Networks of Real-Time Actors
Schedulability Analysis and Coordination

Mohammad Mahdi Jaghoori[1⋆], Ólafur Hlynsson[2], and Marjan Sirjani[2,3]

[1] CWI, Amsterdam, The Netherlands
[2] Reykjavik University, Iceland
[3] University of Tehran, Iran
`jaghoori@cwi.nl`,{`olafurh05`,`marjan`}`@ru.is`

Abstract. We present an automata theoretic framework for modular schedulability analysis of networks of real-time asynchronous actors. In this paper, we use the coordination language Reo to structure the network of actors and as such provide an exogenous form of scheduling between actors to complement their internal scheduling. We explain how to avoid extra communication buffers during analysis in some common Reo connectors. We then consider communication delays between actors and analyze its effect on schedulability of the system. Furthermore, in order to have a uniform analysis platform, we show how to use UPPAAL to combine Constraint Automata, the semantic model of Reo, with Timed Automata models of the actors. We can derive end-to-end deadlines, i.e., the deadline on a message from when it is sent until a reply is received.

1 Introduction

Schedulability analysis in a real-time system amounts to checking whether all tasks can be accomplished within the required deadlines. In a client-server perspective on distributed systems, tasks are created on a client, sent to the server (e.g., as a message), and then finally performed on the server. A deadline given by the client for a task covers three parts: the network delay until the message reaches the server, the queuing time until the task starts executing, and the execution time. In case a reply is sent back to the client, an end-to-end deadline also includes the network delay until the reply reaches the client and is processed.

In previous work [14–16], we employed automata theory to provide a modular approach to the schedulability analysis of real-time actor models, assuming direct and immediate communication between actors, i.e., zero communication delays. An actor [1,12] (à la Rebeca [25]) is an autonomous entity with a single thread of execution. Actors communicate by asynchronous message passing, i.e., incoming messages are buffered and the code for handling each message is defined in a corresponding method. We model each method as a timed automaton [3] where a method can send messages while computation is abstracted in passage of time. In our framework, an actor can define a local scheduler and thus reduce the

⋆ The work by this author is supported by the HATS project (EU FP7-231620).

nondeterminism; a proper choice of a scheduling strategy is indeed necessary to make the actor schedulable.

Section 2 explains a modular way to analyze a system of actors. To be able to do so, the expected usage of each actor is specified in a separate timed automaton, called its *behavioral interface*; this is a contract between the actor and its environment [22], which among other things, includes the schedulability requirements for the actor in terms of deadlines. Every actor is checked individually for schedulability with regard to its behavioral interface. We showed in [15] that schedulable actors need finite buffers; the upper-bound on buffer size can be computed statically. When composing a number of individually schedulable actors, the global schedulability of the system can be concluded from the *compatibility* of the actors [16]. Being subject to state-space explosion, we gave a technique in [16] to test compatibility.

The contribution of this paper is twofold. First in Section 3, we extend the above framework with Reo [4] to enable exogenous coordination of the actors. This provides a separation of concerns between computation and coordination. Reo can be used as a "glue code" language for compositionally building connectors that orchestrate the cooperation between components or services in a component-based system or a service-oriented application. An important feature of Reo is that it allows for anonymous communication, i.e., the sender of a message does not need to know the recipient; instead the Reo connector will forward the message to the proper receiver.

With Reo, individually schedulable actors can be used as off-the-shelf modules in a wider variety of network structures. This requires a new compatibility check for our analysis that incorporates the Reo connectors. Our extension preserves the asynchronous nature of the actors, therefore the Reo connectors must have a buffer at every input/output node, which may lead to state-space explosion. To avoid this problem, we provide techniques to optimize the analysis by reusing internal actor buffers in the Reo connectors that are single-input and/or single-output. We show that in this approach the upper-bound on the size of the buffers of the schedulable actors need not be increased. In Section 5, we give examples of other Reo connectors that can take advantage of the same optimization technique. In any case, we assume coordination and data flow by Reo happens in zero time.

As our second contribution, we analyze in Section 4 the effect of communication delays on the schedulability of a distributed system. For simplicity in presentation, we assume no coordination with Reo in this section. The communication medium between every pair of actors is modeled abstractly by a fixed delay value, called their *distance*. We first describe how to implement the effect of delay on messages in an efficient manner with respect to schedulability analysis. Secondly we extend the compatibility check to take message delays into account. The latter is non-trivial because sending and receiving messages do not happen at the same time any more. Nevertheless, this complication can be hidden from the end user by implementing it in an automatic test-case generation algorithm.

We argue in Section 5 that coordination with Reo and communication delays are orthogonal and can be combined.

As a running example, we consider a client/server composition of two actors. Assuming that the client is faster, the overall system would not be schedulable because the server would not be able to respond in time. This situation can be remedied by using Reo to connect the client to multiple server instances in order to compensate for their slowness. Nonetheless, the client still thinks it is communicating with one server, i.e., coordination is transparent to the client and the server actors. In other words, modularity of the analysis is preserved.

1.1 *Related Work*

Schedulability has usually been analyzed for a whole system running on a single processor, whether at modeling [2, 10] or programming level [7, 18]. We address distributed systems modeled as a network of actors (connected by Reo circuits) where each actor has a dedicated processor and scheduling policy.

The work in [11] is also applicable to distributed systems but is limited to rate monotonic analysis. Our analysis being based on automata can handle non-uniformly recurring tasks as in Task Automata [10]. In Task automata, however, a task is purely specified as computation times and cannot create sub-tasks.

In our approach, behavioral interfaces are key to modularity. A behavioral interface models the most general message arrival pattern for an actor. The behavioral interface can be viewed as a contract, as in 'design by contract' [22], or as a most general assumption in modular model checking [20] (based on assume-guarantee reasoning). Schedulability is guaranteed if the real use of the actor satisfies this assumption.

