# On Accurate Automatic Verification of Publish-Subscribe Architectures

Luciano Baresi, Carlo Ghezzi, and Luca Mottola
Dipartimento di Elettronica e Informazione - Politecnico di Milano
Piazza Leonardo da Vinci, 32 – 20133 Milano (Italy)
{baresi|ghezzi|mottola}@elet.polimi.it

## Abstract

*The paper presents a novel approach based on Bogor for the accurate verification of applications based on Publish-Subscribe infrastructures. Previous efforts adopted standard model checking techniques to verify the application behavior, but they introduce strong simplifications on the underlying infrastructure to cope with the state space explosion problem and make automatic verification feasible.*

*Instead of building on top of existing model checkers, our proposal embeds the asynchronous communication mechanisms of Publish-Subscribe infrastructures* within *Bogor. This way, Publish-Subscribe primitives become part of the specification language as additional, domain-specific, constructs. Accurate models become feasible without incurring in state space explosion problems, thus enabling the automated verification of applications on top of realistic communication infrastructures.*

## 1 Introduction

Pervasive computing, autonomic systems, and ubiquitous applications are pushing for highly dynamic and flexible software environments. The components of a given application, along with their interactions, are not set once and forever. The dynamism of these systems requires that components be able to federate spontaneously: the seamless integration of elements is one of the key requirements for these applications.

If we tackle the problem at architectural level, the Publish-Subscribe interaction style [11] provides an appealing solution for these highly dynamic applications. In Publish-Subscribe applications, components *subscribe* to message (event) patterns and get *notified* when other components *publish* messages matching their subscriptions. Its asynchronous, implicit, and multi-point communication style is particularly amenable to those applications in which components can be added or removed unpredictably, and the communication must be decoupled both in time and space [11]. Based on these features, Publish-Subscribe infrastructures have been developed for a wide range of application scenarios, from wide-area notification services [8] to wireless sensor networks [17].

Despite the clear advantages brought by Publish-Subscribe infrastructures, flexibility hampers their validation. The verification of applications developed using this paradigm is a challenging task. We can easily reason on components in isolation, but the global picture is much more complex. Components are often written independently of the way they are federated and their interactions can change dynamically. Moreover, the Publish-Subscribe paradigm can be instantiated in very different ways. The key features are preserved, but specific guarantees vary from infrastructure to infrastructure. For example, different message delivery policies, reliability guarantees, or concurrency models can easily change the global behavior of the system and thus the final outcome of the verification effort.

Among the many attempts to analyze Publish-Subscribe applications, model checking techniques have been proposed as possible solution, but existing works (e.g., [6, 13, 23]) underestimate the different guarantees provided by the underlying Publish-Subscribe infrastructure. The number of alternatives makes the aforementioned characterization non-trivial per se. The problem becomes even harder if we want to model these features by means of existing model checkers (e.g., SPIN [15]): detailed models cause the well-known *state space explosion* problem, which inherently leads to the inability to verify accurate models. The consequence is that all these approaches adopt simplified or partial models to limit the number of states generated during verification.

In this paper we argue that a fine-grained verification of applications based on Publish-Subscribe architectures requires a different approach. We flip the "classical" solution: instead of building on top of a given model checker, we embed the Publish-Subscribe communication mechanisms

*within* Bogor [20], and export this functionality as primitive constructs of the model checker. This approach enables domain-experts to exploit their specific knowledge to better control the state space explosion, obtain verifiable fine-grained characterizations of the different guarantees, and thus achieve more efficient verification mechanisms.

To achieve this, we employ our knowledge of Publish-Subscribe architectures and the different guarantees they provide to implement *domain-specific state abstraction* mechanisms. Based on the semantics of Publish-Subscribe infrastructures, we can regard states that a standard model-checker would consider different as identical and hide state information not affecting the Publish-Subscribe semantics.

The proposed approach, which is an evolution of the work presented in [23], provides designers with a parametric model of the Publish-Subscribe infrastructure, and lets them model their application components with finite state automata. The designer can choose the guarantees provided by the Publish-Subscribe infrastructure by setting the parameters of our Bogor extension, which is then used to verify the application of interest. This way, developers are free to explore the trade-off between the *assumptions* made on the underlying middleware system, and the *mechanisms* explicitly implemented at the application level.

The properties against which modeled systems are verified are expressed in LTL (Linear Temporal Logic, [1]). We do not concentrate on how to provide the designer with an easier and higher level formalism to express properties, but scenario based notations, for instance, can be easily transformed into LTL expressions.

The first results on customizing Bogor to verify Publish-Subscribe systems were presented in [2], which mainly introduced the problem and compared the state spaces obtained with our and other known approaches. This paper thoroughly motivates and describes the extensions to Bogor for the different guarantees, gives insights into how we exploited our domain-specific knowledge in embedding the Publish-Subscribe extensions in Bogor, and demonstrates them on a realistic case study.

