Loupe: Verifying Publish-Subscribe Architectures with a Magnifying Lens

Luciano Baresi, Carlo Ghezzi, *Fellow*, *IEEE*, and Luca Mottola

Abstract—The Publish-Subscribe (P/S) communication paradigm fosters high decoupling among distributed components. This facilitates the design of dynamic applications, but also impacts negatively on their verification, making it difficult to reason on the overall federation of components. In addition, existing P/S infrastructures offer radically different features to the applications, e.g., in terms of message reliability. This further complicates the verification as its outcome depends on the specific guarantees provided by the underlying P/S system. Although model checking has been proposed as a tool for the verification of P/S architectures, existing solutions overlook many characteristics of the underlying communication infrastructure to avoid state explosion problems. To overcome these limitations, the Loupe domain-specific model checker adopts a different approach. The P/S infrastructure is not modeled on top of a general-purpose model checker. Instead, it is embedded within the checking engine, and the traditional P/S operations become part of the modeling language. In this paper, we describe Loupe's design and the dedicated state abstractions that enable accurate verification without incurring state explosion problems. We also illustrate our use of state-of-the-art software verification tools to assess some key functionality in Loupe's current implementation. A complete case study shows how Loupe eases the verification of P/S architectures. Finally, we quantitatively compare Loupe's performance against alternative approaches. The results indicate that Loupe is effective and efficient in enabling accurate verification of P/S architectures.

Index Terms—Publish-subscribe, verification, model-checking.

1 Introduction

The Publish-Subscribe (P/S) communication paradigm [27] is currently used as a foundation to build sophisticated software systems for diverse application domains, from the business context [56], [59], [61] to pervasive and embedded environments [40], [46]. P/S provides a form of asynchronous, implicit, and multipoint communication which supports applications designed in terms of loosely coupled components. Interactions among components are not carved in stone. Rather, they may change over time, for instance, as the context changes [19].

1.1 Problem

The ability to decouple application components is an asset during the design and implementation phases. However, it becomes a major hindrance to the verification of the system behavior. Developers can easily check whether each individual component matches its specification, but reasoning on the overall federation of components is often difficult, as loose coupling allows components to dynamically change their interactions with the others. In addition, the federation itself may change, as components are free to join and leave the system at any time.

Moreover, although the abstractions and APIs offered by existing P/S systems are very similar, the underlying implementations differ in features and characteristics. For instance, P/S systems for mobile environments rarely offer reliable message delivery [13]. Conversely, this feature is almost always provided by P/S systems for enterprise environments, possibly using different message delivery policies [52]. The different *guarantees* characterizing the operation of P/S systems deeply affect how the application behaves. As a result, the verification is further complicated, as its outcome depends on the guarantees offered by the underlying P/S infrastructure.

Verification of P/S architectures has been tackled using model checking. This approach has already been applied to real-world cases [28], [29], providing an early assessment of the effectiveness of these techniques. In these approaches, *both* the application components and the P/S infrastructure are modeled using the model checker's input language. However, this is often ineffective because of state explosion problems. In addition, current modeling languages are mostly domain-agnostic, being designed as general-purpose solutions. This makes it difficult for developers to describe the intended application behavior in terms of P/S operations.

1.2 Loupe

The above issues require a major change of perspective. We tackle this challenge with Loupe, a *domain-specific* model checker. In Loupe, we embed the P/S paradigm within the checking engine of the Bogor model checker [54] by extending BIR (Bogor's input language) with P/S primitives and by modeling their semantics *inside* the tool. Loupe is publicly available [49].

Our approach allows the checking engine to obtain full control of the state space generation. *Domain-specific* abstractions are implemented to drastically reduce the

- *L. Baresi and C. Ghezzi are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, P.zza L. da Vinci 32, 20133 Milano, Italy. E-mail: {baresi, ghezzi}@elet.polimi.it.*
- *L. Mottola is with the Swedish Institute of Computer Science, Isafjordsgatan 22, 16440 Kista, Stockholm Sweden. E-mail: luca@sics.se.*

Manuscript received 7 June 2008; revised 8 May 2009; accepted 12 Feb. 2010; published online 2 Mar. 2010.
Recommended for acceptance by S. Uchitel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-06-0194.
Digital Object Identifier no. 10.1109/TSE.2010.39.

number of states generated during the verification. This enables accurate models at reasonable cost, for instance, accounting for guarantees such as message priorities and different delivery policies without incurring in state explosion problems.

Moreover, a *customized modeling language* which includes the P/S operations as primitive constructs eases the modeling of the application's behavior and also reduces the conceptual gap between modeling primitives and conventional domain abstractions. This simplifies reasoning on the model checker's outcome and exploiting the insights gained from the verification.

1.3 Road Map

In Section 2, we introduce the P/S paradigm, analyze the characteristics of existing P/S infrastructures, and provide a taxonomy of the guarantees they offer. In Section 3, we describe the language extensions we designed to augment Bogor's input language with P/S operations, and illustrate how these can be used to model the behavior of P/S applications. Modeling the semantics of P/S operations inside the checking engine is the subject of Section 4, where we describe our domain-specific state abstractions. Loupe's internals are described in Section 5, along with our use of state-of-the-art software verification tools to assess the implementation of some of Loupe's key functionalities. Section 6 assesses the effectiveness of our approach in a nontrivial case study, investigating Loupe's ability to provide insights into the interplay between the P/S infrastructure and application components. Section 7 reports on a quantitative study of Loupe's performance compared to alternative approaches and analyzes the contribution of each domain-specific abstraction. We conclude with a survey of related approaches in Section 8 and by providing brief concluding remarks in Section 9.

This paper provides a comprehensive treatment of our work on the accurate verification of P/S architectures—whose initial results appeared in [3], [4], [5]—by presenting a thorough description and evaluation of Loupe in its most mature form. The current version of our tool features more accurate and detailed models of P/S systems, provides a stronger foundation for the correctness of the results obtained, and improves the performance of verification.

2 Publish-Subscribe Infrastructures

In P/S infrastructures, application components either *subscribe* to message patterns, expressing an interest in particular data, or *publish* messages by injecting data into the system. A *dispatcher* mediates the communication by storing subscriptions in a data structure called *subscription table*, and by *filtering* published messages against subscriptions. Components are *notified* of messages matching their subscriptions. Since they are all interactions mediated by the dispatcher, components can join and leave the system dynamically without explicit reconfiguration.

