Modelling, Verifying and Testing the Contract Automata Runtime Environment with Uppaal

Davide Basile✉

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

Abstract. The contract automata runtime environment (CARE) is a distributed middleware application recently introduced to realise service applications specified using a dialect of finite-state automata. In this paper, we detail the formal modelling and verification 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, utilizing exhaustive and statistical model checking techniques. 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 address the issues that have been identified and fixed.

1 Introduction

Behavioural contracts [1] 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 [10]. A composition of contracts is in *agreement* when all requests are matched by corresponding offers of other contracts. A composition can be refined to one in agreement using the orchestration synthesis algorithm [6,9], a variation of the synthesis algorithm from supervisory control theory [26]. The Contract Automata Runtime Environment (CARE) [5] 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 [5] 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 tackle the problem of modelling and verifying 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 consensus [20]) and whose low-level communication implementations were found to have issues, e.g. deadlocks [27]. 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 and statistical model checking. This combination allows for thorough analysis of the model's behavior and ensures scalability when dealing with systems that possess a large state-space. All models, formulas, and logs are publicly available in [3], together with traceability and model-based testing information connecting the model to the source code.

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 [15]. 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.

Related work Several applications of Uppaal to various case studies are available in the literature, including land transport [8], maritime transport [28], medical systems [22], and autonomous agents path planning [17]. 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 [14] 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 [2,7]. This is also the case for industrial systems whose implementations are non-disclosed [18]. 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.

In contrast to the previously mentioned literature, the innovation of this paper lies in the bottom-up formal analysis of an established open-source system that has already been developed [5]. 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. 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.

As reported in [12], the question of qualification and validation of formal methods tools is "absolutely crucial". Other tools for behavioural contracts are present in the literature [16,25], but many of these implementations have not undergone a process of formal verification. In this paper, we provide the formal modelling and verification of `CARE`, which is part of the contract automata toolkit.

Outline In Section 2, we start by providing the background on contract automata, their tool support, Uppaal, and statistical model checking. The modelling phase is described in Section 3, whilst the verification is described in Section 4. Finally, the conclusion and future work are presented in Section 5.

2 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.

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., 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 3 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 [10]. 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 [9]. The contract automaton in Figure 3 right is an orchestration in agreement.

CA and their functionalities are implemented in a software artefact, called Contract Automata Library (`CATLib`) [4]. With `CATLib` it is possible to specify, compose, and synthesise specifications given as contract automata, but `CATLib` does not provide facilities for pairing the specifications computed with `CATLib` with actual implementations of service-based applications. For this purpose, the Contract Automata Runtime Environment (`CARE`) [5] was introduced, which will be described in Section 3.

Uppaal UPPAAL SMC [13] is a variant of UPPAAL [11], 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 via broadcast channels and shared variables. In this paper, we will use both exhaustive and statistical model checking. Statistical Model Checking (SMC) [21] 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 $1 - \alpha$, where $p' = (\#\{\rho_i \mid \rho_i \models \varphi\})/N$, 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 modeling, probabilistic and non-deterministic choices, and capabilities for statistical model checking, simulation, and model-based testing.

CARE We provide a brief overview of `CARE`. Class diagrams, examples of service applications created with `CARE` and other details are in [5]. 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 [5], along with the formal verification of `CARE` as an application. This paper addresses the latter facet. The two core abstract Java classes are the `RunnableOrchestration` and the `RunnableOrchestratedContract`. The first one is the implementation of the orchestrator who reads the synthesised orchestration automaton and communicates with the `RunnableOrchestratedContract` services to realise the prescribed behaviour. Each `RunnableOrchestratedContract` is responsible for pairing its contract automaton specification with an implementation provided as a Java class, where each action of the automaton is in correspondence with a method of the class. 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 ContractViolationException will be raised by the service. As demonstrated in [5], this exception will never be raised if the orchestration in agreement is synthesised using the algorithms of contract automata.

In CARE, two main aspects to implement are the execution of *choices* and *actions*. Principles of object-oriented programming are used to extend the framework to various configurations. For choices, the two currently available implementations are DictatorialChoice (the orchestrator decides autonomously) and MajoritarianChoice (each service involved votes and the majority wins). For actions, the two currently available implementations are CentralisedAction and DistributedAction. In a CentralisedAction, the orchestrator acts as a broker forwarding the offers and requests of the two services to realise the match. In a DistributedAction, the orchestrator makes the two services aware of each other by communicating their addresses and ports. The two services autonomously realise the match action, after which control returns to the orchestrator. Recall that in an orchestration in agreement, labels can be matches or offers and in case of offers the orchestrator interacts with only one service.

