (Java Typestate Checker) [BBG+22].

Other model-based verification approaches for service-oriented systems and distributed middleware are surveyed in [RG21, CHLR23]. In the literature, there exist many formalisms for modelling and analysing the behaviour of services, ranging from behavioural type systems, including behavioural contracts [CGP09, ABZ13, LP15] and session types [BLMT08, HYC08, DCd10, CDCP12, MNF13], to automata-based formalisms, including interface automata [dAH01] and (timed) (I/O) automata [LT89, AD94, DLL$^+$10]. Foundational models for service contracts and session types are surveyed in [tBBG07, BCZ15, HLV$^+$16]. Sessions and session types [BLMT08, HYC08, DCd10, CDCP12, MNF13] have been introduced to reason over the behaviour of services in terms of their interactions. Compared to contract automata, behavioural contracts and session types use process-algebraic frameworks, instead of finite state automata. Indeed, contract automata are similar to other software engineering-based formalisms such as I/O automata [LT89, AD94, DLL$^+$10].

In the literature there are various attempts to model applications that, similarly to CARE, communicates through TCP/IP sockets, also using Uppaal [WWH$^+$22, FZL18, SF17, WVHFM06]. In [CGMZ21] the verification of the transport layer of a TCP/IP protocol stack implementation in C is performed with a translation into SPARK, where contracts are modelling the corresponding behaviour. A method to obtain SPIN formal models from TCP/IP socket applications implemented in C is presented in [dlCGMS09] where the C application is embedded into a PROMELA model equipped with a formalisation of the underlying TCP/IP socket API. Similarly, we provide a model of the underlying socket communications but we also modelled aspects of the application that are targeted by our analysis whilst unnecessary details are abstracted away. Another approach [LSM$^+$20] presents an automatic translation from MAUDE specifications of distributed applications to distributed MAUDE implementations using TCP/IP sockets, where the implementations are formally proven to be in correspondence with the specifications. Our approach is bottom-up and is based on Uppaal modelling of an already realised application.

As reported in [Bou15], the question of qualification and validation of formal methods tools is “absolutely crucial”. Other tools for behavioural contracts are present in the literature [GR17, OPB$^+$21, PMT24], but differently from CARE, many of these implementations have not undergone a process of formal verification.

3. Background

In this section, we will provide background on contract automata, their software support, and the Uppaal statistical model checker. The focus of this paper is on the formal analysis of the runtime environment of contract automata. While we offer a concise overview of contract automata to enhance comprehension of their runtime environment, they are not the focus of our formal analysis.

3.1. Contract Automata.

Contract automata are a dialect of finite-state automata modelling services that exchange offers and requests. A contract automaton models either a single service or a composition of interacting services. Labels of transitions of contract automata are vectors of atomic elements called *actions*. Similarly, states of contract automata are vectors of atomic elements called basic states. The length of the vectors is equal to the number of services in the automaton (this number is called *rank*). A request (resp., offer) action is prefixed by `?` (resp., `!`). The idle action is denoted by `-`. Labels are constrained to be one out of three types. In a *request* (resp., *offer*) label a service performs a request (resp.,

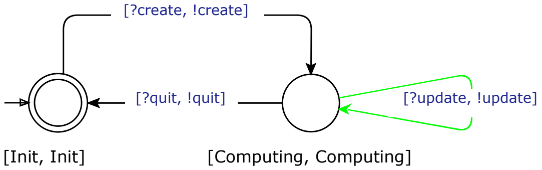


Figure 1. An example of contract automaton

offer) action and all other services are idle. In a *match* label one service performs a request action, another service performs a matching offer action, and all other services are idle. For example, the contract automaton in Figure 1 right has rank 2 and the label `[?quit,!quit]` is a match where the request action `?quit` is matched by the offer action `!quit`. Note the difference between a request label, e.g. `[?coffee, -]`, and a request action, e.g. `?coffee`.

In a composition of contracts various properties can be analysed [BDF16]. For example, the property of *agreement* requires to match all request actions, whereas offer actions can remain unmatched. The synthesis of the orchestration in agreement produces a sub-automaton of the composition where all services can match their requests with corresponding offers to reach a final state. Thus, in the orchestration in agreement labels of transitions are only matches or offers [BtBP20]. The contract automaton in Figure 11 right is an orchestration in agreement.

Contract automata and their functionalities are implemented in a software artefact, called Contract Automata Library (`CATLib`) [BtB22]. With `CATLib` it is possible to specify, compose, and synthesise specifications given as contract automata. Indeed, `CARE` uses the facilities offered by `CATLib` for, among other things, composing contracts and synthesising orchestrations. 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., [BtBD$^+$20, BtBP20, BDF16]). Scalability features offered by `CATLib` include a bounded on-the-fly state-space generation optimised with pruning of redundant transitions and parallel streams computations [BtB23b, BtB24]. The software is open source [BtB22], it has been developed using principles of model-based software engineering [BtB21] and it has been extensively validated using various testing and analysis tools to increase the confidence on the reliability of the library [BtB22].

3.2. Uppaal.

Uppaal SMC [DLL$^+$15] is a variant of Uppaal [BDL$^+$06], which is a well-known toolbox for the verification of real-time systems. Uppaal models are stochastic timed automata, in which non-determinism is replaced with probabilistic choices, and time delays with probability distributions (uniform for bounded time and exponential for unbounded time). These automata may communicate only via broadcast channels and shared variables. In this paper, we will use both exhaustive and statistical model checking. Statistical Model Checking (SMC) [LLT$^+$19] involves running a sufficient number of (probabilistic) simulations of a system model to obtain statistical evidence (with a predefined level of statistical confidence) of the quantitative properties to be checked. Monte Carlo estimation with Chernoff-Hoeffding bound executes $N = \lceil (ln(2) - ln(\alpha))/(2\varepsilon^2) \rceil$ simulations ρ_i, $i \in 1...N$, to provide the interval $[p' - \varepsilon, p' + \varepsilon]$ with confidence $1 - \alpha$. Here, $p' = (\#\{\rho_i \,|\, \rho_i \models \varphi\})/N$, i.e., $\mathtt{Pr}(|p' - p| \leq \varepsilon) \geq 1 - \alpha$ where p is the unknown value of the formula φ being estimated statistically and ε and α are the user-defined precision and confidence, respectively. Crucially, the number of simulations used to estimate a formula is independent of the model's size and depends only on the parameters α and ε. In practice the number of simulations required by Uppaal to reach a specific confidence level is optimized and is thus lower than the above theoretical bound. Uppaal supports template automata used to instantiate different copies (in different experiments) of the same automaton, distinguishable by their parameters. The selection of Uppaal as the chosen tool is influenced by several factors. These include its extensive adoption by the community, expertise of the author, usability, primitive support for real-time and stochastic modelling, probabilistic and non-deterministic choices, and capabilities for statistical model checking, simulation, and model-based testing.

The off-line test case generator of Uppaal is called Yggdrasil, which is integrated since version 4.1 into the Uppaal GUI as a tab. Starting from the Uppaal models, a suite of test cases is created, aiming to cover all syntactic transitions in the model (i.e., edge coverage). The Uppaal model can be decorated with test code, which can be generated from the execution of a transition or at the entry to or exit from a location. Any language can be used for the test code. An abstract test case is generated from a trace of execution of the Uppaal model. During the execution of the trace, the test code of the model is dumped into a text file, forming the abstract test case. When the system is composed of a set of automata, it is possible to generate the test code from a single automaton or from the whole system. In this latter case, the single generated abstract test case contains the interleaved test code from all automata. The traces of execution can be generated as witnesses of some reachability property defined using the Uppaal syntax. Random traces can also be executed to improve test coverage. Parameters to be set include the depth of the trace, the type of search (i.e., breadth, depth), and options for the trace to be generated (i.e., fastest, shortest). The generated abstract test cases need to be decorated and filled with the missing information from the system under test to become concrete test cases. More details will be provided in Section 7.

3.3. CARE.

The Contract Automata Runtime Environment (`CARE`) [BtB23a] provides facilities for pairing the contract automata specifications with actual implementations of service-based applications. Note that our purpose is not to formally analyse the applications created using `CARE`, but to formally analyse `CARE` itself. The formal verification of applications developed using `CARE` is a consequence of the reliance of `CARE` on the formal guarantees provided by contract automata [BtB23a], along with the formal verification of `CARE` as an application. This paper addresses the latter aspect. This software is organised into classes for the orchestrated services (cf. Figure 2) and classes for the orchestrator (cf. Figure 3). The two core abstract Java classes are the `RunnableOrchestration` and the `RunnableOrchestratedContract`.

Services. In Figure 2, `RunnableOrchestratedContract` is an abstract class that implements an executable wrapper responsible for pairing its contract automaton specification (instance variable `contract` storing a contract automaton) with an implementation provided as a Java class (instance variable `service` implementing the service), where each action of the automaton `contract` is in correspondence with a method of the `service` class. The types of the methods of a matching offer and request must also be in correspondence. Each time a new orchestration involving the service is initiated, the `RunnableOrchestratedContract` creates a new service. This service remains in a waiting state to receive commands from the orchestrator, which then triggers the execution of the corresponding methods. In case the orchestrator requires to perform an action not prescribed by its service contract, then a

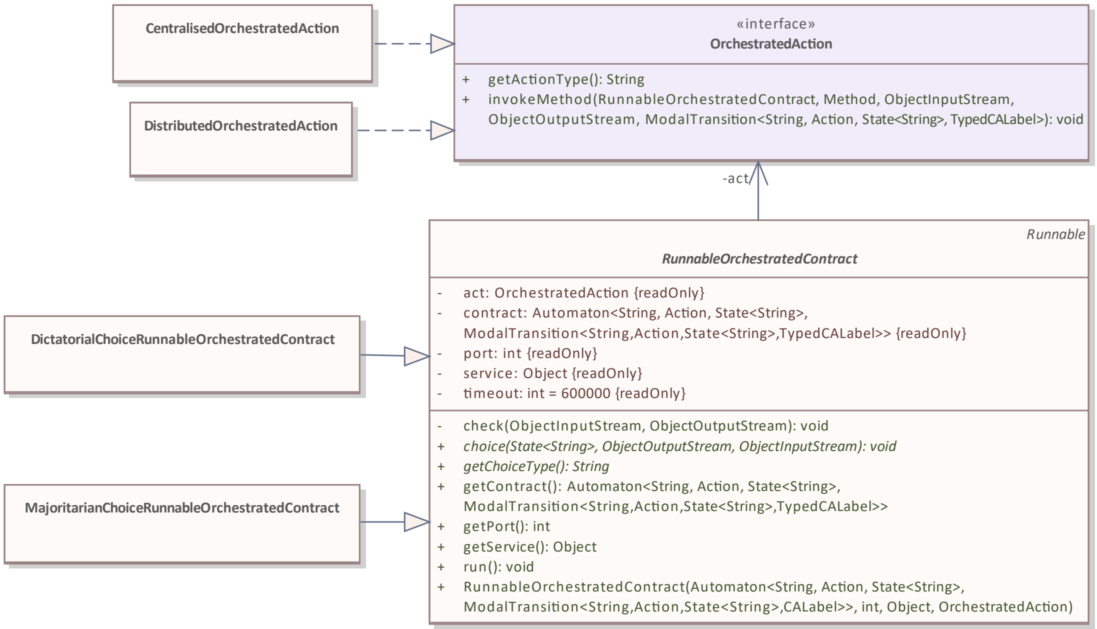

CentralisedOrchestratedAction
DistributedOrchestratedAction
«interface»
OrchestratedAction
+ getActionType(): String
+ invokeMethod(RunnableOrchestratedContract, Method, ObjectInputStream, ObjectOutputStream, ModalTransition<String, Action, State<String>, TypedCALabel>): void
-act
Runnable
RunnableOrchestratedContract
- act: OrchestratedAction {readOnly}
- contract: Automaton<String, Action, State<String>, ModalTransition<String,Action,State<String>,TypedCALabel>> {readOnly}
- port: int {readOnly}
- service: Object {readOnly}
- timeout: int = 600000 {readOnly}
- check(ObjectInputStream, ObjectOutputStream): void
+ choice(State<String>, ObjectOutputStream, ObjectInputStream): void
+ getChoiceType(): String
+ getContract(): Automaton<String, Action, State<String>, ModalTransition<String,Action,State<String>,TypedCALabel>>
+ getPort(): int
+ getService(): Object
+ run(): void
+ RunnableOrchestratedContract(Automaton<String, Action, State<String>, ModalTransition<String,Action,State<String>,CALabel>>, int, Object, OrchestratedAction)
DictatorialChoiceRunnableOrchestratedContract
MajoritarianChoiceRunnableOrchestratedContract

Figure 2. 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)

ContractViolationException will be raised by the service. As demonstrated in [BtB23a], this exception will never be raised if the orchestration in agreement is synthesised using the algorithms of contract automata.

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 Figure 2 and Figure 3.

Orchestrator. The class diagram of the orchestrator side in Figure 3. The abstract class RunnableOrchestration implements a special service that reads the synthesised orchestration (stored in the instance variable contract) and orchestrates the services (the instances of RunnableOrchestratedContract) to realise the overall application. The instance variables port and addresses contain the corresponding addresses and ports of the services involved in the orchestration.

Choices and Actions. Two aspects to implement for both the orchestrator and the services are choices and termination (through the abstract method choice). Indeed, during the execution of an orchestration, the selection of the transition to execute in the presence of multiple enabled transitions is implemented through this abstract method. 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. For the services, MajoritarianChoiceRunnableOrchestratedContract

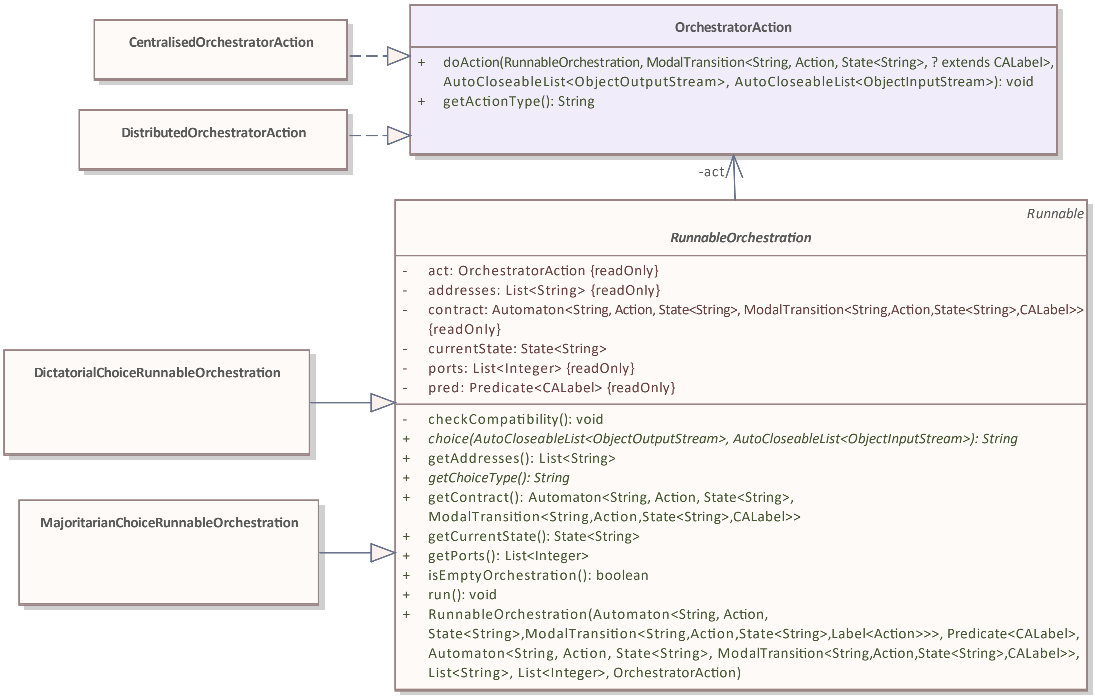

CentralisedOrchestratorAction
DistributedOrchestratorAction
OrchestratorAction
+ doAction(RunnableOrchestration, ModalTransition<String, Action, State<String>, ? extends CALabel>, AutoCloseableList<ObjectOutputStream>, AutoCloseableList<ObjectInputStream>): void
+ getActionType(): String
-act
Runnable
RunnableOrchestration
- act: OrchestratorAction {readOnly}
- addresses: List<String> {readOnly}
- contract: Automaton<String, Action, State<String>, ModalTransition<String,Action,State<String>,CALabel>> {readOnly}
- currentState: State<String>
- ports: List<Integer> {readOnly}
- pred: Predicate<CALabel> {readOnly}
- checkCompatibility(): void
+ choice(AutoCloseableList<ObjectOutputStream>, AutoCloseableList<ObjectInputStream>): String
+ getAddresses(): List<String>
+ getChoiceType(): String
+ getContract(): Automaton<String, Action, State<String>, ModalTransition<String,Action,State<String>,CALabel>>
+ getCurrentState(): State<String>
+ getPorts(): List<Integer>
+ isEmptyOrchestration(): boolean
+ run(): void
+ RunnableOrchestration(Automaton<String, Action, State<String>,ModalTransition<String,Action,State<String>,Label<Action>>>, Predicate<CALabel>, Automaton<String, Action, State<String>, ModalTransition<String,Action,State<String>,CALabel>>, List<String>, List<Integer>, OrchestratorAction)
DictatorialChoiceRunnableOrchestration
MajoritarianChoiceRunnableOrchestration

Figure 3. The class diagram for the orchestrator; the methods of the derived classes are visible in their super-class/interface as abstract methods (in italic)

and `DictatorialChoiceRunnableOrchestratedContract` are the two classes specialising `RunnableOrchestratedContract` according to how the choice is handled and implementing the abstract methods, whilst `MajoritarianChoiceRunnableOrchestration` and `DictatorialChoiceRunnableOrchestration` are specialising `RunnableOrchestration`.