RT-Synchronizers [23] also provide some sort of coordination among actors, however, they are designed for declarative specification of timing constraints over groups of untimed actors. Therefore, they do not speak of schedulability of the actors themselves; in fact, a deadline associated to a message is for the time before it is executed and therefore cannot deal with the execution time of the task itself or sub-task generation.

In [9, 14], our approach is extended to accommodate synchronization statements and *replies* of the Creol language [17]. Asynchronous message passing in Creol is augmented with explicit return values and message synchronization. Therefore, Creol has the natural means to model end-to-end deadlines, however the work in [9, 14] does not support network delays. In present work, an end-to-end deadline including network delays can be computed manually by adding up the deadlines of the message and its corresponding reply message.

There are several coordination languages that can be used to coordinate actors, two of which are worth mentioning. First there is the ARC model [24], which aims at coordinating resource usage and QoS goals, and is based on state transition systems. Secondly there is the PBRD model [21], which aims at logical communication behavior, and is based on rewriting logic. Apart from modeling capabilities, unlike the two above, Reo has automata based semantics which allows us to connect naturally to our automata-theoretic framework in [15].

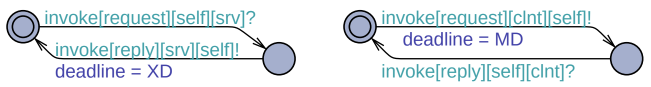

invoke[request][self][srv]?
invoke[reply][srv][self]!
deadline = XD
invoke[request][clnt][self]!
deadline = MD
invoke[reply][self][clnt]?

Fig. 1. The behavioral interfaces of Client (left) and Server (right) are symmetric.

Several semantic models have been suggested for Reo in order to handle data-transfer delays, e.g. [19]. None of these models are yet able to consider the delay in setting up a connection in a distributed way. Therefore in this work, we restrict to centralized Reo connectors and we assume that coordination happens in negligible time. This assumption is reasonable when Reo connectors are deployed local to actors. In this paper, we provide no real-time extensions of Reo; although we propose an algorithm to translate some Reo connectors into Timed Automata.

2 Preliminaries: Real-Time Actors

We use automata theory for modular schedulability analysis of actor-based systems [15, 16]. An actor consists of a set of methods which are specified in Timed Automata (TA) [3]. This enables us to use existing tools, for example Uppaal [6], to perform analysis. Each actor should provide a behavioral interface that specifies at a high level, and in the most general terms, how this actor may be used. As explained later in this section, behavioral interfaces are key to modular analysis of actors. Actors specify local scheduling strategies, e.g., based on fixed priorities, earliest deadline first, or a combination of such policies. Real-time actors may need certain customized scheduling strategies in order to meet their QoS requirements. We describe in this section how to model and analyze actors.

Modeling behavioral interfaces A behavioral interface consists of the messages an actor may receive and send; thus it provides an abstract overview of the actor behavior in a single automaton. A behavioral interface abstracts from specific method implementations, the message buffer in the actor and the scheduling strategy.

To formally define a behavioral interface, we assume a finite global set $\mathcal{M}$ for method names. A behavioral interface B providing a set of method names $M_B \subseteq \mathcal{M}$ is a deterministic timed automaton over alphabet Act^B such that Act^B is partitioned into two sets of actions:

- outputs: $Act^B_O = \{m? | m \in \mathcal{M} \wedge m \notin M_B\}$
- inputs: $Act^B_I = \{m(d)! | m \in M_B \wedge d \in \mathbb{N}\}$

Notice the unusual use of ! and ? signs; this is to simplify the analysis as will be explained later. The integer d associated to input actions represents a deadline. A correct implementation of the actor should be able to finish method m before d time units.

Example. Fig. 1 depicts the Uppaal models for behavioral interfaces of two actors that can communicate in a client-server fashion by sending request and reply

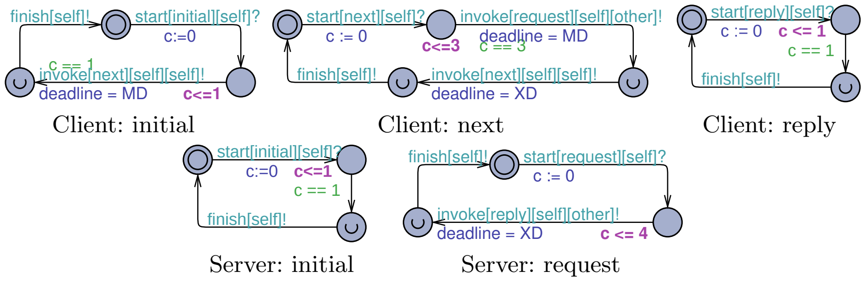


Fig. 2. Method implementations for client and server actors.

messages. In Uppaal, messages are sent along the *invoke* channel and deadlines are passed using the global variable *deadline*. To uniquely identify messages between different actors, every message in $\mathcal{M}$ is represented in Uppaal with three parameters of invoke[msg][snd][rcv] showing the message name, sender and receiver, respectively.

Modeling classes One can define a class as a set of methods implementing a specific behavioral interface. A class R implementing the behavioral interface B is a set $\{(m_1, A_1), \ldots, (m_n, A_n)\}$ of methods, where

- $M_R = \{m_1, \ldots, m_n\} \subseteq \mathcal{M}$ is a set of method names such that $M_B \subseteq M_R$;
- for all i, $1 \leq i \leq n$, A_i is a timed automaton representing method m_i with the alphabet $Act_i = \{m!|m \in M_R\} \cup \{m(d)! \mid m \in \mathcal{M} \land d \in \mathbb{N}\}$;

Method automata only send messages while computations are abstracted into time delays by using a clock c. Receiving and buffering messages is handled by the scheduler automata (explained below). Sending a message $m \in M_R$ is called a self call. A self call with no explicit deadline inherits the (remaining) deadline of the task that triggers it (called delegation); in this case the *delegate* channel must be used.