From the application perspective, the state of the P/S infrastructure is only determined by the current set of subscriptions. Published messages are transient and the data they carry are not persistent. This is in contrast with other communication paradigms, for example, tuple spaces [30], where data remain in the communication infrastructure until explicitly removed. We leverage this observation to devise most of the domain-specific state abstractions we illustrate in Section 4.

TABLE 1
P/S Guarantees

Guarantee	Choices
Dispatcher guarantees	
Message Ordering	Random, Pair-wise FIFO, System-wide FIFO, Causal, Total, Priority-based, Priority-based w/ Scrunching
Filtering	Precise, Approximate
Subscription Delay	Absent, Present
Replies	Absent, Present
Queue Size	Bounded, Unbounded
Queue Drop Policy	None, Tail Drop, Priority Drop
Per-component guarantees	
Publisher Reliability	Absent, Present
Subscriber Reliability	Absent, Present
Queue Size	Bounded, Unbounded
Queue Drop Policy	None, Tail Drop, Priority Drop
Unannounced Disconnections	Absent, Present

Although most existing P/S infrastructures offer similar APIs, they differ in the guarantees they provide, for example, in terms of reliability or message delivery policies. To make Loupe parametric w.r.t. these aspects, we identified the features that may affect the behavior of applications built on top of P/S infrastructures. These features characterize either how the dispatcher coordinates the overall federation of components or the interactions of a particular component with the dispatcher. Table 1 summarizes the dimensions we identified. On the dispatcher side, we consider:

- **Message Ordering.** In current P/S infrastructures, delivery policies for notifications can be *random*, *pair-wise FIFO* (notifications for messages published by the same component reach the same subscriber in the same order) or *system-wide FIFO* (notifications reach the subscribers in the order they are generated). Delivery policies such as *causal* (notifications maintain the causality relations) and *total* (subscribers receive the same subsets of notifications in the same order) are also provided, for instance, in Gryphon [9], [38]. Systems such as DSWare [46] offer *priority-based* (concurrent notifications are delivered according to their priorities) and *priority-based with scrunching* (a mechanism to avoid starvation by increasing a message's priority after it is rescheduled a number of times).
- **Filtering Mechanism.** The algorithm used by the dispatcher to match published messages against subscriptions determines the notifications to deliver. When subscription tables are expected to be large, *approximate* filtering [48], [53] is preferred to an exhaustive search through all subscriptions. These techniques analyze only a subset of subscriptions or a summary of them, and thus may cause false negatives or false positives, respectively.
- **Subscription Delay.** When the P/S dispatcher is centralized, a filter is immediately active when a component performs the corresponding subscribe. Several systems, however, exploit distributed dispatchers to balance the load [13]. At the extreme, every application component may be coupled with a dispatcher on the same host, e.g., in mobile

environments [51]. In these cases, subscriptions take time to propagate throughout the dispatcher network, both when they are issued and when new components—along with their attached dispatcher—join the system and need to receive information on the existing subscriptions.

- **Replies.** In some applications, components need to reply to notifications. Typically, programmers achieve this functionality at the application level by setting up temporary subscriptions to convey replies back to the original publisher [21]. However, for the same reason discussed above, there is no guarantee that these subscriptions are active at the time of publishing the reply. To overcome this limitation, P/S systems may offer an additional `reply` primitive [21]. This is implemented inside the middleware layer using dedicated mechanisms, for instance, by keeping track of the reverse path to the original publisher.
- **Queue Size.** Modern P/S systems sometimes assume that the dispatcher functionality is deployed on powerful hardware where it is safe to assume that the dispatcher queues may grow arbitrarily. However, systems designed for embedded environments, for instance, DSWare [46], have severe restrictions w.r.t. memory occupancy, and thus drastically limit the number of incoming messages. This may cause message losses because of queue overflows.
- **Queue Drop Policy.** When the queue size is limited, messages are dropped according to different policies, such as *tail drop* (a new message is immediately discarded if the queue is full) or *priority drop* (messages with lower priorities are discarded first). Both guarantees are available, for instance, in DSWare [46].

The interactions between application components and dispatcher are characterized by:

- **Reliability.** Existing P/S infrastructures provide different guarantees concerning reliability of publisher-to-dispatcher and dispatcher-to-subscriber communication:
  - Systems providing **publisher reliability** guarantee that all published messages eventually reach the dispatcher. For example, Gryphon [38] supports this feature.
  - Systems providing **subscriber reliability** guarantee that components receive all notifications for messages that match their subscriptions at the dispatcher. For example, OpenJMS [52] supports this feature.

  This distinction is relevant since message demultiplexing and addressing occur at the dispatcher. Thus, if a message is lost before reaching the dispatcher, *none* of the subscribers is notified and the causality relations among messages are not affected. The whole system, with the exception of the publishing component, remains in the same state as if the message was never published.
- **Unannounced Disconnections.** In some P/S systems, the communication between application components and the P/S dispatcher may be suddenly interrupted, preventing the component from accomplishing further operations on the P/S infrastructure. Some P/S systems, for example, REDS [22], provide application components with means to probe the connection to the dispatcher, while others simply suffer from unannounced disconnections.
- **Queue Size and Drop Policy.** These are essentially the same guarantees we described above for the dispatcher, but here they hold on a per-component basis.

By comparing dispatcher-specific and per-component guarantees, it may appear that the effect of approximate filtering and subscription delays is similar to message losses. However, the latter are generally nondeterministic, and thus happen unpredictably. Differently, subscription delays may cause message losses only if subscriptions are in transit when components publish messages, and thus they are not taken into account at the dispatcher. Approximate filtering, instead, is the result of a fully deterministic matching algorithm.

We can characterize most available P/S systems according to the aforementioned dimensions, as shown in Table 2. The most sophisticated guarantees are usually provided by P/S systems for enterprise applications, for example, JMS-compliant systems. As we move toward mobile and embedded scenarios, the guarantees become weaker, especially in terms of reliability, for example, supporting a best effort policy.

3 Loupe

Loupe is built as an extension to the Bogor model checker [54] by adding P/S-specific constructs to its modeling language. Here, we describe Loupe's language constructs and their use in modeling P/S applications. In our discussion, we use Bogor's basic constructs, which should be easily understandable since they are similar to those commonly provided by existing model checkers. The reader can refer to [5], [54] for more details.