`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`, and `RunnableOrchestration` has an `OrchestratorAction` (instance variable `act`) used to implement the corresponding actions that can be either distributed or centralised according to the current implementation.

Example 3.1. We discuss an example of interactions between two services and an orchestrator. The example comprehends both the `CentralisedAction` and `DistributedAction` implementations of `CARE`. The sequence of service invocations is displayed in Figure 4. In this example, the orchestrator is executing a transition of the orchestration automaton that is labelled with the match `[?coffee,!coffee]`, in which `Alice` is requesting a `coffee` and `Bob` is offering a `coffee`. As stated above, the `RunnableOrchestratedContract` class of both `Alice` and `Bob` has two instance variables (`contract` and `service`, resp.) containing the contract automaton object and the implementation provided as a Java class. In this example, the implementation class of `Alice` has a method `Integer coffee(String arg)` that is paired with the corresponding request action `?coffee` of her contract. Similarly, the

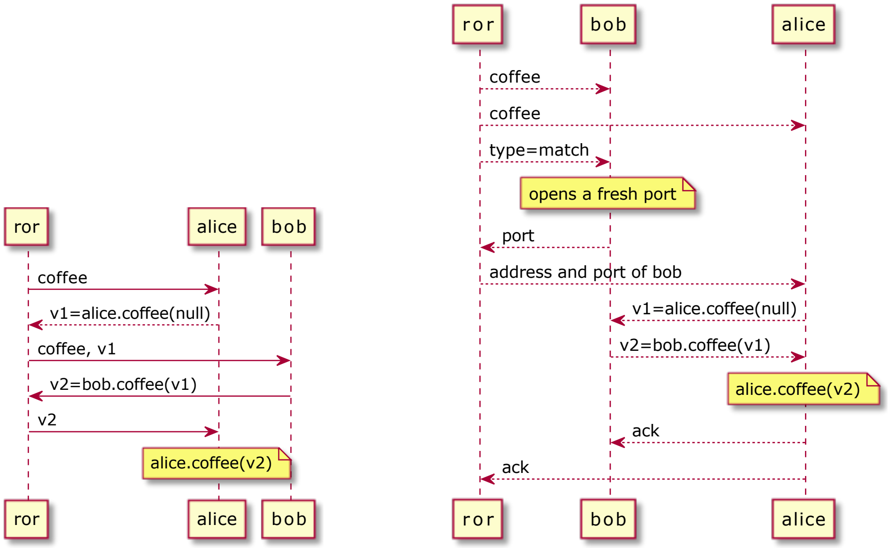


FIGURE 4. The sequence diagrams depicting the interactions implementing a centralised match action (on the left) and a distributed match action (on the right)

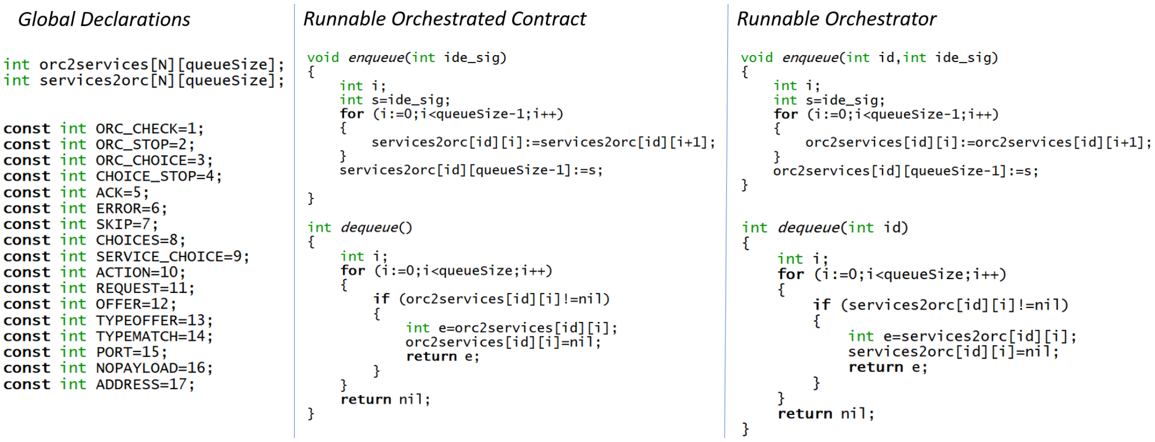
implementation class of `Bob` has a method `String coffee(Integer arg)` that is paired with the corresponding offer action `!coffee` of his contract.

In both centralised and dictatorial configurations, the method `coffee` of `Alice` is invoked twice: firstly, passing no argument, it returns an `Integer` value (e.g., the amount of sugar) that is passed (by the orchestrator `ror`) as an argument to the method `coffee` of `Bob`, which in turn returns a `String` value that is eventually passed as an argument to the method `coffee` of `Alice`, thus fulfilling the `coffee` request.

In the `CentralisedAction` implementation, the orchestrator acts as a broker, forwarding to the involved parties the various invocations and responses. 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 `RunnableOrchestratedContract` updates its contract status, and the orchestrator proceeds with the next invocation according to the overall orchestration contract, whose status is also updated. In Section 5 will discuss how this kind of interaction is modelled.

4. Methodology

In this section, we describe both the methodology used for modelling the communications, abstracting away irrelevant details, and the adequacy of the model to the actual implementation.



Global Declarations

```
int orc2services[N][queueSize];
int services2orc[N][queueSize];


const int ORC_CHECK=1;
const int ORC_STOP=2;
const int ORC_CHOICE=3;
const int CHOICE_STOP=4;
const int ACK=5;
const int ERROR=6;
const int SKIP=7;
const int CHOICES=8;
const int SERVICE_CHOICE=9;
const int ACTION=10;
const int REQUEST=11;
const int OFFER=12;
const int TYPEOFFER=13;
const int TYPEMATCH=14;
const int PORT=15;
const int NOPAYLOAD=16;
const int ADDRESS=17;
```

Runnable Orchestrated Contract

```
void enqueue(int ide_sig)
{
    int i;
    int s=ide_sig;
    for (i:=0;i<queueSize-1;i++)
    {
        services2orc[id][i]:=services2orc[id][i+1];
    }
    services2orc[id][queueSize-1]:=s;

}

int dequeue()
{
    int i;
    for (i:=0;i<queueSize;i++)
    {
        if (orc2services[id][i]!=nil)
        {
            int e=orc2services[id][i];
            orc2services[id][i]=nil;
            return e;
        }
    }
    return nil;
}
```

Runnable Orchestrator

```
void enqueue(int id,int ide_sig)
{
    int i;
    int s=ide_sig;
    for (i:=0;i<queueSize-1;i++)
    {
        orc2services[id][i]:=orc2services[id][i+1];
    }
    orc2services[id][queueSize-1]:=s;
}


int dequeue(int id)
{
    int i;
    for (i:=0;i<queueSize;i++)
    {
        if (services2orc[id][i]!=nil)
        {
            int e=services2orc[id][i];
            services2orc[id][i]=nil;
            return e;
        }
    }
    return nil;
}
```

Figure 5. The Uppaal code used to model the TCP/IP socket communications

4.1. Modelling TCP/IP sockets communication.

Java TCP/IP sockets communications are asynchronous with FIFO buffers [Obj]. In Uppaal, the interactions are via channel synchronisation and global variables. Thus, TCP/IP sockets are solely employed in the actual implementation and are not utilized by the automata of the model. Instead, in the model each party communicates with the partner using two global arrays (one for sending and one for receiving, respectively). Figure 5 shows the code used in the Uppaal models (see Section 5) for modelling the TCP/IP socket communications.

The two global (FIFO) arrays in the model are named `orc2services[N][queueSize]` and `services2orc[N][queueSize]` (see Figure 5 left). These are used to model the TCP/IP socket buffers, where `N` is the number of services in the model and `queueSize` is the size of the queue of the components. These arrays are only modified by functions for enqueueing and dequeuing messages.

The local declarations of the two automata contain methods for sending and receiving from the partners and checking their queues of messages. Both automata declare a method `enqueue` for sending to the partner a message. Messages are modeled as global integer constants, depicted in Figure 5 left. Indeed, in this model the actual payload of each communication is abstracted away. For example, in Figure 3 left, the constants `REQUEST` and `OFFER` abstract payloads that could be any class instantiated by the user, comprehending newly created Java classes (see Section 4.2). The `enqueue` method of the services only takes as parameter the signal `ide_sig` to send (see Figure 5 center). The identifier of the service `id` is only needed on the orchestrator side to identify the partner (there is only one orchestrator thus no identifier is needed for it, see Figure 5 right). Similarly, both automata have a method `dequeue` for consuming messages from their respective arrays.

The default mode for Java TCP/IP sockets is *blocking* [Jav], meaning that the sender blocks when the buffer of the receiver is full, and waits until there is enough space to proceed. Accordingly, a transition having a send in its effect will check in its guard whether there is enough space left in the array of the partner by calling either the method `available` (returning the space left) or `isFull`. Similarly, in Java TCP/IP sockets the read operation is *blocking*. Accordingly, before reading it is always checked whether the array is not empty with the method `!isEmpty`. When the array is empty the automaton blocks until a message is received.

Source locations of sending and receiving transitions are neither committed nor urgent. In fact, an enabled committed transition (i.e., whose source location is committed, denoted with C) must be executed before any other non-committed transition in the network. Instead, an urgent transition must be executed without any delay. If a sending or receiving transition were to be either committed or urgent, it could introduce the possibility of false positive deadlocks. This scenario arises when, for example, the receiver has a full buffer (i.e., array) and is prepared to free it, but the sender transition is enabled and committed. Similarly, this false positive can occur when the buffer is full, the send transition is urgent but the receive operation is not urgent, or vice versa, when the buffer is empty, the read transition is urgent, and the send operation is not urgent.

Hence, the operations of writing to and reading from a buffer are represented using stochastic delays, specifically following an exponential distribution. Two rates are employed to capture the delay associated with reading and writing. These exponential delays (variables write and read) are present in all non-committed and non-final locations of both automata in Figure 9 and Figure 10 (modelling the orchestrator and the services, cf. Section 5). However, the presence of unbounded delays introduces scenarios where the receiver (resp., sender) may wait indefinitely without executing its read (resp., write) transition, even when it is enabled. These scenarios would invalidate the exhaustive model checking of reachability properties that are satisfied by the actual system, leading to false positives. In the real implementation, Java TCP/IP sockets offer a timeout mechanism wherein an exception is thrown if no message exchange occurs within a specified time frame. All sockets used by CARE have this timeout. Consequently, a dedicated automaton called SocketTimeout is replicated for each service to model the timeout operation (see Figure 11 left). Each send or receive operation in every socket resets the SocketTimeout clock c. If no reset operation is received within a certain duration (variable timeout), SocketTimeout enters a location called Timeout and broadcasts the signal fail, indicating that an exception has been thrown. All automata have a transition from every location (except for Terminated) to a location Timeout that is reached upon receiving the signal fail. For readability, these Timeout locations and all their incoming fail transitions are not shown in Figure 9 and Figure 10.

4.2. Abstractions.

We have discussed the modelling of Java TCP/IP socket communications. We now discuss other aspects that have been abstracted away in the model. We note that the abstracted aspects are irrelevant for the analysis discussed in Section 6. We remark that the paper objective is to verify the interactions of the CARE middleware itself. Therefore, the underlying application that is executed by CARE is abstracted away. Therefore, in the model, the underlying orchestration automaton is abstracted together with the contracts of the services. Thus, all conditionals that are dependent from the underlying orchestration are abstracted as probabilistic choices. We assume that the orchestration has been correctly synthesised from the services contracts. This allows us to verify the interactions for any possible valid orchestration. If a specific orchestration would be modelled, then we would lose such generality. In particular, the payloads of the communications (e.g., which specific action, which choices) are abstracted. The conditions used to decide whether to perform a choice, an action or to stop are also abstracted away. The only modelled condition is that no two consecutive choices are allowed (i.e., after choosing, the chosen step must be performed). When executing a transition, the conditions used to check whether the label of the transition is an offer or a match are abstracted away in the model. Moreover, the identifiers of the services involved in performing a choice or an action (which are concretely extracted from the labels of the transitions) are also chosen non-deterministically. Indeed, all services are distinguished replicas of the same automaton. Finally, we model a single orchestration. In fact, when multiple orchestrations are executed, they operate independently of each other and can be verified individually.

As stated above, the underlying application executed by `CARE` is abstracted away. Therefore, properties related to the underlying application are not verifiable under the current abstraction. An example of property not verifiable under the current abstraction could be to show that the returned payload of an action (e.g., an `Integer` object) is always non-negative, by reasoning on the code of the corresponding method returning the value. This kind of properties cannot be verified under the current abstraction. In this case, a theorem prover like KeY [ABB^{+}16] would be more suited than Uppaal.

4.3. **Traceability.** This paper adopts a lightweight approach to integrate the formal model with the concrete implementation, using the facilities provided by Uppaal. Traceability and model-based testing have been used to ascertain the adequacy of the model with respect to the implementation. Traceability involves connecting the items of the abstract model with the lines of source code that are being abstractly represented. As stated previously, some aspects of the application are abstracted away. Therefore, only the relevant source code lines are connected with the abstract model. In [Basb] each transition of the model is traced back to the corresponding source code instructions using comments in the model. Traceability information is related to a specific version of the source code[1]. For example, Figure 6 shows a portion of the automaton modelling the `RunnableOrchestratedContract` (discussed in details in the next section). Specifically, from state `Ready` a transition is executed which is traced back to line 98 of the class `RunnableOrchestratedContract`, also showed in Figure 6. Indeed, at line 98 the first read operation from the `ObjectInputStream`, reading from the TCP/IP socket, is performed by the service. This is abstracted in the model by the operation `dequeue()` of the automaton.