Classes have an *initial* method which is implicitly called upon initialization and is used for the system startup. Execution of a method begins after receiving a signal on the start channel and terminates by sending a signal on the finish channel; this way the scheduler can control execution of the methods. Fig. 2 shows an implementation of the methods of our example.

Modeling schedulers The scheduler for each actor, containing also its message buffer, is modeled separately as a timed automaton (see Fig. 3). The *buffer* is modeled using arrays in Uppaal and thus it can be modeled compactly, i.e., without different locations for different buffer states. The scheduler automaton begins with putting an *initial* message in the buffer via the *initialize* function.

The scheduler is input-enabled, i.e., it allows receiving any message from any sender on the *invoke* channel. The buffer stores along each message its sender and deadline. A free clock is assigned to each message and reset to zero upon

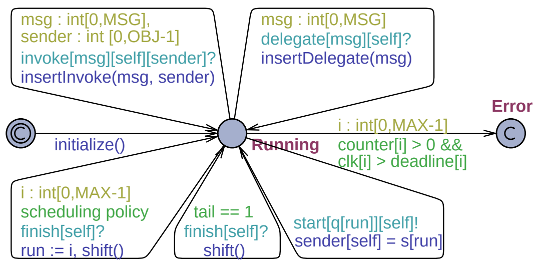


Fig. 3. A general scheduler automaton

insertion in the buffer. These are in the *insertInvoke* function. By reusing this clock, a new message may inherit the remaining deadline of another message; this is captured in the *insertDelegate* function. If a clock assigned to a message ($counter[i] > 0$) passes its deadline, the scheduler moves to an Error location.

When there are multiple messages in the buffer, the scheduler decides the order of their execution. The next method to be executed (via a signal on the start channel) should be chosen based on a specific *scheduling strategy*. If the index 0 of the buffer is always selected during context switch, the automaton serves as a First Come First Served (FCFS) scheduler. The remaining deadline of each message i can be used in the scheduling policy (e.g., Earliest Deadline First) as $deadline[i] - clk[i]$. When a method is finished (via synchronization on the finish channel), it is taken out of the buffer by shifting.

For more details on modeling actors and schedulers, please refer to our previous work [13].

2.1 Modular Schedulability Analysis

An actor is an instance of a class together with a scheduler. A closed system of actors is schedulable if and only if all tasks finish within their deadlines. We have shown in [15] that schedulable actors do not put more than $\lceil d_{max}/b_{min} \rceil$ messages in the buffer, where d_{max} is the longest deadline for the messages and b_{min} is the shortest termination time of its method automata. One can calculate the best case runtime for timed automata as shown by Courcoubetis and Yannakakis [8]. Formally, schedulability is defined as follows.

Definition 1 (System Schedulability). *A closed system of actors is schedulable if and only if none of the scheduler automata can reach the Error location or exceeds the buffer limit of* $\lceil d_{max}/b_{min} \rceil$.

Thus, schedulability analysis can be reduced to reachability analysis in a tool like UPPAAL. The intrinsic asynchrony of actors and their message buffers practically lead to state-space explosion. Our approach to modular analysis of the actors (as in [15]) combines model checking and testing techniques to overcome this problem. This is done in the two steps described below.

Individual actor analysis The methods of an actor can in theory be called in infinitely many ways, which makes their analysis impossible. However, it is reasonable to restrict only to the incoming method calls specified in its behavioral interface. Input actions in the behavioral interface correspond to incoming messages. Incoming messages are buffered in the actor; this can be interpreted as creating a new task for handling that message. The behavioral interface doesn't capture internal tasks triggered by self calls. Therefore, one needs to consider both the internal tasks and the tasks triggered by the behavioral interface, which abstractly models the acceptable environments. We can analyze all possible behaviors of an actor in UPPAAL by model checking the network of timed automata consisting of its method automata, behavioral interface automaton B and a scheduler automaton. Inputs of B written $m!$ match inputs in the scheduler written $m?$, and outputs of B written $m?$ match outputs of method automata written $m!$. An actor is schedulable w.r.t. its behavioral interface iff the scheduler cannot reach the Error location and does not exceed its buffer limit.

Compatibility check Once an actor is verified to be schedulable with respect to its behavioral interface, it can be used as an off-the-shelf component. In this section, we assume that actors communicate directly with no communication delays. As in modular verification [20], which is based on assume-guarantee reasoning, individually schedulable actors can be used in systems *compatible* with their behavioral interfaces. Schedulability of such systems is then guaranteed. Intuitively, the product of the behavioral interfaces, called B, shows the acceptable sequences of messages that may be communicated between actors.

Definition 2 (Compatibility). *Compatibility is defined as the inclusion of the visible traces of the system in the traces of B [16], where visible actions correspond to messages communicated between actors.*

Checking compatibility is prone to state-space explosion due to the size of the system; we avoid this by means of testing techniques. A naive approach could take a trace from the system S as a test case and check whether it exists also in B. This test case generation method is not efficient due to the great deal of nondeterminism in S. As proposed in [16], we generate test-cases from B. A test-case, first of all, *drives* the system along a trace taken from B and thus restricts system behavior. Secondly, it *monitors* the system along this trace checking for any action that is forbidden in B (as a possible witness for incompatibility). To do the monitoring, every communication between different actors has to be intervened by the test-case automaton. Receiving and forwarding these messages in the test-case are separated by a 'committed' location so that UPPAAL executes them with no interruption.