3.1 P/S Operations

Fig. 1 illustrates the preamble that must be included in all Bogor models using Loupe. The signatures are intuitive as they mimic those found in real P/S systems. The P/S infrastructure is made available as a generic abstract data type. An instance of this data type represents a connection between an application component and the P/S dispatcher, and is customized based on the type of messages exchanged.

Fig. 2 shows a model of two application components interacting via P/S using Loupe. Every component is mapped onto a Bogor thread. The model includes a `Publisher` that generates a message possibly received by a `Subscriber`. Initially, the `Subscriber` opens the connection to the dispatcher using the `register` operation (`loc0`), specifying as argument the length of the input queue for this connection (`0` is used to denote an unbounded queue), the policy used to drop messages in case of overflows, and three Boolean values stating whether the connection provides publisher/subscriber reliability and if the component may be subject to unannounced disconnections. Next, the component issues a subscription using `subscribe`. In our

TABLE 2
Examples of Existing P/S Systems Classified along the Dimensions of Table 1

Guarantee	OpenJMS [52]/ ActiveMQ [1]	Gryphon [38]	DSWare [46]	Siena [13]	REDS [22]	Mires [57]
Dispatcher guarantees						
Message Ordering	Pair-wise FIFO	Total/Random	Random	Random	Pair-wise FIFO/Causal	Random
Filtering	Precise	Precise	Precise	Precise	Precise/Approximate	Approximate
Subscription Delay	Absent	Present	Present	Present	Present	Present
Replies	Present	Absent	Absent	Absent	Present	Absent
Queue Size	Unbounded	Bounded	Bounded	Unbounded	Unbounded	Bounded
Queue Drop Policy	Priority Drop	Tail Drop	Tail Drop	None	None	Tail Drop
Per-component guarantees						
Publisher Reliability	Present	Present	Absent	Present	Absent	Absent
Subscriber Reliability	Present	Absent	Absent	Absent	Absent	Absent
Queue Size	Bounded	Bounded	N/A	Bounded	Bounded	N/A
Queue Drop Policy	Tail Drop	Tail Drop	N/A	Tail Drop	Tail Drop	N/A
Unannounced Disconnections	Absent	Absent	Absent	Present	Present	Absent

approach, the filter is an arbitrary Boolean function accepting a message as input (in the example, `isGreaterThanZero`). The value returned tells whether the message matches the filter. This approach supports high expressive power in specifying the matching semantics, in contrast to earlier work that severely constrained the nature of filters [29]. Finally, the `Publisher` component is started.

The `Publisher` opens a connection to the P/S dispatcher using `register`. Next, in `loc1`, it creates a new message carrying a constant value. The message is given as input to the `publish` operation, which requires as additional parameters the connection the message is being sent over (`ps`) and the message priority. Concurrently, the `Subscriber` component moves to `loc1` where the `waitingMessage` expression acts as a guard, preventing a further transition from firing until there is at least one message in the input queue. If so, the component executes the `getNextMessage` operation, passing a reference to an empty message as a parameter. Using Bogor's `lazy` modifier as a "pass-by-reference," the empty message is filled with the information received from the publisher.

Besides the P/S operations used in this example, Loupe also provides an `isConnected` expression, as well as `unsubscribe` and `reply` operations. The former can be used as a guard to check whether a connection to the P/S dispatcher is active at the time of performing an operation. The `unsubscribe` operation has the opposite semantics of `subscribe`, and may experience the same propagation delays we discussed in Section 2. The `reply` operation can be used only when Loupe is configured to model a P/S infrastructure supporting replies. Otherwise, our tool signals an exception and aborts the verification.

3.2 Timing Aspects

A large body of work exists on model checking real-time systems (e.g., see [2]). Our objective, however, is not to embed a generic notion of time, but rather to include only the temporal aspects relevant to P/S architectures. Our approach builds on the work by Deng et al. [24] and extends it to account for different P/S guarantees and message delays.

In Loupe, one can control the *execution rate* of components and *message delays*. The former dictates the maximum frequency of a component's interactions with the P/S dispatcher, i.e., the number of publish, (un)subscribe, and reply operations allowed in a time unit. A relevant class of real systems can be modeled similarly (e.g., [26], [32]).

```
typealias MsgPriority int (0,9);
enum DropPolicy {NO_DROP, TAIL_DROP, PRIORITY_DROP};
extension PSConnection for polimi.bogor.loupe.PubSubModule {
  typedef type<'a>;
  // Opening a connection
  expdef PSConnection.type<'a>
    register<'a>(int, DropPolicy, publisher_reliability, subscriber_reliability, disconnection);
  // Checking a connection
  expdef boolean isConnected<'a>(PSConnection.type<'a>);
  // P/S operations
  actiondef subscribe<'a>(PSConnection.type<'a>, 'a -> boolean);
  actiondef unsubscribe<'a>(PSConnection.type<'a>, 'a -> boolean);
  actiondef publish<'a>(PSConnection.type<'a>,'a, MsgPriority);
  actiondef reply<'a>(PSConnection.type<'a>, 'a, MsgPriority);
  // Receiving notifications
  expdef boolean waitingMessage<'a>(PSConnection.type<'a>);
  actiondef getNextMessage<'a>(PSConnection.type<'a>, lazy 'a);
}
```

Fig. 1. Loupe P/S preamble.

```
// Message definition
record MyMessage { int value; }
MyMessage receivedEvent := new MyMessage;
// Filter definition
fun isGreaterThanZero(MyMessage event) returns boolean = event.value > 0;
active thread Subscriber() {
  PSConnection.type<MyMessage> ps;
  loc loc0:  // Connection setup and subscription
    do { ps := PSConnection.register<MyMessage>(0, NO_DROP, true, true, false);
         PSConnection.subscribe<MyMessage>(ps, isGreaterThanZero);
         start Publisher();
    } goto loc1;
  loc loc1:  // Message receive
    when PSConnection.waitingMessage<MyMessage>(ps) do {
         PSConnection.getNextMessage<MyMessage>(ps, receivedEvent);
    } return;
}
thread Publisher() {
  MyMessage publishedEvent;
  PSConnection.type<MyMessage> ps;
  loc loc0:  // Connection setup
    do { ps := PSConnection.register<MyMessage>(0, NO_DROP, true, true, false);
    } goto loc1;
  loc loc1:  // Publishing a message
    do { publishedEvent := new MyMessage;
         publishedEvent.value := 1;
         PSConnection.publish<MyMessage>(ps, publishedEvent, 0);
    } return;
}
```

Fig. 2. Modeling a publisher and a subscriber component in Loupe.