3 Methodology and Formal Model

We now describe the methodology used for modelling the communications, abstracting away irrelevant details and ascertain the adequacy of the model with respect to the implementation.

3.1 Methodology

Modelling TCP/IP sockets communication Java TCP/IP sockets communications are asynchronous with FIFO buffers [24]. 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, global (FIFO) arrays are used to model the TCP/IP sockets buffers. These arrays are only modified by functions for enqueueing and dequeuing messages. Each party communicates with the partner using two global arrays (one for sending and one for receiving, respectively). 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 (that is one of the global constants described above). Indeed, in this model the actual payload of each communication is abstracted away. 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). Similarly, both automata have a method dequeue for consuming messages from their respective arrays.

The default mode for Java TCP/IP sockets is *blocking* [19], 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 transition in the network. Instead, an urgent transition must be executed without any delay. If a send or receiving transition were to be either committed or urgent, it would 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. 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 3 left). Each send or receive operation in every socket resets the `SocketTimeout` clock. 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 transitions are not shown in Figures 1 and 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 4. 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.

Adequacy The adequacy of the adopted level of abstraction is ascertained as follows. Each transition in the model is traced back to the specific lines of source code in [3]. The model-based testing functionality of UPPAAL has been employed to generate tests that demonstrate the model's adherence to the actual implementation. Roughly, each transition that involves enqueuing or dequeuing messages produces test code for writing to or reading from a socket, respectively. Due to lack of space, we refer to [3] for more details. The generated tests cover all transitions of the model and all interactions between the orchestrator and the services. The code coverage indicates that the 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, the modelling and verification allowed us to discover and fix some issues (see Section 4). This would not have been possible in case the abstraction were too coarse.

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 Figures 1 and Figure 2 succinctly and accurately specify the interaction logic of `CARE`.

3.2 Formal model

The formal model of `CARE` is described. All models used in this paper together with the evaluated formulas are available in [3].

Figure 1 displays the template automaton for the `RunnableOrchestration` (i.e., the orchestrator), while the template automaton for the `RunnableOrchestratedContract` (i.e., the service) is depicted in Figure 2 and has as parameter the `id` of the service. 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 4, in order to perform different analyses. In particular, in Figures 1 and 2 the configurations of choice and action (for both the services and the orchestrator) are instantiated non-deterministically. Another version of the model is also available where the configuration options are also parameters of the templates. The repository [3] 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 [3].

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.

Description of the automata We now describe Figure 1 and Figure 2. The initial states of the automata are depicted with a double circle. 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. 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. 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.

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

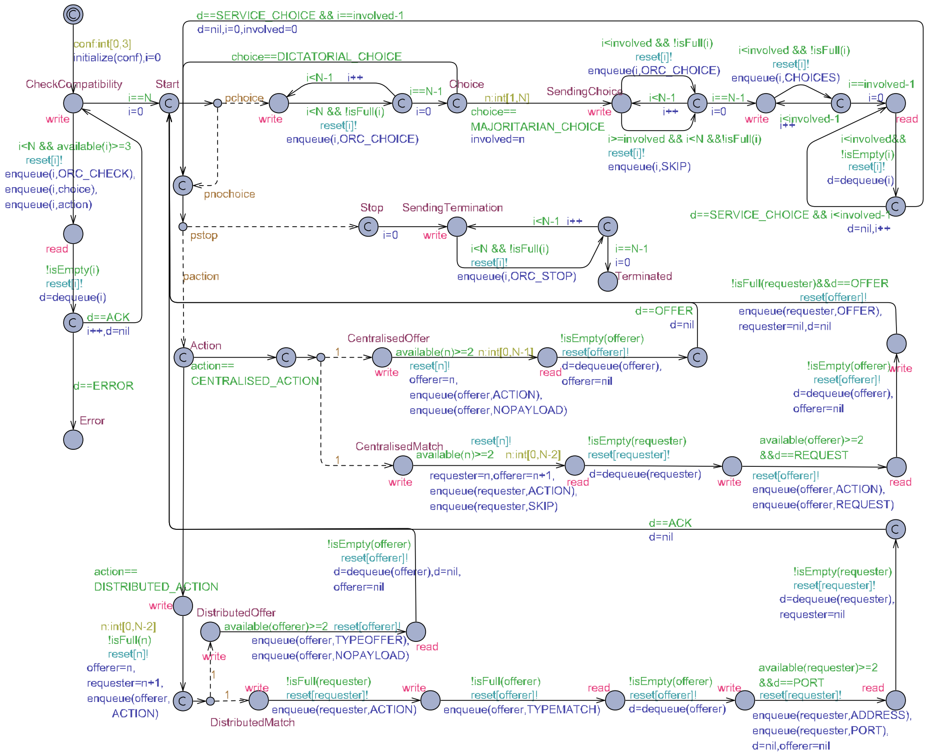

Fig. 1. The `RunnableOrchestration` UPPAAL template

service now replies with its choice, abstracted as a message `SERVICE_CHOICE`. After all involved services have voted, the orchestrator will decide accordingly.