The rest of the paper is organized as follows. Section 2 introduces Publish-Subscribe systems and identifies a set of significant guarantees to characterize them. Section 3 briefly presents Bogor and Section 4 describes our extensions to embed the Publish-Subscribe primitives into the model checker. Section 5 exemplifies the approach on a case study, Section 6 surveys related proposals, and Section 7 concludes the paper.

## 2 Publish-Subscribe Architectures

Publish-Subscribe is an asynchronous communication paradigm where application components do not interact directly. Instead, their communications are mediated by an additional logical layer, called *dispatcher*. Components declare the messages they are interested in by means of subscriptions. These are collected by the dispatcher in a suitable data structure, called *subscription table*. When a component publishes a message, the dispatcher matches this against existing subscriptions, and delivers the message to all those application components that issued matching subscriptions. This process is usually referred to as *message filtering*. Using this style of interaction, the sender does not know the identity of the receivers: it is the dispatcher that identifies them dynamically. As a consequence, new components can join a federation, become immediately active, and cooperate with the others without requiring any reconfiguration of the architecture.

| **Guarantee** | **Choices** |
|---|---|
| *Message Reliability* | Absent, Present |
| *Message Ordering* | Random, Pair-wise FIFO, System-wide FIFO, Causal Order, Total Order |
| *Filtering* | Precise, Approximate |
| *Real-Time Guarantees* | None, Soft RT, Hard RT |
| *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: Publish-Subscribe guarantees.

Both commercial products [21] and advanced research prototypes [8] implement the Publish-Subscribe paradigm, but they differ in a number of significant characteristics, as well as in the guarantees they support. To provide fine-grained verification of Publish-Subscribe infrastructures, we studied existing literature to devise a possible classification of the different characteristics. Table 1 summarizes the results of this investigation, and shows the dimensions along which existing Publish-Subscribe systems can be classified and the instantiations we found reasonable to consider.

**Message reliability** refers to the fact that the Publish-Subscribe infrastructure can either guarantee that *all messages are eventually delivered*, or some may be *lost*. **Message ordering** defines the policy with which the dispatcher delivers 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, *causality* chain among messages, or *total order*, to mean that all the components with the same set of subscriptions deliver the same set of messages in the same order. **Message filtering** can be either *precise*, i.e., components only receive the messages they are subscribed to, or *approximate*, where components can receive messages they are not subscribed to

(false positives), or miss events in which they are interested (false negatives)[1]. **Real-time guarantees** describe the ability of the Publish-Subscribe infrastructure to deliver messages within *soft* or *hard real-time* constraints, or *without considering time* at all. **Subscription propagation** identifies the ability of the dispatcher in setting up subscriptions *immediately*, or with some *propagation delay*, after which they actually start to be matched against published messages. **Repliable messages** account for the possible *support for replies* to published messages, which are hence guaranteed to arrive to the requesting node, as opposed to systems where subscriptions must be used to convey the replies, and there is no guarantee that these are active before publishing the reply. **Message priorities** deal with the capability of the infrastructure to *prioritize* the order of delivered messages, and possibly support *queue scrunching* to avoid starvation of low-priority messages. **Queue drop policy** describes how the dispatcher identifies the messages to be dropped in the presence of queues of finite length, i.e., either considering their *order of arrival* (messages arriving to a full queue are immediately discarded), or looking at their *priorities* (message dropping starts from low priorities).

To the best of our knowledge, this kind of fine-grained classification is new. Table 2 exemplifies how these different dimensions can be used to precisely classify three deeply different Publish-Subscribe infrastructures. The first column considers a centralized Publish-Subscribe infrastructure implementing the JMS APIs and semantics. In this case, the dispatcher runs on a single machine, to which publishers and subscribers connect. With this architecture, the system trivially provides pair-wise FIFO delivery, as there is only one path used to route messages towards subscribers. For the same reason, there is no subscription delay: everything is managed in a centralized fashion. According to the JMS specifications [21], the system also supports repliable messages and priorities.

The second column describes the Gryphon middleware [16], which is a Publish-Subscribe architecture geared towards peer-to-peer and dynamic settings. In this case, the dispatcher is realized by distributing subscriptions and messages through a network of servers. Therefore, subscriptions must propagate to all the servers before being active throughout the whole system. The work in [4] also shows how to implement reliable total order in this network of servers.

The third column examines the DSWare system [17], which is an event detection middleware for wireless sensor networks. In such scenarios, message delivery with prespecified time constraints is of paramount importance. A form of hard real-time delivery is provided, but because of the severe limitations of the devices for which DSWare is designed, all the remaining features are implemented using the least resource-consuming approach. For instance, there is no support for particular message orderings.