Message delays are modeled by mapping the traveling time to the execution rates of the intended receivers. The way timing constructs are modeled inside Loupe is described in Section 4. Hereafter, we discuss how they can be used to specify a P/S application.

To control timing aspects, designers include the preamble of Fig. 3, which essentially refines the one of Fig. 1. The `register` operation now takes three additional integer parameters, specifying the execution rate for the registering component and the upper and lower bound of a (discrete) random delay that messages experience when addressed to this component. The execution rates of components are controlled by guard `canProceed`, which application designers must prepend to every P/S operation. The guard yields true iff the component can perform a state transition without violating any time constraint. The `waitingMessage` expression now returns a value among: 1) `CAN_PROCEED`, 2) `CANNOT_PROCEED`, and 3) `QUEUE_EMPTY`, meaning that 1) the component can proceed and the incoming queue is nonempty, 2) the component cannot proceed without violating the time model, and 3) the component can execute, yet its input queue is empty. In the last case, a component may decide to perform a different P/S operation or it can suspend itself using `suspend` until at least one message arrives in its queue.

The current implementation of Loupe checks for the correct use of the timing constructs by raising exceptions if the model is structured incorrectly, for instance, if the `canProceed` guard is not used before every P/S operation.

3.3 Verification

As previously illustrated, the `register` operation specifies the assumed per-component guarantees. Dispatcher guarantees are specified in a separate configuration file that Loupe parses before starting the verification. Different instances of

```
includes "PSConnection.bir"
enum ExecGuards {QUEUE_EMPTY, CAN_PROCEED, CANNOT_PROCEED};
extension TimedPSConnection for polimi.bogor.loupe.rate.TimedPubSubModule {
  typedef type<'a>;
  // Opening a timed connection
  expdef TimedPSConnection.type<'a>
    register<'a>(int, DropPolicy,
                 publisher_reliability, subscriber_reliability, disconnection,
                 int, int, int);
  // Receiving notifications
  expdef ExecGuards waitingMessage<'a>(TimedPSConnection.type<'a>);
  // Timing guards
  expdef boolean canProceed<'a>();
  actiondef suspend<'a>();
}
```

Fig. 3. Loupe preamble for timing—expressions and operations not included here remain the same as in Fig. 1.

the checking engine are generated depending on the specified configuration. During the verification, Loupe is triggered whenever any of the operations in the preamble of Fig. 1 (or Fig. 3) is executed. This allows our tool to control how the state space evolves, depending on the assumed P/S guarantees. For the remaining operations, the verification exploits the standard procedures inside Bogor, including the ones used to verify assertions and properties and to check for deadlocks. For instance, running Loupe with the example of Fig. 2 and the dispatcher guarantees of OpenJMS in Table 2, the model is found to be correct as it is deadlock-free and we did not specify any property or assertion to check.

Loupe allows designers to explore the interplay between the application model and the guarantees provided by the P/S infrastructure. Doing so is as simple as changing some values in Loupe's configuration file or modifying some of the parameters given to the `register` operation. For instance, by setting the `publisher_reliability` parameter to `false` in Fig. 2, designers are able to study a scenario where published messages may not reach the dispatcher. The verification of the model in Fig. 2 now fails: The transition specified in `loc1` of the `Subscriber` component may be never enabled if messages do not reach the dispatcher. Therefore, the system may enter a deadlock state. Designers may then decide to assume that the underlying P/S system provides publisher reliability or to account for this issue at application level.

Loupe enables the reasoning above based on a model of the application at hand. Once the key design choices are settled and the application functionality is accordingly verified, Loupe models may be input to a code generation tool to produce a running implementation. Our tool thus provides a stepping stone for this process. Generating running code, as well as testing the resulting implementation, is beyond the scope of our work and can be achieved with existing techniques [63], [64].

4 Domain-Specific Abstractions and Heuristics

Embedding the P/S infrastructure within the model checker enables the implementation of domain-specific state abstractions to reduce the number of states generated during the verification. In addition, we leverage dedicated heuristics to take advantage of the interplay between P/S guarantees and timing aspects. Both features are described next.

4.1 State Abstractions

In P/S architectures, the information determining the system state is mostly inside the application components, not in the communication infrastructure. By modeling the P/S infrastructure inside the model checker, we get close to this ideal picture. In contrast, by modeling the dispatcher through the model checker's input language, we would expose the dispatcher's internals as an explicit part of the state, although these are transparent to the application.

Here, we illustrate the aspects that we deem most important to provide the above degree of abstraction over the communication infrastructure. Throughout the discussion, we first describe the specific feature of the P/S paradigm that motivates the abstraction, and then discuss how it is supported in Loupe.

4.1.1 Subscription Table

P/S systems are expected to deal with a large number of subscriptions [51]. Typically, however, a component is notified once, regardless of how many of its subscriptions match a message. The dispatcher is thus free to examine the current subscriptions in any order, provided that all of them are eventually checked. Also, once a subscription matches, it is unnecessary to examine the same message against other subscriptions issued by the same component.

Abstraction in Loupe. Taking advantage of the above considerations is fairly complex when the P/S infrastructure is modeled using a model checker's input language (e.g., [29]). Normally tables that store the same subscriptions, but in different orders, correspond to different executions during the verification. Similarly, notifications addressed to the same component but generated by different subscriptions yield different executions. From the application perspective, all such executions are equivalent. Loupe leverages this observation by abstracting away executions that differ only in the ordering of subscriptions, or where the same component is notified of the same message because of different matching subscriptions. The functionality to detect such situations is hidden inside Loupe, and hence no explicit states are generated.

4.1.2 Multipoint Communication

In contrast to traditional interaction paradigms such as client-server, P/S is inherently multipoint. A single published message may cause multiple notifications delivered to different subscribers. Moreover, the binding between publishers and subscribers is implicitly determined by the current set of subscriptions, and may thus change over time.

