Analysis of Timing Constraints in Heterogeneous Middleware Interactions

Ajay Kattepur, Nikolaos Georgantas, Georgios Bouloukakis, Valerie Issarny

Analysis of Timing Constraints in Heterogeneous Middleware Interactions

Ajay Kattepur[1], Nikolaos Georgantas[2], Georgios Bouloukakis[2] & Valérie Issarny[2] *

[1] PERC, TCS Innovation Labs, Mumbai, India.
ajay.kattepur@tcs.com
[2] MiMove Team, Inria Paris-Rocquencourt, France.
firstname.lastname@inria.fr

Abstract. With the emergence of *Future Internet* applications that connect web services, sensor-actuator networks and service feeds, scalability and heterogeneity support of interaction paradigms are of critical importance. Heterogeneous interactions can be abstractly represented by *client-service*, *publish-subscribe* and *tuple space* middleware connectors that are interconnected via bridging mechanisms providing interoperability among the services. In this paper, we make use of the *eXtensible Service Bus* (XSB), proposed in the *CHOReOS* project as the connector enabling interoperability among heterogeneous choreography participants. XSB models transactions among peers through generic `post` and `get` operations that represent peer behavior with varying time/space coupling. Nevertheless, the heterogeneous `lease` and `timeout` constraints of these operations severely affect latency and success rates of transactions. By precisely studying the related timing thresholds using *timed automata* models, we verify conditions for successful transactions with XSB connectors. Furthermore, we statistically analyze through simulations, the effect of varying `lease` and `timeout` periods to ensure higher probabilities of successful transactions. Simulation experiments are compared with experiments run on the XSB implementation testbed to evaluate the accuracy of results. This work can provide application developers with precise design time information when setting these timing thresholds in order to ensure accurate runtime behavior.

Keywords: Heterogeneous Services, Middleware Interoperability, Interaction Paradigms, Timed Automata, Uppaal, Statistical Analysis.

1 Introduction

Service Oriented Architectures (SOA) allow heterogeneous components to interact via standard interfaces and by employing standard protocols. Choreographies [4] of such components allow large scale integration of devices (exposed as services) via SOA. However, these principally use the *client-service* interaction paradigm, as for instance, with RESTful services [21]. With the advent of paradigms such as the *Internet of Things* [13] that involve not only conventional

* This work has been partially supported by the European Union's Horizon 2020 Research and Innovation Programme H2020 / 2014-2020 under grant agreement number 644178 (project CHOReVOLUTION, http://www.chorevolution.eu).

services but also sensor-actuator networks and data feeds, additional middleware level abstractions are needed to ensure interoperability.

In particular, heterogeneous platforms, such as REST [21] supporting *client-service* interactions, *publish-subscribe* based Java Messaging Service [20], or JavaSpaces [11] offering a shared *tuple space*, can be made interoperable through middleware protocol converters [15]. In this paper, we use the *eXtensible Service Bus* (XSB) proposed by the *CHOReOS* project[3] [9,12] for dealing with heterogeneous choreographies at the middleware level. XSB prescribes a connector that abstracts and unifies the three aforementioned interaction paradigms: *client-service* (CS), *publish-subscribe* (PS) and *tuple space* (TS). Furthermore, XSB is implemented as a common bus protocol that enables interoperability among services employing heterogeneous interactions following one of these paradigms.

While our previous work [16] studies the effect of heterogeneous choreographies on multi-dimensional end-to-end QoS properties, we now analyze heterogeneous middleware interactions with specific emphasis on timing behavior. We propose a timing model that can represent a system relying on not only any of the CS, PS, TS paradigms, but also any interconnection between them. This model can be used to compare between paradigms, select among them, tune the timing parameters of the overlying application, and also do the previous when interconnection is involved. Our model captures data availability and validity in time with the `lease` parameter, as well as intermittent availability of the data recipients with the `timeout` parameter. Hence, this model allows us to study, in a unified manner, time coupling and decoupling among interacting peers.

We examine the conditions for successful transactions with timed automata [2], and verify reachability and safety properties by employing the Uppaal [6] model-checker. This analysis provides us with formal conditions for successful XSB transactions and their reliance on the `lease` and `timeout` parameters as well as on the stochastic behavior of interacting peers. We further perform statistical analysis through simulation of transactions over multiple runs, and study the success rate and latency trade-off with varying `lease` and `timeout` periods. Simulation outputs are compared with experiments run on the XSB testbed with respect to the accuracy of predicted results. By analyzing the related timing thresholds, we enable designers to leverage the `lease` and `timeout` periods effectively in order to obtain maximal transaction success rates. Moreover, designers can evaluate the impact of interconnecting heterogeneous systems having different timing behaviors, or the impact of replacing a middleware paradigm by another.

The rest of the paper is organized as follows. An overview of heterogeneous interaction paradigms and XSB is provided in Section 2. The model for timing analysis of XSB transactions is introduced in Section 3. This is further refined with timed automata models and verification of properties in Section 4. The results of our analysis through simulation experiments are presented in Section 5, which includes comparison with experiments on the XSB implementation. This is followed by related work and conclusions in Sections 6 and 7, respectively.