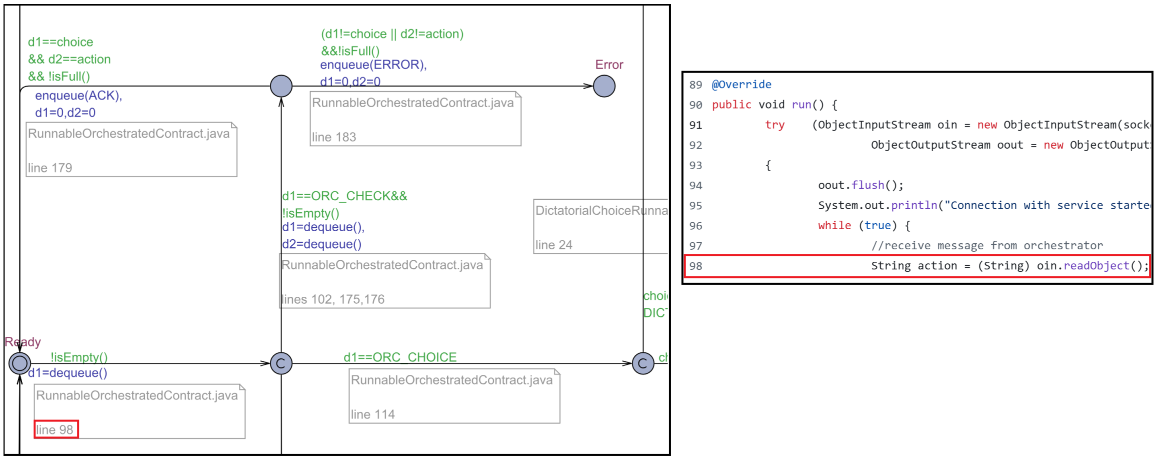
Testing is used to show the adherence of the model to the actual implementation, and will be discussed in Section 7.

Remark. We highlight the advantages of employing graphical diagrams. When it comes to the implementation phase, developers typically work with the source code, which currently consists of 770 lines of code in the case of `CARE`. On the other hand, during the modelling phase with Uppaal, designers graphically edit automata. The automata depicted in Figure 9 and Figure 10 succinctly and accurately specify the interaction logic of `CARE`.

5. Formal model

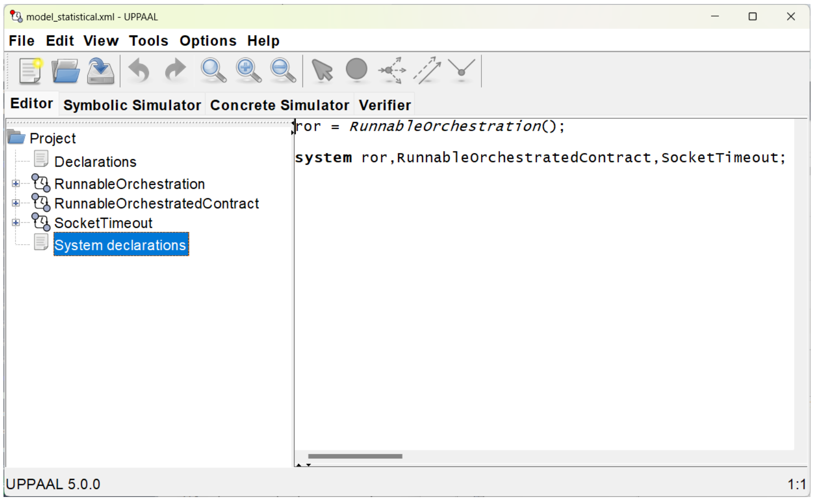
This section describes the formal model of `CARE`. All models used in this paper together with the evaluated formulas are available in [Basb]. The network of automata is composed of a `RunnableOrchestration` automaton modelling the orchestrator, and each service is modeled by two automata: `RunnableOrchestratedContract` and `SocketTimeout`. Figure 7 shows the system declarations of the model. In particular, there is a single instance of `RunnableOrchestration` whilst there are `N` instance of `RunnableOrchestratedContract` and `SocketTimeout`. Indeed, both `RunnableOrchestratedContract` and `SocketTimeout`

[1] `https://github.com/contractautomataproject/CARE/releases/tag/v1.0.1`



```
89 @Override
90 public void run() {
91         try    (ObjectInputStream oin = new ObjectInputStream(sock
92                 ObjectOutputStream oout = new ObjectOutput
93         {
94                 oout.flush();
95                 System.out.println("Connection with service starte
96                 while (true) {
97                         //receive message from orchestrator
98                         String action = (String) oin.readObject();
```

FIGURE 6. A fragment of the `RunnableOrchestratedContract` automaton equipped with traceability information (left) and a fragment of the class `RunnableOrchestratedContract.java` (right). The transition from state `Ready` is traced back to line 98 of the source code of the respective class.


model_statistical.xml - UPPAAL
File Edit View Tools Options Help
Editor Symbolic Simulator Concrete Simulator Verifier
Project
Declarations
RunnableOrchestration
RunnableOrchestratedContract
SocketTimeout
System declarations
UPPAAL 5.0.0
1:1

```
ror = RunnableOrchestration();

system ror,RunnableOrchestratedContract,SocketTimeout;
```

FIGURE 7. A snapshot of the UPPAAL model showing the system declarations

templates take as parameter `id_t`, which is a special global declaration of UPPAAL (namely, `typedef int[0,N-1] id_t`) to instruct the model to produce `N` replicas of these template, where as stated above `N` is also a global constant of the model.

Figure 9 displays the template automaton for the `RunnableOrchestration` (i.e., the orchestrator), while the template automaton for the `RunnableOrchestratedContract` (i.e., the service) has as parameter the `id` of the service (of type `id_t`) and is depicted in Figure 10. Recall that, for readability, the `Timeout` location is not displayed (see Section 4.1). The behaviour according to the given configuration of action and choice is modelled inside each automaton. We anticipate that we will use variants of these two automata in Section 6, in order to perform different analyses. In particular, in Figure 9 the configurations of choice and action (for both the services and the orchestrator) are instantiated non-deterministically by

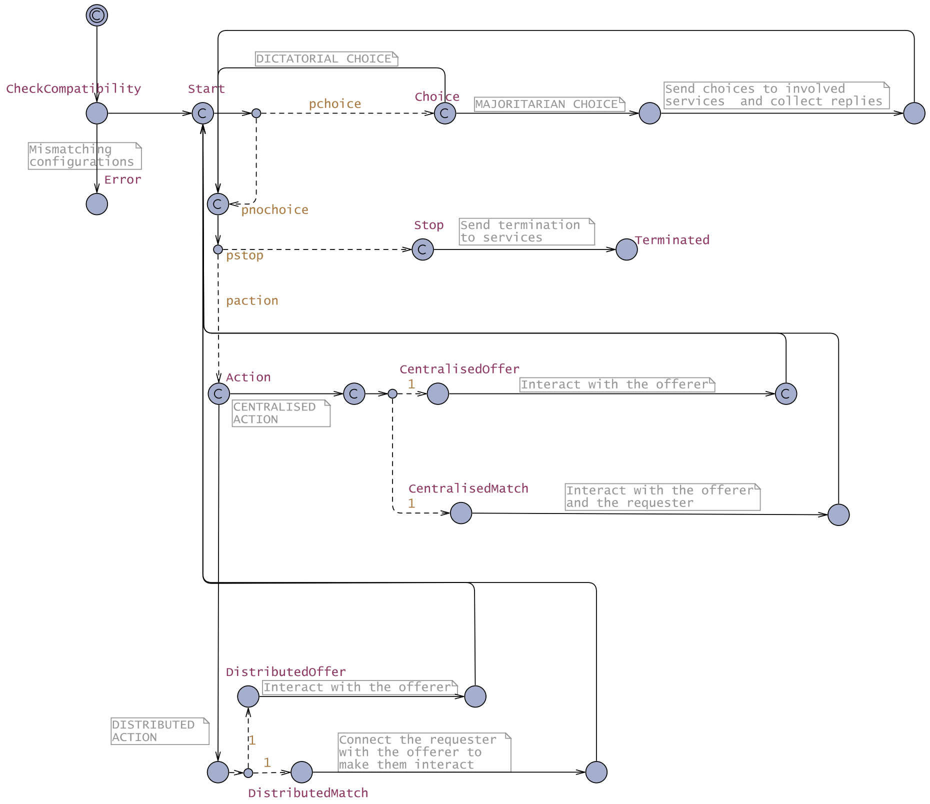

CheckCompatibility
Start
DICTATORIAL CHOICE
pchoice
Choice
MAJORITARIAN CHOICE
Send choices to involved services and collect replies
Mismatching configurations
Error
pnochoice
Stop
Send termination to services
Terminated
pstop
paction
Action
CentralisedOffer
1
Interact with the offerer
CENTRALISED ACTION
CentralisedMatch
1
Interact with the offerer and the requester
DistributedOffer
Interact with the offerer
DISTRIBUTED ACTION
1
1
Connect the requester with the offerer to make them interact
DistributedMatch

FIGURE 8. A simplified schema of the `RunnableOrchestration` automaton

the transition outgoing the initial state (see Section 5.1). Note that, in this case, the variables `choice` and `action` are not parameters of the templates. Another version of the model is also available, where the configuration options, instead of being selected non-deterministically, are fixed as parameters of the templates. Note that in UPPAAL templates cannot be primitively instantiated non-deterministically. The repository [Basb] contains the model equipped with traceability information, i.e., each transition of the model has a comment describing the class and the lines of the source code that correspond to the specific behaviour of the transition. The models used for generating tests, together with the generated tests, are also available in [Basb].

The UPPAAL model is composed of a list of global declarations, the two automata (with their local declarations) and the system set-up (i.e., the instantiation of the automata). Global declarations include the number of services `N`, the size of the buffers, the timeout threshold, the rates of the exponential distributions, two variables `action` and `choice` storing the corresponding configuration for all automata, and the communication buffers. Constants are also defined globally and their identifiers are in capital letters. Some names of locations are displayed in the automata for readability. Labels of transitions contain guards and effects and in a few cases also probabilistic selections.

5.1. **Description of the automata.** Before discussing the two automata, we describe a simplified template, depicted in Figure 8.

5.1.1. *Simplified template.* Note that Figure 8 is not a Uppaal automaton. The initial state is depicted with a double circle. The first activity consists in checking whether all services and the orchestrator have the same configuration (`CheckCompability`). If this check is successful, the orchestration starts. From the location `Start`, the orchestrator internally decides (based on the orchestration) whether to perform a choice or an action. The choice of termination is modelled as a third alternative. Probability weights are used to model the probability of performing a choice, an action or to terminate (`pchoice`, `pstop`, `paction`). As stated previously, after a choice is completed, the orchestrator moves to a state where only an action or termination can be chosen. After an action is completed, the orchestrator returns to the `Start` location. The choice can be either dictatorial or majoritarian, according to the assigned configuration. In the majoritarian choice, the orchestrator interacts with the services to collect their choice. Similarly, the configuration of action can be either centralised or distributed. In both cases, an offer label or a match label will be executed. As stated in Section 3, in a distributed match the orchestrator makes the two services aware of each other so that they can interact and execute the match. In case of termination, the orchestrator sends the termination message to all services and terminates.

5.1.2. *Detailed description.* We now briefly describe Figure 9 and Figure 10. In the first transition of the orchestrator, one of four options encoding the combinations of choice and action is selected non-deterministically. The function `initialize(conf)` instantiates accordingly the `choice` and `action` global variables such that all have the same configuration, and also initialises the communication buffers.

From the location `CheckCompatibility` of the orchestrator a loop is executed to communicate with all services to check if all have the same configuration. The orchestrator sends the message `ORC_CHECK` and its configuration (action and choice). If a service has the same configuration an `ACK` is sent, or an `ERROR` message otherwise. In case of no errors, the orchestration starts. As stated above, from the location `Start` the orchestrator internally decides whether to perform a choice, an action or terminate.

In the case of termination, the orchestrator sends to all services an `ORC_STOP` message and the orchestration terminates.

Choice. In the case of a choice, firstly all services receive the message `ORC_CHOICE`. If the choice is dictatorial, the orchestrator decides autonomously, and no further interactions are necessary. Otherwise, in the case of a majoritarian choice, the services involved in the choice are selected non-deterministically, and a message `ORC_CHOICE` is sent to them. Concretely, the involved services are those who perform an action in one of the outgoing transitions from the current state of the orchestration. The services that are not involved will receive a `SKIP` message. The involved services then receive from the orchestrator the available choices (i.e., the forward star of the current state). These concrete choices are abstracted by the constant `CHOICES`. Each involved service now replies with its choice, abstracted as a message `SERVICE_CHOICE`. After all involved services have voted, the orchestrator will decide accordingly.

Action. In case of an action, the orchestration behaves differently depending on whether the configuration is centralised or distributed. In both cases, a probabilistic choice is made on whether the transition is an offer or a match. After that, the orchestrator picks non-deterministically an offerer and a requester (only in the case of a match transition) with

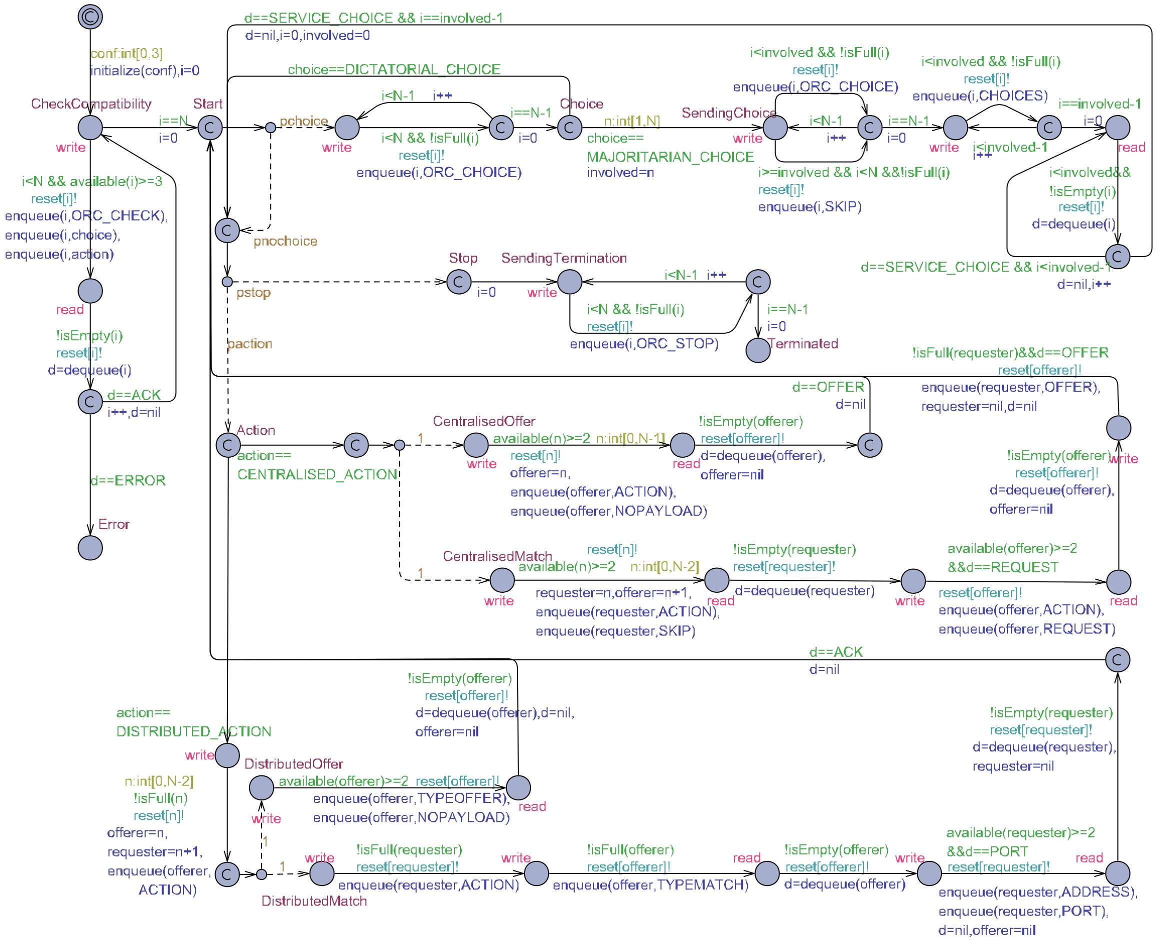


Figure 9. The `RunnableOrchestration` Uppaal template

consecutive identifiers (recall that their IDs are immaterial). In the case of a centralised offer action, the offerer receives from the orchestrator the invocation of the action, abstracted by the constant `ACTION`. At this point the offerer is still not aware of whether it is involved in an offer or a match transition. The reception of the `NOPAYLOAD` message (i.e., there is no payload from the requester) disambiguates the offerer who replies with a payload abstracted here by the constant `OFFER`.

