Playing with Time in Publish-Subscribe using a Domain-Specific Model Checker

Luciano Baresi, Giorgio Gerosa, Carlo Ghezzi, and Luca Mottola
Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
{baresi, gerosa, ghezzi, mottola}@elet.polimi.it

ABSTRACT

Thanks to the sharp decoupling it fosters, the Publish-Subscribe paradigm is particularly suited to the implementation of dynamic applications where components join and leave the system unpredictably, and their distributed interactions change over time. Although this feature represents an asset during the implementation phases, it is usually difficult to reason on the global behavior at design time. The problem is exacerbated by the variety of Publish-Subscribe systems available that greatly differ in the guarantees provided, e.g., in terms of message reliability or delivery order.

Some of the authors already tackled the problem with a domain-specific model checker, whose internals are customized depending on the guarantees assumed on the communication infrastructure. However, we essentially disregarded the timing aspects, which are nonetheless pivotal in many applications exploiting a Publish-Subscribe infrastructure. In this paper we augment our tool to verify temporal properties, and explore the interplay between time and different Publish-Subscribe semantics through a case study. Moreover, we report on an effort to formally verify the correctness of the temporal extension, in an attempt to provide a strong foundation for the results obtained using our tool.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms

Modeling, verification, distributed architectures.

Keywords

Model checking, Publish-Subscribe, time.

1. INTRODUCTION

In recent years, the rise of pervasive and embedded applications has increasingly demanded for highly dynamic and reconfigurable software architectures. In these scenarios, the application components require the ability to federate spontaneously, and dynamically change the nature of their interactions as new requirements arise. As a result, traditional architectural paradigms (e.g., client-server) are ill-suited to the requirements at hand. In contrast, the Publish-Subscribe (P/S) [13] paradigm provides an asynchronous, implicit, and multi-point communication style that well adapts to dynamic scenarios. As illustrated in Figure 1, in a P/S system components *subscribe* to specific message (event) patterns, and are *notified* when other components *publish* messages matching their subscriptions. A *dispatcher* mediates the communication by storing subscriptions in a dedicated table, and matching them against published messages. Based on this interaction pattern, P/S systems have been developed for a wide range of scenarios, from wide-area notification services [8] to wireless sensor networks [18].

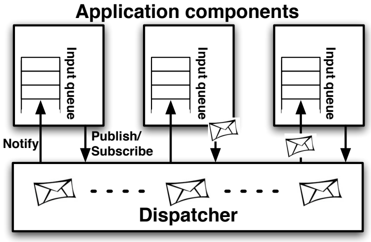


Figure 1: P/S architecture.

Although the P/S paradigm makes it easier to implement dynamic applications, the strong decoupling it fosters renders the global interactions among components difficult to capture and to reason about. This ultimately hinders the verification and validation of the overall federation. Moreover, available P/S systems provide radically different guarantees that may affect the outcome of the verification effort. For instance, different message delivery orderings may have an impact on a component's execution flow, which may reflect in a different system-wide behavior.

To address these issues, model checking has been proposed as a tool to analyze the behavior of applications built on top of P/S infrastructures, e.g., as in [14]. These approaches, however, do not capture many of the guarantees provided by existing P/S systems, thus limiting their applicability. In [2], we proposed a novel approach to the problem: instead of using standard tools, we leverage off the extensible model checker Bogor [19], and augment its input language and internal mechanisms to include P/S operations as primitive constructs. By doing so, we can model the various P/S guarantees *within* the checking engine, and customize the verification based on a specific incarnation of the P/S paradigm. This approach, summarized in Section 2, allows us to achieve a *domain-specific, state abstraction* mechanism, which dramatically reduces the cost of performing the verification.

In this paper, we make a step forward by adding a notion of *time* to our tool. Our temporal model, illustrated in Section 3, is in-

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

Guarantee	**Choices**
Message Reliability	Absent, Present
Message Ordering	Random, Pair-wise FIFO, System-wide FIFO, Causal Order, Total Order
Filtering	Precise, Approximate
Subscription Propagation Delay	Absent, Present
Repliable Messages	Absent, Present
Message Priorities	Absent, Present, Present w/ Scrunching
Queue Drop Policy	None, Tail Drop, Priority Drop

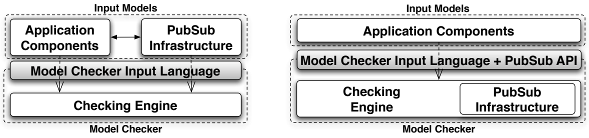
Table 1: P/S guarantees.

spired by the work on real-time event-based middleware by Deng et al. [10]. In their work, however, time was still tied to a particular incarnation of the P/S paradigm, namely the CORBA Event Service. Furthermore, they did not account explicitly for message delays, that may also impact the execution flow of a component. As such, their work cannot be reused as is. In our approach we bring time as an additional dimension next to those we use to characterize the semantics provided by P/S systems, explicitly accounting for message delays. By enabling the interplay between time and the various P/S guarantees, we enable the verification of P/S applications in *realistic* environments, going beyond the simplistic communication models of previous work. The effectiveness of our approach is assessed in a non-trivial case study, illustrated in Section 4, using properties expressed in Linear Temporal Logic (LTL).