2 Interconnecting Heterogeneous Interaction Paradigms

To deal with heterogeneous service choreographies of the *Future Internet*, we make use of the modeling solution proposed in the *CHOReOS* [9] project. While

[3] `http://www.choreos.eu/bin/view/Main/`

Primitives	Arguments
post	mainscope, subscope, data, lease
get	↑mainscope, ↑subscope, ↑data, timeout

Table 1. XSB connector API.

typical service choreographies utilize pure client-service interactions between participants, Future Internet applications require inclusion of service feeds (via publish-subscribe) and sensor-actuator networks (via shared tuple spaces). We briefly review salient properties of these interaction paradigms:

- *Client-Service (CS)* is a commonly used paradigm for web services. A client (`source`) communicates directly with a server (`destination`) either by direct messaging (one-way `send`) or by a remote procedure call (RPC, two-way) through an `operation`. Both synchronous and asynchronous reception of messages (`receive`) are possible at the receiving entity (within a `timeout` period). CS represents tight *space coupling*, with the client and service having knowledge of each other. There is also tight *time coupling*, with service availability being crucial for successful message passing.
- *Publish-Subscribe (PS)* is a commonly used paradigm for content broadcasting/feeds. Peers interact using an intermediate `broker` service; publishers produce (`publish`) events characterized by a specific topic (`filter`) to the broker; subscribers subscribe their interest for specific topics to the broker; and the broker matches received events with subscriptions and delivers a copy of each event to an interested subscriber (`retrieve`) until a `lease` period. PS allows *space decoupling*, as the subscribers need not know each other. Additionally, *time decoupling* is possible, with the disconnected subscribers receiving updates synchronously or asynchronously when reconnected to the broker.
- *Tuple Space (TS)* is commonly used for shared data with multiple read/write users. Peers interact with a *tuple space* (`tspace`) and have write (`out`), `read` and tuple removal (`take`) access to the commonly shared data. Further, peers are able to choose a `template` to select the tuples they procure from the tuple space. TS enables both space and time decoupling between interacting peers.

We employ the *eXtensible Service Bus (XSB) connector* [4], which ensures interoperability across the above interaction paradigms. XSB extends the conventional ESB system integration paradigm [12]. The XSB API is depicted in Table 1, where we only refer to *one-way* interactions; *two-way* interactions are built by combining two of the former. It employs primitives such as `post` and `get` to abstract CS (`send, receive`), PS (`publish, retrieve`), and TS (`out, take/read`) interactions. The `data` argument can represent a CS `message`, PS `event` or TS `tuple`. The `mainscope` and `subscope` arguments are used to unify space coupling (addressing mechanisms) across CS, PS and TS. We employ the ↑ symbol in Table 1 to denote a return argument of a primitive. XSB primitives and arguments can be mapped one-to-one to typical primitives and arguments of CS, PS and TS as shown in Table 2.

We exploit the semantics of the `lease` and `timeout` parameters in each interaction paradigm as follows: `lease` refers to emitted messages, events or tuples, and characterizes both data availability in time, e.g., thanks to storing by a broker, and data validity, e.g., for data that become obsolete as part of a data

[4] `http://xsb.inria.fr/`

Interaction	Native Primitives	XSB Primitives
CS	**send**(destination, operation, message) **receive**(↑source, ↑operation, ↑message, timeout)	**post**(destination, operation, message, 0) **get**(↑source, ↑operation, ↑message, timeout)
PS	**publish**(broker, filter, event, lease) **retrieve**(↑broker, ↑filter, ↑event, timeout)	**post**(broker, filter, event, lease) **get**(↑broker, ↑filter, ↑event, timeout)
TS	**out**(tspace, template, tuple, lease) **take**(↑tspace, ↑template, ↑tuple, timeout) **read**(↑tspace, ↑template, ↑tuple, timeout)	**post**(tspace, template, tuple, lease) **get**(↑tspace, ↑template, ↑tuple, timeout) **get**(↑tspace, ↑template, ↑tuple, timeout)

Table 2. APIs of Interaction Paradigms mapped to XSB Primitives.

feed. Hence, `lease` equals to zero in the client-service paradigm, as shown in Table 2. `timeout` characterizes the interval during which a receiving peer is connected and available. During this active period, the peer can receive one or more sent messages, events or tuples, either synchronously or asynchronously. Between active periods, the peer is disconnected, e.g., for energy-saving or other application-related reason. In the next section, we study in further detail the effect of these parameters on successful interactions.

3 Timing Analysis of Interactions

In this section, we examine the timing thresholds for `timeout`/`lease` periods and their effects on successful data passing in choreography interactions, where data is our generic representation of messages, events and tuples. We examine, in a unified manner, both time (de)coupled and synchronous/asynchronous data passing. This analysis is critical for inter-operating heterogeneous distributed systems, where the designer has to reconcile varying system timing behaviors.

In order to enrich choreography interactions with timing constraints, we focus on *one-way transactions* over a client-service, publish-subscribe, or tuple space connection, abstractly represented by the XSB connector: a transaction represents an end-to-end interaction enabling posting and getting of data. We examine latency increments δ for such transactions. Our analysis considers in particular the "steady state" behavior of publish-subscribe and tuple space interactions. For PS, this means that subscribers are already subscribed and do not unsubscribe during the study period. For TS, this means that peers accessing the tuple space properly coordinate for preventing early removal of data by one of the peers before all interested peers have accessed this data.

In an XSB transaction, a *poster* entity posts data with a validity period `lease`; this data can be procured using `get` within the `timeout` period at the *getter* side. Fig. 1 depicts a XSB transaction as a correlation in time between a `post` operation and a `get` operation. The `post` and `get` operations are independent and have individual time-stamps. We assume that application entities (undertaking the poster and getter roles) enforce their semantics independently (no coordination). The `post` operation is initiated at $t_{\texttt{post}}$. A timer is started also at $t_{\texttt{post}}$, constraining the data availability to the `lease` period $\delta_{\texttt{post-on}}$. Note that the $\delta_{\texttt{post-on}}$ period may be set to 0, as in the case of CS messages. The period when the `lease` period elapses and the next `post` operation is yet to begin is denoted as $\delta_{\texttt{post-off}}$.