[1]Notice that approximate filtering is deterministic, while reliability problems are in general unpredictable.

| Guarantee | JMS | Gryphon | DSWare |
|---|---|---|---|
| *Message Reliability* | Present | Present | Absent |
| *Message Ordering* | Pair-wise FIFO | Total | None |
| *Filtering* | Precise | Precise | Precise |
| *Real-Time Guarantees* | None | None | Hard RT |
| *Subscription Propagation Delay* | Absent | Present | Present |
| *Repliable Messages* | Present | Absent | Absent |
| *Message Priorities* | Present | Absent | Absent |
| *Queue Drop Policy* | Priority Drop | Tail-drop | Tail-drop |

Table 2: Examples of existing Publish-Subscribe infrastructures classified along the dimensions of Table 1.

## 3 Bogor

Bogor is a state-of-the-art extensible model checker implemented in Java. Its architecture is designed to let domain-experts easily customize its input language and checking engine.

The input language is an evolution of the Bandera Intermediate Language (BIR) [9], initially conceived as an intermediate step for translating Java programs into the input language of existing model checkers. It provides the basic constructs commonly available in modeling languages of well-established verification tools, (e.g., Promela in the case of SPIN [15]), including non-primitives data types such as records and arrays. Additionally, it supports advanced constructs such as function pointers, generic types, and dynamic threads. These features ease the modeling of dynamic applications using Bogor.

Concurrent processes are modeled as *threads*. A thread usually defines a set of local variables, and a number of *locations*[2] to represent different values for the program counter inside the thread. The control flow is based on the interplay between guard *expressions* and *actions*. The former must produce values with no side-effects, and are normally used to control the transitions between different locations, whereas the latter actually affect the system state. Figure 1 illustrates how to model two concurrent threads that access a shared resource. `Thread1` and `Thread2` concurrently modify the state of `t1`, `t2`, and `x`. The third thread (`Thread0`) checks whether `x` is equal to a predefined constant. Needless to say, if we run Bogor on this example, it returns all the interleavings of the thread transitions that falsify the assertion in `Thread0`.

The internal architecture of Bogor is highly modular and customizable. Well-defined interfaces allow the user

[2]Notice that the code associated with locations define atomic actions as for thread evolution.

```
system SumToN {
  const PARAM { N = 1; }
  int x := 1; int t1; int t2;

  active thread Thread1() {
    loc loc0:
      do { t1 := x; }
      goto loc1;

    loc loc1:
      do { t2 := x; }
      goto loc2;

    loc loc2:
      do { x := t1 + t2; }
      goto loc0;
  }

  active thread Thread2() {
    loc loc0:
      do { t1 := x; }
      goto loc1;

    loc loc1:
      do { t2 := x; }
      goto loc2;

    loc loc2:
      do { x := t1 + t2; }
      goto loc0;
  }

  active thread Thread0() {
    loc loc0:
      do { assert (x != PARAM.N); }
      return;
  }
}
```

Figure 1: Bogor: sample code.

```
extension GenericRandom
  for genericRandom.GenericRandomModule {
    typedef type<'a>;
    expdef GenericRandom.type<'a>
      choose<'a>(GenericRandom.type<'a>,
                 GenericRandom.type<'a>);
}
```

(a) language extension for random choice.

```
package genericRandom;
public class GenericRandomModule implements IModule {
  public IMessageStore
         connect (IBogorConfiguration bc) {
    // Retrieving Bogor hooks
  }
  public IValue
         choose(IextArguments args) {
    // Specifies the semantics of choose
  }
}
```

(b) Java excerpt that implements the extension.

Figure 2: Bogor: special-purpose choose.

to customize the verification engine. For instance, one can replace the module that implements the state space exploration with a dedicated version to better drive the search process under particular assumptions. Moreover, one can also add further primitive constructs (expressions or actions) as well as data types. Figure 2(a) illustrates the Bogor code needed to augment the input language with a new expression that takes two instances of a generic data type, and non-deterministically returns one of the two. In particular, we must first define an identifier for the extension —`GenericRandom` in this case— and specify the Java class that implements the new constructs (`genericRandom.GenericRandomModule`, in this example). Within the extension body, the `typedef` construct declares a generic data type, used in the signature of the `choose` expression. After these declarations, the additional constructs defined can be used as if they were primitive ones. For instance, we could use the `choose` expression as a guard condition.

Figure 2(b) presents an excerpt of the Java class that implements this extension. Bogor provides the necessary hooks to access its internal mechanisms and influence the state space search. Using these features, one can actually implement the semantics required for each expression/action defined in the extension. For instance, to model check a thread by exploiting the `choose` construct, we must explore all the possible system executions generated by the random choice. Therefore, in the body of method `choose`, we ask Bogor to generate two different paths in the state space exploration, and generate two different "next states" for the two paths, corresponding to the different values returned by the `choose` expression.