In a centralised match the requester receives the action invocation, abstracted by the constant `ACTION`, and a `SKIP` command (i.e., the offer has not yet been generated). Note that in the implementation the service is informed of being an offerer or a requester upon receiving the action. Since the concrete action is abstracted, in the model this disambiguation occurs when receiving either the message `SKIP` (i.e., a requester) or the messages `NOPAYLOAD` (i.e., an offerer in an offer transition) or `REQUEST` (i.e., an offerer in a match transition). The requester replies with the message `REQUEST` (the concrete payload is abstracted away) that is forwarded by the orchestrator to the offerer, who receives in sequence the messages `ACTION` and `REQUEST`. Similarly to the previous case, the offerer sends to the orchestrator its offer payload (now based on the payload of the requester), abstracted again by the constant `OFFER`.

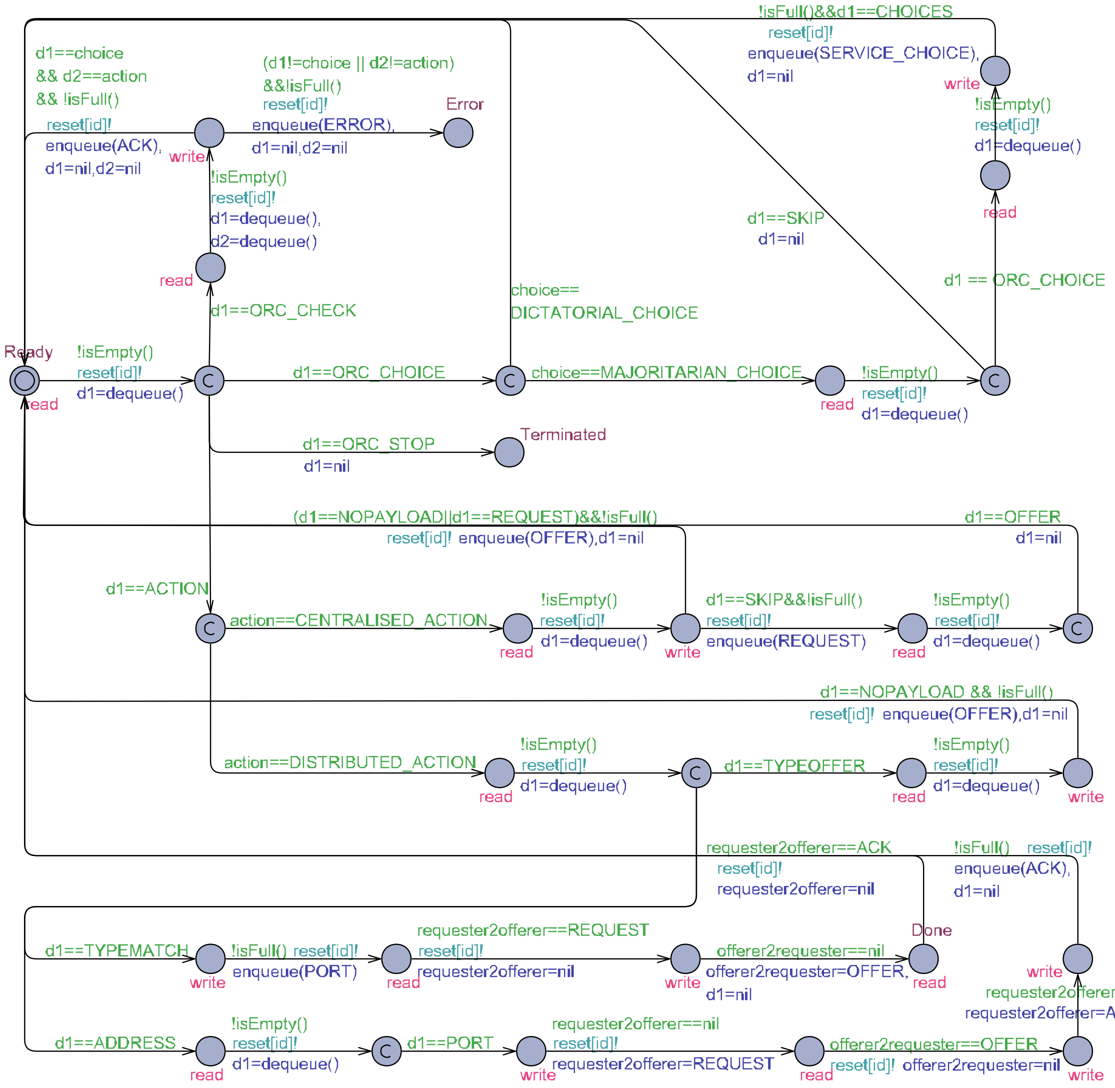


Figure 10. The `RunnableOrchestratedContract` Uppaal template

Concerning the distributed configuration, in case of an offer or match first the offerer receives the `ACTION` command from the orchestrator. In case of an offer, a `TYPEOFFER` message is received by the offerer followed by a `NOPAYLOAD` message. The offerer replies with an `OFFER` message. In case of a distributed match, the `ACTION` message is also sent to the requester and the `TYPEMATCH` message is sent to the offerer. The offerer opens a fresh port and communicates this port (abstracted by the constant `PORT`) to the orchestrator. The offerer waits for a connection from the requester. The orchestrator communicates to the requester (who was waiting after receiving the `ACTION` command) the address (constant `ADDRESS`) and `PORT` of the offerer. Now the requester and the offerer can interact without the orchestrator. The interactions between any two services in the orchestration all use two (one-position) buffers, implemented by the variables `requester2offerer` and `offerer2requester`. Indeed, it is never the case that some service interferes in a match in which it is not involved, and using different buffers for each pair of services would unnecessarily increase the state space.

Initially, the requester sends its REQUEST payload to the offerer. The offerer replies to the requester with its payload OFFER (based on the request) and reaches location Done. The requester receives the payload and sends an ACK to the offerer and the orchestrator (who both terminate the execution of the match transition and returns to the Start and Ready state, respectively).

6. Analysis

We now describe the analyses we performed on the model. The modelling activity led to some issues in both the implementation and the model, which were all fixed (we remark that an already existing application was modelled). Other formal checks on the model were performed to ensure the properties described in this section (e.g., absence of deadlocks, absence of orphan messages). We have used Uppaal version 4.1.26-1, released in February 2022. Note that a more recent version of Uppaal is available, which integrates the previously separate tool Uppaal Stratego. We opted to continue using the version we initially started working with, since the capabilities of Uppaal Stratego are not used in this paper.

6.1. **Validation through modelling.** The first validation was performed during the modelling phase. Indeed, formal modelling requires an accurate analysis and review of the source code. Interactive simulation is used during modelling to animate and analyse the portion of the model designed so far, similarly to how the source code is debugged interactively (e.g., by choosing the next step). We note that in many model-based engineering tools, behavioural models (e.g., state charts) are validated by only relying on graphical interactive simulations [BMF23]. Issues can be detected during this phase in particular if the source code has not been thoroughly tested, as was the case for CARE.

We report an issue detected in the source code during the modelling of the automaton in Figure 9, and more specifically during the modelling of the loop in which the orchestrator is reading the SERVICE_CHOICE messages sent by the involved services. In the implementation, the orchestrator was waiting for a choice from all services, also comprehending those who received a SKIP message. This means that a deadlock could occur in case there was a service not involved in a choice. Initially, this issue was undetected because the tests had all services involved in all choices. It was fixed thanks to the activity described in this paper.

On a side note, thorough testing is generally more time-consuming than designing a formal model similar to the one in this paper. This is a further benefit derived from the usage of formal methods. For example, the library implementing contract automata operations (CATLib) [BtB22] has been tested up to 100% coverage of all lines and branches, also using mutation testing [BtB22]. In CATLib, the lines of code of the tests are more than three times those of its source code [BtB22]. A similar effort for CARE is more demanding than the one needed for designing the models in Figures 9 and 10. Uppaal has been used to automatically generate tests from the formal model.

6.2. **Formal Verification.** We discuss the formal verification performed on the model, encompassing both exhaustive and statistical model checking. In the initial phase, we employ a model variant that guarantees consistent configurations between the services and the orchestrator. In this variant, the selection of the choice and action configuration is performed non-deterministically by the first transition of RunnableOrchestration (variable conf, see Figure 9), and it is always consistent. Subsequently, we move to another variant

Task	Property	Parameters' tuning	Interval	Confidence
Buffer size	Probability of filling a buffer	**queueSize** set to 5	$[0, 0.00996915]$	0.95
Timeout	Probability of timeout	**timeout** set to 15	$[0, 0.00996915]$	0.95
Delays	Probability of termination	**write** and **read** set to 5	$[0.990031, 1]$	0.95
Orphan messages	Probability of terminating with non-empty buffers	-	$[0, 0.00999882]$	0.995
Interference	Probability of a service interfering on a match between two other services	-	$[0, 0.00999882]$	0.995

Table 1. A summary of the properties analysed with statistical model checking

Task	Property	Conf.	Time	Memory	State space
Termination	If orchestrator terminates then all terminate or a timeout occurs	c1	3 m.	1.5 GB	52 M st.
		c2	30 m.	12 GB	426 M st.
Deadlocks	Absence of deadlocks (no enabled transitions, timeout, or termination)	c1	3 m.	1.5 GB	25 M st.
		c2	40 m.	13 GB	189 M st.
Orphan Messages	Upon termination, no messages left in buffers	c1	2 m.	1.5 GB	25 M st.
		c2	27 m.	12.5 GB	189 M st.
Dummy Execution	No dummy execution where no interaction occurs	c1	3 sec.	1.4 GB	24 st.
		c2	24 s.	10 GB	28 st.
Compatibility Check	Mismatch leads to `Error` location	Special config.	1 ms.	49 MB	79 st.

Table 2. A summary of the properties analysed with exhaustive model checking that are satisfied by the model

where the configuration of `choice` and `action` are treated as parameters. In this case, we formally prove that when configurations do not match, an error state is reached.

Performances. Statistical model checking has been used to scale to larger systems, and the verification has been performed in a few seconds on a standard laptop. Conversely, exhaustive model checking necessitated more resources and has been limited to smaller parameter setup generating hundreds of millions of states (see below). In this case, the verification has been performed on a machine with Intel(R) Core(TM) i9-9900K CPU @ 3.60 GHz equipped with 32 GB of RAM. Uppaal has been configured for maximising the state space optimisation and reusing the generated state space. Logs of the experiments are available in [Basb]. The summaries of the analysis performed with statistical (resp., exhaustive) model checking are in Table 1 (resp., Table 2).

6.3. Parameters Tuning.

The verification process involves employing specific parameter setup within the model. This encompasses various aspects, such as setting the delays in reading and writing, setting socket timeout thresholds, adjusting buffer sizes, assigning probability weights, and determining the number of services involved (i.e., the instantiations of the `RunnableOrchestratedContract` template). For deriving the desired set-up of

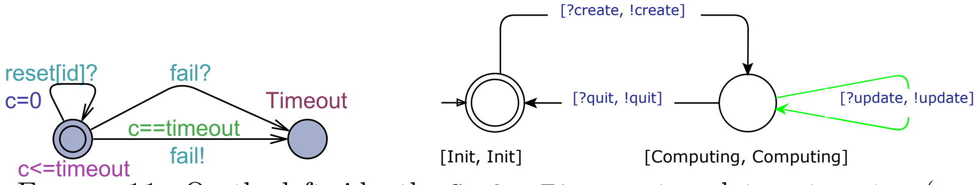


Figure 11. On the left side, the `SocketTimeout` template automaton (one such automaton is istantiated for each service). On the right side, the orchestration automaton taken from the composition service example in [BtB23a]

parameters, we employ statistical model checking. If not stated otherwise, the parameters α and ε of the statistical model checker are set to 0.05 and 0.005, respectively (see Section 3).

In this section, when reporting the evaluation of a probabilistic formula, instead of reporting an interval containing zero (e.g., $[0, 0.00996915]$), sometimes, for conciseness, we will say that the property has a probability near zero. To be more precise, we mean that all runs do not satisfy the property, thus the probability lies within the interval $[0, 0 + \varepsilon]$ with probability $1 - \alpha$. We remind readers that, unlike probabilistic model checking, in statistical model checking, the unknown probability value is estimated to lie within an interval with a given confidence.

It is crucial to note that we do not employ statistical model checking to determine values (such as buffers size) for use in the concrete implementation. Instead, these quantified values are used solely within the model. For instance, in the actual implementation, the size of Java TCP/IP socket buffers is fixed (more below). Our objective is to employ parameter setup that ensure realistic modelling and improve the performances of the exhaustive model checking. Realistic modelling entails accurately representing the behaviour of the real system, by reducing the probability of filling the buffers, timeouts, or excessive communication delays. Failure to maintain these conditions could potentially invalidate the results of the formal verification. Improving performances, on the other hand, entails reducing the state space of the model.

Probabilities weights. The values of the *probabilities weights* `pstop`, `pchoice`, `pnochoice`, and `paction` (see Figure 9) can be tuned based on average values extracted from the orchestrations subject of the analysis. Indeed, as stated in Section 4, the underlying orchestration automaton is abstracted away. Note that, e.g., in location `Start` the probability of performing a choice is $\frac{\texttt{pchoice}}{\texttt{pchoice+pnochoice}}$ (`pchoice` is the probability weight of performing a choice and `pnochoice` is the probability weight of not performing a choice). For example, the orchestration in Figure 11 (right) can be modelled by tuning the probabilities to `pstop=25`, `pchoice=1`, `pnochoice=0`, and `paction=75`.

For readability, in the reminder of this section we will use in the formulae the abbreviations `ror` for the instantiation of the `RunnableOrchestration` template (i.e., the orchestrator), and `ROC(i)` for the i-th instantiation of the `RunnableOrchestratedContract` template (i.e., a service).

Buffer size. Next, we address the *buffer size* (denoted as variable `queueSize`). It is important to note that the buffers in the model are represented by global arrays utilized by the automata for enqueuing and dequeuing values. These global arrays serve as models of the actual buffers found in Java TCP/IP sockets, where the default size is typically 8 KB. Our objective is to prevent unnecessary growth in the model's state space while ensuring a low probability of the buffers filling up, similar to the behaviour observed in the real application. The formula:

```
E[<=500; 10000](max:  sum(i:int[0,N-1]) (sum(j:int[0,queueSize-1])(orc2services[i][j]!=nil)))
```

computes, using 10000 simulations of 500 time units, the expected maximum number of non-empty positions of the buffers used by the orchestrator to send messages to the services (called `orc2services`). For all statistical evaluations, we set the number of services to 10. With buffer size set to 10, this formula evaluates to 4.5 ± 0.019. This indicates that, on average, the maximum number of utilized buffer positions is between 4 and 5. Consequently, in order to reduce the state space, it is safe to decrease the value of `queueSize` to less than 10 for the exhaustive model checking phase. This observation is further supported by the following formula:

```
Pr[<=500] (<>(exists (i:id_t) ror.isFull(i)))
```

which measures the probability that, within 500 time units, one of the `orc2services[i]` arrays becomes full (`ror` is the orchestrator automaton). In three separate experiments where `queueSize` was varied between 3, 4, and 5, this formula yielded the respective evaluation intervals of $[0.990031, 1]$, $[0.00632077, 0.0163207]$ (with $\alpha = 0.005$), and $[0, 0.00996915]$. Based on these results, if not stated otherwise, `queueSize` is set to 5 for the subsequent experiments.