Similarly at the getter side, the `get` operation is initiated at $t_{\texttt{get}}$, together with a timer controlling the active period limited by the `timeout` interval, denoted as $\delta_{\texttt{get-on}}$. If `get` returns within the `timeout` period with valid data (not ex-

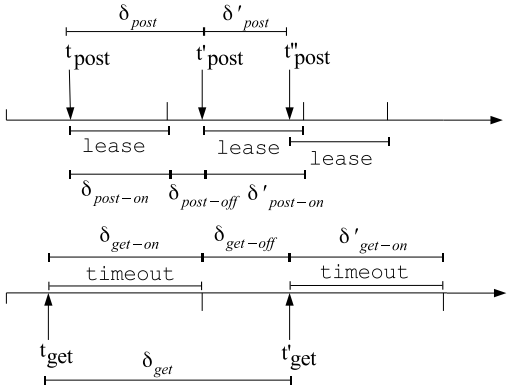

Fig. 1. Analysis of `post` and `get` δ increments.

ceeding the `lease`), then the transaction is successful. We consider this instance also as the end of the `post` operation. `post` operations are initiated repeatedly, with an interval rate $\delta_{\texttt{post}}$ (set as a random valued variable) between two successive `post` operations. Similarly, `get` operations are initiated repeatedly, with a random valued interval equal to $\delta_{\texttt{get}}$ between the start of two successive $\delta_{\texttt{get-on}}$ periods; the interval between `timeout` and the next $t_{\texttt{get}}$ qualifies the disconnection period of receivers ($\delta_{\texttt{get-off}}$). While `lease` and `timeout` are in general set by application/middleware designers, inter-arrival delays $\delta_{\texttt{post}}$ and $\delta_{\texttt{get}}$ are stochastic random variables dependent on multiple factors such as concurrent number of peers, network availability, user (dis)connections and so on.

Note that this model allows concurrent `post` messages; buffers of active receiving entities (including the broker and tuple space) are assumed to be infinite. The data processing, transmission and queueing (due to processing and transmission of preceding data) times inside the transaction are assumed to be negligible (or of the same order in the CS case of $\delta_{\texttt{post-on}} \approx 0$) compared to durations of $\delta_{\texttt{post-on}}$ and $\delta_{\texttt{get-on}}$ periods. In particular regarding queueing, we assume that we have no heavy load effects. This means that: all posts arriving during an active period are immediately served; all posts arriving during an inactive period are immediately served at the next $\delta_{\texttt{get-on}}$ period, unless they have expired before. This corresponds to a G/G/∞/∞ queueing model, where there are an infinite number of on-demand servers, hence there is no queueing. We assume that the general distribution characterizing service times incorporates the disconnections of receivers. Extending this model with actual queueing is part of our ongoing unfinished work.

Successful transactions depend on either of the disjunctive conditions:

$$t_{\texttt{get}} < t_{\texttt{post}} < t_{\texttt{get}} + \texttt{timeout} \tag{1}$$

$$t_{\texttt{post}} < t_{\texttt{get}} < t_{\texttt{post}} + \texttt{lease} \tag{2}$$

meaning that a successful transaction occurs as long as a `post` and a `get` operation overlap in time. Otherwise, there is no overlapping in time between the two operations: only one of them takes place, and goes up to its maximum duration, i.e., `lease` for `post` and `timeout` for `get`. Precisely:

1. If `get` occurs first, and then `post` occurs before `timeout`: the transaction is *successful*. Else, `timeout` is reached, and the `get` operation yields no transaction.

2. If `post` occur first, and then `get` occurs before `lease`: the transaction is *successful*. Else, `lease` is reached, and the transaction is a *failure*.

The above analysis of XSB transactions not only can represent the individual CS, PS, TS interactions, but also any heterogeneous interconnection between them, e.g., a PS publisher interacting with a TS reader. Interconnection is performed through the XSB bus, i.e., an ESB-style middleware implementing the XSB connector. We assume that the effect of the XSB bus on the timings of the end-to-end interactions is negligible.

4 Timed Automata Model

In this section, we build a timed automata model that represents the typical behavior of the XSB connector and of application components using this connector for performing the timed interactions described in the previous section. By relying on the expressive power of timed automata, we are able not only to model the timing conditions of such interactions, but also to introduce basic stochastic semantics for the poster and getter behavior. Using the Uppaal model checker, we provide and verify essential properties of our timed automata model, including formal conditions for successful XSB transactions.

4.1 Timed Automata Model of XSB

A timed automaton [2] is essentially a finite automaton extended with real-valued clock variables. These variables model the logical clocks in the system, which are initialized with zero when the system is started, and then increase synchronously at the same rate. Clock constraints are used to restrict the behavior of the automaton. A transition represented by an edge can be taken only when the clock values satisfy the *guard* labeled on the edge. Clocks may be reset to zero when a transition is taken. Clock constraints are also used as *invariants* at locations represented by vertices: they must be satisfied at all times the location is reached or maintained.

In order to study XSB interactions with timed automata, we make use of Uppaal [6]. Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata. In such networks, automata synchronize via *binary synchronization channels*. For instance, with a channel declared as `chan c`, a transition of an automaton labeled with `c!` (sending action) synchronizes with the transition of another automaton labeled with `c?` (receiving action). Uppaal makes use of computation tree logic (CTL) [10] to specify and verify temporal logic properties. We employ the *committed location* qualifier (marked with a 'C') for some of the locations. In Uppaal, time is not allowed to pass when the system is in a committed location; additionally, outgoing transitions from a committed location have absolute priority over normal transitions. The *urgent location* qualifier (marked with a 'U') is also used: time is not allowed to pass when the system is in an urgent location, either (without the priority clause of committed locations, though).