# 4 A Domain-specific Model Checker

We used the customization features supplied by Bogor both to augment its input language, with additional constructs that mimic Publish-Subscribe functionality, and to embed a parametric dispatcher within the model-checker itself to provide a special-purpose verification engine based on domain-specific state abstraction mechanisms. The verification is based on the semantics defined by the particular combination of selected guarantees.

## 4.1 Publish-Subscribe Primitives

The Publish-Subscribe infrastructure is represented as a generic and abstract data type named `PubSubConnection` (Figure 3). An instance of this data type, seen as local variable in each thread, provides application components with operations to open a connection to the Publish-Subscribe infrastructure, issue subscriptions, and publish and receive messages.

```
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: Modeling constructs to export the Publish-Subscribe infrastructure in Bogor.

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

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

active thread Publisher() {
  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 Subscriber() {
  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: Publish-Subscribe extensions in a Bogor model.

Figure 4 shows how these extensions can be used to model a simple message exchange between two threads. They both exploit the `register` operation in location `loc0` to open a connection to the infrastructure and implicitly create a virtually infinite incoming queue. In doing this, they need to specify the kind of messages sent or received (`MyMessage` in this case). A further primitive `registerWithDropping` is also provided to specify an incoming queue of finite length and the corresponding dropping policy while registering.

To publish a message, the user creates an instance of a previously defined data structure, and uses the `publish` operation, as illustrated in location `loc1` in the publisher thread. The same message can alternatively be published according to a given priority, with construct `publishWithPriority`. A message is received by any application component that issued a matching subscription. For instance, a component subscribed to messages of type `MyMessage` having the `data` field greater than zero should receive the message published in location `loc1` in the publisher thread.

Subscriptions are represented as pointers to boolean functions taking messages as inputs. Notice that this solution does not constrain the subscription language and format of messages, that can be defined freely. To issue a subscription, the user simply defines the corresponding function and passes it as parameter to our construct `subscribe`. This is exemplified in location `loc0` in the subscriber thread. To receive a message, our extension provides expression `waitingMessage` that can be used as a guard condition for incoming messages. When it evaluates to true, there is at least a message waiting in the incoming queue of the considered application thread. The message can be retrieved by using construct `getNextMessage`, as illustrated in location `loc1` in the subscriber thread.

Remarkably, the primitives defined as extensions to the Bogor modeling language mimic the API commonly found in real Publish-Subscribe infrastructures. We argue that this ease the transformation of high-level behavioral specifications into the Bogor (extended) input language, and enables the use of our approach in existing software development/verification processes.

### 4.2 State Abstraction

The state abstractions supplied by the customized verification engine, which embeds the parametric dispatcher, depend on the guarantees assumed with respect to the dimensions of Table 1. To illustrate how this is done, we describe the state of a Publish-Subscribe system as a tuple:

$$SysState = \langle T_1, T_2, ..., T_n, DispState \rangle \quad (1)$$

where the generic $T_i$ represents the state of the Bogor thread modeling the $i^{th}$ application component, which includes the state of the connection to the Publish-Subscribe infrastructure, whereas $DispState$ is the current state of the Publish-Subscribe dispatcher embedded in the model checker. In turn, this can be expressed as:

$$DispState = \langle SbTable, RtData, M_1, M_2, ..., M_m \rangle \quad (2)$$

where $RtData$ is bookkeeping information to implement specific features (e.g., causal order delivery), $SbTable$ is the subscription table, and $M_j$ represents a message in transit from the publisher node to the matching subscribers.

Space limitations do not allow us to concentrate on the implementation, which is thoroughly described in [19]. The mapping from the system state to the Java code we implemented is straightforward. Class `PubSubModule` is the main responsible for the implementation of the Publish-Subscribe dispatcher, whereas the per-thread state of the Publish-Subscribe connections is modeled with different instances of class `PubSubConnection`. The former class is instantiated differently depending on the guarantees the user wants to assume: a vector with the specific guarantees selected among the ones in Table 1 is passed as parameter before the actual verification starts. The verification then proceeds according to the semantics defined by the particular combination of selected guarantees. In the following, we give an overview of the techniques used to implement the different guarantees —and thus the different state abstraction policies— by means of examples.