To achieve our objective, we must delve into the internals of Bogor to modify critical aspects, such as the inter-component schedule. In doing so, we may run the risk of breaking the checking engine itself, thus producing unsound results. To address this issue, Section 5 illustrates how we formally verified the correctness of our temporal extension using existing tools for software verification. Notably, these are based on Bogor itself. This allowed us to make the verification process feasible, by leveraging off the expertise in Bogor we gained while developing the temporal extension. As we discuss in Section 5, a brute-force approach would instead make the same problem intractable. Brief concluding remarks and directions for future work conclude the paper in Section 6.

2. MODEL CHECKING P/S ARCHITECTURES

The P/S paradigm revolves around a few primitives, which allow application components to interact by publishing messages or issuing (un)subscriptions. Although the programming interface mostly remains the same across different P/S incarnations, the way the paradigm is implemented greatly differs. Table 1 illustrates a classification of P/S guarantees and semantics we found in existing systems. These characterize the features that may impact the behavior of components running on top of such infrastructures, and therefore affect whether a given requirement is actually verified. For instance, *message ordering* refers to the policy used to deliver messages: *random* order, *pair-wise FIFO* order to deliver messages to a given subscriber in FIFO order with respect to publish operations from the same component, *system-wide FIFO* order to deliver messages in the same order as publish operations also across different components, according to the *causality* chain among messages, or *total order* to deliver the same messages in the same order to all components with the same set of subscriptions. The remaining dimensions in Table 1 are thoroughly described in [2]. In the following, we briefly overview how the problem of verifying P/S architectures has been tackled in previous work, and how we addressed the same problem trough a domain-specific model checker.



(a) Standard. (b) Domain-Specific.

Figure 2: Approaches to model checking P/S architectures.

2.1 Approaches Using Standard Tools

Garlan et al. investigated the problem of model checking P/S architectures in [14]. They provide a set of pluggable modules that allow the user to choose one configuration out of a predefined set. Nonetheless, available models are far from fully capturing the different characteristics of existing P/S systems shown in Table 1. Also, application components cannot change their subscriptions at run-time. The same approach is extended in [5] by adding more expressive events, dynamic delivery policies and dynamic event-method bindings. Still, the dispatching mechanism is only characterized in terms of delivery policy (*asynchronous*, *synchronous*, *immediate* or *delayed*). Similarly, some of the authors of this paper addressed similar issues in [21] using the SPIN model checker [17]. The P/S infrastructure is characterized in terms of reliability, message delivery order, and subscription propagation delay. Therefore, several of the dimensions in Table 1 are still missing.

Techniques applicable to specific P/S systems have been considered in [3, 6]. Beek et al. [3] concentrate on the addition of a P/S notification service to an existing groupware protocol. They also show how the P/S paradigm improves the user awareness of the status of a project when used to coordinate a large development team. Caporuscio et al. [6] develop a compositional reasoning technique based on an assume-guarantee methodology. The methodology is applied on a specific case study, i.e., the development a file sharing system on top of the Siena P/S system [8]. These approaches lose generality in that they do not allow the user to customize the checking engine to model various P/S guarantees.

2.2 A Change of Perspective

Standard tools easily show their limitations when it comes to implement fine-grained, customizable models to describe guarantees such as those in Table 1. Essentially, this is due to the lack of parametrization in the input language, and state space explosion problems. Based on this observation, we reverted the traditional approach, by embedding the P/S communication paradigm *within* the model checker, and exporting the P/S API as primitive constructs of the modeling language. This is intuitively illustrated in Figure 2.

This approach provides several advantages over traditional solutions. We can easily customize the state space generation depending on the particular combination of guarantees assumed on the P/S infrastructure. This achieves a *domain-specific, state abstraction* mechanism that sensibly reduces the cost of accomplishing the verification by minimizing the number of states generated. By the same token, it is easy to customize the behavior of the P/S infrastructure. To this end, before starting the verification, the user selects a combination of the guarantees shown in Table 1. For instance, s/he may want to check the behavior of application components while assuming an underlying P/S infrastructure that guarantees causal order and precise filtering. Based on this, our tool instantiates a parametric dispatcher within the checking engine, to model the behavior the user desires. As a nice side-effect, describing the behavior of components running on top of a P/S infrastructure becomes straightforward, as the input language now comprises a set of constructs mimicking the P/S API found in real systems.

```
typealias MessagePriority int (0,9); enum DropPolicy {TAIL_DROP, PRIORITY_DROP}
extension PubSubConnection for polimi.bogor.bogorps.PubSubModule {
  typedef type<’a>;
  expdef PubSubConnection.type<’a> register<’a>();
  expdef PubSubConnection.type<’a> registerWithDropping<’a>(int, DropPolicy);
  actiondef subscribe<’a>(PubSubConnection.type<’a>, ’a -> boolean);
  actiondef publish<’a>(PubSubConnection.type<’a>, ’a);
  actiondef publishWithPriority<’a>(PubSubConnection.type<’a>, ’a, MessagePriority);
  expdef boolean waitingMessage<’a>(PubSubConnection.type<’a>);
  actiondef getNextMessage<’a>(PubSubConnection.type<’a>, lazy ’a);
}
```