We represent XSB transactions with the connector roles *XSB poster*, *XSB getter*, and with the corresponding *XSB glue*. The two roles model the behavior expected from application components employing the connector, while the glue represents the internal logic of the connector coordinating the two roles. We detail in the following the modeling of these components.

Fig. 2 shows the *poster* behavior. Typically, a poster entity repeatedly emits a `post!` message to the *glue* without receiving any feedback about the end (successful or not) of the `post` operation. We have enhanced (and at the same time constrained) the poster's behavior with a number of features. The committed locations `post_event` (`post!` sent to the glue) and `post_end_event` (`post_end?` received from the glue) have been introduced to detect the corresponding events. Upon these events, the automaton oscillates between the `post_on` and `post_off` locations, which correspond to the $\delta_{\texttt{post-on}}$ and $\delta_{\texttt{post-off}}$ intervals presented in Fig. 1. `delta_post` is a clock that controls the $\delta_{\texttt{post}}$ interval between two successive `post` operations. `delta_post` is reset upon a new `post` operation and set to `lease` at the end of this operation (note that the `post_init` location and its outgoing transition serve initializing `delta_post` at the beginning of the poster's execution – this unifies verification also for the very first `post` operation). The invariant condition `delta_post<=max_delta_post` (where `max_delta_post` is a constant) at the `post_off` location ensures that a new `post` operation will be initiated before the identified boundary. This setup results in at most one `post` operation active at a time. This `post` remains active ($\delta_{\texttt{post-on}}$ interval) for `lease` time (and then it expires) or less than `lease` time (in case of successful transaction). In both cases, we set `delta_post` to `lease` at the end of the `post` operation (this enables verification, since we can not capture absolute times in UPPAAL). Hence, the immediately following $\delta_{\texttt{post-off}}$ interval will last a stochastic time uniformly distributed in the interval [`lease`, `max_delta_post`]. With regard to the timing model of Section 3, we opted here for restraining concurrency of `post` operations for simplifying the architecture of the glue. The present model (poster, getter and glue) can be compared to one of the infinite on-demand servers of the $G/G/\infty/\infty$ model of Section 3. Nevertheless, this model is sufficient for verifying Conditions (1) and (2) for successful XSB transactions. These conditions relate any `post` operation with an overlapping `get` operation; possible concurrency of `post` operations has no effect on this. Moreover, we will see that these conditions are independent of the probability distributions characterizing the poster and getter's stochastic behavior.

Fig. 3 shows the *getter* behavior. Typically, a getter entity repeatedly emits a `get!` message to the glue, with at most one `get` operation active at a time. The duration of the `get` operation is controlled by the getter with a local `timeout`; upon the `timeout`, a `get_end!` message is sent to the glue. Before reaching the `timeout`, multiple data items (posted by posters) may be delivered to the getter by the glue, each with a `get_return?` message. We have enhanced the getter's behavior with similar features as for the poster. Hence, we capture the events and time intervals presented in Fig. 1 with the `get_event`, `get_end_event`, `get_on`, `get_off` locations, as well as with the `delta_get` clock and the invariant conditions `delta_get<=timeout` (at `get_on`) and `delta_get<=max_delta_get` (at `get_off`). This setup results in a succession of $\delta_{\texttt{get-on}}$ and $\delta_{\texttt{get-off}}$ intervals, with the former lasting `timeout` time and the latter lasting a stochastic time uniformly distributed in the interval [`timeout`, `max_delta_get`]. We have additionally introduced the committed location `no_trans`, which, together with the Boolean variable `get_ret`, helps detecting whether the whole `timeout` period elapsed with no transaction performed or at least one data item was received.

The *glue* automaton is shown in Fig. 4. It determines the synchronization of the incoming `post?` and `get?` operations. A successful synchronization between such

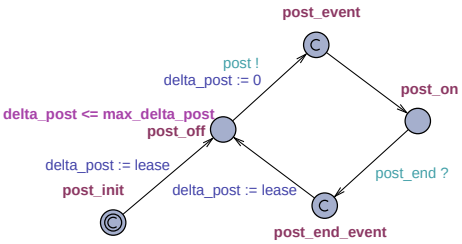

Fig. 2. *XSB poster* automaton.

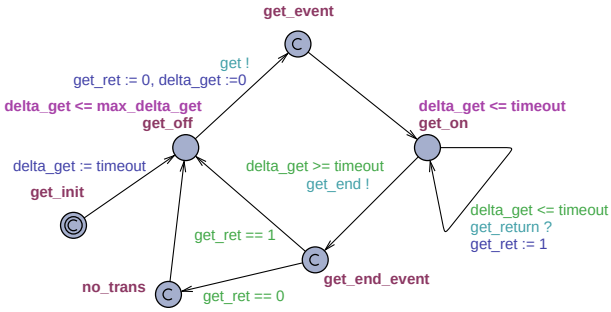

Fig. 3. *XSB getter* automaton.

operations leads to a successful transaction, which is represented in the automaton by the `trans_succ` location. Note that the timing constraints specified in Section 3 regarding the lifetime of posted data have been applied here with the additional clock `delta_post_on` employed to guard transitions dependent on the `lease` period. Two ways for reaching the `trans_succ` location are considered:

- If the `get?` operation occurs from the initial location (leading to location `glue_get`), a consequent `post?` operation results in a `get_return!` message and eventually the successful transaction location `trans_succ` (Eq. 1). At the same time, the poster is notified of the end of the `post` operation with `post_end!`. Note that we employ the *urgent location* qualifier for `glue_get_post`; thus, the glue completes instantly the successful transaction and is ready for a new one. At the `glue_get` location, if the `get_end?` message is received from the getter automaton (suggesting `delta_get >= timeout`), the glue is reset to the initial location `glue_init`.
- If the `post?` operation occurs initially (leading to location `glue_post`), a `get?` operation before the constraint `delta_post_on <= lease` results again in a successful transaction (Eq. 2). Exceeding the `lease` period without any `get?` results in location `trans_fail`, and the automaton returns to its initial location `glue_init`, notifying at the same time the poster with `post_end!`. This is done without any delay, thanks to the invariant `delta_post_on <= lease` at the `glue_post` location.