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 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`, followed by the `NOPAYLOAD` message (i.e., there is no payload from the requester) and the offerer replies with a payload abstracted 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

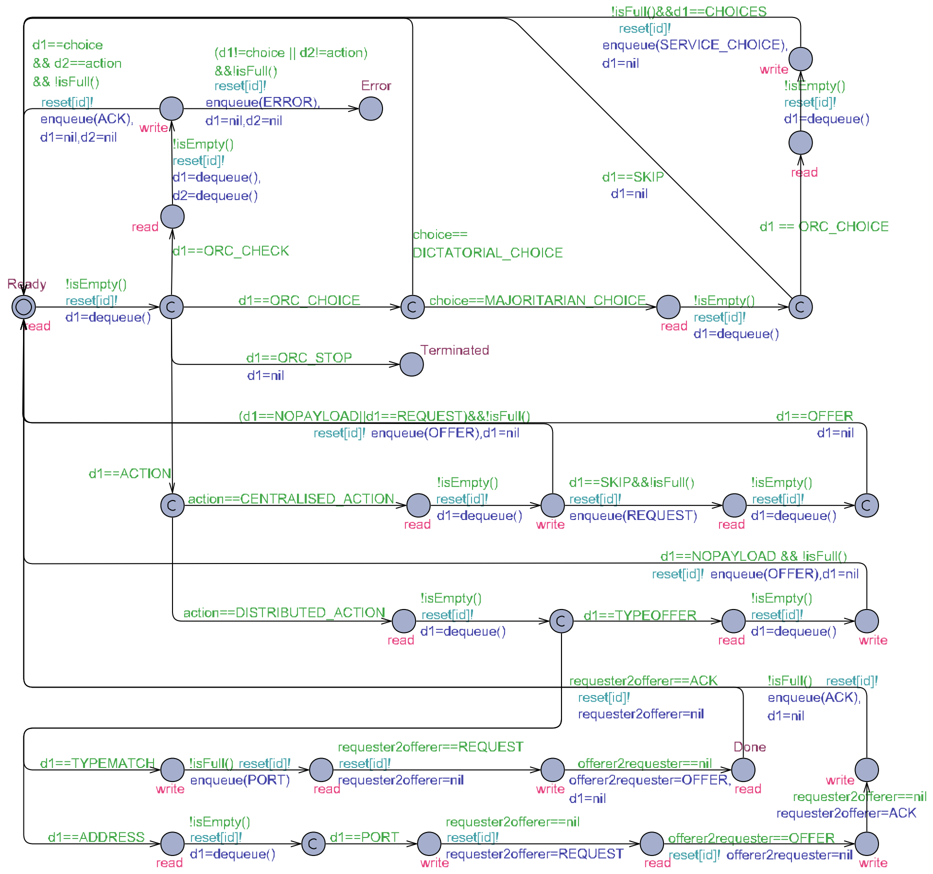

Fig. 2. The `RunnableOrchestratedContract` Uppaal template

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`.

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).

4 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, February 2022.

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 [23]. 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 1, 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`) [4] has been tested up to 100% coverage of all lines and branches, also using mutation testing [4]. In `CATLib`, the lines of code of the tests are more than three times those of its source code [4]. A similar effort for `CARE` is more demanding than the one needed for designing the models in Figures 1 and 2. Moreover, Uppaal has been used to automatically generate tests from the formal model.

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. Subsequently, we move to another variant where the configurations 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 configurations 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 optimization and reusing the generated state space. Logs of the experiments are available in [3].

Parameters Tuning The verification process involves employing specific parameter configurations 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 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 2).

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 configurations that ensure realistic modelling and improve the performances of the exhaustive model checking. Realistic modeling entails accurately representing the behavior 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.