Figure 3: Bogor preamble to export the P/S infrastructure.

```
// Message definition
record MyMessage { int value;}
MyMessage receivedEvent := new MyMessage;

// Subscription definition
fun isGreaterThanZero(MyMessage event)
    returns boolean = event.value > 0;

active thread PublisherComponent() {
  MyMessage publishedEvent;
  PubSubConnection.type<MyMessage> ps;

  loc loc0:  // Connection setup
    do {
      ps := PubSubConnection.register<MyMessage>();
    } goto loc1;

  loc loc1:  // Publishing a message
    do {
      publishedEvent := new MyMessage;
      publishedEvent.value := 1;
      PubSubConnection.
          publish<MyMessage>(ps, publishedEvent);
    } return;
}

active thread SubscriberComponent() {
  PubSubConnection.type<MyMessage> ps;

  loc loc0:  // Connection setup and subscription
    do {
      ps := PubSubConnection.register<MyMessage>();
      PubSubConnection.
          subscribe<MyMessage>(ps, isGreaterThanZero);
    } goto loc1;

  loc loc1:  // Message receive
    when PubSubConnection.
      waitingMessage<MyMessage>(ps) do {
      PubSubConnection.
          getNextMessage<MyMessage>(ps, receivedEvent);
    } return;
}
```

Figure 4: Using P/S extensions in a Bogor model.

To assess the feasibility of the approach, we use Bogor [19], an extensible model checker written in Java. With respect to similar tools, Bogor eases the definition of domain-specific constructs in its input language. Additionally, it provides out-of-the-box support for function pointers and dynamic threads, that are pivotal in modeling the dynamic applications we target. Adding further constructs to Bogor requires the developers to prepend a preamble to the Bogor models exploiting the new constructs, and provide one or more Java classes implementing the required semantics.

Figure 3 illustrates the preamble containing the P/S constructs available in our tool. Instead, Figure 4 shows an example use, where two components initially register with the P/S extension within Bogor, as shown in `loc0:`. In a sense, this models the operation of opening a connection to the P/S dispatcher, represented by a dedicated handler returned by the `register` operation. The handler is used to issue (un)subscriptions and publish messages over a specific connection to the dispatcher. The former is accomplished by providing as parameter to `subscribe` a *boolean function* representing the actual subscription, as in `loc0:` for the subscriber component. Notably, this gives the user great flexibility in defining the format of messages and the matching semantics. Instead, publishing is achieved with `publish` or `publishWithPriority`, depending on whether messages have associated priorities. To process incoming messages, a guard statement named `waitingMessage` is provided, which holds true when at least one message is available in the incoming queue. To retrieve the actual message content, we use the `getNextMessage` construct.

The mechanisms underpinning the constructs in Figure 3 focus on guaranteeing a given P/S semantics while reducing the number of states generated during the verification. For instance, consider the processing triggered by a message published: the dispatcher matches the message against the subscriptions issued so far, and delivers its content to a component if *at least one* of its subscriptions matches. However, in the presence of multiple subscriptions, the order in which these are examined is immaterial. Therefore, we can model the subscription table as a *set*, thus avoiding the generation of explicit states when it would make no difference from the application perspective. This already provides improvements over standard tools not equipped with a notion of set. Much greater improvements are achieved in controlling the generation of states representing message routing with particular delivery orderings, and in modeling message duplications. More details can be found in [2].

3. TIME EXTENSION

A large body of work exists in the field of model checking embedded systems with time constraints, e.g., [1]. However, our objective here is *not* to embed a generic notion of time. Being our approach specific to the P/S domain, we rather aim to include a temporal model suited to the requirements of applications built on top of a P/S infrastructure. Additionally, this must be sufficiently lightweight to enable its interplay with the other P/S dimensions in a clear and intuitive manner. Based on these reasons, we adapted the model presented in [10] to suit the aforementioned needs.

3.1 Time Model

Components running on top of P/S infrastructures are usually implemented as *passive* threads executing in an event-driven environment. Thread activation is accomplished implicitly by the P/S infrastructure in delivering a message to a subscribing component. This executes a message handler where further operations are usually performed, e.g., to issue new (un)subscriptions or publish messages. The thread is then suspended again waiting for further messages. In our approach, the *execution rate* of a component dictates the frequency of its operations on the P/S dispatcher, i.e., how many P/S primitives can be executed in a single time unit. Notably, a relevant class of real systems can be similarly modeled, e.g., [12,16]. In addition, in distributed environments it is unreasonable to assume that messages are delivered with zero delays. Therefore, differently

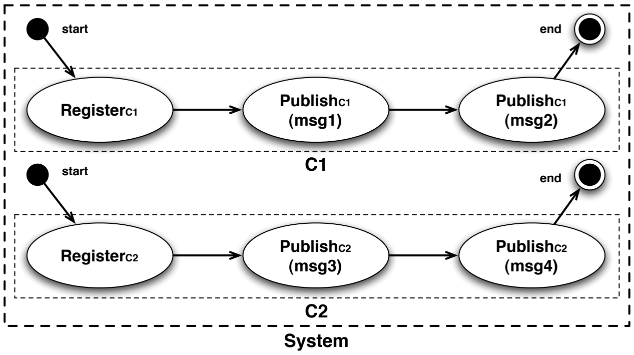
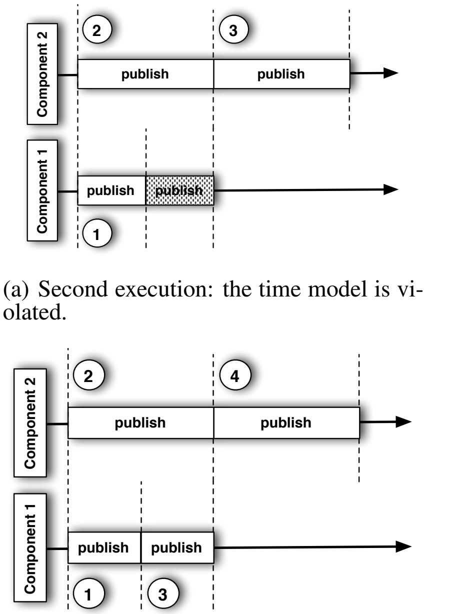


Figure 5: Two components publishing two messages each.