Delays and timeout. Next, we consider the real-time behaviour of the model and focus on determining appropriate values for the *rates* `write` and `read`, which represent the delays in writing to and reading from a buffer, respectively. The average message delay in Java TCP/IP sockets is affected by multiple factors, such as network conditions and server load. Delays can be sampled using tools like `tcpdump`, and the rate can be estimated by calculating the inverse of the sample mean. Additionally, we consider the variable `timeout`, which represents the timeout threshold. Our aim is to achieve three objectives. First, we strive to maintain a low probability of encountering timeouts. Second, we seek to ensure a high probability of terminating within a specific timeframe, which we have set to be 500 time units, corresponding to the duration used in our experimental setup. Third, we aim to keep the timeout threshold at a lower value in order to decrease the state space and facilitate model checking.

The formula:

```
Pr[<=500] (<> ror.Timeout)
```

measures the likelihood of one or more service sockets experiencing a timeout (recall that in this case a failure signal is broadcasted and all automata enter their respective `Timeout` location). The probability of all services and the orchestrator successfully concluding their operations is evaluated with the formula:

```
Pr[<=500](<>ror.Terminated&&(forall (i:id_t) ROC(i).Terminated))
```

(`ROC` is used as an abbreviation of `RunnableOrchestratedContract`). In different experiments where we varied the values of the pair (rate,timeout), specifically in $(5, 14)$, $(4, 15)$, $(5, 15)$, respectively, the first formula (timeout probability) yielded the respective evaluation intervals $[3.06006e^{-06}, 0.00999494]$ (with $\alpha = 0.005$), $[0.0433422, 0.053341]$, and $[0, 0.00996915]$, while the second formula (probability of termination) yielded $[0.990005, 0.999997]$ (with $\alpha = 0.005$), $[0.948625, 0.958625]$ (with $\alpha = 0.005$), and $[0.990031, 1]$. Therefore, to fulfill the aforementioned three objectives, we have set the values of `write` and `read` to 5, while the value of `timeout` has been set to 15 for the subsequent experiments. Indeed, in the experiments we just mentioned, the value (5,15) for the pair (rate,timeout) yields the best values for both the probability of experiencing timeout and the probability of reaching termination.

Concerning the *instances* of the templates, in all experiments there is one instance of the orchestrator template and either 4 or 5 instances of the service template. For the exhaustive model checking phase, we used two small parameter setup of (number of services, buffers size). The first is c1=(4,5), the second is c2=(5,3). In fact, the parameter setup (5,4) remained inconclusive in the experiments due to the need for generating billions of states and the inadequate memory capacity of the utilized machine. The verification of larger parameter setup necessitates either relying exclusively on statistical model checking or employing more powerful machines.

6.4. **Verification.**

Once the model's parameter setup is determined, our next step involves verifying additional formal properties.

Termination. We have already assessed the probability of non-termination and found it to be nearly zero. However, it may be worthwhile to conduct an exhaustive verification specifically for this property. The property that in all executions eventually all services and the orchestrator terminate is not valid. Indeed, as described in Section 4, the orchestration contract automaton is abstracted away and at each iteration, a choice is performed to decide whether to terminate or not. Hence, there exists an execution in which the orchestration never terminates. A milder property does hold:

```
ror.Stop-->((ror.Terminated&&(forall(i:id_t)ROC(i).Terminated))||
        (exists(i:id_t)SocketTimeout(i).Timeout))
```

i.e., if the orchestrator starts to terminate then eventually all services and the orchestrator terminate. The formula `p-->q` is a shortcut for `A[](p imply A<>q)`. Thus, this formula states that for all executions and for all states either the orchestrator is not in the location `ror.Stop` or all executions passing through that location will eventually lead to a state where all services and the orchestrator have terminated or a timeout failure is experienced. The formula holds in both parameter setup c1 and c2. The first (resp. second) parameter setup required roughly 3 (resp. 30) minutes, 1.5 (resp. 12) Giga of memory, and explored 52 (resp. 426) million states.

Absence of deadlocks. The likelihood of non-termination being very low also implies an almost negligible probability of encountering deadlocks. While it may be of interest to exhaustively prove the absence of deadlocks, the previous formula is insufficient for this purpose. Hence, to prove that there is no deadlock, we perform exhaustive model checking of the formula:

```
A[](not deadlock || (exists(i:id_t) SocketTimeout(i).Timeout) ||
    (ror.Terminated && (forall (i:id_t) ROC(i).Terminated)))
```

The formula states that for all executions and for all states of the composed system, either there is always at least one enabled transition or either a timeout failure has been experienced or all services and the orchestrator are in the `Terminated` location. In the formula, `not deadlock` is a special predicate provided by Uppaal. As expected, this property is satisfied in both parameter setup. The first (resp. second) parameter setup required roughly 3 (resp. 40) minutes, 1.5 (resp. 13) Giga of memory, and explored 25 (resp. 189) million states. This also proves that in a correct configuration (of action and choice) the `Error` location is never reached, because this would result in a deadlock and the above property would not be satisfied. This property is also satisfied when the size of the buffers is 3. However, if we further reduce the size, then a deadlock occurs. This is because from location `CheckCompatibility` the orchestrator requires to insert three messages in the buffer of the receiver in one step. By dividing these three send operations in three non-committed transitions it is possible to further reduce the buffer size.

We report a modelling issue detected during model checking the above formula. In an earlier version, it was assumed that the socket mode was non-blocking (i.e., sending to a recipient with a full buffer would cause an error). This was modelled by making committed (`C`) all source states of transitions with sending operations. In this way, if the buffer of the receiver would not have enough free space then an attempt to send to the receiver would cause a deadlock. In fact, in this earlier version of the model the above formula (absence of deadlocks) was not satisfied, for any possible size of the buffer. The counterexample trace of the model checker helped to understand and eventually fix this issue. Basically, in the model configured with a majoritarian choice there exists a loop in which the orchestrator alternates choices and actions, and enqueues a sequence of `ORC_CHOICE` and `SKIP` messages to a service that never consumes them and is never involved in neither choices nor actions, thus eventually filling its buffer and deadlocking. A similar issue also exists in the case of a dictatorial choice.

Note that this kind of problems are hard to detect without model checking. Indeed, the counterexample trace was generated automatically, and the counterexample trace is composed of hundreds of steps. Without model checking this would require to manually execute each step of this trace, and the longer the trace the less chance to discover it. After we detected this issue, we analysed the underlying Java TCP/IP socket semantics [Jav] and fixed the model as described in Section 4 (i.e., by modelling these sockets with default blocking mode). Another fix could be to include an ack after the reception of a `SKIP` message (this would however require to modify also the implementation).

Absence of orphan messages. We now prove that upon termination of an orchestration no messages are left in any buffer, i.e., all messages are consumed. To expedite the verification process, we begin by conducting statistical model checking of the following property:

```
Pr[<=500](<>!allEmpty()&&ror.Terminated&&(forall(i:id_t)ROC(i).Terminated))
```

this property quantifies the probability of termination with at least one message remaining in any buffer. To verify whether all buffers are empty, we utilize the predicate `allEmpty()`. As expected, the probability is found to be nearly zero. We proceed with an exhaustive verification by employing the property:

```
A[]((ror.Terminated && (forall (i:id_t) ROC(i).Terminated)) imply allEmpty())
```

the above formula can be read as follows: in all states of all executions either all buffers are empty or someone has not terminated yet. The above property is valid in both parameter setup, as expected. The first (resp. second) parameter setup required roughly 2 (resp. 27) minutes, 1.5 (resp. 12.5) Giga of memory, and explored 25 (resp. 189) million states. It is also possible to verify that there is no dummy execution in which the services and the orchestrator never interact (and thus all buffers are trivially empty). This can be verified with the formula:

```
E[] (allEmpty() && !ror.Timeout)
```

that, as stated above, checks if there exists an execution where all states have empty buffers (excluding the dummy execution scenario in which the timeout occurs at the beginning). As expected, this property is not valid in the model, for both parameter setup. The first (resp. second) parameter setup required roughly 3 (resp. 24) seconds, 1.4 (resp. 10) Giga of memory, and explored 24 (resp. 28) states.

No interference. When discussing the distributed match action in Section 5, we stated that it is never the case that some service interferes in a match in which it is not involved. This guarantees that it is safe to use two one-position buffers for all communications between any two services involved in a match. To verify this, we perform statistical model checking of the following formula:

```
Pr[<=500](<>exists(i:id_t)(i<N-1&&(ROC(i).d1==TYPEMATCH||ROC(i).d1==ADDRESS||ROC(i).d1==PORT))
           &&((ROC(i+1).d1==TYPEMATCH||ROC(i+1).d1==ADDRESS||ROC(i+1).d1==PORT))&&
 (exists(j:id_t)(j!=i&&j!=i+1&&(ROC(j).d1==TYPEMATCH||ROC(j).d1==ADDRESS||ROC(j).d1==PORT))))
```

the formula measures the probability of reaching a state where one service (index `j`) is interfering on a match between two other services (indexes `i` and `i+1`). We recall that in the model two matching services have consecutive indexes. We detect a service to be involved in a distributed match when its temporary variable `d1` has one of the three values (`TYPEMATCH`, `ADDRESS`, `PORT`). The probability is found to be nearly zero.

We also include in the repository [Basb] a version of the model where each pair of services has its own buffers. All results in this section also hold in that model.

Compatibility check. Next, we formally prove that if some service is not matching the configuration of the orchestrator, then the orchestration will not start and an `Error` location will always eventually be reached. Indeed, the possibility of mismatching configurations is allowed in the real system. However, in the model discussed in Section 5 this scenario is not possible because, by construction, all services and the orchestrator share the same configuration. The configuration is selected non-deterministically. In this way, for each formula being verified all possible configurations are checked automatically.

Only for this check the model has been slightly modified by adding two parameters `action` and `choice` to the templates and by updating accordingly the model. For this verification, the set-up of the system is of an orchestrator `ror` and three services `alice`, `bob` and `carl` and the size of each buffer is 4:

```
ror = RunnableOrchestration(MAJORITARIAN_CHOICE,DISTRIBUTED_ACTION);
alice = RunnableOrchestratedContract(0,MAJORITARIAN_CHOICE,DISTRIBUTED_ACTION);
bob = RunnableOrchestratedContract(1,MAJORITARIAN_CHOICE,DISTRIBUTED_ACTION);
carl = RunnableOrchestratedContract(2,DICTATORIAL_CHOICE,DISTRIBUTED_ACTION);
ast = SocketTimeout(0);bst = SocketTimeout(1);cst = SocketTimeout(2);
system ror,alice,bob,carl,ast,bst,cst;
```

Note that, differently from Figure 7, the configurations of choice and action are now parameters assigned to each automaton. This allows us to assign a mismatching configuration to verify that the `Error` location will be reached. Indeed, in the above set-up `carl` has a different configuration. We use the formula:

```
A<>((ror.Error && carl.Error)||ror.Timeout)
```

stating that in all executions eventually the orchestrator and the service with a wrong configuration reach an `Error` location or a timeout is experienced. Alternatively, the formula:

```
A[](!ror.Start)
```

states that in all executions the `Start` location of the orchestrator is never traversed (i.e., the orchestration never starts). Both properties are satisfied in this setup, thus verifying the correctness of the compatibility check. The first (resp. second) formula required visiting 19 (resp. 79) states. Both formulae used roughly 48 Megabytes of memory and a few milliseconds of CPU. We have proved that in case of mismatching configurations, the orchestration will not start.

7. Testing

The model-based testing functionality of Uppaal has been employed to generate tests that demonstrate the model's adherence to the actual implementation.

The abstract test cases generated by Uppaal have been transformed into concrete JUnit tests for the source code. Uppaal provides various methods for test generation (see Section 3). We utilized the generation from reachability queries (i.e., of the form `E<>`). In our models, these queries encode specific simulation traces that are relevant to the specific orchestration employed in the tests and are designed to cover all transitions of the model. The specific simulation traces also guide the generation of the concrete test cases from the abstract test cases. Uppaal provides functionalities for offline test generation, such as achieving transition coverage or generating traces that reach a specified final state (reachability queries). However, Uppaal does not primitively offer a mechanism within their query language to explicitly enforce a sequence of intermediate states or events that a test trace must follow. To achieve this, two global variables, namely `steps[]` and `step`, are introduced in the models to encode the desired orchestration in the query. Whenever a specific transition is executed, the variable `steps[step]` is updated with a value encoding the transition being executed, and the counter `step` is incremented. For example (see Figure 12), whenever the transition reaching state `Terminated` is executed, the value `ORC_STOP` is assigned to `steps[step]`. As another example, whenever a centralised match transition is executed by the orchestration, the value `CENTRALISED_MATCH` is assigned to `steps[step]`, encoding the execution of the request action in the match (see Section 5). The length of the array `steps[]` is equal to the desired number of transitions that the trace must pass through, and the variable `step` is a pointer to the current position in the array. As a small example, let `alice` be an instatiation of the automaton in Figure 12. The query `E<>(alice.steps[0]==ORC_CHECK)` encodes a trace where the check operation is executed first. In Section 7.3, we will provide a complete example showing how these additional variables are used in a query.

7.1. **Testing the Orchestrator.**

Different variants of the model have been developed, each one equipped with specific test code to test one of the components. One version of the model is equipped with test code for testing the orchestrator (i.e., `RunnableOrchestration`). In this case, during concrete testing, the orchestrator will be the actual system, whilst the tester will mimic the services. Therefore, the test code will be generated from the services model. In this version, the automaton template `RunnableOrchestratedContract` contains test code in its transitions, whilst the automaton template `RunnableOrchestration` does not contain test code. However, useful comments are inserted as test code in some transitions of `RunnableOrchestration` to guide the generation of concrete tests from abstract tests. This mainly involves printing the actual value during simulation of variables `offerer`, `requester` and `i` used by the `RunnableOrchestration` as indices to identify the services the orchestrator is interacting with. In the generated tests, no timeout is experienced, therefore the `SocketTimeoutAutomaton` is a dummy automaton receiving signals from `RunnableOrchestratedContract` and doing nothing. Indeed, a timeout in the concrete system could be experienced due to a worsening of the physical network, which is outside the scope of the model-based testing performed in this paper to ascertain the adequacy.

All transitions of RunnableOrchestratedContract containing a dequeue() operation in the effects will also contain as test code the instruction msg = (String) oin.readObject(). Basically, the abstract operation of reading a message from the queue is concretised as a read operation from the variable oin of class ObjectInputStream of the Java TCP/IP socket of the service. Similarly, transitions containing an enqueue effect will contain as test code instructions for writing into a socket. The information on the actual payload to be sent by the concrete services is abstracted away in the Uppaal model. For example, consider the transition reaching state Ready of RunnableOrchestratedContract with effect enqueue(SERVICE_CHOICE) (see Figure 12, top right corner). In this case, SERVICE_CHOICE is a constant abstracting from the choice that the service will be expressing, which will be based at runtime on the transitions outgoing the current state of the contract automaton of the service, abstracted in the Uppaal model. Accordingly, the test code generated from that transition is oout.writeObject(choice);oout.flush();. In this case, choice is a placeholder that needs to be replaced in the concrete test with the actual choice made at that point. Furthermore, oout is the ObjectOutputStream of the service's TCP/IP socket, on which a write operation is performed.

Finally, some read operations and other transitions with guards will include assertions in their test code. These assertions will be concretised in the concrete tests by adding values that were abstracted away in the model and will be used by the JUnit tests to verify whether the tests are successful or not. For example, the transition of RunnableOrchestratedContract reaching state Terminated will contain in its test code the instruction assertEquals(msg,RunnableOrchestration.stop_msg);. In this case, it is tested whether the last received message is the constant identifying the termination of the current orchestration.