Abstraction in Loupe. To the best of our knowledge, this style of interprocess communication is not supported natively by existing model checkers. The closest example in this respect is Promela channels [36]. However, they describe point-to-point interactions, and most importantly, it is not possible to create channels dynamically to model subscribe operations. A way to circumvent this problem might be to demultiplex at a fictitious, additional process. By doing so, however, the checking engine would generate one or more explicit states for every published message. This operation, however, is atomic from the application perspective. In Section 7.3, we show quantitatively how this impacts on the number of states generated during the verification. Alternatively, designers might overprovision the number of channels and use them as a pool. As already observed in [50], however, this method would unnecessarily increase the size of the state vector. In Loupe, demultiplexing and addressing occur within the checking engine, and no additional states are generated to handle them.

4.1.3 Message Filtering

P/S supports content-based information filtering. Often, a large fraction of published data is filtered out before it reaches any subscriber [35]. Publish operations with no matches, however, have no effect but on the publishing

component. From the perspective of the rest of the system (including the dispatcher), it is as if the above publish operation never occurred.

Abstraction in Loupe. Modeling the P/S dispatcher alongside the application components necessarily exposes the bookkeeping data needed during the filtering process. For instance, the approach presented in [62], based on Promela, generates explicit states even during the evaluation of a filter that eventually generates no matches. This is unavoidable in Promela, as the filtering process is too complex to be expressed in a single atomic step. If the dispatcher determines that no notifications are to be sent, these states are useless. In contrast, in Loupe, the filtering process is not exposed to the checking engine. Therefore, if no subscriptions match a message, no additional states are generated but the one for the publishing component.

4.1.4 Message Ordering

To enforce a specific delivery ordering, the dispatcher must track a sizable amount of routing information. Generally, it needs to be aware of which processes published which messages within a given time window. For instance, with N components in the system, totally ordered delivery requires an $N \times N$ matrix of integers representing per-component logical clocks [60]. If a message is published whose delivery would violate total ordering, it is temporarily buffered at the dispatcher and some values in the matrix are modified, changing the dispatcher's internal state. Again, this operation is transparent to the application.

Abstraction in Loupe. Modeling any specific message ordering alongside the application, as traditional approaches do, causes the model checker to explore multiple states that are actually equivalent from the application perspective. For instance, modeling total ordering as in [62] causes SPIN to generate a new state for every modification of the values in the aforementioned matrix. In addition, some ordering policies only partially constrain the set of possible executions. Total ordering, for instance, only dictates that messages must be received in the same order, without specifying the exact intramessage schedule. Therefore, different contents of the routing matrix above may lead to the same ordering of notifications [44]. While the checking engine would generate all permutations of message deliveries to fully explore the state space, each combination may be reflected in the same information stored in the dispatcher's routing matrix. This corresponds to further explicit states if the dispatcher's internals are exposed to the checking engine. In Loupe, this information is hidden within the verification engine. No additional states are generated due to routing information being updated at the dispatcher. In Loupe, different delivery ordering guarantees thus show the *same* overhead in terms of states generated, regardless of their complexity.

4.1.5 Message Loss

As in any distributed infrastructure, in P/S architectures, messages may be lost for a number of reasons. Published messages may be lost on the way to the dispatcher and notifications may be lost before getting to the subscribing components. Notifications may also overflow in the incoming queue of application components or of the dispatcher if its size is limited. In addition, subscribers may not receive their notifications because of approximate filtering and/or propagation delays. Regardless of the reason, the loss occurs within the communication infrastructure. Application components should therefore not be affected by the particular cause of a message loss. Rather, they should only see its effect.

Abstraction in Loupe. Using standard model checking approaches, the cost of accounting for different causes of message loss would be prohibitive since losses due to different causes would be treated independently. This may lead to a set of executions that the model checker perceives as different, and yet they are equivalent from the application's perspective. To address this issue, in Loupe, the decision whether a message is lost is taken only once, depending on the combination of P/S guarantees the designer selected. This is possible in our tool as the checking engine is aware of the complete system state. Thus, once again, our solution does not generate multiple executions that are equivalent from the application perspective.

4.2 Timing Heuristics

To model timing aspects in Loupe, we replace Bogor's state space exploration module with a custom one to account for message delays and component execution rates. The latter is the maximum rate at which components execute P/S operations. Given a set of components C_i with execution rates $\mathcal{R}_i$, time is divided into *frames* of different length f_i, where $f_i = 1/\mathcal{R}_i$. We define an *hyperperiod* (*hp*) to be the least common multiple frame among the different f_i. Our state space exploration module abstracts time as discrete "ticks" corresponding to the passing of time in the highest rate component (shortest frame). Based on this, every component C_i, with $hp = k \cdot f_i$ for some integer k, is scheduled to perform at most k P/S operations in every hyperperiod.[1] Loupe explores all possible interleavings of P/S operations executed within the same hyperperiod and resets the internal representation of time at the end of it.

Fig. 4 depicts an example where two components perform a sequence of P/S operations. In this example, $\mathcal{R}_1 = 3/2 \cdot \mathcal{R}_2$; thus, the hyperperiod is equal to three frames of component 1 (or two frames of component 2). The scheduling of operations shown in Fig. 4a is correct, as component 1 executes three operations before component 2 executes its third one, that is, all operations have been performed in a hyperperiod by component 1. In contrast, the execution of Fig. 4b is invalid, as component 2 must not proceed to the following hyperperiod before component 1 performs its third P/S operation.