**Publish-Subscribe Connections.** Let us consider a generic $T_i$ in tuple (1). This normally depends on the current value of the variables local to that particular thread, and on the program counter[3]. Our approach considers a connection to the Publish-Subscribe infrastructure as an instance of an abstract data type, used by the different threads as a special-purpose, local variable (`ps` in Figure 5). The different values of $T_i$, with respect to the selected guarantees, are defined by characterizing the semantics of this data type within the model checker.

For example, if we consider *causal order* delivery, the incoming queue of a given thread may receive a message without being able to deliver it to the application. This may happen when a causally related message has not been received yet. As far as the corresponding application component is concerned, the state of the incoming queue remains unchanged. We can therefore omit to consider the messages buffered in the incoming queue of a thread when we want to distinguish different values for $T_i$, and thus reduce the number of generated states.

**Dispatcher.** Let us now consider tuple (2). Using the proposed approach, we can implement the semantics of the dispatcher directly, and distinguish different values for $DispState$ only when the chosen guarantees require that. This way, we can explicitly control the number of different tuples (2) we generate.

For example, if we consider *message filtering*, an application component is included in the set of matching components if *at least one* of its subscriptions matches. The result is not affected by the *order* in which the different subscriptions are examined. Based on this observation, we model the subscription table as a *set*, and generate different values for $SbTable$ in (2) only when the content of the table changes, regardless of the order in which subscriptions arrive at the dispatcher. This cannot be achieved in model checkers whose input language lacks native support for order-insensitive data containers, e.g, [15].

**Dispatcher-Connections Interplay.** Our approach shows the biggest advantages when it comes to exploit Publish-Subscribe as a *point-to-multipoint* communication mechanism, e.g., when we duplicate messages addressed to different message queues. This message duplications must be transparent to application components, as this is hidden by the Publish-Subscribe infrastructure (i.e., the dedicated component embedded in Bogor). By controlling what influences tuples (1) and (2), we can avoid to deal with mechanisms and data structures needed to duplicate and distribute messages from the computation of the system state. This avoids generating states that should not be perceived as different by the application. To the best of our knowledge, this form of communication is not natively supported by any existing model checker. The closest example might be Promela channels [15], but they are limited to point-to-point communication. Moreover, once attached to a pair of processes, they cannot be detached to serve different processes, and cannot model advanced features like the ones in Table 1 (apart from fixed-size queues). Actually, achieving multi-point communication using Promela channels is feasible, but highly inefficient in terms of number of generated states, as we have already shown in [2].

A further example is *total order delivery*. To achieve this, one needs to augment messages with timestamps, and have the dispatcher enforcing the proper order of message delivery by keeping track, for each application component,

[3] This is called *location* in Bogor terminology.

of published or received messages. This processing does not affect the application behavior. To implement total order, the dispatcher must store bookkeeping information in data structures that are modified each time a message is published or delivered. In our approach, these are not part of (1) or (2). Therefore, they cannot be responsible for generating further system states. Standard model checkers, in contrast, would generate these states even when they are not needed. For instance, a message matching no subscription can be ignored, as far as the total order is concerned.

## 5 Case Study

To assess the effectiveness of our approach, here we report on the verification process adopted while developing an embedded, safety-critical application on top of a Publish-Subscribe infrastructure. We first describe the scenario and highlight the corresponding application requirements, which also represent the properties we want to verify. Then, we illustrate our experience in checking the system model to demonstrate how the proposed approach effectively enables fine-grained verification.

### 5.1 Scenario and Model

An application for fire monitoring in a tunnel comprises the following components:

- A set of *temperature and smoke sensors* installed along the tunnel to obtain environmental data;
- A *central control station* that periodically gathers data from the temperature sensors;
- A set of *actuators* to control the *ventilation* inside the tunnel and the *traffic lights* at the entrances;
- *RFID readers* to monitor the trucks that enter the tunnel that may transport dangerous materials (whose characteristics are encoded in the tags themselves);
- An *external fire brigade* ready for intervention in case of emergency.

The proposed scenario is not just an academic exercise: researchers are investigating the use of similar solutions for real deployments, while some of the authors are involved in a EU-funded project[4], that is taking similar settings as motivating scenarios. In similar systems, the goal is to implement a decentralized control-loop able to tolerate the dynamics of components failing or joining the system. Because of this, the Publish-Subscribe interaction paradigm turns out to be well suited to these applications.

[4] `www.ist-runes.org`

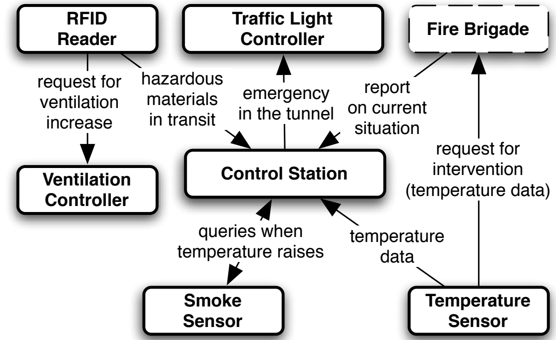


Figure 5: System architecture for fire control in a tunnel.