(a) Second execution: the time model is violated.

(b) Third execution: the timing constraints are met.

Figure 6: A graphical representation of the second and third executions in Table 2. (The numbers in the circles represent a possible system-wide schedule).

from [10], we also consider *random message delays*, thus providing an even more realistic environment to modeling P/S applications in case message delays are to be taken into account [7].

The above time model does *not* modify the individual states of the system. Rather, it *limits* the way the system state space is explored, by preventing some of the transitions to be taken. Let us consider the example in Figure 5: two components register with the P/S infrastructure, and publish two messages each. In the absence of any notion of time, our tool would explore all the possible interleavings of the operations of the two components. As the system global state is given by the combination of the per-component local states, the model checker would generate a high number of possible executions. Some example schedules are shown in Table 2.

When we enable our time model with component $C1$ being assigned an execution rate twice as that of $C2$, both executions 1 and 2 in Table 2 become unfeasible. The former trivially violates the timing constraints, as all operations of component $C2$ are executed before $C1$ starts. As for the latter, Figure 6 graphically compares execution 2 and 3: our time model is violated in transitioning from $\langle publish(msg1)_{C1},\ publish(msg3)_{C2}\rangle$ to $\langle publish(msg1)_{C1},\ publish(msg4)_{C2}\rangle$. Indeed, being $C1$ running at twice the rate of $C2$, it should be allowed to perform two operations for each operation executed by $C2$. Instead, Figure 6(a) shows $C2$ performing a further operation while $C1$ has not yet performed the second publish. Also note that Figure 6(b) is not the only possible correct schedule. For instance, a different, correct execution is obtained by swapping the relative order of the initial publish operations. This indeed represents a different inter-leaving of concurrent operations.

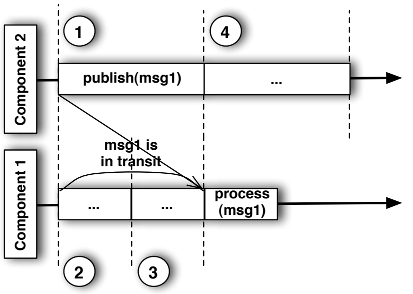


Figure 7: A possible schedule between two components when a message is sent with a non-zero delay.

Message delays are modeled similarly, by marking a message as "in transit" until the time constraints at the subscribing components are met. An example is illustrated in Figure 7: a component whose execution rate is of one operation per time unit publishes a message. This travels towards the subscribing components with a delay of a single time unit. The receiving component, whose execution rate is of two operations per time unit, has two available slots before the message appears in its input queue. During this time frame, it can either perform other operations, or decide to be suspended waiting for its input queue to fill.

3.2 Implementation

Implementing the above time model in our Bogor P/S extension essentially requires the ability to control the inter-component scheduling. To this end, we further augment the P/S preamble in Figure 3, by adding the constructs needed to control how the components proceed, and providing the corresponding semantics within the existing implementation of the P/S extension.

Bogor Language Constructs. Figure 8 illustrates the same example as in Figure 4, now using the additional constructs of our time extension. After registering the connection to the dispatcher, each component configures the time extension using **`configureTimeParams`**. This requires the component execution rate, and two values representing the lower and upper bounds of a (discrete) random delay for incoming messages. In case the user needs to temporarily revert to the original untimed behavior, it is sufficient to set to zero the execution rate of all components.

The inter-component schedule is controlled using two guard statements: **`canProceed`** and **`timedWaitingMessage`**. The former yields true when a component is allowed to proceed without violating any time constraint. Instead, the latter yields one value among **`CAN_PROCEED`**, **`CANNOT_PROCEED`**, and **`QUEUE_EMPTY`**. Note that we explicitly distinguish whether the component cannot proceed because higher priority components must be scheduled first (**`CANNOT_PROCEED`**), or the timing constraints would allow the component to proceed, but no messages are waiting in its input queue (**`QUEUE_EMPTY`**). The latter is needed to give the ability not to lose available slots in the current schedule and perform some operations instead of waiting for an incoming message, as it is possible in the untimed version of the tool.