Message delays are modeled by relating their traveling time to component execution rates, thus mapping message latency to a given multiple of the shortest frame. In case messages are in transit at the end of an hyperperiod, they are realigned to the beginning of the following hyperperiod. Loupe also applies a basic form of partial order reduction [14] to model random message delays. The objective is to identify the minimum set of concurrent executions that must be checked for the verification to be complete. At the

1. If a component does not perform any P/S operation, it is moved to the next frame to avoid blocking the scheduler.

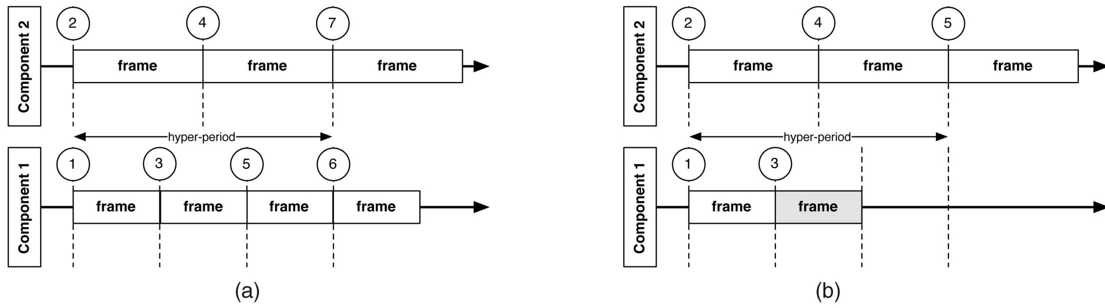

Fig. 4. Timed executions in Loupe. The numbers in circles represent a system-wide schedule of operations. (a) Example of correct execution. (b) Example of invalid execution.

beginning of every hyperperiod and after every P/S operation, Loupe performs the following steps:

1. It identifies the set of notifications that are received within the current hyperperiod. This set is determined by the possible delays of messages: Loupe checks every possible (discrete) value within the bounds for message delays specified in `register`, marking the values that allow the notifications to arrive at the subscribers before the end of the hyperperiod.
2. It partitions the notifications into subsets that involve nonoverlapping subsets of subscribers. These correspond to independent transitions in the state space. Therefore, the verification is also complete if Loupe considers only one of the possible interleavings.
3. Within each subset of independent transitions, it identifies the values of message delays that generate different delivery orderings at the target components. This is needed to avoid exploring executions characterized by different message delays that would not impact the execution at the receiving component.
4. It generates the states representing the delivery of the first notification according to different delivery ordering identified in the previous step, and hands these states over to Bogor. This can now proceed with the verification, eventually triggering Loupe again.

This way, we only generate executions that differ in the interleavings among components being notified of the same messages, or in different delivery orderings at the same component.

Our approach also lends itself to the use of heuristics exploiting the interplay between timing aspects and the P/S guarantees in Table 1. For instance, when Loupe models a P/S system providing causal order, it often happens that a message might be delivered before others (e.g., because it experiences a smaller delay), but so doing would violate the ordering because some causally connected message is still in transit. In our experience, the impact of this situation on the number of enabled transitions is much greater than that imposed by the time model alone. Therefore, whenever possible, we apply the mechanisms that model message orderings *before* computing the intracomponent schedule to reduce the number of states possibly visited.

Another example deals with situations where `waitingMessage` returns `QUEUE_EMPTY`. If so, the corresponding component passed the time checks. Thus, the checking engine lets another component proceed and reschedules the first component later without rerunning the time checks. This is correct because once a component passed the time checks, there is no way for another component to create a situation where the first component is no longer allowed to proceed. Based on this, we can alleviate the processing overhead generated during the verification.

5 Loupe Internals

We illustrate the architecture and implementation of Loupe and report on how we assessed the implementation of some key Loupe's functionality.

5.1 Architecture

Loupe's architecture is shown in Fig. 5. The tool, implemented in Java, is designed to decouple the modeling of different guarantees and yet to provide the necessary hooks to exploit the interplay among them. The `PubSubModule` class implements the model of the dispatcher guarantees and directly interacts with Bogor's checking engine. Most of the domain-specific abstractions take place within this module. `TimedPubSubModule` is a refinement of `PubSubModule` that implements our timing heuristics. The scheduling of P/S operations performed by application components is handled by an independent `Scheduler` module, while message delays are modeled inside `DispatchingManager`.

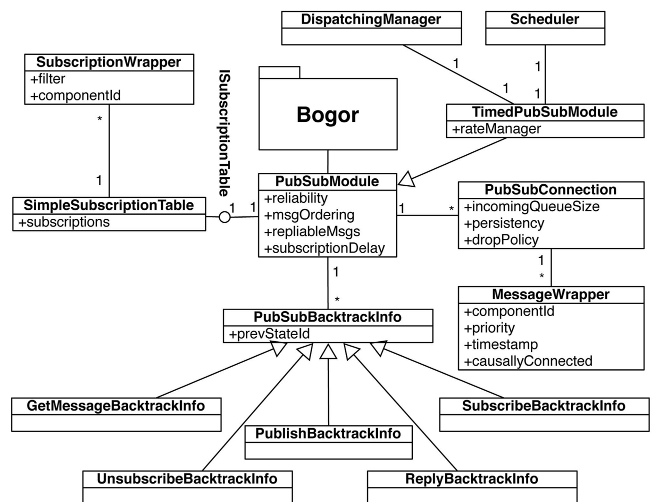

Fig. 5. Loupe architecture.

The `PubSubConnection` class represents a connection to the P/S infrastructure. It stores the set of messages in a component's input queue and enforces a particular message ordering. It also implements the reliability model described in Section 2. `MessageWrapper` is used to piggyback additional information on messages, for instance, to store references to causally connected messages when causally ordered delivery is assumed.

Information on previous states, required to backtrack actions, is stored in dedicated classes. The `PubSubBacktrackInfo` class factors out information common to all P/S operations, while dedicated classes are used to retain operation-specific information. For instance, the `PublishBacktrackInfo` class stores the message corresponding to the publish operation to backtrack.

The filtering process is decoupled from the rest of the architecture, being modeled inside an independent class with a specific `ISubscriptionTable` interface. This allows one to experiment with different filtering mechanisms. Loupe's current implementation includes three such schemes: a solution based on hash maps, an approximate filtering mechanism ported from the REDS middleware [22], and a simplified implementation of the scheme by Ouksel et al. [53]. Different filtering mechanisms may be easily incorporated if necessary, or even ported form existing systems to reflect the semantics expected in the final implementation.

5.2 Assessing Loupe's Implementation

We carried out a hybrid approach to assess Loupe's implementation. Portions of Loupe's code were tested using traditional techniques, while critical parts were formally verified on a set of significant scenarios. In this section, we focus on the latter methodology, as it brings interesting insights into the limitations of today's software verification tools and how these can be overcome.