Example. Fig. 4 shows a test-case that proves the Server and Client implementations in Fig. 2 to be incompatible. This test-case considers one round of *expected* request-reply scenario. This scenario is captured in the main line of the test-case (leading to PASS verdict). For the sake of simplicity, we only monitor for one *forbidden* behavior in this test-case which leads to the FAIL verdict: a lack of a

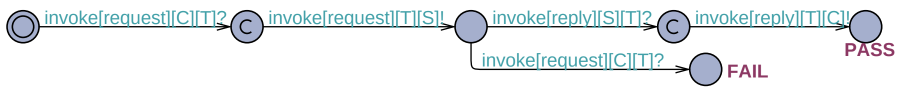


Fig. 4. In this test case, `C`, `S` and `T` represent Client, Server and Test-case, respectively.

timely reply is captured as sending two requests without an intermediate reply. When executing this test-case, the FAIL location is indeed reachable because the client in Fig. 2 (i.e., its 'next' method) is faster than the server (i.e., its 'request' method). We show in Section 3 how Reo can bring flexibility in composing actors such that we can remedy this problem; specifically by allowing us to use two servers with one client.

3 Using Reo for Coordination

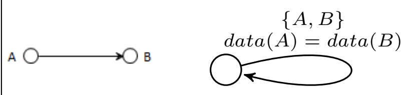
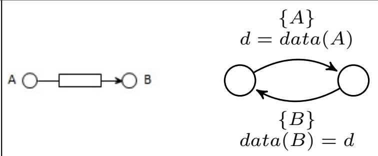
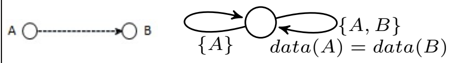
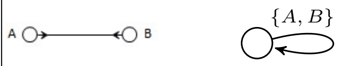
Reo can help us coordinate the actors to avoid unexpected message-passing scenarios. That is, we can impose a strict communication pattern on the components, e.g., replicating requests and merging replies or ordering the messages. This can be seen as an exogenous scheduler that might be crucial in schedulability of a composed system. An advantage of Reo for us is its automata-theoretic semantic model, namely Constraint Automata (CA). The idea is that CA models of Reo networks have a high potential to be used in combination with Timed Automata models of actors and thus allow us to analyze our models in Uppaal.

Complex Reo connectors can be composed out of a basic set of channels. Each channel has exactly two ends that have their own unique identities. A channel end can be a source or a sink. Data enters at the source end and leaves the channel through the sink. To build complex connectors, channels are connected by means of nodes (also called ports). A node is like a pumping station that takes the data on one of the incoming 'sink' ends and replicates the data onto all of its outgoing 'source' ends. Therefore, channels can be connected by: sequential composition where the data flows from one channel to the next one; a non-deterministic choice of data from multiple channels merging to one; or, replication of data from one channel to many. All this happens in one synchronous step.

Table 1 illustrates a set of primitive channels. The synchronous channel accepts data at the source and dispenses data through the sink as soon as both source and sink are ready. The lossy synchronous channel can always accept data at the source. The data flows from the source to the sink if the sink can accept data at that instance; otherwise, it is lost. The synchronous drain has two source ends; it takes the data on its sources if and only if they are both ready. It acts like a channel synchronizer and does not transfer any data. The FIFO1 channel transfers data from the source to the sink in two transitions, thereby acting like a one-place data storage. A FIFO channel can also be unbounded.

Transitions of constraint automata are labeled with a set of port names and a data constraint. A transition is taken when all of the ports on its label are ready. In that case, the data constraint determines the data flow in a declarative

Table 1. Basic Channels and their constraint automata.

 Synchronous channel	 A B $\{A\}$ $d = data(A)$ $\{B\}$ $data(B) = d$ FIFO1 channel with variable d
 Lossy synchronous channel	 Synchronous drain

fashion, e.g., when a synchronous channel fires the data at both ends will be the same. Direction of data flow is understood from the types of channel ends. As in FIFO1, a CA can have variables to temporarily store data values. The initial state of the CA for FIFO1 depends on whether it is initially full or empty.

When channels are composed into a connector, the behavior of the connector is derived compositionally as the product of the CA of its constituent channels. Furthermore, the hiding operator can be applied to create a simple and intuitive CA that accurately describes how the connector works, without exposing the internal ports. Please refer to [5] for a formal definition of product and hiding.

3.1 Integrating Real-Time Actors with Reo

Integrating actors with Reo is complicated by the asynchronous nature of actors: Actors can send messages whenever they have to; therefore, a Reo connector may not block them exogenously. A natural way of solving this issue is to add a FIFO channel as a message buffer at every input port of a Reo connector. The problem is that for model checking, a suitable bound for these FIFOs is necessary. Furthermore, the number of buffers needed quickly blows up the state-space. As a workaround, we suggest using the buffers that already exist in the actors for this purpose. Nevertheless, the upper bound for these buffers need not be increased as discussed below. This approach can be thought of as a low-level optimization of the schedulability check, where we produce a behavior which at a high-level is indistinguishable from adding buffers to the input ports of the connectors. Before explaining the details, we need to restrict the allowable Reo connectors.

A Reo connector may not lose a message. In fact when a message is lost, it can never meet its deadline, and the system will not be schedulable. If we were to allow lossy connectors, one may argue that lost messages can be seen as having met their deadlines; this can be justified by assuming that the Reo connector is in charge and has rightfully decided to lose the message. But this causes a problem if the connector has a buffer to store messages before they are lost (which is the case as explained above). Since we assume that a Reo connector operates in zero time, it may lose any arbitrary number of messages in zero time and therefore, we cannot statically compute a bound on the size of this buffer for a schedulable system. This restriction, however, does not greatly reduce the expressiveness of Reo as witnessed by the examples provided in this section and in Section 5. Notice that drain and lossy synchronous channels can still be used.

Another restriction is that only bounded FIFO channels may be used. Therefore, the CA for these connectors is finite-state. Now we explain how to optimize analysis for two patterns of Reo connectors:

– **Single-input, multiple-output (e.g. Fig. 5.a):** Since the output ports are directly connected to a message buffer in an actor, they are always enabled. Therefore, as soon as there is a message on the input port of this connector, it can decide the destination of the message. Since the connector does not lose the message, it may directly go to an actor or it is stored in a FIFO channel. In either case, we do not need an extra buffer at the input port.
– **Multiple-input, single-output (e.g. Fig. 5.b):** In this case, the destination of all messages is the same, namely the actor connected to the output node. Therefore, we can reuse the buffer of this actor to hold also the messages pending at the input ports. To distinguish these messages from the ones actually in the actor's buffer, these pending messages are flagged so that the actor scheduler cannot select them. This flag will be removed from a message whenever the Reo connector decides that this message can actually be delivered to the recipient actor.