7.2. **Testing the Services.** For testing the RunnableOrchestratedContract services another version of the model is used. The orchestrator will be mocked by the test, whilst the services will be the actual classes of CARE. The abstract test code will be generated through the automaton RunnableOrchestration. The Uppaal automaton RunnableOrchestration will also be decorated with the effects on the array steps[step], similar to Figure 12. The generation of test code is similar to the case of testing the orchestrator, discussed previously. The main difference is that the orchestrator manages a list of input/output streams, one for each service. For example, when sending the message ORC_CHOICE to the services (see Figure 9), the transition with the effect enqueue(i,ORC_CHOICE) generates the test code oout.get($(ror.i)).writeObject(RunnableOrchestration.choice_msg); oout.get($(ror.i)).flush();, where the object oout is a list of ObjectOutputStream, one for each service. Similarly, the dequeue(i) effect generates the test code response = oin.get($(ror.i)).readObject();, where oin is a list of ObjectInputStream. Finally, similar to the case of testing the orchestrator, abstract values need to be instantiated in the concrete tests, and assertions are included in the tests to check whether the received messages are as expected.

For testing the interactions between two services in a distributed match, there are two further versions of the model. In these versions, the tester is represented by the orchestrator and one of the two interactive services, either the requester or offerer in the match.

In the following, we will provide an example on how the abstract test code is generated and transformed into a concrete test. We will focus on a specific version of the model, i.e., the one used for testing the orchestrator, and a specific set-up.

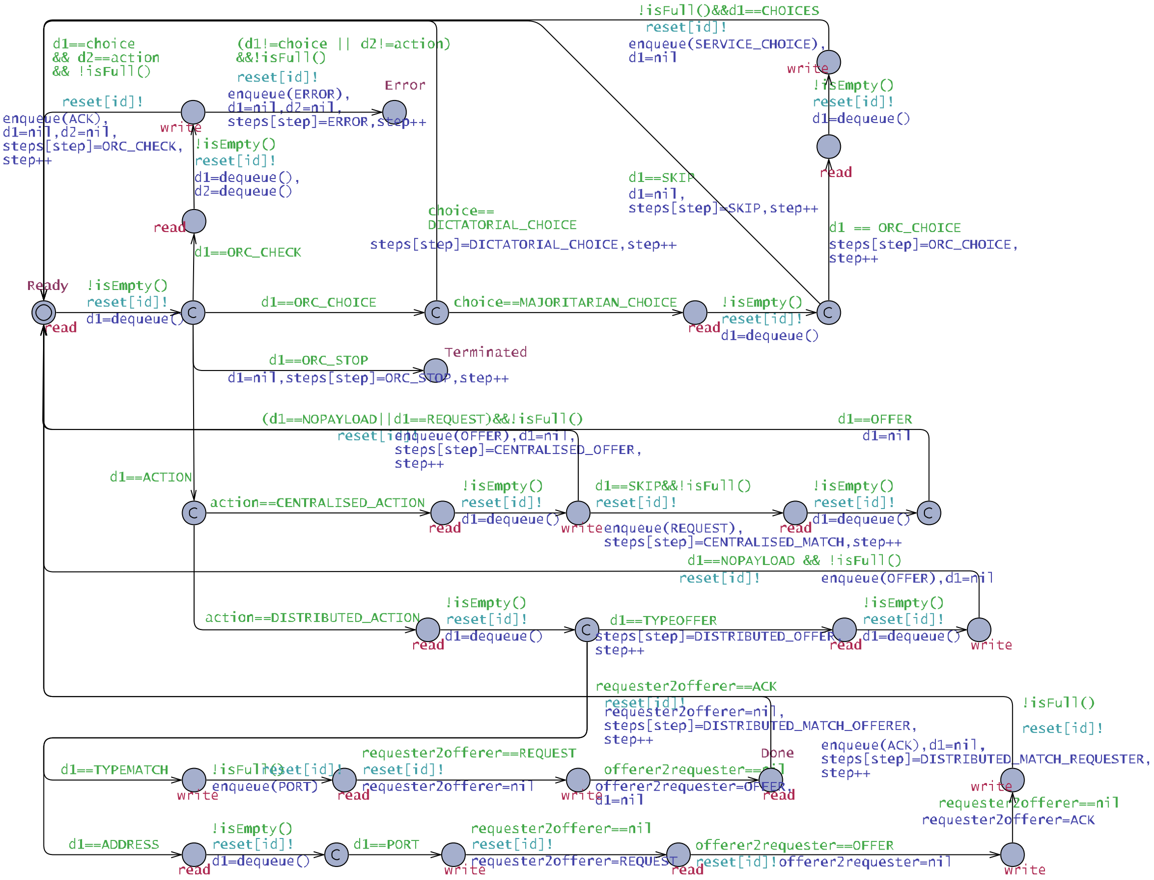


FIGURE 12. The RunnableOrchestratedContract UPPAAL template used for testing the orchestrator

7.3. **An example of test generation.** We now discuss an example of testing an orchestration whose set-up is dictatorial choice with centralised action. This means that the orchestrator will decide each choice and the orchestrator will act as a broker in a match between the two services.

Query. All versions of the model contains queries for testing the various configurations of the orchestrator (e.g., majoritarian choice, distributed action). We refer to [Basb] for more details about these other queries. In the concrete tests, the orchestrator will interact with two mocked services. Furthermore, the orchestration contract automaton, a parameter of the RunnableOrchestration object (see Figure 3), is dummy and is instrumental to cover the desired portions of code of CARE. In particular, the orchestration contract automaton that will be used in the concrete tests accepts the trace (!euro,-)(?coffee,!coffee)[(!euro,-)]*. The corresponding query used to generate the abstract test is:

```
E<>(alice.steps[0]==ORC_CHECK &&
bob.steps[1]==ORC_CHECK &&
alice.steps[2]==CENTRALISED_OFFER && //alice offer action !euro
alice.steps[3]==CENTRALISED_MATCH && //alice request action ?coffee
bob.steps[4]==CENTRALISED_OFFER && //bob offer action !coffee
alice.steps[5]==DICTATORIAL_CHOICE &&
bob.steps[6]==DICTATORIAL_CHOICE &&
alice.steps[7]==CENTRALISED_OFFER && //alice offer action !euro
alice.steps[8]==DICTATORIAL_CHOICE &&
bob.steps[9]==DICTATORIAL_CHOICE &&
alice.steps[10]==ORC_STOP &&
bob.steps[11]==ORC_STOP)
```

The query above is endowed with comments to pinpoint each step with the corresponding action, if any. An evidence to the query above simulates a run accepted by the orchestration contract automaton. We now analyse the query. As discussed in Section 5, the orchestration always starts by checking if the configurations of the services and the orchestrator are matching. Therefore, both services are instructed to check their configuration (`alice.steps[0]==ORC_CHECK` and `bob.steps[1]==ORC_CHECK`). In this test, the configuration is of centralised action and dictatorial choice. After the configuration checks, the orchestration starts. In the query, Alice is instructed to execute the offer action `!euro` with `alice.steps[2]==CENTRALISED_OFFER`. This corresponds to execution of the offer label `(!euro,-)` in the orchestration (Alice is the first service). After that, the match label `(?coffee,!coffee)` of the orchestration is executed. The match starts with Alice performing the request `?coffee`, encoded in the query as `alice.steps[3]==CENTRALISED_MATCH`. After Alice performs the request action, it is Bob's turn to match the request of Alice with the corresponding offer action`!coffee`. In Figure 12, it is possible to observe that the transition of the `RunnableOrchestratedContract` Uppaal template covering the execution of an offer action in both the cases of offer label or match label (respectively, `d1==NOPAYLOAD` or `d1==REQUEST`, see Section 5) has in its effect the instruction `steps[step]=CENTRALISED_OFFER`. Therefore, Bob is instructed to execute the offer action `!coffee` in the query with `bob.steps[4]==CENTRALISED_OFFER`.

After the offer label `(!euro,-)` and the match label `(?coffee,!coffee)` have both been executed, a choice has to be made on whether to execute the offer label `(!euro,-)` or terminate (recall that the trace `(!euro,-)(?coffee,!coffee)[(!euro,-)]`* is accepted by the orchestration contract automaton). In this set-up (i.e., dictatorial choice), the orchestrator performs the choice and communicates it to the services. The orchestrator will decide the first time to execute the offer label `(!euro,-)` and the second time to terminate. thus covering both cases. The first choice is encoded in the query with `alice.steps[5]==DICTATORIAL_CHOICE` and `bob.steps[6]==DICTATORIAL_CHOICE`. As before, the offer label is encoded in the query with `alice.steps[7]==CENTRALISED_OFFER`. Finally, the orchestrator performs the choice to terminate. This is encoded in the query with `alice.steps[8]==DICTATORIAL_CHOICE` and `bob.steps[9]==DICTATORIAL_CHOICE` (the orchestrator communicates the choice to the services) and `alice.steps[10]==ORC_STOP`, `bob.steps[11]==ORC_STOP` (both services terminate their execution).

7.3.1. *Test generation.* We now discuss the abstract test code generated by the query and how it is concretised into a JUnit test for the real system. Uppaal also allows the insertion of abstract test code in the system declaration of a model, which will be printed out in the abstract test either at the start (i.e., pre-fix) or upon termination of the test (i.e., post-fix).

In all tests, pre-fix test code is used to declare some variables that will be utilised during the test.

Configuration check. The first transitions executed during the test are those related to the configuration check. Both Alice and Bob generate the same portion of abstract test code, reported below.

```
1  msg = (String) oin.readObject();
2
3  //INIT CHECK
4  assertEquals(msg,RunnableOrchestration.check_msg);
5
6  String received_choiceType = (String) oin.readObject();
7  String received_actionType = (String) oin.readObject();
8  assertEquals(received_actionType,actionType);
9  assertEquals(received_choiceType,choiceType);
10
11 oout.writeObject(RunnableOrchestration.ack_msg);
12 oout.flush();
13 //STOP CHECK
```

In Figure 12, for both services, after the transition from the initial state `Ready` is executed, the transition with the guard `d1==ORC_CHECK` is fired, followed by the subsequent transition, and finally, the transition having the effect `steps[step]==ORC_CHECK` is executed. In order to satisfy the query these four transitions must first be executed by Alice (when `step=0`) and then by Bob (when `step=1`).

The first transition performs a `dequeue` operation, which appends the `readObject` instruction (line 1). The second transition executed, with the guard `d1==ORC_CHECK`, generates an assertion to check whether the message received by the orchestrator is the command for performing the configuration check (line 4). The next transition performs two `dequeue` operations, generating two `readObject` instructions, followed by two assertions to check whether the two messages are those expected by the orchestrator (lines 6-9). Recall that the orchestrator communicates the choice and action configuration (see Figure 9) to all services. It is important to note here that the actual values to check (`actionType` and `choiceType`) will be instantiated in the concrete test, discussed next. The fourth executed transition performs an `enqueue` operation, generating the test code `writeObject`, which writes the corresponding acknowledgement message into the socket (lines 11-13).

We now move to the structure of the concrete JUnit test and the concretisation of the abstract test code discussed until now. After that, we will discuss the remaining part of the abstract test code and its concretisation. The concrete test file is available at [Basb] and is called `DictatorialCentralisedRunnableOrchestrationTest.java`. All concrete tests used for testing the orchestrator follow the same schema, reported below (for the case of dictatorial choice and centralised action).

Listing 1. The JUnit method testing the centralised dictatorial orchestrator

```
@Test
public void test() throws IOException, ClassNotFoundException, InterruptedException {
    Thread alice = new Thread(()->{ //mocked service
        check(8080);
        try (ServerSocketChannel server = ServerSocketChannel.open()){
            server.bind(new InetSocketAddress(8080));
            Socket socket = server.accept().socket();
            ObjectInputStream oin = new ObjectInputStream(socket.getInputStream());
            ObjectOutputStream oout = new ObjectOutputStream(socket.getOutputStream());
10              oout.flush();
11              uppaal_alice(oin,oout);
12
13          } catch (IOException | ClassNotFoundException e) {
14              throw new RuntimeException(e);
15          }
16      });
17
18      Thread bob = new Thread(()->{ //mocked service
19          check(8081);
20          try (ServerSocketChannel server = ServerSocketChannel.open()){
21              server.bind(new InetSocketAddress(8081));
22              Socket socket = server.accept().socket();
23              ObjectInputStream oin = new ObjectInputStream(socket.getInputStream());
24              ObjectOutputStream oout = new ObjectOutputStream(socket.getOutputStream());
25              oout.flush();
26              uppaal_bob(oin,oout);
27          } catch (IOException | ClassNotFoundException e) {
28              throw new RuntimeException(e);
29          }
30      });
31      alice.start();
32      bob.start();
33      RunnableOrchestration ror = new DictatorialChoiceRunnableOrchestration(new Agreement(),
34              adc.importMSCA(dir+"Orchestration.data"),
35              Arrays.asList(null,null),
36              Arrays.asList(8080,8081),
37              new CentralisedOrchestratorAction());
38
39      Thread tror = new Thread(ror);
40      tror.start();
41      tror.join();
42  }
```

The test is organised as follows. Both Alice and Bob services are encoded as parallel threads. Alice's thread is declared at lines 3–16, whilst Bob's thread is declared at lines 18–30. They are both executed at lines 31–32. After that, the orchestration is declared at lines 33–37 and executed as another parallel thread (lines 39–41). It is important to note that, whilst Alice and Bob are two mocked services, the orchestrator is the real `RunnableOrchestration` class of `CARE`. In this case, at lines 33–37, we can see that the `RunnableOrchestration` object `ror` is instantiated with dictatorial choice and centralised action. The other tests for the orchestrator will have different instantiations (majoritarian choice and/or distributed action). Furthermore, it uses the orchestration contract automaton discussed above, stored in the file `Orchestration.data`. The `ror` object also takes as a parameter the ports of the services. Alice is listening at port `8080`, and Bob is listening at port `8081`.

Alice's thread starts by invoking the method `check`, passing its port as an argument (line 4). After that, the socket and object streams are initialised, and the method `uppaal_alice` is invoked, passing the input and output object streams as arguments. Bob's thread is equivalent. The method `check` (not reported for brevity), similarly to lines 5–10 of method `test` above, opens a socket awaiting connection from the orchestrator and invokes the method `uppaal_check`. The method `uppaal_check` contains the concretisation of the abstract test code discussed above, which is used for checking the configuration of the corresponding service. The methods `uppaal_alice` and `uppaal_bob` (line 11 and line 26) contain the remaining concretisation of the abstract test code generated by Alice and Bob, respectively, and will be discussed in the following. The method `uppaal_check` is reported below.

Listing 2. The JUnit method testing the configuration of the service

```
1  private void uppaal_check( ObjectInputStream oin, ObjectOutputStream oout) throws IOException {
2      String choiceType = "Dictatorial";
3      String actionType = "Centralised";
4      String msg;
5      msg = (String) oin.readObject();
6      assertEquals(msg,RunnableOrchestration.check_msg);
7      String received_choiceType = (String) oin.readObject();
8      String received_actionType = (String) oin.readObject();
9      assertEquals(received_actionType,actionType);
10     assertEquals(received_choiceType,choiceType);
11     oout.writeObject(RunnableOrchestration.ack_msg);
12     oout.flush();
13 }
```

It is easy to see that this method includes the abstract test case instructions discussed above. In this case, the concretisation of the abstract test case simply instantiates the variables `choiceType` and `actionType` to their respective values (lines 2-3). Note that the method `uppaal_check` will be invoked by both services.

We have discussed the main structure of the concrete test, the abstract test case instructions covering the configuration check and their concretisation. We now discuss the remaining instructions of the abstract test case and their concretisation in the methods `uppaal_alice` and `uppaal_bob`.