Id	Execution
1	$\langle start_{C1}, start_{C2}\rangle, \langle start_{C1}, register_{C2}\rangle, \langle start_{C1}, publish(msg3)_{C2}\rangle, \langle start_{C1}, publish(msg4)_{C2}\rangle,$ $\langle start_{C1}, end_{C2}\rangle, \langle register_{C1}, end_{C2}\rangle, \langle publish(msg1)_{C1} end_{C2}\rangle, \langle publish(msg2)_{C1}, end_{C2}\rangle, \langle end_{C1}, end_{C2}\rangle$
2	$\langle start_{C1}, start_{C2}\rangle, \langle register_{C1}, start_{C2}\rangle, \langle register_{C1}, register_{C2}\rangle, \langle publish(msg1)_{C1}, register_{C2}\rangle,$ $\langle publish(msg1)_{C1}, publish(msg3)_{C2}\rangle, \langle publish(msg1)_{C1}, publish(msg4)_{C2}\rangle, \langle publish(msg2)_{C1}, publish(msg4)_{C2}\rangle,$ $\langle end_{C1}, publish(msg4)_{C2}\rangle, \langle end_{C1}, end_{C2}\rangle$
3	$\langle start_{C1}, start_{C2}\rangle, \langle register_{C1}, start_{C2}\rangle, \langle register_{C1}, register_{C2}\rangle, \langle publish(msg1)_{C1}, register_{C2}\rangle,$ $\langle publish(msg1)_{C1}, publish(msg3)_{C2}\rangle, \langle publish(msg2)_{C1}, publish(msg3)_{C2}\rangle, \langle publish(msg2)_{C1}, publish(msg4)_{C2}\rangle,$ $\langle end_{C1}, publish(msg4)_{C2}\rangle, \langle end_{C1}, end_{C2}\rangle$

Table 2: Some of the possible executions for the example in Figure 5, when no time notion is employed.

```
// Message definition
record MyMessage { int value;}
MyMessage receivedEvent := new MyMessage;

// Subscription definition
fun isGreaterThanZero(MyMessage event)
    returns boolean = event.value > 0;

active thread PublisherComponent() {
  MyMessage publishedEvent;
  PubSubConnection.type<MyMessage> ps;

  loc loc0:  // Connection setup
    do {
      ps := PubSubConnection.register<MyMessage>();
      PubSubConnection.configureTimeParams(ps, 2, 1, 0);
    } goto loc1;

  loc loc1:  // Publishing a message
    when (PubSubConnection.canProceed<MyMessage())
    do {
      publishedEvent := new MyMessage;
      publishedEvent.value := 1;
      PubSubConnection.
          publish<MyMessage>(ps, publishedEvent);
    } return;
}

active thread SubscriberComponent() {
  PubSubConnection.type<MyMessage> ps;

  loc loc0:  // Connection setup and subscription
    do {
      ps := PubSubConnection.register<MyMessage>();
      PubSubConnection.configureTimeParams(ps, 1, 1, 0);
      PubSubConnection.
          subscribe<MyMessage>(ps, isGreaterThanZero);
    } goto loc1;

  loc loc1:  // Message receive
    when (PubSubConnection.
      timedWaitingMessage<MyMessage>(ps)==CAN_PROCEED)
      do {
      PubSubConnection.
          getNextMessage<MyMessage>(ps, receivedEvent);
    } return;
    when (PubSubConnection.
      timedWaitingMessage<MyMessage>(ps)==QUEUE_EMPTY)
      do {
        // Do something...
    } return;
}
```

Figure 8: Adding time to the example model in Figure 4.

Bogor Internals. The mechanisms underlying the above Bogor constructs divide time into *frames*, whose length corresponds to a single operation of the lowest priority component. Based on this, higher priority components are scheduled multiple times in a single frame. Within a frame, all possible inter-leavings are generated. When this is achieved, the execution proceeds to the next frame.

Our implementation is tied to that of the P/S operations, to leverage off the *domain-specific semantics* of the executions involved, and cut down on the processing overhead whenever possible. Here we provide some examples as to where we take advantage of this. Interested readers are referred to [15] for more information.

- Several of the dimensions listed in Table 1 also somehow constrain the inter-component schedule. For instance, when *causal order* is assumed, a component is suspended waiting for incoming messages until all causally connected messages are in its input queue. In our experience, the impact of these guarantees on the number of enabled transitions is much greater than that imposed by the time model, especially when safety properties are to be checked. Therefore, whenever possible, we apply the mechanisms modeling the P/S guarantees *before* computing the time-based schedule and running the corresponding checks. This saves in the processing overhead during the verification.

- To model a random message delay between two bounds, we must generate all the possible executions corresponding to each (discrete) value in the interval. However, leveraging off the semantics of this value —which represents the time taken for a message to be transmitted from a component to another— it can be observed that not every value in the interval generates a different execution. Based on this observation, we can apply basic results of rate monotonic theory, and identify the subset of values that need to be checked to ensure the completeness of the verification. This way, we save the processing to generate executions that do not differ in the inter-leavings among components. Note that the above can be done while the verification proceeds, by looking at the execution rate and current state of the components about to receive the message in transit.

- When **timedWaitingMessage** returns **QUEUE_EMPTY**, the corresponding component already passed the time checks. Unless the component includes some alternative behaviors (as in Figure 8), it is suspended waiting for incoming messages. In this case, the checking engine lets another component proceed, and reschedules the suspended component immediately after, without re-running the time extension. Note that this is semantically correct because once a component passed the time checks for the current frame, there is no way for another component to subtract an allocated time slot from it. Based on this observation, we alleviate the processing overhead generated during the verification whenever a message is to be received.

Our tool performance is such that time-related properties can be checked within reasonable time under realistic assumptions on the P/S infrastructure, as illustrated next.