We chose Bandera [17], a tool for the automatic verification of Java programs. Notably, Bandera itself is based on Bogor, as it translates Java code into Bogor models. Programmers instrument Java code by expressing pre and postconditions on the values of method parameters. The instrumented code is given as input to Bandera, where code analysis techniques are employed to reduce the size of the model handed over to Bogor, for instance, by eliminating portions of code that are not relevant to check the properties of interest. In our case, however, these techniques were insufficient to yield tractable models. In the following, we illustrate how we managed to overcome this limitation.

We focus on the modeling of causal and total ordering, as well as on the generation of correct component schedules when timing aspects are accounted for, hence also checking the time heuristics and partial order reduction described in Section 4.2. Indeed, these are the most subtle features of our tool.

5.2.1 Slicing

A brute-force approach with the whole Bogor code plus Loupe given as input makes Bandera fail the translation. This is essentially due to the use of Java reflection in Bogor to dynamically load the extension classes. Bandera cannot handle this feature, as it makes the control flow dependent on the class being loaded. Therefore, we manually linked the implementation of Loupe to the rest of Bogor. In addition, Bandera refuses to process Java classes with direct bindings to the underlying virtual machine. Therefore, we also removed all references to Java native libraries. For instance, in the case of functionality for file I/O, we hard-coded the clear text that Bogor would read from files in the code itself.

Although Bandera was now able to complete the translation, the resulting models were intractable. A closer look at Bandera's output revealed that large portions of the input code were processed unnecessarily. Most often, this is due to situations where there exists some execution path that Bandera cannot exclude because of the lack of runtime information. In almost all cases, however, we could safely carve out only the relevant portions of code based on our knowledge of Loupe internals and the properties to verify. Thus, we manually assembled the minimal functionality of Loupe plus a handful of Bogor classes necessary to carry out the validation. At the end of this process, the code input to Bandera included:

- The subset of Loupe modules strictly needed to run the verification with a given combination of P/S guarantees. For instance, if system-wide FIFO is not assumed, some code can be eliminated as it will never be used.
- The minimal Bogor code to create the initial system state. In doing so, we almost completely eliminated the BIR parser by hard-coding most of the information that Bogor would normally read from the input models.
- The code to *generate* the state space, but *not* the one driving its exploration. Indeed, this is dictated by our timing heuristics, which are already part of Loupe.

In quantitative terms, the above functionality accounts for 411 Java classes, about 32,000 methods, and over 400,000 Java statements. The models output by Bandera at this stage become tractable.

5.2.2 Causal Ordering

A P/S infrastructure providing causally ordered delivery must satisfy the following condition for any two messages m and m':

$$Publish(m) \rightarrow Publish(m') \Rightarrow Notify(m) \rightarrow Notify(m'), \quad (1)$$

where $\rightarrow$ indicates the *happens-before* relation [45]. In Loupe, the model of causally ordered delivery is implemented in a single Java method that returns an ordered list of notifications addressed to a given component. Therefore, property (1) is stated as a postcondition to the aforementioned method by referring to sequence numbers in messages to capture the happens-before relation. In contrast, whenever the antecedent of condition (1) does not hold, Loupe must generate all possible interleaving of message deliveries. We specified this property as a postcondition of the method implementing causal ordered delivery, using a fragment of Java code to compute the

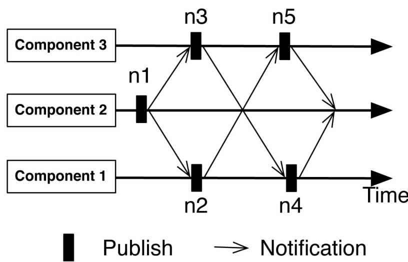

Fig. 6. Distributed execution corresponding to the bug found by Bandera. The receipts of $n4$ and $n5$ are *not* causally related, although our implementation incorrectly recognized the opposite. (Communication to/from the dispatcher is not shown.)

possible message permutations given the notifications currently being delivered.

We used a scenario where three components alternate in publishing messages, creating all possible combinations of publish operations from different components. Only two components cannot create a situation where causal ordering is violated without violating pairwise FIFO ordering as well. Since we verified the latter policy using traditional testing, these scenarios are already covered. On the other hand, any additional component beyond the three we use would not generate situations that cannot be mapped to a distributed execution with three components [43]. We checked property (1) using a setting without time constraints.

Bandera revealed a subtle bug in our implementation. Fig. 6 depicts the distributed execution corresponding to one of the counterexamples returned. The situation is rather pathological: The receipts of $n4$ and $n5$ are *not* causally related. Therefore, Loupe should generate two executions at component 2, corresponding to the receipt of $n4$ before $n5$ and vice versa. Because of a missing recursive call in our code, Loupe incorrectly recognized the two messages as being causally related, forcing either of the two to be received before the other. Bandera consequently failed the verification. It took us a couple of days to understand Bandera's output and recreate the situation in Fig. 6, as Bandera's counterexamples do not easily relate to the original Java code. However, once we figured out the conditions under which the verification failed, fixing the problem was straightforward.

5.2.3 Total Ordering

A system provides totally ordered delivery if the same subsets of notifications are received in the same order by the same components [44]. Thus, if both message m and m' are delivered to both component $C1$ and $C2$, we must guarantee the satisfaction of the following condition:

$$Notify(m)_{C1} \rightarrow Notify(m')_{C1} \Leftrightarrow Notify(m)_{C2} \rightarrow Notify(m')_{C2}. \quad (2)$$

Note that when no two components receive the same two messages and thus the definition above does not apply, total ordering does not prescribe any delivery ordering. As we did for causal ordering, we can specify this property as a postcondition to the method in Loupe responsible for scheduling messages according to total ordering.

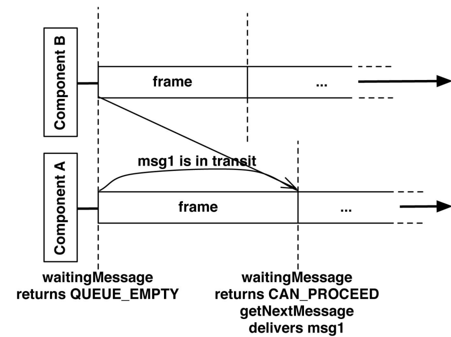

Fig. 7. A scenario to verify component execution rates and message delays.

As input models, we designed a scenario with three components taking turns in publishing messages. This is needed to check the two possible conditions of interest [44]: 1) two components receiving the same two notifications published by a third component[2] and 2) no two components receiving the same two notifications. The former serves to check that the specification of total ordering is satisfied, whereas the latter controls that all possible interleavings of message receptions are explored when definition (2) does not apply. This time, the verification succeeded immediately.