Offer label. After the orchestration check has been performed, the orchestration starts, and the offer label `(!euro,-)` of the orchestration will be executed. The following instructions are appended to the abstract test case being generated.

```
1  //centralised offerer 0
2
3  msg = (String) oin.readObject();
4
5  assertEquals(msg, expected action);
6
7  msg = (String) oin.readObject(); // reading payload centralised action
8  assertEquals(msg,expectedPayload);
9
10 oout.writeObject(INSERT OFFER);
11 oout.flush();
```

The transition outgoing state `CentralisedOffer` of the `RunnableOrchestration` automaton (see Figure 9) has as test code `//centralised offerer $(ror.offerer)`. This will append the comment at line 1 in the abstract test, indicating that the next instructions of the abstract test will cover the first offer action performed by Alice (service at index 0). This is helpful for generating the concrete test from the abstract test. After performing the check, both Alice and Bob are again in state `Ready` (Figure 12). The first `readObject` operation (line 3) is again appended by the outgoing transition from state `Ready` of Alice. The next transition executed by Alice, with guard `d1==ACTION`, will append the instruction at line 5. The assertion will check whether the message received by the orchestrator will be the expected action, which will be concretised in the concrete test with the value `"euro"` (i.e., the orchestrator is invoking the method `euro` of Alice).

After that, Alice automaton will execute the centralised action, and the subsequent transition will perform a `dequeue` operation, appending to the abstract test case the instructions at lines 7-8. In the assertion, `expectedPayload` is a placeholder that will be concretised with value `null`. Recall that in CARE each action of the contract automaton of each service is in

correspondence with a method of the corresponding `service` class (see Section 3), which has input parameters and a returned value. In this case, the parameters and returned values of the method in correspondence with the offer `euro` of Alice are dummy (i.e., concretised with `null` values). The next transition with effect `enqueue` is executed by Alice. This transition also has in the effect the instruction `steps[step]=CENTRALISED_OFFER`, thus fulfilling the query (at this point of the simulation it holds `step=2`). The execution of the transition will append instructions at lines 10-11 to the abstract test case. Indeed, the `enqueue` operation corresponds to a `writeObject` operation in the test. The placeholder `INSERT OFFER` will be replaced by the concrete value `null` in the concrete test.
This concludes the test code for the first offer label `(!euro,-)` of the orchestration.

Match label. Next, the match label `(?coffee,!coffee)` is executed. The instructions appended to the abstract test case during the execution of the match label are reported below.

```
1   //requester: 0
2
3   msg = (String) oin.readObject();
4
5   assertEquals(msg, expected action);
6
7   msg = (String) oin.readObject(); // reading payload centralised action
8   assertEquals(msg,expectedPayload);
9
10  oout.writeObject(INSERT REQUEST);
11  oout.flush();
12
13  //matching offerer: 1
14
15  msg = (String) oin.readObject();
16
17  assertEquals(msg, expected action);
18
19  msg = (String) oin.readObject(); // reading payload centralised action
20  assertEquals(msg,expectedPayload);
21
22  oout.writeObject(INSERT OFFER);
23  oout.flush();
24
25  //requester: 0
26
27  msg = (String) oin.readObject(); //receiving the matching offer
28  assertEquals(msg, expected payload);
```

The three comments (lines 1,13,25) are appended by the `RunnableOrchestration` automaton, after state `CentralisedMatch` is reached (see Figure 9), and are useful to identify the instructions generated by Alice (`requester 0`) and Bob (`matching offerer 1`). The instructions at lines 3-11 are appended by Alice when executing the request action `?coffee`. The placeholder `expectedPayload` at line 8 will be concretised with the value `"coffee"` (i.e., the orchestrator is invoking the method `coffee` of Alice), whilst the placeholder at line 10 will be replaced with the concrete value `"request payload"` (i.e., the request sent by Alice to Bob is the dummy String `"request payload"`). Indeed, although the services are mocked by the test, in case of a match these values will be handled by the real `CARE` orchestrator, which will act as a broker between the two services. After executing the transition with effect `steps[step]=CENTRALISED_MATCH` (Figure 12), thus fulfilling the query, Alice waits for Bob to send the offer. The instructions at lines 15-23 are appended by Bob when executing the

offer action `!coffee`. Note that these are identical to the previous offer action `!euro` fired by Alice, because the same transitions of Figure 12 have been executed. The placeholders at line 17, line 20 and line 22 will be replaced by the concrete values `"coffee"` (the orchestrator invokes the method `coffee` of Bob), `"request payload"` (the payload sent by Alice to Bob is checked) and `"offer payload"` (the dummy reply of Bob to Alice), respectively. After that, the instructions at lines 27-28 are executed by Alice for completing the request action. The placeholder at line 28 will be replaced by the concrete value `"offer payload"` (the reply of Bob to Alice is checked).
This concludes the match `(?coffee,!coffee)`.

Choice. As stated earlier, the orchestrator will now perform a choice on whether to continue and execute the offer label `(!euro,-)` or terminate. For brevity, we only report the abstract test code dumped by a service when executing a dictatorial choice, reported below.

```
1  msg = (String) oin.readObject();
2
3  assertEquals(msg,RunnableOrchestration.choice_msg);
```

The instruction at line 1 is again appended when executing the `dequeue` effect outgoing state `Ready` (Figure 12). After that, the transition with guard `d1 == ORC_CHOICE` is executed, which will append the test code at line 3. The next executed transition, with effect `steps[step]=DICTATORIAL_CHOICE`, has no test code generated but it is used to fulfill the query. No other test code is generated for the dictatorial choice, as the orchestrator decides on its own the next move. This test code is generated by both Alice and Bob, performing in turn the choice. In this example, after the first dictatorial choice, the orchestrator will execute again the offer label `(!euro,-)`, and again a dictatorial choice. We have already covered the test code that will be generated.

Termination. After that, the orchestrator decides to terminate. The test code generated by each service is:

```
1  msg = (String) oin.readObject();
2
3  assertEquals(msg,RunnableOrchestration.stop_msg);
```

The instruction at line 1 is the first `dequeue` operation as above. The instruction at line 3 is instead appended by the test code of the transition with guard `d1==ORC_STOP` (Figure 12). This transition has also in its effect the instruction `steps[step]=ORC_STOP`, thus fulfilling the query. Note that both values `RunnableOrchestration.choice_msg` and `RunnableOrchestration.stop_msg` are constants of the class `RunnableOrchestration` of `CARE`, used to identify the two commands of the orchestrator.
We have already discussed how the values of the abstract test code are concretised. For completeness, we report below the code of the method `uppaal_alice` discussed before, which contains test code for the service Alice, obtained by concretising the abstract test code as discussed above.

Listing 3. The JUnit method testing the Alice service

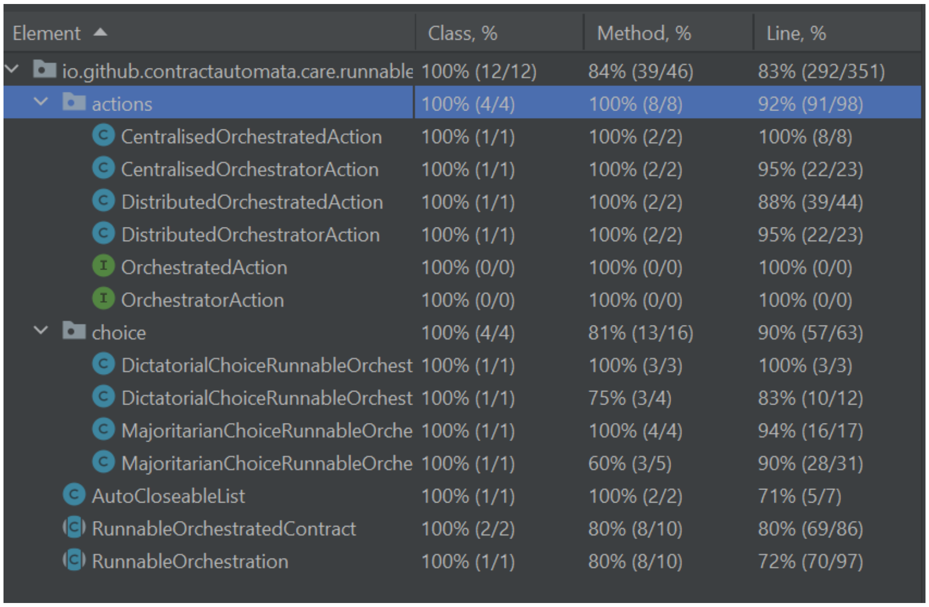
```
private void uppaal_alice( ObjectInputStream oin, ObjectOutputStream oout) throws IOException {
    String msg;
    msg = (String) oin.readObject();
    assertEquals(msg, "euro");
    msg = (String) oin.readObject(); // reading payload centralised action
    assertEquals(msg,null);
    oout.writeObject(null);
    oout.flush();
```

```
 9
10      msg = (String) oin.readObject();
11      assertEquals(msg, "coffee");
12      msg = (String) oin.readObject(); // reading payload centralised action
13      assertEquals(msg,null);
14      oout.writeObject("request payload");
15      oout.flush();
16
17      msg = (String) oin.readObject(); //receiving the matching offer
18      assertEquals(msg, "offer payload");
19
20      msg = (String) oin.readObject();
21      assertEquals(msg,RunnableOrchestration.choice_msg);
22
23      msg = (String) oin.readObject();
24
25      //if the orchestrator decides not to terminate
26      while (msg.equals("euro")) {
27          msg = (String) oin.readObject(); // reading payload centralised action
28          assertEquals(msg, null);
29          oout.writeObject(null);
30          oout.flush();
31          msg = (String) oin.readObject();
32          assertEquals(msg, RunnableOrchestration.choice_msg);
33          msg = (String) oin.readObject();
34      }
35
36      assertEquals(msg,RunnableOrchestration.stop_msg);
37  }
```

Note that the above code allows Alice to perform more than a single offer "euro" before terminating (lines 26 34). In this way it is possible to handle all possible traces of the orchestration automaton, and all internal choices of the orchestrator.
Similarly, we report below the code of the method uppaal_bob.

Listing 4. The JUnit method testing the Bob service

```
private void uppaal_bob(ObjectInputStream oin, ObjectOutputStream oout) throws IOException {
    String msg;

    msg = (String) oin.readObject();
    assertEquals(msg, "coffee");
    msg = (String) oin.readObject(); // reading payload centralised action
    assertEquals(msg,"request payload");
    oout.writeObject("offer payload");
    oout.flush();

    msg = (String) oin.readObject();
    assertEquals(msg,RunnableOrchestration.choice_msg);

    //in case the orchestrator decides not to terminate
    do {
        msg = (String) oin.readObject();
    } while (msg.equals(RunnableOrchestration.choice_msg));

    assertEquals(msg,RunnableOrchestration.stop_msg);


}
```



Element ▲	Class, %	Method, %	Line, %
io.github.contractautomata.care.runnable	100% (12/12)	84% (39/46)	83% (292/351)
actions	100% (4/4)	100% (8/8)	92% (91/98)
CentralisedOrchestratedAction	100% (1/1)	100% (2/2)	100% (8/8)
CentralisedOrchestratorAction	100% (1/1)	100% (2/2)	95% (22/23)
DistributedOrchestratedAction	100% (1/1)	100% (2/2)	88% (39/44)
DistributedOrchestratorAction	100% (1/1)	100% (2/2)	95% (22/23)
OrchestratedAction	100% (0/0)	100% (0/0)	100% (0/0)
OrchestratorAction	100% (0/0)	100% (0/0)	100% (0/0)
choice	100% (4/4)	81% (13/16)	90% (57/63)
DictatorialChoiceRunnableOrchest	100% (1/1)	100% (3/3)	100% (3/3)
DictatorialChoiceRunnableOrchest	100% (1/1)	75% (3/4)	83% (10/12)
MajoritarianChoiceRunnableOrche	100% (1/1)	100% (4/4)	94% (16/17)
MajoritarianChoiceRunnableOrche	100% (1/1)	60% (3/5)	90% (28/31)
AutoCloseableList	100% (1/1)	100% (2/2)	71% (5/7)
RunnableOrchestratedContract	100% (2/2)	80% (8/10)	80% (69/86)
RunnableOrchestration	100% (1/1)	80% (8/10)	72% (70/97)

FIGURE 13. The coverage information generated by IntelliJ

Similar to Alice, also the code of Bob allows to handle many unfoldings of the final loop of the orchestration.

This concludes the example of test generation. Recall that this is one query of one version of the model (for testing the orchestrator). There are other queries used for testing the other configurations (e.g., majoritarian choice, distributed offer) and other versions of the model for testing the other components. However, the generation of abstract tests and their concretisation follows the same approach covered by this example. All these versions and the concrete tests are available at [Basb].

7.4. **Coverage.** The coverage information generated by IntelliJ is in Figure 13. The code coverage indicates that the concrete tests derived from the model cover a significant portion of the source code. This suggests that the model is not excessively abstract compared to the actual implementation. Furthermore, from the queries we know that the generated abstract tests cover all transitions of the model and all interactions between the orchestrator and the services.

On a side note, the modelling and verification allowed us to fix some issues (see Section 6). This would not have been possible in case the abstraction were too coarse.

8. Conclusion and Future Work

We have presented the formal modelling and verification of the Contract Automata Runtime Environment (CARE), an open-source platform. Both statistical and exhaustive model checking techniques have played a crucial role in formally verifying numerous desired properties of the modeled system, such as the absence of deadlocks, while also enhancing the accuracy of the formal model. Statistical model checking has also been employed to fine-tune parameter settings within the formal model, such as the buffer size. The adequacy of the abstract model has been described, and the transitions of the formal model have been linked to the corresponding lines of source code. The tests generated from the formal model have been employed to test the source code.

At present, the different artifacts such as the model, source code, tests, and tracing information are manually kept aligned. This process demands substantial effort whenever a new version of `CARE` is introduced, as each artifact needs to be updated accordingly. Future work involves studying techniques for automatic alignment of these artifacts. Furthermore, it would be interesting to exploit the facilities provided by the recently introduced tool `Uppex` [PPN$^+$23] to factorize the different variants of the model discussed in the paper under a single, configurable `Uppex` model, to automate the selection of a configuration and more generally to facilitate the collaboration between the software developer and the Uppaal modeller, whenever they are different persons.

Acknowledgment. Part of this study was carried out within the MUR PRIN 2022 PNRR P2022A492B project ADVENTURE (ADVancEd iNtegraTed evalUation of Railway systEms) funded by the European Union - NextGenerationEU, and the CNR project “Formal Methods in Software Engineering 2.0”, CUP B53C24000720005.

References

[ABB$^+$16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification – The KeY Book: From Theory to Practice*, volume 10001 of *LNCS*. Springer, 2016. doi:10.1007/978-3-319-49812-6.

[ABZ13] L. Acciai, M. Boreale, and G. Zavattaro. Behavioural contracts with request-response operations. *Sci. Comp. Program.*, 78(2):248–267, 2013. doi:10.1016/j.scico.2011.10.007.

[AD94] R. Alur and D. Dill. A Theory of Timed Automata. *Theoret. Comp. Sci.*, 126(2):183–235, 1994. doi:10.1016/0304-3975(94)90010-8.

[AD24] Arwa Hameed Alsubhi and Ornela Dardha. Coconut: Typestates for embedded systems. In Ilaria Castellani and Francesco Tiezzi, editors, *Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings*, volume 14676 of *Lecture Notes in Computer Science*, pages 219–238. Springer, 2024. doi:10.1007/978-3-031-62697-5_12.

[Basa] Davide Basile. CATApp. URL: https://github.com/contractautomataproject/ContractAutomataApp.

[Basb] Davide Basile. Formal analysis of the Contract Automata Runtime Environment with Uppaal, complementary material. doi:10.5281/zenodo.14671729.

[Bas24] Davide Basile. Modelling, verifying and testing the contract automata runtime environment with uppaal. In Ilaria Castellani and Francesco Tiezzi, editors, *Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings*, volume 14676 of *Lecture Notes in Computer Science*, pages 93–110. Springer, 2024. doi:10.1007/978-3-031-62697-5_6.

[BBG$^+$22] Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota, and António Ravara. A Java typestate checker supporting inheritance. *Sci. Comput. Program.*, 221, 2022. doi:10.1016/j.scico.2022.102844.