As a consequence, the Reo models do not need to include extra buffers at the input, and rather focus on the coordination logic (cf. Fig. 5). Compared to a normal buffer (as in Section 2, disabling a message only delays its execution, whereas its deadline counts since it is generated. Therefore, as before, a queue with more than $\lceil d_{max}/b_{min} \rceil$ messages is not schedulable. Subsection 3.2 describes how we can implement the above solutions in UPPAAL. Section 5 introduces more patterns in which such optimizations are possible.

Client-Server connectors In our example of client-server we have one client and two servers. The requests and replies between the client and the servers are routed through the connectors shown in Fig. 5. The request sequencer accepts messages from the client through the input port I and routes them to the servers through the output ports O_1 and O_2 in a strict sequence. The reply sequencer accepts messages through input ports I_1 and I_2 and routes them back to the client through output port O, in the order in which they were sent. In both connectors we have a circular configuration of FIFO1 channels, this is to produce an alternating behavior of port selection. For the request sequencing we see that one FIFO1_1 channel is initially full, this causes ports $\{I, x_0, x_1, x_2, f_1, O_1\}$ to become enabled when a message is put on input port I and the request flows through output port 1 O_1. Now the FIFO1_2 channel is full, so for the next request the ports $\{I, x_0, x_3, x_2, f_1, O_2\}$ are enabled for the next message, causing the data to flow through output port O_2. Similarly, for the request sequencing

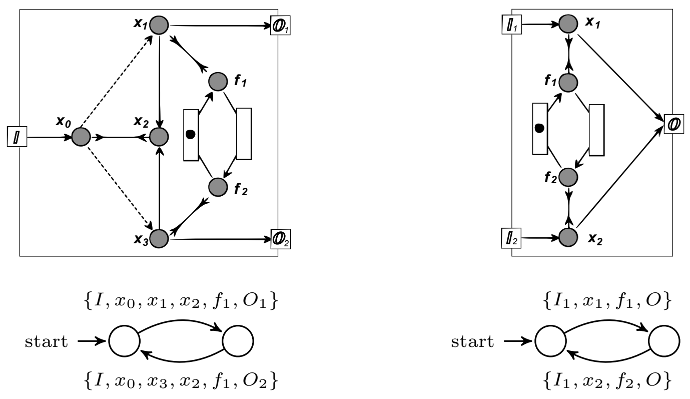


Fig. 5. Request and Reply sequencing.

we have that FIFO1_1 channel is initially full, which forces a strict sequencing on the order in which the replies are put into the buffer of the client. To avoid blocking the input ports I_1 or I_2, in principle we need to add extra buffers on the input ports; this extra buffer is avoided by reusing the buffer of Client as we explained above. In the next section, we show how to implement this in Uppaal. In the sequel, we hide internal ports $\{x_0, x_1, x_2, x_3, f_1, f_2\}$ in the CA models.

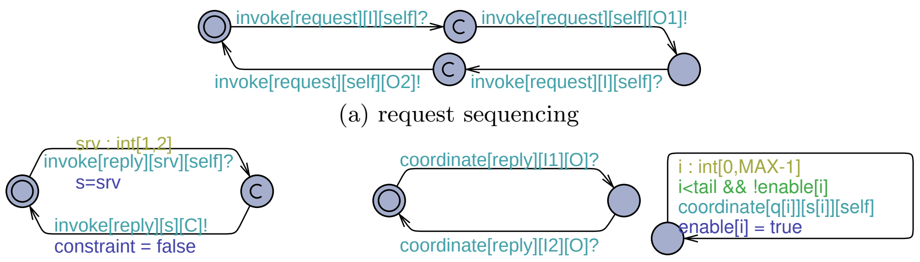
3.2 Analysis in Uppaal

To be able to perform analysis in Uppaal, we need to give a representation of CA in terms of Uppaal timed automata. We work with the CA representing each connector, i.e., after the product of the CA of the constituent channels has been computed. Furthermore, all internal ports should be hidden. Therefore, we are not concerned with composing two translated CA.

The idea is that synchronization on port names can be translated to channel synchronization in Uppaal. We can reuse the *invoke* channel for this purpose. Recall from Section 2 that *invoke* is used for sending messages. An action on an input (resp. output) port is translated to a 'receive' (resp. 'emit') on the channel. Variables in CA can be directly translated to variables in Uppaal, therefore, data constraints can be simply translated to assignments in Uppaal.

The main challenge is that transitions in CA may require synchronization on multiple ports, whereas in Uppaal channels provide binary synchronization. To solve this, whenever multiple ports should synchronize, they are put on consecutive transitions separated by committed locations. This produces an equivalent behavior as these transitions are all taken in zero time and without being interleaved with other automata instances. In the following, we show how to implement the optimizations for the two Reo patterns mentioned previously.

– **Single-input, multiple-output:** In this case (e.g., the request sequencer), the message can immediately be processed and the sender will never be


invoke[request][I][self]?
C
invoke[request][self][O1]!
C
invoke[request][self][O2]!
invoke[request][I][self]?
srv : int[1,2]
invoke[reply][srv][self]?
s=srv
invoke[reply][s][C]!
constraint = false
coordinate[reply][I1][O]?
coordinate[reply][I2][O]?
i : int[0,MAX-1]
i<tail && !enable[i]
coordinate[q[i]][s[i]][self]
enable[i] = true

(a) request sequencing

(b) forwarding the reply

(c) sequencing the reply by manipulating the queue

Fig. 6. Integrating Constraint Auomata into UPPAAL.

blocked. Therefore, the above translation from CA to timed automata is enough and the CA can directly intermediate between the sender and receiver actors. For example in Fig. 6.(a), the synchronous step on I and O_1 is modeled by first reading a request message on I and then writing the message on O_1. Similarly, I and O_2 are synchronized at the next step.