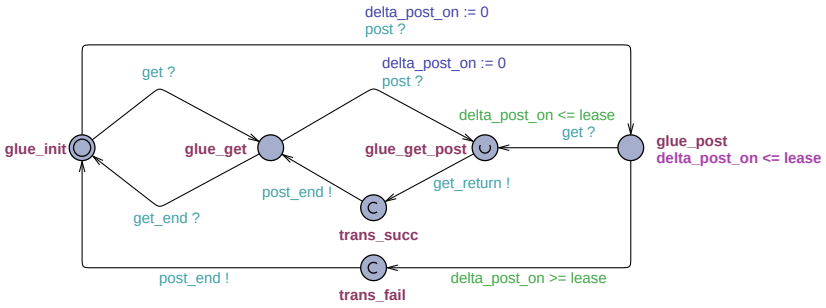

Fig. 4. *XSB glue* automaton.

4.2 Verification of Properties

We verify reachability and safety properties of the combined automata *XSB poster*, *XSB getter* and *XSB glue*, by using the model checker of UPPAAL. A

reachability property, specified in UPPAAL as `E<>`φ, expresses that, starting at the initial state, a path exists such that the condition φ is eventually satisfied along that path. A safety property, specified in UPPAAL as `A[]`φ, expresses that the condition φ invariantly holds in all reachable states.

Poster Automaton. We verify a set of reachability and safety properties that characterize the timings of the poster's stochastic behavior.

```
A[] poster.post_event imply delta_post==0                          (3)

A[] poster.post_on imply delta_post<=lease                         (4)

A[] poster.post_off imply (delta_post>=lease and
                           delta_post<=max_delta_post)             (5)

E<> poster.post_end_event and delta_post<lease                     (6)
```

Eq. 3 states that `post` events occur at time 0 captured by the `delta_post` clock. Eq. 4 and 6 together state that $[0, \texttt{lease}]$ is the maximum interval in which a `post` operation is active, nevertheless, the operation can end before `lease` is reached. Eq. 5 states that $[\texttt{lease}, \texttt{max\_delta\_post}]$ is the maximum interval in which there is no active `post` operation. This confirms the fact that we artificially "advance time" to `lease` at the end of the `post` operation.

Getter Automaton. We verify similar properties that characterize the timings of the getter's stochastic behavior.

```
A[] getter.get_event imply delta_get==0                            (7)

A[] getter.get_on imply delta_get<=timeout                         (8)

A[] getter.get_off imply (delta_get>=timeout and
                          delta_get<=max_delta_get)                (9)

A[] getter.get_end_event imply delta_get==timeout                  (10)
```

Hence, Eq. 7 states that `get` events occur at time 0 captured by the `delta_get` clock. Eq. 8 and 10 together state that a `get` operation precisely and invariantly terminates at the end of the $[0, \texttt{timeout}]$ interval. Eq. 9 states that $[\texttt{timeout}, \texttt{max\_delta\_get}]$ is the maximum interval in which there is no active `get` operation.

Glue Automaton. Finally, we verify conditions for successful transactions using the glue automaton.

```
A[] glue.trans_succ imply (poster.post_on and getter.get_on
                           and (delta_post==0 or delta_get==0))     (11)
```

In addition to the reachability property (`E<> glue.trans_succ`), we verify the safety property in Eq. 11. According to this, a successful transaction event implies that while a `post` operation is active a `get` event occurs, or while a `get` operation is active a `post` event occurs.

```
A[] glue.trans_fail imply (poster.post_on and getter.get_off
          and delta_post==lease and delta_get-timeout>=lease)      (12)
```

In addition to the reachability property (`E<> glue.trans_fail`), we verify the safety property in Eq. 12. A failed transaction event means that `lease` is reached for an active `post` operation and no `get` operation is active. Additionally, the ongoing inactive `get` interval entirely includes the terminating active `post` interval. With regard to the stochastic `post` and `get` processes of our specific setting, we explicitly checked that if the condition `max_delta_get-timeout>=lease` does not hold for the given values of the included constants, then the reachability property `E<> glue.trans_fail` is indeed not satisfied.

```
A[] getter.no_trans imply (getter.get_on and poster.post_off
      and delta_get==timeout and delta_post-lease>=timeout)
```
(13)

In addition to the reachability property (`E<> getter.no_trans`), we verify the safety property in Eq. 13. Symmetrically to Eq. 12, a no-transaction event implies that `timeout` is reached for an active `get` operation and no `post` operation is active. Additionally, the ongoing inactive `post` interval entirely includes the terminating active `get` interval. Similarly to Eq. 12, we check that if this safety property is not satisfied, then the state `getter.no_trans` is indeed not reachable.