[BCZ15] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. Compliance in Behavioural Contracts: A Brief Survey. In *Programming Languages with Applications to Biology and Security*, volume 9465 of *LNCS*, pages 103–121. Springer, 2015. doi:10.1007/978-3-319-25527-9_9.

[BDF16] Davide Basile, Pierpaolo Degano, and Gian-Luigi Ferrari. Automata for specifying and orchestrating service contracts. *Log. Methods Comput. Sci.*, 12(4):6:1–6:51, 2016. doi:10.2168/LMCS-12(4:6)2016.

[BDL^{+}06] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *Proceedings 3rd International Conference on the Quantitative Evaluation of SysTems (QEST)*, pages 125–126. IEEE, 2006. doi:10.1109/QEST.2006.59.

[BLMT08] R. Bruni, I. Lanese, H. C. Melgratti, and E. Tuosto. Multiparty Sessions in SOC. In *COORDINATION*, volume 5052 of *LNCS*, pages 67–82. Springer, 2008. doi:10.1007/978-3-540-68265-3_5.

[BMF23] Davide Basile, Franco Mazzanti, and Alessio Ferrari. Experimenting with formal verification and model-based development in railways: The case of UMC and sparx enterprise architect. In Alessandro Cimatti and Laura Titolo, editors, *Formal Methods for Industrial Critical Systems (FMICS)*, volume 14290 of *LNCS*, pages 1–21. Springer, 2023. doi:10.1007/978-3-031-43681-9_1.

[Bou15] Jean-Louis Boulanger. Tool Qualification. In *CENELEC 50128 and IEC 62279 Standards*, chapter 9, pages 287–308. John Wiley & Sons, 2015. doi:10.1002/9781119005056.ch9.

[BtB21] Davide Basile and Maurice H. ter Beek. A Clean and Efficient Implementation of Choreography Synthesis for Behavioural Contracts. In Ferruccio Damiani and Ornela Dardha, editors, *COORDINATION*, volume 12717 of *LNCS*, pages 225–238. Springer, 2021. doi:10.1007/978-3-030-78142-2_14.

[BtB22] Davide Basile and Maurice H. ter Beek. Contract Automata Library. *Sci. Comput. Program.*, 221:102841, 2022. URL: https://github.com/contractautomataproject/ContractAutomataLib, doi:10.1016/j.scico.2022.102841.

[BtB23a] Davide Basile and Maurice H. ter Beek. A Runtime Environment for Contract Automata. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *FM*, volume 14000 of *LNCS*, pages 550–567. Springer, 2023. URL: https://github.com/contractautomataproject/CARE, doi:10.1007/978-3-031-27481-7_31.

[BtB23b] Davide Basile and Maurice H. ter Beek. Research challenges in orchestration synthesis. In Clément Aubert, Cinzia Di Giusto, Simon Fowler, and Larisa Safina, editors, *Proceedings of the 16th Interaction and Concurrency Experience (ICE)*, volume 383 of *EPTCS*, pages 73–90, 2023. doi:10.4204/EPTCS.383.5.

[BtB24] Davide Basile and Maurice H. ter Beek. Advancing orchestration synthesis for contract automata. *J. Log. Algebraic Methods Program.*, 141:100998, 2024. doi:10.1016/J.JLAMP.2024.100998.

[BtBC18] Davide Basile, Maurice H. ter Beek, and Vincenzo Ciancia. Statistical Model Checking of a Moving Block Railway Signalling Scenario with Uppaal SMC. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 11245 of *LNCS*, pages 372–391. Springer, 2018. doi:10.1007/978-3-030-03421-4_24.

[BtBD^{+}20] Davide Basile, Maurice H. ter Beek, Pierpaolo Degano, Axel Legay, Gian-Luigi Ferrari, Stefania Gnesi, and Felicita Di Giandomenico. Controller synthesis of service contracts with variability. *Sci. Comput. Program.*, 187:102344, 2020. doi:10.1016/j.scico.2019.102344.

[BtBFL22] Davide Basile, Maurice H. ter Beek, Alessio Ferrari, and Axel Legay. Exploring the ERTMS/ETCS full moving block specification: an experience with formal methods. *Int. J. Softw. Tools Technol. Transf.*, 24(3):351–370, 2022. doi:10.1007/s10009-022-00653-3.

[BtBL20] Davide Basile, Maurice H. ter Beek, and Axel Legay. Strategy Synthesis for Autonomous Driving in a Moving Block Railway System with Uppaal Stratego. In Alexey Gotsman and Ana Sokolova, editors, *FORTE*, volume 12136 of *LNCS*, pages 3–21. Springer, 2020. doi:10.1007/978-3-030-50086-3_1.

[BtBP20] Davide Basile, Maurice H. ter Beek, and Rosario Pugliese. 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. doi:10.23638/LMCS-16(2:9)2020.

[CDCP12] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On Global Types and Multi-Party Sessions. *Log. Methods Comput. Sci.*, 8(1:24):1–45, 2012. doi:10.2168/LMCS-8(1:24)2012.

[CGMZ21] Guillaume Cluzel, Kyriakos Georgiou, Yannick Moy, and Clément Zeller. Layered formal verification of a TCP stack. In *SecDev*, pages 86–93. IEEE, 2021. doi:10.1109/SecDev51306.2021.00028.

[CGP09] G. Castagna, N. Gesbert, and L. Padovani. A Theory of Contracts for Web Services. *ACM Trans. Program. Lang. Syst.*, 31(5):19:1–19:61, 2009. doi:10.1145/1538917.1538920.

[CHLR23] Héléne Coullon, Ludovic Henrio, Frédéric Loulergue, and Simon Robillard. Component-based distributed software reconfiguration:a verification-oriented survey. *ACM Comput. Surv.*, 56(1), August 2023. `doi:10.1145/3595376`.

[dAH01] L. de Alfaro and T. Henzinger. Interface Automata. In *ESEC/FSE*, pages 109–120. ACM, 2001. `doi:10.1145/503209.503226`.

[DCd10] M. Dezani-Ciancaglini and U. de'Liguoro. Sessions and Session Types: An Overview. In *WS-FM*, volume 6194 of *LNCS*, pages 1–28. Springer, 2010. `doi:10.1007/978-3-642-14458-5_1`.

[dlCGMS09] Pedro de la Cámara, María-del-Mar Gallardo, Pedro Merino, and David Sanán. Checking the reliability of socket based communication software. *Int. J. Softw. Tools Technol. Transf.*, 11(5):359–374, 2009. `doi:10.1007/s10009-009-0112-7`.

[DLL$^+$10] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed I/O Automata: A Complete Specification Theory for Real-time Systems. In *HSCC*, pages 91–100. ACM, 2010. `doi:10.1145/1755952.1755967`.

[DLL$^+$15] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. Uppaal SMC tutorial. *Int. J. Softw. Tools Technol. Transf.*, 17(4):397–415, 2015. `doi:10.1007/s10009-014-0361-y`.

[FtB22] Alessio Ferrari and Maurice H. ter Beek. Formal Methods in Railways: A Systematic Mapping Study. *ACM Comput. Surv.*, 2022. `doi:10.1145/3520480`.

[FZL18] Yuan Fei, Huibiao Zhu, and Xin Li. Modeling and verification of NLSR protocol using UPPAAL. In Jun Pang, Chenyi Zhang, Jifeng He, and Jian Weng, editors, *TASE*, pages 108–115. IEEE Computer Society, 2018. `doi:10.1109/TASE.2018.00022`.

[GJP$^+$22] Rong Gu, Peter Gjøl Jensen, Danny Bøgsted Poulsen, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. Verifiable strategy synthesis for multiple autonomous agents: a scalable approach. *Int. J. Softw. Tools Technol. Transf.*, 24(3):395–414, 2022. `doi:10.1007/s10009-022-00657-z`.

[GR17] Simon Gay and António Ravara, editors. *Behavioural Types: from Theory to Tools*. River, 2017. `doi:10.13052/rp-9788793519817`.

[GtBvdP20] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 Expert Survey on Formal Methods. In M.H. ter Beek and D. Ničković, editors, *FMICS*, volume 12327 of *LNCS*, pages 3–69. Springer, 2020. `doi:10.1007/978-3-030-58298-2_1`.

[HL23] Mohamed Amin Hammami and Mariam Lahami. Model-based testing approach for EIP-1559 ethereum smart contracts. In Mohamed Mosbah, M. Tahar Kechadi, Ladjel Bellatreche, and Faïez Gargouri, editors, *Model and Data Engineering - 12th International Conference, MEDI 2023, Sousse, Tunisia, November 2-4, 2023, Proceedings*, volume 14396 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2023. `doi:10.1007/978-3-031-49333-1_4`.

[HLM$^+$08] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer, 2008. `doi:10.1007/978-3-540-78917-8_3`.

[HLV$^+$16] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. - M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. Torres Vieira, and G. Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. `doi:10.1145/2873052`.

[HMG$^+$23] Benedek Horváth, Vince Molnár, Bence Graics, Ákos Hajdu, István Ráth, Ákos Horváth, Robert Karban, Gelys Trancho, and Zoltán Micskei. Pragmatic verification and validation of industrial executable sysml models. *Systems Engineering*, 2023. `doi:10.1002/sys.21679`.

[HYC08] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM, 2008. `doi:10.1145/1328438.1328472`.

[Jav] URL: `https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html`.

[KDPG18] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Sci. Comput. Program.*, 155:52–75, 2018. `doi:10.1016/j.scico.2017.10.006`.

[KLN$^+$15] Jin Hyun Kim, Kim G. Larsen, Brian Nielsen, Marius Mikucionis, and Petur Olsen. Formal analysis and testing of real-time automotive systems using UPPAAL tools. In Manuel Núñez and Matthias Güdemann, editors, *Formal Methods for Industrial Critical Systems*

- 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings, volume 9128 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2015. doi:10.1007/978-3-319-19458-5_4.

[LLN18] Kim G. Larsen, Florian Lorber, and Brian Nielsen. 20 years of UPPAAL enabled industrial model-based validation and beyond. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, volume 11247 of *Lecture Notes in Computer Science*, pages 212–229. Springer, 2018. doi:10.1007/978-3-030-03427-6_18.

[LLT$^+$19] Axel Legay, Anna Lukina, Louis-Marie Traonouez, Junxing Yang, Scott A. Smolka, and Radu Grosu. Statistical Model Checking. In Bernhard Steffen and Gerhard J. Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, volume 10000 of *LNCS*, pages 478–504. Springer, 2019. doi:10.1007/978-3-319-91908-9_23.

[LP15] C. Laneve and L. Padovani. An algebraic theory for web service contracts. *Form. Asp. Comp.*, 27(4):613–640, 2015. doi:10.1007/s00165-015-0334-2.

[LRN$^+$22] Sascha Lehmann, Antje Rogalla, Maximilian Neidhardt, Anton Reinecke, Alexander Schlaefer, and Sibylle Schupp. Modeling $\mathbb{R}^3$ Needle Steering in Uppaal. In Clemens Dubslaff and Bas Luttik, editors, *Proceedings of the 5th Workshop on Models for Formal Analysis of Real Systems (MARS@ETAPS)*, volume 355 of *EPTCS*, pages 40–59, 2022. doi:10.4204/EPTCS.355.4.

[LSM$^+$20] Si Liu, Atul Sandur, José Meseguer, Peter Csaba Ölveczky, and Qi Wang. Generating correct-by-construction distributed implementations from formal MAUDE designs. In Ritchie Lee, Susmit Jha, and Anastasia Mavridou, editors, *NFM*, volume 12229 of *LNCS*, pages 22–40. Springer, 2020. doi:10.1007/978-3-030-55754-6_2.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.

[LT89] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI Q.*, 2:219–246, 1989. URL: https://ir.cwi.nl/pub/18164/18164A.pdf.

[MNF13] J. Michaux, E. Najm, and A. Fantechi. Session types for safe Web service orchestration. *J. Log. Algebr. Program.*, 82(8):282–310, 2013. doi:10.1016/j.jlap.2013.05.004.

[Obj] URL: https://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html.

[OPB$^+$21] Simone Orlando, Vairo Di Pasquale, Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Corinne, a Tool for Choreography Automata. In Gwen Salaün and Anton Wijs, editors, *FACS*, volume 13077 of *LNCS*, pages 82–92. Springer, 2021. doi:10.1007/978-3-030-90636-8_5.

[PMT24] Carlos Gustavo López Pombo, Pablo Montepagano, and Emilio Tuosto. Search: An execution infrastructure for service-based software systems. In Ilaria Castellani and Francesco Tiezzi, editors, *Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings*, volume 14676 of *Lecture Notes in Computer Science*, pages 314–330. Springer, 2024. doi:10.1007/978-3-031-62697-5_17.

[PPN$^+$23] José Proença, David Pereira, Giann Spilere Nandi, Sina Borrami, and Jonas Melchert. Spreadsheet-based Configuration of Families of Real-Time Specifications. In Maurice H. ter Beek and Clemens Dubslaff, editors, *Proceedings of the 1st Workshop on Trends in Configurable Systems Analysis (TiCSA@ETAPS)*, volume 392 of *EPTCS*, pages 27–39, 2023. doi:10.4204/EPTCS.392.2.

[RCS$^+$22] Markus Roggenbach, Antonio Cerone, Bernd-Holger Schlingloff, Gerardo Schneider, and Siraj Ahmed Shaikh. *Formal Methods for Software Engineering: Languages, Methods, Application Domains.* TTCS. Springer, 2022. doi:10.1007/978-3-030-38800-3.

[RG21] Gopal N. Rai and G. R. Gangadharan. Model checking based web service verification: A systematic literature review. *IEEE Transactions on Services Computing*, 14(3):747–764, 2021. doi:10.1109/tsc.2018.2845401.

[RW87] Peter J. Ramadge and Walter M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987. doi:10.1137/0325013.

[SF17] Shruti Saini and Ansgar Fehnker. Evaluating the Stream Control Transmission Protocol Using Uppaal. In Holger Hermanns and Peter Höfner, editors, *Proceedings of the 2nd Workshop on*

Models for Formal Analysis of Real Systems (MARS@ETAPS), volume 244 of *EPTCS*, pages 1–13, 2017. `doi:10.4204/EPTCS.244.1`.

[SVW20] Fatima Shokri-Manninen, Jüri Vain, and Marina Waldén. Formal Verification of COLREG-Based Navigation of Maritime Autonomous Systems. In Frank S. de Boer and Antonio Cerone, editors, *SEFM*, volume 12310 of *LNCS*, pages 41–59. Springer, 2020. `doi:10.1007/978-3-030-58768-0_3`.

[SY86] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986. `doi:10.1109/TSE.1986.6312929`.

[tBBG07] Maurice H. ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Web Service Composition Approaches: From Industrial Standards to Formal Methods. In *Proceedings of the 2nd International Conference on Internet and Web Applications and Services (ICIW'07)*. IEEE, 2007. `doi:10.1109/ICIW.2007.71`.

[TMR20] André Trindade, João Mota, and António Ravara. Typestates to Automata and back: a tool. In Julien Lange, Anastasia Mavridou, Larisa Safina, and Alceste Scalas, editors, *Proceedings of the 13th Interaction and Concurrency Experience (ICE)*, volume 324 of *EPTCS*, pages 25–42, 2020. `doi:10.4204/EPTCS.324.4`.

[WVHFM06] Daniel Witsch, Birgit Vogel-Heuser, Jean-Marc Faure, and Gaëlle Marsal. Performance analysis of industrial ethernet networks by means of timed model-checking. In *INCOM*, volume 39, pages 101–106, 2006. `doi:10.3182/20060517-3-FR-2903.00063`.

[WWH$^+$22] Jie Wang, Xintao Wu, Gang Hou, Pengfei Li, Ao Gao, Zhichao Chen, and Haoyu Gao. Modeling and reliability verification of industrial control network protocol based on time state transition matrix. *Int. J. Commun. Syst.*, 35(9), 2022. `doi:10.1002/dac.5140`.