– **Multiple-input, single output:** As explained in previous subsection, actor buffers need to be extended such that every message has a boolean flag called 'enabled'. As long as this flag is false, the message will be not be selected by the scheduler. The extended *insertInvoke* function (cf. Section 2) assigns variable 'constraint' to the 'enabled' field corresponding to every incoming message. The variable 'constraint' is always set to true, except when a message is sent via a "multiple-input, single-output" connector (cf. Fig. 6.(b)). Via this connector, all messages are directly passed on to the buffer of the single receiver with their 'enabled' flag set to false.
Another automaton, shown in Fig. 6.(c), captures the coordination logic, i.e., it has the exact form of the constraint automata for the Reo connector. The second automaton in Fig. 6.(c) is an extension to the scheduler automata which follows the coordination logic to enable messages in the queue. Therefore, these messages are enabled at the moment that is allowed by the CA. In this figure q[i] shows a message at index i of the queue which was sent by s[i]. Note that this automaton selects only disabled messages, i.e., it does not consider a message twice. However, as shown in this figure, it does not distinguish between different instances of the same message. Since every message already has a clock assigned to it which keeps track of how long it has been in the queue, we can use that clock to select the oldest message instance. To do so, we need to extend the guard like this:

```
i < tail && ! enable[i] &&
forall (m : int[0,MAX-1]) (
  enable[m] ||  m>=tail ||
  q[i] != q[m] || s[i] != s[m] ||
  clk[ca[i]]-clk[ca[m]]>=0
)
```

where clk[ca[i]] shows the clock assigned to the message at q[i].

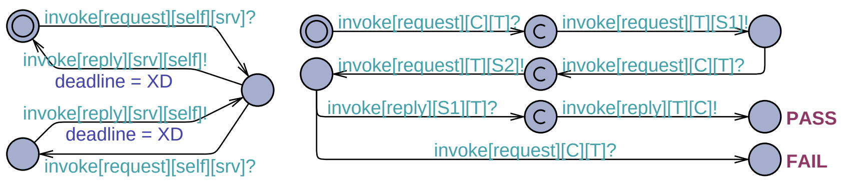


Fig. 7. A client that can send two requests in a row and a corresponding test-case.

Compatibility Check To check the compatibility of actors coordinated using Reo connectors, we need to compose the behavioral interfaces of the actors with the Constraint Automata models of the Reo connectors. This composed automaton will serve as the basis for test case generation. In this composition, we will use the transformed version of the constraint automata into Uppaal format. However, the coordinate channels need to be converted back to invoke channel so that the behavioral interfaces can communicate with them. Note that converting Constraint Automata to Timed Automata can ideally be automated such that these conversions would be safe from human error.

Fig. 7 shows a new behavioral interface for the client that accommodates a late reply by incorporating the possibility of sending two requests in a row. On the right side, a (simplified) test case is shown that is generated from the composition of behavioral interfaces of one client and two servers connected with the sequencer Reo connectors. Compared to the test case in Section 2, this test case can identify two servers S1 and S2. This test case cannot reach the FAIL verdict. This is because before the client wants to send a third request, the servers will provide the replies.

4 Actors with Communication Delays

In this section, we show how to extend the modeling framework of Section 2 and the corresponding schedulability analysis to take account of communication delays between actors. We assume here that actors communicate directly, i.e., there is no Reo connector.

We assume a fixed delay for communications between every pair of actors, called their *distance*. This is a reasonable assumption if the communication medium between the actors is fixed for all messages. Therefore, the delays in the whole network can be modeled as a matrix; this matrix will be symmetric if we assume the uplink and downlink connections have the same properties. For example, for the client-server example, we assume the distance 1 between the client and the server (see Fig. 8). The distance of an actor to itself is then zero.

$$\begin{array}{c c} & \begin{array}{cc} C & S \end{array} \\ \begin{array}{c} C \\ S \end{array} & \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \end{array}$$

Fig. 8. The distance matrix.

Extension of the actor framework with network delays must properly address the following concerns:

1. The time difference since a message is sent and is executed (at receiver) cannot be smaller than the distance between the sender and the receiver.
2. The deadline associated to each message (specified by the sender) should also include the network delay.
3. The modularity of the analysis techniques should be preserved.

A naive solution to handle network delays is to introduce network buffers, e.g., by adding an extra actor. This actor should delay each message exactly as intended and reduce its remaining deadline correspondingly. This, however, introduces a great overhead in the size of the model: there will be at least a buffer (and its corresponding clocks) between every pair of actors in each direction, i.e., an exponential number of buffers and clocks. Additionally, finding a reasonable upper bound on the size of these buffers is not trivial.

To avoid introducing this overhead, we place the messages directly into the buffer of the receiving actors. To model the distances, the messages in the buffer should be *disabled* as long as the network delay has not passed (concern 1). As explained in Section 2, $clk[msg]$ is reset to zero when msg is added to the buffer. With the distance matrix available, we can use this guard:

$$distance[sender][receiver] < clk[msg]$$

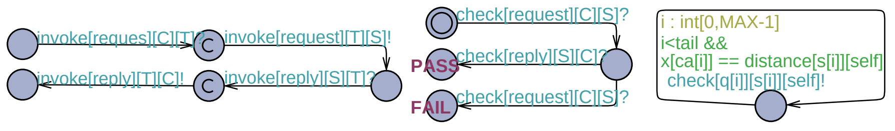
as the enabling condition for each message. Recall that scheduling policies are implemented as guards in the scheduler automata in Uppaal, which model the selection condition of every message. The above enabling condition can therefore be hard coded into this guard. Thus we avoid extra variables in the buffer representation to capture the enabling conditions of messages, which leads to a very efficient implementation. Additionally, using $clk[msg]$ together with the original deadline of the message satisfies the second concern in a straightforward way.