The system must be designed to cope with *normal* and *emergency* situations inside the tunnel. Some of the high-level requirements can be described as follows:

**R1** When a truck enters the tunnel carrying hazardous materials, the nodes controlling the ventilation must preventively increase the air exchange.

**R2** As soon as an increase in temperature is detected at the central station, smoke sensors must be queried to detect the possible presence of smoke.

**R3** In the same situation (increase of temperature), the sensor must contact the fire brigade, which in turn reports the anomaly to the control station as soon as it arrives. The report from the brigade must match the information received from the sensor directly.

**R4** The messages sent from the control station to command the traffic lights at the tunnel entrances must be delivered with a maximum delay of five time units.

We used Bogor to develop our system model, whose components and relationships in terms of exchanged data are depicted in Figure 5. The different message routes are expressed according to the interests of the application components, that is, in terms of subscriptions. Notice that the fire brigade is normally not part of the system. It is only involved under emergency conditions.

To verify each of the aforementioned requirements, we considered example scenarios triggering specific system executions. For instance, to check requirement **R3**, we assume that the component representing a temperature sensor suddenly reports a value above the safety threshold, and we verify if the rest of the system behaves correctly. Each of these requirements can be expressed by defining suitable predicates on the internal states of the application components, and by making use of LTL temporal operators to constrain event sequences.

For instance, if we consider requirement **R1**, we can define predicates $HazardEntering$ and $VentilationIncreased$ to describe a read of a tag on a truck carrying dangerous material, and an increase in the tunnel ventilation, respectively. Using these definitions, requirement **R1** can be verified by checking the following LTL expression[5]:

$$\Box(HazardEntering \Longrightarrow \Diamond\, VentilationIncreased) \quad (3)$$

## 5.2 Verification Effort

Our approach allows system designers to explore the trade-offs between the assumptions made on the underlying middleware system and the mechanisms explicitly implemented at application level. This analysis can proceed along two different, yet parallel, dimensions. For each required characteristic, the developer can either pose new requirements on the Publish-Subscribe infrastructure to satisfy the desired properties (hence increasing the number of guarantees assumed on the underlying middleware), or maintain the same assumptions on the Publish-Subscribe infrastructure, and implement required mechanisms at application level.

We analyzed the system for fire control in a tunnel starting from a coarse-grained model of the various application components and from the features guaranteed by a Publish-Subscribe middleware for embedded sensing environments like DSWare (whose guarantees are described in Table 2). The design choices driven by our approach are:

**Step 1 - Message Reliability.** Our initial model implicitly took message reliability for granted. Indeed, all the properties we wanted to verify fail immediately in all the system executions where messages are lost. For instance, requirement **R1** trivially fails if the message published by the *RFID reader* never reaches the *ventilation controller*. Message reliability is an active topic of research in the embedded networking community (e.g., [22]). Therefore, we decided to customize the Publish-Subscribe infrastructure with some form of message reliability, and did not modify the application at this stage.

**Step 2 - Causal Order Delivery.** To verify requirement **R3**, the data regarding an emergency must be available at the control station at the time the fire brigade publishes its report. Our verification failed in checking this requirement when the *fire brigade* reports on a fire in the tunnel and the *control center* does not know anything about it. This happens in those systems in which causal order delivery is violated at the control center. According to this delivery policy, the message from the temperature sensor carrying the too high value should arrive at the control station before the firefighters' report, which is indeed caused by the former. Since causal order is key to relate the occurrence of events at different processes —a characteristic of paramount importance in our scenario— we decided to push the causal order delivery guarantee in the Publish-Subscribe middleware, and again did not modify the application.

[5]The verification of LTL expressions uses the Bogor LTL plug-in [5].

**Step 3 - Subscription delays.** Despite causal order, requirement **R3** keeps failing in systems where, because of delays in propagating subscriptions, the fire brigade publishes a report before the corresponding subscription issued by the control station is active. This is caused by the fire brigade that dynamically joins the system at unpredictable times, which implies a non null setup time for the related subscriptions. Differently from before, it is unreasonable to assume that subscriptions propagate instantaneously in such a distributed environment. Therefore, we modified the initial design of the application to let the fire brigade repeat the publish operation of the report until an acknowledgement from the control station is received. If subscriptions propagate within a finite time frame, the message from the fire brigade is eventually delivered.

**Step 4 - Replies.