Checking Eqs. 11, 12, 13, successful transactions are determined by the durations and relative positions in time of the $\delta_{\texttt{post-on}}$, $\delta_{\texttt{post-off}}$, $\delta_{\texttt{get-on}}$ and $\delta_{\texttt{get-off}}$ intervals. These depend on the deterministic parameter constants `lease`, `timeout` and on the stochastic parameters $\delta_{\texttt{post}}$ and $\delta_{\texttt{get}}$. Nevertheless, Eqs. 11, 12, 13 are expressed in a general way, independently of the specific `post` and `get` processes. Hence, the analysis results of this section provide us with general formal conditions for successful XSB transactions and their reliance on observable and potentially tunable system and environment parameters. Using these results, we perform experiments to quantify the effect of varying these parameters for successful transactions in the next section.

5 Results: Analysis of Timing Thresholds

In this section, we provide results of simulations of XSB transactions with varied `timeout` and `lease` periods. We demonstrate that varying these periods has a significant effect on the rate of successful transactions. In case of choreographies, the trade-off involved between success rates and latency (depending on `timeout`/`lease` periods) is also evaluated.

5.1 Transaction Success Rates

In order to test the effect of varying `lease` and `timeout` periods on transaction success rates, we perform simulations over the timing analysis model described in Section 3. *Poisson* arrival rates are assumed for subsequent t_{post} instances (hence, $\delta_{\texttt{post}}$ follows the corresponding exponential distribution). The data is valid for a deterministic `lease` period and then discarded. Similarly, there are *exponential* intervals between subsequent $t_{\texttt{get}}$ periods ($\delta_{\texttt{get}}$ follows this distribution). The getter entity is active for a deterministic `timeout` period and can disconnect for random valued intervals. Applying the timing model in Section 3, the simulation enables concurrent posts with no-queueing. As the arrivals follow a *Poisson* process, this simulates an M/G/∞/∞ queueing model.

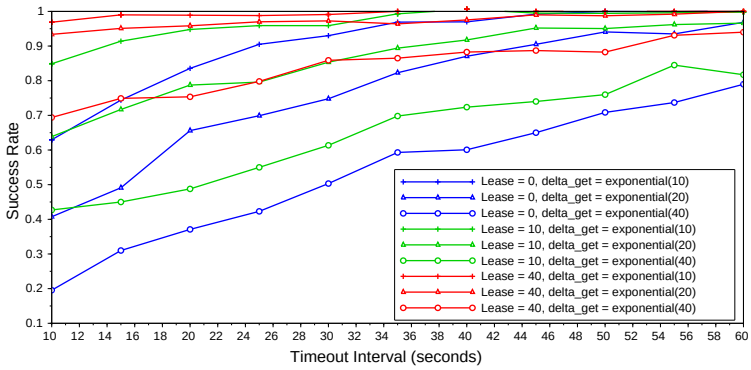

Fig. 5. Transaction success rates with varying `timeout` and `lease` periods.

The simulations done in *Scilab*[5] analyze the effect of varying `lease` and `timeout` periods on XSB transactions. We set $\delta_{\texttt{post}}$ between subsequent `post` messages to have a mean of 10 s. The `get` messages are simulated with varying exponential active periods ($\delta_{\texttt{get}}$). This procedure was run for $10,000$ $t_{\texttt{get}}$ periods to collect transaction statistics, by applying the formal conditions of Section 4.

The rates of successful transactions are shown in Fig. 5 for various values of `lease`, `timeout` and δ_{get} periods. As expected, increasing `timeout` periods for individual `lease` values improves the success rate. However, notice that the success rate is severely bounded by `lease` periods. For time/space coupled CS interactions, where the `lease` period is very low (0 s), the success rate, even at higher `timeout` intervals, remains bound at around 70% for $\delta_{\texttt{get}}$ with mean 40 s. Reducing `get` disconnection intervals (by properly setting $\delta_{\texttt{get}}$ and `timeout`) produces a significant improvement in the success rate, especially for the CS case. For the other interaction paradigms (PS/TS), where the `lease` period can be varied: a higher `lease` period combined with higher `timeout` or lower $\delta_{\texttt{get}}$ intervals would guarantee better success rates.

5.2 Latency vs. Success Rate

In order to study the trade-off between end-to-end latency and transaction success rate, we present cumulative latency distributions for transactions in Fig. 6. Note that we assume that all posts arriving during an active `get` period are immediately served; all posts arriving during an inactive `get` period are immediately served at the next active period, unless they have expired before. All failed transactions are pegged to the value: `lease`.

We set $\delta_{\texttt{post}} = \mathit{Poisson}(10)$ s and $\delta_{\texttt{get}} = \mathit{Exponential}(20)$ s for all simulated cases. From Fig. 6, lower `lease` periods produce markedly improved latency. For instance, with `lease` = 10s, `timeout` = 20s, all transactions complete within 10s. Comparing this to Fig. 5, the success rate with these settings is 78%. Changing to `lease` = 40s, `timeout` = 20s, we get a success rate of 95%, but with increased latency. So, with higher levels of `lease` periods (typically PS/TS), we notice high success rates, but also higher latency. While individual success rates and latency values depend also on the network/middleware efficiency, our analysis

[5] `http://www.scilab.org`

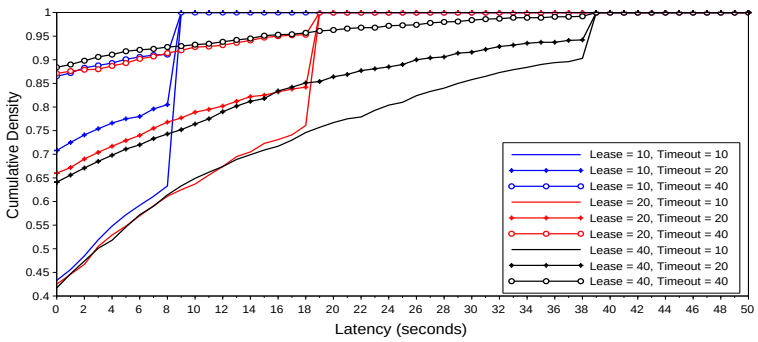

Fig. 6. Latency distributions for transactions with varying `timeout` and `lease` periods.

provides general guidelines for setting the `lease` and `timeout` periods to ensure successful transactions.