This approach brings about two new concerns:

4. In this approach, the order of messages in a buffer are based on their sending time rather than their arrival time, i.e., when they become enabled.
5. While preserving schedulability, the buffer of every actor needs to be big enough to contain all messages, including disabled messages.

Since messages may arrive from different actors with different distances, multiplexing them into the same buffer should preserve their order of arrival rather than their order of sending. This is important in scheduling strategies that depend on the arrival order of the messages, e.g., FIFO. To address this issue, we need to re-implement such schedulers based on the waiting time of messages after they become enabled, which is equal to $clk[msg] - distance[sender][receiver]$; this value should be used when it is not negative.

Finally, we show that the size of buffer for schedulable actors does not need to be increased in presence of network delays. As argued in previous section, disabling a message may only delay its execution, whereas the deadline associated to all messages (disabled or enabled) is still in effect and approaching. Therefore, if there are $n > \lceil d_{max}/b_{min} \rceil$ messages in the buffer, one of them inevitably



(a) Test-driver (b) Monitor (c) Scheduler extension

Fig. 9. Checking compatibility while considering network delays

misses its deadline. This means that individually schedulable actors can still be used provided that the compatibility check is adapted, i.e., modularity is preserved.

4.1 Compatibility Check

Definition 2 defines compatibility as the inclusion of visible traces of the system S in the traces of B, where B is the composition of the behavioral interfaces. The actions in these traces are instantaneous communication of messages; however, in presence of network delays, communication is not instantaneous any more. The main challenge here is to bridge the time gap between the traces in S which capture the sending times and the traces in B which reflect the arrival times.

Definition 3 (Compatibility with delay). *For every trace from S, say $\sigma = (m_1, t_1) \ldots (m_i, t_i) \ldots$, which captures the sending time of each message, there should exist a corresponding trace $\sigma' = (m_1, t_1 + x_1) \ldots (m_i, t_i + x_i) \ldots$ in B, where x_i is the distance between the sender and receiver of m_i (cf. Fig. 8).*

Furthermore, a deadline on the server side (in the behavioral interface) only includes the buffer time and the execution time, whereas a deadline on the client side (in a method) includes also the network delay. In other words, the compatibility check must ensure that the client side deadline is not smaller than the deadline on the server side plus the distance between the actors.

To check compatibility, as explained in Section 2, we generate test cases from the more abstract side, i.e., the composition of the behavioral interfaces B. A test-case in the original framework [16] both drives the system under test and monitors it for unexpected behavior. These tasks must be separated now: a *test-driver* automaton communicates with the system based on the send times (cf. Fig. 9.(a)); a *monitor* automaton checks whether the arrival time of messages matches the expectations in B (cf. Fig. 9.(b)). The latter is not trivial as the arrival time of a message is when it become enabled. Therefore, the scheduler automata must send a signal on a new channel, *check*, at the actual arrival time of the message, i.e., $clk[msg]$ reaches $distance[sender][receiver]$ (cf. Fig. 9.(c)).

A test-driver is a linear timed automaton generated from a trace taken from B. To be able to drive the system under test, the arrival times must be changed to sending times. As a result, we may need to reorder the transitions of the original trace so that the messages are sent in the correct chronological order.

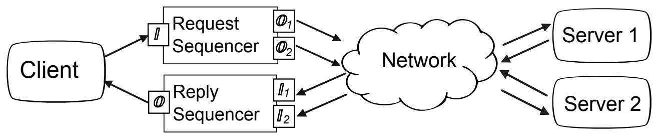


Fig. 10. Graphical illustration of the client-server example.

The monitor automaton is obtained in the same way as in Section 2 when no delays are present. However, it does not drive the system behavior any more. Instead it uses the check channel to see if an actor in the system could receive a message outside the expected time as specified in its behavioral interface. Fig. 9.(b) considers the client/server model in Section 2 where two consecutive request messages are disallowed.

5 Discussion and Future Work

We extended our previous work on schedulability analysis of real-time actors to consider complex networks of actors. On one hand, the coordination language Reo is applied. Reo can be used to take better advantage of off-the-shelf components, where in our case components are modeled as actors. We showed with our simple example that with the help of Reo we can combine actors in such a way that their combination becomes schedulable; in addition, more complicated systems can be built. On the other hand, we showed how to consider communication delays between actors. This is especially important when actors are to be deployed on remote machines.

In an ideal situation, Reo connectors can carry timing information and as such also include the network delays. However, as already mentioned, there is currently no fully satisfactory real-time extension of Reo. As a result, we continue with the assumption that the coordination in Reo connectors happens in negligible time (as in Section 3). Furthermore, we assume that Reo connectors are local to actors. Therefore, the use of a distance matrix as introduced in Section 4 is orthogonal to using Reo. This means that one can directly combine the techniques in the previous two sections to analyze coordinated networks of actors in presence of delays.

In Fig. 10 we illustrate this implementation graphically for our running example. The request and reply sequencing connectors are local to the Client actor. The real delay happens in the network cloud (formally modeled in the distance matrix). By assuming a fixed delay between every pair of actors, we can essentially look at the network as a black-box, i.e. we don't need to know any details about the network, only how long it takes to send messages through the network.

For checking compatibility, we need to generate the separate test-driver and monitor automata because of the delay in the network. Nevertheless, the test cases should be generated from the composition of the behavioral interfaces and the constraint automata models of the Reo connectors, as depicted in Section 3.