Firstly, the values of the *probabilities weights* `pstop`, `pchoice`, `pnochoice`, and `paction` (see Figure 1) can be tuned based on average values extracted from the orchestrations subject of the analysis. Indeed, as stated in Section 3, the underlying orchestration automaton is abstracted away. Note that, e.g., in location `Start` the probability of performing a choice is $\frac{\texttt{pchoice}}{\texttt{pchoice}+\texttt{pnochoice}}$. For example, the orchestration in Figure 3 (right) can be modelled by tuning the probabilities to `pstop=25`, `pchoice=1`, `pnochoice=0`, and `paction=75`.

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

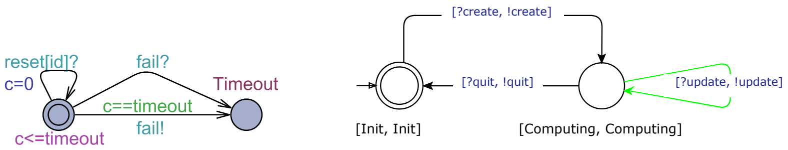

Fig. 3. On the left side, the `SocketTimeout` automaton. On the right side, the orchestration automaton taken from the composition service example in [5]

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 behavior 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 arrays `orc2services[i]` 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.

Next, we consider the real-time behavior 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 `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.

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 configurations of (number of services, buffers size). The first is c1=(4,5), the second is c2=(5,3). In fact, the configuration (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 configurations necessitates either relying exclusively on statistical model checking or employing more powerful machines.

Verification Once the model's parameter configuration 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 (i.e., the probability lies within the interval $[0, 0+\varepsilon]$ with probability $1-\alpha$). 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 3, 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 decides 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 configurations c1 and c2. The first (resp. second) configuration 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) RunnableOrchestratedContract(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 configurations. The first (resp. second) configuration 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 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 [19] and fixed the model as described in Section 3 (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 configurations, as expected. The first (resp. second) configuration 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 configurations. The first (resp. second) configuration 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 3, 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`) (see Figure 2). The probability is found to be nearly zero.

We also include in the repository [3] 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 3 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 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.

5 Conclusion and Future Work

We have presented the formal modeling and verification of the Contract Automata Runtime Environment (`CARE`). This paper witnesses the utility of formal modeling in identifying and addressing issues within the source code of a distributed application. Notably, `CARE` is an open-source platform. 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. 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 been employed to fine-tune parameter settings within the formal model, such as the buffer size. The methodology utilized for modeling `CARE` can be applied to other Java-based distributed applications that communicate over TCP/IP sockets, enabling their formal verification with similar rigor and precision. 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. Exploring techniques for automatic alignment of these artifacts is a subject of future research.

References

1. Bartoletti, M., Cimoli, T., Zunino, R.: Compliance in Behavioural Contracts: A Brief Survey. In: Programming Languages with Applications to Biology and Security. LNCS, vol. 9465, pp. 103–121. Springer (2015). https://doi.org/10.1007/978-3-319-25527-9_9

2. Basile, D., ter Beek, M.H., Ciancia, V.: Statistical Model Checking of a Moving Block Railway Signalling Scenario with Uppaal SMC. In: Margaria, T., Steffen, B. (eds.) ISoLA. LNCS, vol. 11245, pp. 372–391. Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_24
3. Basile, D.: Modelling and verifying the Contract Automata Runtime Environment, complementary material. https://doi.org/10.5281/zenodo.8017613
4. Basile, D., ter Beek, M.H.: Contract Automata Library. Sci. Comput. Program. **221** (2022). https://doi.org/10.1016/j.scico.2022.102841, https://github.com/contractautomataproject/ContractAutomataLib
5. Basile, D., ter Beek, M.H.: A runtime environment for contract automata. In: Chechik, M., Katoen, J., Leucker, M. (eds.) Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, Proceedings. LNCS, vol. 14000, pp. 550–567. Springer (2023). https://doi.org/10.1007/978-3-031-27481-7_31, https://github.com/contractautomataproject/CARE
6. Basile, D., ter Beek, M.H., Degano, P., Legay, A., Ferrari, G.L., Gnesi, S., Di Giandomenico, F.: Controller synthesis of service contracts with variability. Sci. Comput. Program. **187** (2020). https://doi.org/10.1016/j.scico.2019.102344
7. Basile, D., ter Beek, M.H., Ferrari, A., Legay, A.: Exploring the ERTMS/ETCS full moving block specification: an experience with formal methods. Int. J. Softw. Tools Technol. Transf. **24**(3), 351–370 (2022). https://doi.org/10.1007/s10009-022-00653-3
8. Basile, D., ter Beek, M.H., Legay, A.: Strategy Synthesis for Autonomous Driving in a Moving Block Railway System with Uppaal Stratego. In: Gotsman, A., Sokolova, A. (eds.) FORTE. LNCS, vol. 12136, pp. 3–21. Springer (2020). https://doi.org/10.1007/978-3-030-50086-3_1
9. Basile, D., ter Beek, M.H., Pugliese, R.: Synthesis of orchestrations and choreographies: Bridging the gap between supervisory control and coordination of services. Log. Methods Comput. Sci. **16**(2), 9:1–9:29 (2020). https://doi.org/10.23638/LMCS-16(2:9)2020
10. Basile, D., Degano, P., Ferrari, G.L.: Automata for specifying and orchestrating service contracts. Log. Methods Comput. Sci. **12**(4), 6:1–6:51 (2016). https://doi.org/10.2168/LMCS-12(4:6)2016
11. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: Proceedings 3rd International Conference on the Quantitative Evaluation of SysTems (QEST). pp. 125–126. IEEE (2006). https://doi.org/10.1109/QEST.2006.59
12. Boulanger, J.L.: Tool Qualification. In: CENELEC 50128 and IEC 62279 Standards, chap. 9, pp. 287–308. John Wiley & Sons (2015). https://doi.org/10.1002/9781119005056.ch9
13. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B.: Uppaal SMC tutorial. Int. J. Softw. Tools Technol. Transf. **17**(4), 397–415 (2015). https://doi.org/10.1007/s10009-014-0361-y
14. Ferrari, A., ter Beek, M.H.: Formal Methods in Railways: A Systematic Mapping Study. ACM Comput. Surv. (2022). https://doi.org/10.1145/3520480
15. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 Expert Survey on Formal Methods. In: ter Beek, M., Ničković, D. (eds.) FMICS. LNCS, vol. 12327, pp. 3–69. Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_1
16. Gay, S., Ravara, A. (eds.): Behavioural Types: from Theory to Tools. River (2017). https://doi.org/10.13052/rp-9788793519817