We provide in the following an illustrative use case, where our fine-grained timing analysis can be employed to properly configure a concrete application. In a transport information management system based on both authoritative and mobile crowd-sourced information from multiple heterogeneous sources, `posts` carrying events of interest for the average user arrive with a mean rate of 1 event every 10 min. To guarantee the freshness of provided information, notifications are maintained by the system for a `lease` period of 10 min. We assume that users access the system every 20 min on average to receive up-to-date transport information on their hand-held devices. They stay connected for a `timeout` period and then disconnect, also for resource saving purposes. Actual connection/disconnection behavior is based on the user's profile and context at the specific time. By relying on our statistical analysis, an application designer may configure the `timeout` period of user access to 10 min. Using scaled values from Figs. 5 and 6, this guarantees that the user will receive on average 65% of the posted notifications, within at most 8 min of latency with a probability of 0.63. If these values are insufficient and the designer re-configures the `timeout` to 20 min, this guarantees that now the user will receive on average 80% of the posted notifications, within at most 4 min of latency with a probability of 0.77. This technique can be extended to other scenarios, where varying such parameters would provide improvements in performance metrics.

5.3 Comparison with XSB Implementation

In order to validate the simulations performed in Section 5.1, we implement realistic transactions using the XSB framework. Specifically, we use two middleware implementations: i) for $\texttt{lease} = 0$ transactions, the DPWS[6] CS middleware provides an API to set a poster and a getter interacting with each other directly; and ii) for ($\texttt{lease} > 0$) transactions, the JMS[7] PS middleware provides an API to set a poster, a getter, and the intermediate entity through which they interact. Applying the same settings as in Section 5.1, posters and getters perform operations based on probability distributions (exponential $\delta_{\texttt{post}}$ with mean of

[6] `http://ws4d.e-technik.uni-rostock.de/jmeds`
[7] `http://activemq.apache.org`

10 s and $\delta_{\texttt{get}}$ with various mean periods). At the intermediate entity we set various `lease` periods, using the JMS API. Note that in these XSB implementation settings, we have concurrent `post`s and queueing. This corresponds to an M/G/1/∞ queueing model; however, the queueing time of data due to processing of preceding data is negligible in our specific settings. All the transactions

`lease` (s)	$\delta_{\texttt{get}}$ (s)	**Simulation**	**Measurement**
0	`exponential(20)`	0.65	0.717
0	`exponential(40)`	0.35	0.42
10	`exponential(20)`	0.75	0.778
10	`exponential(40)`	0.48	0.554
40	`exponential(20)`	0.93	0.91
40	`exponential(40)`	0.75	0.81

Table 3. Simulated vs Measured Transaction Success Rates.

are performed using an Intel Xeon W3550e 3.08 GHz $\times$ 4 (7.8GB RAM) under a *Linux Mint* OS. For getting reliable results, the mean values of $\delta_{\texttt{post}}$ and $\delta_{\texttt{get}}$ intervals are expected to be close to the expected mean values. To do so, we create sufficient number of `post` operations and `get` connections/disconnections by running each experiment for at least 2 hours. In Table 3, we compare the results of simulated and measured success rates for `timeout` = 20s, $\delta_{\texttt{post}} = Poisson(10)$s, `lease` = $0, 10, 40$s and various $\delta_{\texttt{get}}$. The absolute deviation between the two is no more than 10%. This deviation may be attributed to implementation factors such as network delays and buffering at each entity (poster, getter, intermediate entity) which may affect the success rates. As this deviation is not too high, it allows developers to rely on our simulation model to tune the system.

6 Related Work

With an always increasing number of heterogeneous devices being interconnected among them and with conventional services through the Internet of Things [13], extensions to standard (client-service oriented) ESB-style bridging middleware [8] are required. The XSB connector [12], which resulted from the CHOReOS project [9], explicitly incorporates multiple interaction paradigms, including PS and TS schemes. XSB relies on protocol conversion [19], which allows reasoning about diverse interaction paradigms using the unifying XSB semantics. In our previous work [16], we extended the XSB connector with multi-dimensional QoS metrics that can incorporate timeliness, security and resource efficiency levels. However, we did not consider limited data lifetime, disconnections of peers, or asynchronous reception, as we do in this paper.

Our work upgrades middleware connectors for heterogeneous interaction paradigms with timing analysis. In [24], service composition models are studied where synchronous, asynchronous or parallel interaction may provide superior success rates under time constraints. Similar tuning of time parameters has been applied in distributed real time systems [17] for resource management, while checking end-to-end performance across multiple layers. Besides, middleware-based QoS control has been proposed by [7], where the QoS-aware adaptation and reconfiguration of systems is performed by reflective middleware. In [22], a grid quorum based publish-subscribe system is proposed to deal with delay-sensitive aspects of Internet of Things applications. In comparison, the contribution of our work is a unified timing analysis across heterogeneous middleware paradigms.

Timed automata [2] have been applied to a variety of real time system models to ensure accurate behavior under timed guards. Model checkers such as

Uppaal [6] and PRISM [18] have been proposed for timed and probabilistic properties of such systems. We make use of such tools for design time analysis of heterogeneous middleware interactions. Timed automata are used in [23] for studying fault tolerant behavior (safety, bounded liveness) in distributed asynchronous real timed systems. In [14], the transmission channels of publish-subscribe middleware are modeled using probabilistic timed automata to verify properties of supported interactions. The same authors do model-checking of publish-subscribe applications using Bogor [3] and the PRISM probabilistic model checker [14]. A closely related work is [1], where formal analysis (using colored Petri-Nets) of various types of time synchronization in distributed middleware architectures has been performed. Indeed, alternatives to simulation based approaches, such as statistical model checking [5], may be applied in the context of our work in order to verify, for instance, probabilistic reachability properties. However, simulation techniques are needed as a starting point, in order to elicit distributions needed as inputs to statistical model checkers.