4. CASE STUDY

In this section, we describe the application scenario we have chosen to exemplify the approach, discuss the insights we gained by exploring the interplay between time and the various P/S guarantees, and report on some performance results assessing the effectiveness of our tool.

Scenario. Systems exploiting a P/S style of interaction span several applications domains. Among them, telemedicine is one of the most promising, as it has the potential to drastically lower the costs of maintaining hospital facilities, while letting patients enjoy better quality of life [20]. Here we consider a remote patient monitoring system, consisting of the following components:

- A variable number of *patients*, equipped with several sensing devices to monitor critical parameters, such as heart rate or blood pressure.
- The *medical laboratory*, responsible for monitoring the patients' status. In case of moderate danger, the lab personnel can immediately decide on corrective actions when no physical intervention is required. For instance, a dose change for a treatment can be remotely communicated to the patient.
- If the patient is in severe danger and is to be picked up by a first-response team, the medical laboratory informs a *flying squad* about the emergency, communicating all the relevant information to reach the patient and cope with the situation.
- In the same conditions, the medical laboratory also notifies the *hospital* about a possible request for hospitalization. On the way to it, the flying squad also keeps the hospital posted about the patient's current conditions, until a final notification is sent when the patient is handed over to the hospital personnel.

Interactions are expressed in terms of P/S operations. Specifically, the medical laboratory issues a subscription to collect the data gathered by the patient's sensors when the values are outside the allowed ranges. The hospital, as well as the flying squad, subscribe to all messages regarding possible requests for hospitalization. In addition, the hospital is also interested in messages coming from the flying squad while carrying a patient.

Essentially, three patterns of interactions characterize our scenario, depending on the patient's status. Under *normal conditions*, the component modeling the patient periodically publishes messages that, by virtue of not representing any possible danger, are not delivered to any remote component. When the patient parameters represent a *moderate danger*, the medical laboratory interacts with the patient only, e.g., to adjust the doses, without involving any further component. Differently, under *severe danger*, all the components in the system are involved until the patient's responsibility is passed to the hospital personnel.

Running the Verification. To verify our initial design, we checked whether the following requirements were satisfied:

Requirement 1 When a patient's status turns into a situation of moderate danger, any corrective action must be communicated by the medical laboratory within $T1$ time units.

Requirement 2 Whenever a patient is in a situation of severe danger, the hospital must receive a request for hospitalization within $T2$ time units.

Requirement 3 When a patient arrives at the hospital, the personnel there must have received the corresponding request for hospitalization in advance.

The above requirements are straightforwardly expressed as LTL formulae over the variables representing the components' current state[1]. In particular, the first and second requirement exploit a hook into our time extension that allows properties to be expressed depending on time intervals.

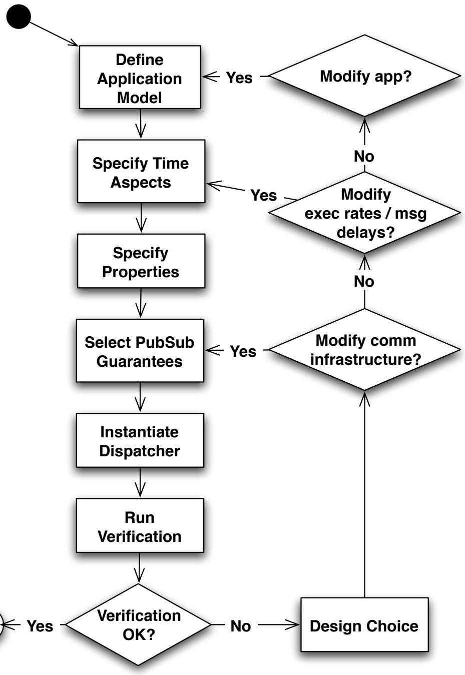


Figure 9: Verification flow with Bogor and our P/S extension.

As illustrated in Figure 9, our tool allows the application designer to iterate in a loop where either the application model evolves, the timing aspects are tuned, or the guarantees assumed on the P/S infrastructure change. This allows the designers to explore the interplay between the application and the underlying communication infrastructure. Further, time adds another dimension to this, enabling an additional degree of freedom.

For instance, we realized that in our application the characteristics of the input queues and the component execution rates are tightly intertwined. Indeed, the first requirement easily fails if the component modeling the medical laboratory is not assigned an execution rate sufficient to handle multiple concurrent notifications coming from different patients. However, even if this component is running at a sufficiently high rate, the patients' notifications can easily fill up the component's input queue if this is assumed to be finite. In this case, depending on the drop policy adopted, some messages are discarded. Similarly, when multiple patients are in severe danger, the medical laboratory may send multiple requests for hospitalization. Therefore, to meet the second requirement, the component modeling the hospital must be able to process these messages within a given time bound, and have sufficiently large queues not to drop any of them.

An interesting relation also exists between message delays and delivery order. To verify the third requirement, our application needs an underlying communication infrastructure providing causal order delivery. Indeed, with random message delays, it may happen that the message coming from the medical lab is delayed w.r.t. the one sent by the flying squad when handing over the patient. However, if message delays are constant, the system essentially proceeds in a *delayed-synchronous* manner, which makes assuming any specific message orderings superfluous.