Reo Patterns. In this paper, we considered only two patterns of Reo connectors, i.e., single input or single output. Although this may seem a strict restriction on use of Reo, many useful connectors can still be used. Another example of such connectors is shown in Fig. 11.(a). In this example, the client actor requires two services $m1$ and $m2$ (say 'BookFlight' and 'BookHotel') but there is no server actor that can provide both. The connectors in this figure can be used to connect such a client to two servers each providing one of these services. In this connector filter channels are used which may pass the incoming data only if it matches the pattern provided and thus e.g. distinguishing $m1$ and $m2$. The replies from the two servers can be simply merged using a merger as shown in Fig. 11.(b).

Although applying a multiple input multiple output connector may in general require an extra buffer at its input, this can be avoided again in several kinds of connectors, which need to be considered individually. Another example where we can optimize the implementation is a barrier synchronizer, shown in Fig. 11.(c). A barrier synchronizer delays the messages from the fast client actors until all inputs are ready and only then forwards them to their destinations. In this connector, the destination actor for each input port is statically known; therefore, the buffer of that actor can be used to store messages on the respective input port.

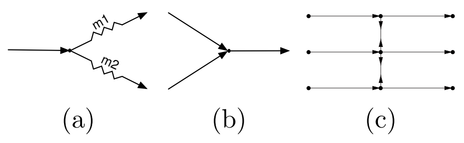


Fig. 11. More Reo connectors

References

1. G. Agha, "The structure and semantics of actor languages," in *Proc. the REX Workshop*, 1990, pp. 1–59.
2. K. Altisen, G. Gößler, and J. Sifakis, "Scheduler modeling based on the controller synthesis paradigm," *Real-Time Systems*, vol. 23, no. 1-2, pp. 55–84, 2002.
3. R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.
4. F. Arbab, "Reo: A channel-based coordination model for component composition," *Mathematical Structures in Computer Science*, vol. 14, pp. 329–366, 2004.
5. F. Arbab, C. Baier, J. J. Rutten, and M. Sirjani, "Modeling component connectors in Reo by constraint automata," in *Proceedings of FOCLASA'03*, ser. ENTCS, vol. 97. Elsevier, 2004, pp. 25–46.
6. G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Proc. Formal Methods for the Design of Computer, Communication, and Software Systems*, ser. LNCS, M. Bernardo and F. Corradini, Eds., vol. 3185, 2004, pp. 200–236.
7. E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine, "TAXYS: A tool for the development and verification of real-time embedded systems," in *Proc. CAV'01*, ser. LNCS, vol. 2102. Springer, 2001, pp. 391–395.
8. C. Courcoubetis and M. Yannakakis, "Minimum and maximum delay problems in real-time systems," *Formal Methods in System Design*, vol. 1, no. 4, pp. 385–415, 1992.

9. F. de Boer, T. Chothia, and M. M. Jaghoori, “Modular schedulability analysis of concurrent objects in Creol,” in *Proc. Fundamentals of Software Engineering (FSEN’09)*, vol. 5961, 2009, pp. 212–227.
10. E. Fersman, P. Krcal, P. Pettersson, and W. Yi, “Task automata: Schedulability, decidability and undecidability,” *Information and Computation*, vol. 205, no. 8, pp. 1149–1172, 2007.
11. J. J. G. Garcia, J. C. P. Gutierrez, and M. G. Harbour, “Schedulability analysis of distributed hard real-time systems with multiple-event synchronization,” in *Proc. 12th Euromicro Conference on Real-Time Systems.* IEEE, 2000, pp. 15–24.
12. C. Hewitt, “Procedural embedding of knowledge in planner,” in *Proc. the 2nd International Joint Conference on Artificial Intelligence*, 1971, pp. 167–184.
13. M. M. Jaghoori, “Time at your service,” Ph.D. dissertation, LIACS, Leiden University, 2010.
14. M. M. Jaghoori and T. Chothia, “Timed automata semantics for analyzing Creol,” in *Proc. 9th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA’10)*, ser. EPTCS 30, 2010, pp. 108–122.
15. M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani, “Schedulability of asynchronous real-time concurrent objects,” *J. Logic and Alg. Prog.*, vol. 78, no. 5, pp. 402 – 416, 2009.
16. M. M. Jaghoori, D. Longuet, F. S. de Boer, and T. Chothia, “Schedulability and compatibility of real time asynchronous objects,” in *Proc. RTSS’08.* IEEE CS, 2008, pp. 70–79.
17. E. B. Johnsen and O. Owe, “An asynchronous communication model for distributed concurrent objects,” *Software and Systems Modeling*, vol. 6, no. 1, pp. 35–58, 2007.
18. C. Kloukinas and S. Yovine, “Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems,” in *Proc. ECRTS’03.* IEEE CS, 2003, pp. 287–294.
19. N. Kokash, B. Changizi, and F. Arbab, “A semantic model for service composition with coordination time delays,” in *Proc. 12th International Conference on Formal Engineering Methods (ICFEM’10)*, ser. LNCS, vol. 6447, 2010, pp. 106–121.
20. O. Kupferman, M. Y. Vardi, and P. Wolper, “Module checking,” *Information and Computation*, vol. 164, no. 2, pp. 322–344, 2001.
21. J. Meseguer and C. L. Talcott, “Semantic models for distributed object reflection,” in *Proc. 16th European Conference on Object-Oriented Programming.* Springer, 2002, pp. 1–36.
22. B. Meyer, *Eiffel: The language.* Prentice-Hall, 1992.
23. S. Ren and G. Agha, “RTsynchronizer: language support for real-time specifications in distributed systems,” *ACM SIGPLAN Notices*, vol. 30, no. 11, pp. 50–59, Nov. 1995.
24. S. Ren, Y. Yu, N. Chen, K. Marth, P.-E. Poirot, and L. Shen, “Actors, roles and coordinators: A coordination model for open distributed and embedded systems,” in *Coordination Models and Languages*, ser. LNCS, 2006, vol. 4038, pp. 247–265.
25. M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, “Modeling and verification of reactive systems using Rebeca,” *Fundamamenta Informaticae*, vol. 63, no. 4, pp. 385–410, 2004.