17. Gu, R., Jensen, P.G., Poulsen, D.B., Seceleanu, C., Enoiu, E., Lundqvist, K.: Verifiable strategy synthesis for multiple autonomous agents: a scalable approach. Int. J. Softw. Tools Technol. Transf. **24**(3), 395–414 (2022). https://doi.org/10.1007/s10009-022-00657-z
18. Horváth, B., Molnár, V., Graics, B., Hajdu, A., Ráth, I., Horváth, A., Karban, R., Trancho, G., Micskei, Z.: Pragmatic verification and validation of industrial executable sysml models. Systems Engineering (2023). https://doi.org/10.1002/sys.21679
19. https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html
20. Lamport, L., Shostak, R.E., Pease, M.C.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982). https://doi.org/10.1145/357172.357176
21. Legay, A., Lukina, A., Traonouez, L., Yang, J., Smolka, S.A., Grosu, R.: Statistical Model Checking. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science: State of the Art and Perspectives, LNCS, vol. 10000, pp. 478–504. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_23
22. Lehmann, S., Rogalla, A., Neidhardt, M., Reinecke, A., Schlaefer, A., Schupp, S.: Modeling $\mathbb{R}^3$ Needle Steering in Uppaal. In: Dubslaff, C., Luttik, B. (eds.) Proceedings of the 5th Workshop on Models for Formal Analysis of Real Systems (MARS). EPTCS, vol. 355, pp. 40–59 (2022). https://doi.org/10.4204/EPTCS.355.4
23. Mazzanti, F., Basile, D.: 4SECURail Deliverable D2.2, Formal development demonstrator prototype, 1st release, https://www.4securail.eu/pdf/4SR-WP2-D2.2-Formal-development-demonstrator-prototype-1st%20release-CNR-3.0.pdf
24. https://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html
25. Orlando, S., Pasquale, V.D., Barbanera, F., Lanese, I., Tuosto, E.: Corinne, a Tool for Choreography Automata. In: Salaün, G., Wijs, A. (eds.) FACS. LNCS, vol. 13077, pp. 82–92. Springer (2021). https://doi.org/10.1007/978-3-030-90636-8_5
26. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987). https://doi.org/10.1137/0325013
27. Roggenbach, M., Cerone, A., Schlingloff, B., Schneider, G., Shaikh, S.A.: Formal Methods for Software Engineering: Languages, Methods, Application Domains. TTCS, Springer (2022). https://doi.org/10.1007/978-3-030-38800-3
28. Shokri-Manninen, F., Vain, J., Waldén, M.: Formal Verification of COLREG-Based Navigation of Maritime Autonomous Systems. In: de Boer, F.S., Cerone, A. (eds.) SEFM. LNCS, vol. 12310, pp. 41–59. Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_3