** We initially customized the Publish-Subscribe middleware not to support repliable messages. Since requirement **R2** requires a query-reply form of interaction, we issued subscriptions at the control station to convey the reply from the smoke sensors when the former needs to query the latter. However, if subscriptions do not propagate instantaneously, there is no guarantee to receive needed replies, as pointed out in Section 2. For this reason, the verification of requirement **R2** fails. However, the support to query-reply message can be imposed in the kind of environments we are considering, as dedicated mechanisms exist in the literature, e.g., [18]. We therefore continue the verification of our system by imposing a further guarantee provided by the Publish-Subscribe infrastructure.

**Step 5 - Real-time.** requirement **R4** demands for a timed system[6] able to deliver messages within strict time constraints. This feature is at the core of any application targeted to control and monitoring since actions in response to sensed events must be performed when gathered data are still valid. We already imposed this guarantee since the beginning of our verification effort[7]. Indeed, it was already part of the features of DSWare, which we took as starting point. Based on this guarantee, requirement **R4** can be verified without modification to the existing architecture.

These example steps show that our approach enables fine-grained analysis on the application components, allowing the designer to investigate the interplay between the guarantees provided by the Publish-Subscribe infrastructure

[6]If needed, application components deal with time-related constraints by means of dedicated local variables.

[7]Notice that our implementation of real-time guarantees is a "light" interpretation of what proposed in [10].

| Requirement | Memory (Mb) | States | Time |
|---|---|---|---|
| Requirement R1 | 96.38 | 786 | 133 sec |
| Requirement R2 | 126.72 | 1098 | 287 sec |
| Requirement R3 | 261.1 | 2067 | 547 sec |
| Requirement R4 | 503.98 | 2854 | 9,33 min |

Table 3: Performance of the extended checking engine (with all the guarantees adopted in steps 1-5).

| Scenario | Publish ops | Bogor with P/S | | | SPIN - Promela | | |
|---|---|---|---|---|---|---|---|
| | | Memory (Mb) | States | Time | Memory (Mb) | States | Time |
| Causal1 | 2 | 32.8 | 98 | 103 sec | 298.3 | 364 | >15 min |
| Causal2 | 5 | 45.6 | 133 | 132 sec | 589.6 | 604 | >1 hour |
| Causal3 | 7 | 52.3 | 213 | 158 sec | OM | NC | NC |
| Causal4 | 10 | 61.1 | 301 | 189 sec | OM | NC | NC |
| Priority1 | 2 | 18.3 | 75 | 47 sec | 192 | 259 | >10 min |
| Priority2 | 5 | 26.9 | 125 | 103 sec | 471.2 | 467 | >30 min |
| Priority3 | 7 | 37.9 | 166 | 134 sec | OM | NC | NC |
| Priority4 | 10 | 49.1 | 207 | 163 sec | OM | NC | NC |

OM = Out of Memory - NC = Not Concluded

Table 4: Comparing the Bogor-based approach with [23] (taken from [2]).

and the mechanisms implemented within the application. This witness the advantages brought by our approach from a qualitative, process-oriented standpoint.

To complement this description, Table 3 provides some experimental data related to the verification of the different requirements we discussed. All considered properties can be checked within reasonable time, and by consuming an amount of memory normally available on conventional desktop PCs. This quantitatively assesses the feasibility of our approach.

Finally, for the sake of completeness, Table 4 reports a more detailed version of the data already presented in [2], to highlight the gains enabled by our approach with respect to the one in [23] while verifying a set of synthetic executions. These simply perform a number of publish operations (second column) according to causal order and pre-specified priorities, to stress the underlying model checker. While not being meaningful per se, these system executions provide a measure of the gains enabled by our approach with respect to existing solutions. Our solution achieves gains in the order of 500% in terms of consumed memory, while reducing the time needed to carry out the verification process by two orders of magnitude. Most importantly, it allows concluding the verification effort, while the original approach runs out of memory. More information can be found in [2].

## 6 Related Work

The work presented in this paper extends previous efforts of some of the authors [2,23]. With respect to [23], we acknowledge the need for a radically different approach to the provision of fine-grained verifiable models of Publish-Subscribe architectures. In particular, [23] still models the Publish-Subscribe infrastructure using the primitives made available by the SPIN model checker. In contrast, this work reverts that approach by providing a domain-specific model checker that exports the Publish-Subscribe API as constructs of the modeling language. Furthermore, [23] characterizes the Publish-Subscribe infrastructure in terms of reliability, message delivery order, and subscription propagation delay. Therefore, it does not consider several of the dimensions listed in Table 1. Conversely, the work in [2] introduces our novel approach, and illustrates an initial, quantitative analysis of the gains in terms of verification time and consumed memory on a set of synthetic scenarios. The goal was to foster further investigation, whose results are presented here with the details of the mechanisms we implemented and an assessment of the effectiveness of our approach on a realistic case study.