In our paper, we unify the verification of the timing behavior of multiple heterogeneous interactions using timed automata and their statistical analysis. While our prior work focused mainly on the functional interoperability or QoS upgrade of heterogeneous middleware systems, we further model here the fine-grained effect of timing thresholds on both coupled and decoupled distributed systems as well as their combinations. By leveraging the analysis of timing thresholds, designers of heterogeneous choreographies can accurately set constraints to ensure high success rates for transactions.

7 Conclusions

Timing constraints have typically been used for time-sensitive systems to ensure properties such as deadlock freeness and time-bounded liveness. In this paper, we study the XSB interoperable middleware connector from the *CHOReOS* project, by accurately modeling its timing behavior through timed automata. Verification of conditions for successful XSB transactions is done in Uppaal in conjunction with the timing guards specified. We demonstrate that accurate setting of `lease` and `timeout` periods significantly affects the transaction success rate. By providing a fine-grained analysis of the related timing thresholds for designers of choreographies, increased probability of successful transactions can be ensured. This is crucial for accurate runtime behavior, especially in the case of heterogeneous space-time coupled/decoupled interactions with variable connectivity of peers. Furthermore, we demonstrate that the latency vs. success rate tradeoff can be suitably configured for heterogeneous choreographies. Finally, we confirm the sufficient accuracy of our results by comparing with experimental outcomes from the XSB implementation framework.

References

1. L. Aldred, W. M. Aalst, M. Dumas, and A. H. M. Hofstede. On the notion of coupling in communication middleware. In *CoopIS, DOA, and ODBASE*, volume 3761, pages 1015–1033. Springer, 2005.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

3. L. Baresi, C. Ghezzi, and L. Mottola. On accurate automatic verification of publish-subscribe architectures. In *IEEE Intl. Conf. on Software Engineering*, 2007.
4. A. Barker, C. D. Walton, and D. Robertson. Choreographing web services. *IEEE Trans. on Services Computing*, 2:152–166, 2009.
5. A. Basu, S. Bensalem, M. Bozgt, B. Delahaye, and A. Legay. Statistical abstraction and model-checking of large heterogeneous systems. *Int. J. Softw. Tools Techno.l Transfer*, 14:53–71, 2012.
6. G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal 4.0. Technical report, Aalborg University, Denmark, 2006.
7. G. S. Blair, A. Andersen, L. Blair, G. Coulson, and D. Sanchez. Supporting dynamic QoS management functions in a reflective middleware platform. *Proc. IEE Software*, 147(1):2000, 13–21.
8. D. A. Chappell. *Enterprise Service Bus*. O'Reilly Media, 2004.
9. CHOReOS. Final CHOReOS architectural style. Technical report, Large Scale Choreographies for the Future Internet, 2013.
10. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
11. E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Professional, 1999.
12. N. Georgantas, G. Bouloukakis, S. Beauche, and V. Issarny. Service-oriented Distributed Applications in the Future Internet: The Case for Interaction Paradigm Interoperability. In *Euro. Conf. on Service-Oriented and Cloud Computing*, 2013.
13. D. Guinard, S. Karnouskos, V. Trifa, B. Dober, P. Spiess, and D. Savio. Interacting with the SOA-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *IEEE Trans. on Services Computing*, 3:223–235, 2010.
14. F. He, L. Baresi, C. Ghezzi, and P. Spoletini. Formal analysis of publish-subscribe systems by probabilistic timed automata. In *Formal Techniques for Networked and Distributed Systems FORTE 2007*, volume 4574, pages 247–262. Springer, 2007.
15. V. Issarny, A. Bennaceur, and Y.-D. Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In *Formal Methods for Eternal Networked Soft. Sys.*, volume 6659, pages 217–255. Springer, 2011.
16. A. Kattepur, N. Georgantas, and V. Issarny. QoS analysis in heterogeneous choreography interactions. In *Intl. Conf. on Service Oriented Computing*, 2013.
17. M. Kim, M.-O. Stehr, C. Talcott, N. Dutt, and N. Venkatasubramanian. Combining formal verification with observed system execution behavior to tune system parameters. In *Formal Modeling and Analysis of Timed Systems*. Springer, 2007.
18. M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Proc. Tools Session of Aachen Intl. Multiconf. on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 7–12, 2001.
19. S. S. Lam. Protocol conversion. *Software Engineering, IEEE Transactions on*, 14(3):353–362, 1988.
20. M. Richards, R. Monson-Haefel, and D. A. Chappell. *Java Message Service*. O'Reilly, second edition, 2009.
21. L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, 2007.
22. Y. Sun, X. Qiao, B. Cheng, and J. Chen. A low-delay, lightweight publish/subscribe architecture for delay-sensitive IoT services. In *IEEE 20th International Conference on Web Services*, 2013.
23. L. Waszniowski, J. Krakora, and Z. Hanzalek. Case study on distributed and fault tolerant system modeling based on timed automata. *The Journal of Systems and Software*, 82:1678–1694, 2009.
24. T. Zhang, J. Ma, C. Sun, Q. Li, and N. Xi. Service composition in multi-domain environment under time constraint. In *IEEE Intl. Conf. on Web Services*, 2013.