In addition, to evaluate the performance of our tool, we measured the *time* and *memory* taken, as well as the *number of states*

[1] To run LTL verification, we used the Bogor extension in [4]

Req.	No. Patients	Mem. (Mb)	States	Time
R1	10	278.38	70234	≈ 16 min
R1	20	312.31	123122	≈ 20 min
R2	10	412.21	113213	≈ 22 min
R2	20	502.75	209123	≈ 26 min
R3	10	498.1	232123	≈ 30 min
R3	20	591.1	289124	≈ 35 min

Table 3: Performance of our tool when R1-R3 are verified.

generated during the verification. We considered a system with 10 or 20 patients, each publishing 10 messages that may randomly trigger the actions corresponding to moderate or severe danger. We gathered the aforementioned metrics on an Intel Core Duo 1,83Ghz processor running Apple OSx, using the DJProf [11] profiler to evaluate the memory occupancy.

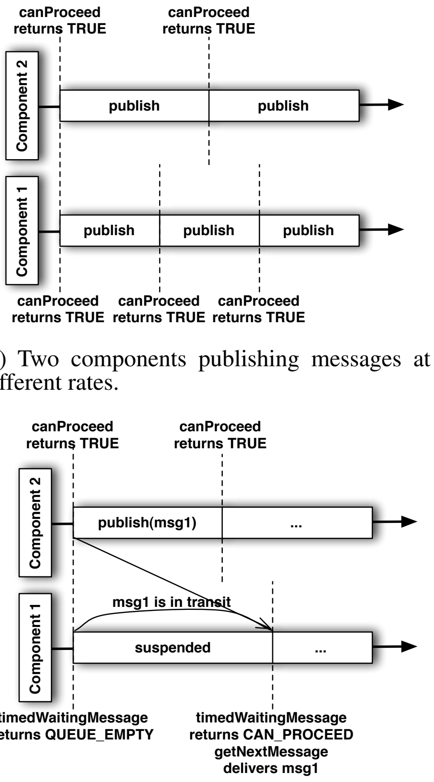
When a requirement turns out not to be verified, a counterexample is returned within a few seconds. Instead, Table 3 reports the performance of our tool in case the verification succeeds. Note that *doubling* the number of patients corresponds to a sharp increase in the message traffic modeling the interactions among components, as well as in the number of possible inter-leavings. The additional complexity of the model, however, yields only a *moderate* overhead in the time taken to accomplish the verification, about 25% in the worst case. We believe this is due to our implementation of the time extension, described in Section 3.2, that exploits the domain-specific semantics of P/S to cut down the processing. By the same token, the above metrics only slightly change assuming different P/S guarantees that still make the verification succeed. For instance, as already mentioned, the third requirement can be verified by either assuming causal ordering, or by imposing a constant message delay. In both cases, the verification completes in about half hour.

5. VERIFYING THE TIME EXTENSION

For our tool to prove useful, we must provide a strong foundation upon which our implementation can substantiate the soundness of presented results. In general, it is hard to achieve this objective. Moreover, in our approach the challenge is even more critical, as we are extending an existing model checker by augmenting its internal mechanism. To address this issue, we checked the correctness of our temporal extension using Bandera [9], a tool for the automatic verification of Java code. Notably, Bandera itself is based on Bogor. Essentially, it translates the Java code into a Bogor model upon which the actual verification is run. Consequently, we exploited our expertise in Bogor to reduce the size of the code fed as input to Bandera, therefore making the verification feasible. In this section we report on our experience in this respect, highlighting the lessons we learned on the way.

Bandera is a set of tools for the transformation of Java code into verifiable models. To this end, code analysis techniques are used to reduce the size of the models produced. Essentially, Bandera aims to i) eliminate from the input code all the elements (e.g., classes, methods, variables) that do not affect the verification of a given property, ii) abstract the type of, and infer bounds for, the remaining variables to reduce the state space generated during the verification. This is achieved through multiple translation steps, whose final output is a runnable Bogor model encompassing the properties to be verified. These are generally specified in terms of the values taken by input or output parameters of relevant methods.

Despite the degree of sophistication of Bandera, a brute-force approach whereby the entire Bogor code plus the P/S and time extensions are input to Bandera easily fails: the model output by Bandera is intractable. Nonetheless, by examining the outcome of Bandera, one can recognize how large parts of those models are not relevant to the verification of the time extension. Indeed, as we already observed, our notion of time does not alter the state space w.r.t. the untimed version of our tool. Rather, it limits the way the state space is explored. Therefore, the correctness of our extension can be checked by simply making sure that the guards controlling the inter-component schedules return the correct values for every possible situation.



(a) Two components publishing messages at different rates.

(b) A subscriber and a publisher component, messages have a non-zero delay.

Figure 10: Scenarios for verifying the time extension.

Based on the above observation —that again exploits our domain-specific knowledge— we carved out the time extension plus a few Bogor components needed to trigger its functionality. Specifically, almost the entire Bogor code enabling the extension capabilities was removed, as well as parts of the Bogor parser. Moreover, the state space generation mechanism was greatly reduced, as only the ability of *generating* the state space was required. Instead, how to *explore* this is dictated by the time extension, that is our verification target. To let the entire package compile, we implemented empty stubs in place of the parts removed.

To check our implementation, we must explore all the possible inter-component schedules. Notably, this can be accomplished with only two components, and four scenarios where these components publish or receive messages:

Scenario 1. As shown in Figure 10(a), two components publish messages with a non-integer ratio between their execution rates. No component is subscribed to these messages, hence they are discarded at the dispatcher. The scenario essentially checks whether the inter-component schedules are generated correctly in the absence of any message in transit.