Model checking for Publish-Subscribe architectures is also investigated by Garlan et al. in [12], where they provide a set of pluggable modules that allow the modeler to choose one possible configuration out of a set of possible choices. However, available models are far from fully capturing the different characteristics of existing Publish-Subscribe systems. For instance, application components cannot change their subscriptions at run-time, and the message dispatching mechanism is only characterized in terms of delivery policy (*asynchronous*, *synchronous*, *immediate* or *delayed*). The same approach is extended in [6] by adding more expressive events, dynamic delivery policies and dynamic event-method bindings. These features are then used in [24] to implement a transformational framework that, starting from a dedicated programming language, produces XML data for model checking as well as executable artifacts for testing. The resulting approach only deals with the specification of different delivery policies depending on the overall state of the model, and still does not capture fine-grained guarantees such as real-time constraints.

Techniques applicable to specific Publish-Subscribe middleware systems have been considered in [3, 7, 10, 14]. Beek et al. [3] concentrate on the addition of a Publish-Subscribe notification service to an existing groupware protocol, and reports on the improvements in user awareness of the development status achieved in this way. Caporuscio et al. [7] develop a compositional reasoning technique based on an assume-guarantee methodology. The methodology is applied on a specific case study, i.e., on developing a file sharing system on top of the Siena Publish-Subscribe

middleware [8]. The proposals in [10, 14] describe an approach similar to ours based on an early version of Bogor. The authors focus on modeling the real-time features of the CORBA Communication Model (CCM). Their time model is certainly more detailed than ours. All the aforementioned approaches lose generality in that they do not allow users to customize the checking engine to model Publish-Subscribe systems that provide various guarantees.

## 7 Conclusions

The paper presents a novel approach based on Bogor for the fine-grained verification of applications based on Publish-Subscribe architectures. Our approach does not build on top of an existing model checker, but extends both the input language and the verification engine of Bogor with Publish-Subscribe primitives and guarantees to supply a *domain-specific* model checker. To assess the correctness of our extensions we devised a wide set of test cases and we thoroughly used benchmark applications.

The approach allows the designer to balance the guarantees demanded to the Publish-Subscribe infrastructure adopted with the functionality embedded in the application components. Besides the flexibility, the results presented in the paper demonstrate the gain in terms of computational time and memory use. This means that models that would be too heavy for "conventional" model checkers become analyzable with our approach.

## References

[1] M. Autili, P. Inverardi, and P. Pelliccione. A scenario based notation for specifying temporal properties. In *Proc. of the 2006 Int. Wkshp. on Scenarios and state machines: models, algorithms, and tools*, pages 21–28, 2006.

[2] L. Baresi, C. Ghezzi, and L. Mottola. Towards fine-grained automated verification of publish-subscribe architectures. In *Proc. of the* $26^{th}$ *Int. Conf. on Formal Methods for Networked and Distributed Systems (FORTE06)*, pages 131–135, 2006.

[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)*, pages 596–603, 2005.

[4] S. Bhola, R. E. Strom, S. Bagchi, Y. Zhao, and J. S. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *Proc. of the 2002 Int. Conf. on Dependable Systems and Networks*, pages 7–16, 2002.

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

[6] 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.*, pages 78–87, 2003.

[7] M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *Proc. of the* $19^{th}$ *Int. Conf. on Software engineering (ICSE04)*, pages 221–230, 2004.

[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*, pages 439–448, 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*, pages 154–181, 2002.

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

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

[13] D. Garlan, S. Khersonsky, and J. Kim. Model checking publish-subscribe systems. In *Proc. of the* $10^{th}$ *Int. SPIN Workshop on Model Checking Software (SPIN03)*, pages 166–180, 2002.

[14] J. Hatcliff, X. Deng, M.-B. Dwyer, G. Jung, and V. Ranganath. Cadena: an integrated development, analysis, and verification environment for component-based systems. In *Proc. of the* $25^{th}$ *Int. Conf. on Software Engineering (ICSE03)*, pages 160–173, 2003.

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

[16] IBM Research. The Gryphon Middleware. `www.research.ibm.com/gryphon`.

[17] 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):351–368, June 2004.

[18] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[19] L. Mottola. Accurate Verification of Publish-Subscribe Architectures. Technical report, Politecnico di Milano, Italy, 2006.

[20] 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.*, pages 267–276, 2003.

[21] Sun Microsystems. JMS Specifications and Reference Implementation. `java.sun.com/products/jms/docs.html`.

[22] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. of the* $1^{st}$ *Int. Conf. on Embedded networked sensor systems (SENSYS03)*, pages 14–27, 2003.

[23] L. Zanolin, C. Ghezzi, and L. Baresi. An approach to model and validate publish/subscribe architectures. In *Proc. of the SAVCBS'03 Workshop*, pages 35–41, 2003.

[24] H. Zhang, J. S. Bradbury, J. R. Cordy, and J. Dingel. Implementation and verification of implicit-invocation systems using source transformation. In *Proc. of the* $5^{th}$ *Int. Wkshp. on Source Code Analysis and Manipulation (SCAM05)*, pages 87–96, 2005.