Scenario 2. With a non-zero message delay, a component subscribes to a message published by another component, as depicted in Figure 10(b). Therefore, **timedWaitingMessage** returns **QUEUE_EMPTY** while the message is in transit, and switches to **CAN_PROCEED** as soon as the message appears in the input queue. The component execution rates are assigned so that only the receiving component is allowed to proceed at the time of message reception. The scenario verifies the functioning of **timedWaitingMessage**, and how the inter-component schedule is generated when a message is received with the subscribing component being given higher priority.

Scenario 3. Similarly to the previous scenario, this time the component execution rates are assigned so that the publishing component has higher priority. Therefore, when the message is inserted in the input queue of the receiving component, this is not immediately scheduled, and the publishing component can proceed. This scenario checks the situation dual to scenario 2.

Scenario 4. To test the combination of scenarios 2 and 3, the component execution rates and message delays are assigned so that *both* components can be scheduled when the message arrives at the intended recipient. The objective is to check whether both possible schedules are correctly generated.

Overall, Bandera generated about 100 assertions to verify the correctness of our implementation. As for the results, we actually discovered a bug in our initial prototype. Bandera showed a counterexample in the third scenario where **timedWaitingMessage** returned the wrong value after backtracking from the state that represents component A receiving the message. This was caused by a non-initialized variable, whose default value worked for most (but not all) combinations of the input parameters. Apparently, in our initial tests we were lucky in picking the “right” inputs. This result witnesses the importance of our efforts in verifying our time extension. In their absence, this bug would have probably survived.

6. CONCLUSION

In this paper we presented a time model to investigate time-sensitive P/S architectures. In our approach, time is embedded within the model checker as an additional dimension characterizing the system behavior. This work completes previous efforts by some of the authors, by providing the missing tile in a framework for the verification of P/S architectures. Our approach opens up opportunities for better investigations during the early design stages, which ultimately hold the potential to produce more reliable implementations.

Our research agenda includes a deeper assessment of the effectiveness of our approach through several case studies, as well as further work in the direction of the formal verification of the correctness of our Bogor implementation.

7. REFERENCES

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proc. of the* 5^{th} *Int. Symposium on Logic in Computer Science*, 1990.

[2] L. Baresi, C. Ghezzi, and L. Mottola. On accurate automatic verification of publish-subscribe architectures. In *Proc. of the* 29^{th} *Int. Conf. on Software Engineering (ICSE07)*, 2007.

[3] M.-H. Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, and M. Sebastianis. A case study on the automated verification of groupware protocols. In *Proc. of the* 27^{th} *Int. Conf. on Software engineering (ICSE05)*, 2005.

[4] Bogor Extensions for LTL Checking. `projects.cis.ksu.edu/projects/gudangbogor/`.

[5] J.-S. Bradbury and J. Dingel. Evaluating and improving the automatic analysis of implicit invocation systems. In *Proc. of the* 9^{th} *European software engineering Conf.*, 2003.

[6] M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *Proc. of the* 26^{th} *Int. Conf. on Software Engineering (ICSE04)*, 2004.

[7] N. Carvalho, F. Araujo, and L. Rodrigues. Reducing latency in rendezvous-based publish-subscribe systems for wireless ad hoc networks. In *Proc. of the* 5^{th} *Int. Wkshp. on Distributed Event-Based Systems (DEBS)*, 2006.

[8] A. Carzaniga, D.-S. Rosenblum, and A.-L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3), 2001.

[9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *Proc. of the* 22^{nd} *Int. Conf. on Software engineering*, 2000.

[10] X. Deng, M.-B. Dwyer, J. Hatcliff, and G. Jung. Model-checking middleware-based event-driven real-time embedded software. In *Proc. of the* 1^{st} *Int. Symposium on Formal Methods for Components and Objects*, 2002.

[11] DJProf Java Profiler, `www.mcs.vuw.ac.nz/djp/djprof/`.

[12] B. S. Doerr and D. C. Sharp. Freeing product line architectures from execution dependencies. In *Proc. of the* 1^{st} *Conf. on Software product lines : experience and research directions*, 2000.

[13] P.-Th. Eugster, P.-A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), 2003.

[14] D. Garlan and S. Khersonsky. Model checking implicit-invocation systems. In *Proc. of the* 10^{th} *Int. Workshop on Software Specification and Design*, 2000.

[15] G. Gerosa. Design and Implementation of a Time Extension for a Domain-Specific Model Checker. Master Thesis (in italian), Politecnico di Milano (Italy), 2007.

[16] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time corba event service. In *Proc. of the* 12^{th} *ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA)*, 1997.

[17] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[18] S. Li, Y. Lin, S.H. Son, J.A. Stankovic, and Y. Wei. Event detection services using data service middleware in distributed sensor networks. *Telecommunication Systems*, 26(2), June 2004.

[19] Robby, M.-B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. of the* 9^{th} *European software engineering Conf.*, 2003.

[20] U. Varshney. Pervasive healthcare. *Computer*, 36(12), 2003.

[21] L. Zanolin, C. Ghezzi, and L. Baresi. An approach to model and validate publish/subscribe architectures. In *Proc. of the SAVCBS Workshop*, 2003.