5.2.4 Time Extension Validation

To check the behavior of Loupe when timing aspects are accounted for, it is important to observe that our time extension does *not* alter the individual system states. Rather, it limits the way the state space is explored by excluding sequences of operations that violate the time model. Based on this, the implementation of our time extension can be checked by ensuring that the guards controlling the component schedules return the right values in the right order, as the guards themselves implicitly slice the state space.

We devised a set of input models to investigate the situations that may arise when scheduling components with different execution rates and with the possible presence of messages in transit. These scenarios trigger different combinations of values returned by `canProceed` and `waitingMessage` in Fig. 3. To this end, we use four scenarios with two components:

- **Scenario 1.** This is the case of Fig. 4a, where two components publish messages with a noninteger ratio between their execution rates. There are no active subscriptions; hence, messages are discarded at the dispatcher. The scenario essentially checks whether the intercomponent schedules are correct when no messages are in transit.
- **Scenario 2.** Fig. 7 shows a situation where `waitingMessage` must return `QUEUE_EMPTY` while the message is in transit, and switch to `CAN_PROCEED` when the message arrives. The execution rates are assigned in a way that only the receiving component is allowed to proceed upon message reception. The scenario verifies the correct behavior of `waitingMessage`, and how the intercomponent schedule is

2. Note that a component is not notified of locally published messages.

generated when a message is traveling toward a component that should immediately execute.

- **Scenario 3.** Dually w.r.t. the previous scenario, here the execution rates are assigned in a way that forces the publishing component to execute first, even if the subscriber has a notification waiting in its input queue.
- **Scenario 4.** To test the combination of scenarios 2 and 3, component execution rates and message delays are assigned so that *both* components can be scheduled when the message arrives at the subscriber. This checks if the two possible schedules are correctly generated.

We determined offline the correct schedules in these scenarios. Based on this, we could uniquely determine the values returned by `canProceed` and `waitingMessage` based on the system state at the end of the previous hyperperiod. We specified this as a precondition for the methods implementing the semantics of `canProceed` and `waitingMessage`, and the values we expected as postconditions.

This time, we discovered another bug. Bandera showed a counterexample in the third scenario where `waitingMessage` returned the wrong value after backtracking from the state that represents component A receiving the message. This was caused by a noninitialized variable whose default value worked for most (but not all) combinations of the input parameters.

5.2.5 Summary

Bandera checked the correctness of our implementation w.r.t. the scenarios and properties we specified in reasonable time and with moderate resource consumption. Despite the diversity of the mechanisms being checked, the Bogor models output by Bandera involved a comparable number of states (about 130,000) and the process completed within half an hour, occupying at most 160 Mb of memory in the worst case, i.e., the time extension.

Although the use of Bandera required a significant effort, its results were beneficial. Without undertaking a similar effort, the bugs we found would have remained uncaught. This increased our confidence in the soundness of our domain-specific extensions and heuristics.

6 Case Study

Loupe has been used for the verification of P/S architectures ranging from control of road tunnels [5] to remote assistance to elderly people [3]. Here, we illustrate the use of Loupe in the design of an information system for transport scenarios [23].

6.1 Scenario and Requirements

Consider the problem of monitoring a fleet of buses in a metropolitan area. The scenario is a realistic one, as demonstrated by large efforts currently under way [23], [58]. A system to achieve this goal is composed of the following actors:

- *Buses* traveling along a route, equipped with sensors to detect the number of passengers and a GPS receiver to determine the current stop. These data are published along with notifications of possible bus breakdowns.
- *Bus stops* along a route, equipped with displays that show information about buses (e.g., on time or delayed) and alerts about incoming buses.
- *In-field personnel*, equipped with devices to receive breakdown notifications, move across routes to support bus drivers.
- The fleet *headquarter*, where operators monitor breakdowns within the fleet. If there is a breakdown, they send out replacement buses and inform the passengers along the routes affected.

Because of the dynamic interactions in this scenario, developers must carefully verify their design. Sample requirements to meet are as follows:

- **R1:** In case of a breakdown, all stops along involved routes must eventually display an alert message that a breakdown occurred.
- **R2:** In case of a breakdown, all stops along involved routes must eventually display a message informing that a replacement bus is in operation.
- **R3:** In case of a breakdown involving a bus with more than P passengers, members of the in-field personnel within T stops from the mishap must be eventually notified.
- **R4:** Position updates from the same bus must appear in the order in which they are issued when displayed at a given stop, in order to not confuse travelers with inconsistent information.

We describe next how we model this scenario and specify the requirements above.

6.2 Model

6.2.1 Components

We map every actor in our scenario to a P/S component. Table 3 illustrates the corresponding subscriptions. Buses dynamically join the system at the beginning of their route and leave once it is over. Likewise, in-field personnel enter the system at a any point in time. Buses publish their current route, position, number of passengers aboard, and breakdown notifications. The headquarters subscribes to information reporting breakdowns. After possibly receiving such notification, the headquarters eventually publishes information on a replacement bus sent out. The in-field personnel are interested in breakdown notifications when the route involved is the one they are currently inspecting, the location of the breakdown is within T stops from their location, and the breakdown involves a bus with more than P passengers aboard. Bus stops receive information from the buses if they are yet to pass by. They also subscribe to information reporting breakdowns and replacement buses.

As an example,[3] Fig. 8 shows a finite-state model describing how a bus stop controls the information displayed at the stops. During normal operation, notifications are used to display information on bus positions

3. The finite-state model is only comprised of states and transitions corresponding to the interaction with the P/S infrastructure to avoid cluttering the figures.

TABLE 3
Subscriptions in the Transport Scenario

Component	Identifier	Format
Headquarter	S1	breakdown = true
In-field personnel member	S2	route = this.route AND stop = this.stop ± T AND passengers > P AND breakdown = true
	S3	route = this.route AND replacement = true
Bus Stop	S4	route = this.route AND replacement = true
	S5	route = this.route AND stop = this.stop -1
	S6	route = this.route AND stop < this.stop -1
	S7	route = this.route AND stop = this.stop
	S8	route = this.route AND breakdown = true

The keyword `this` *is used to refer to the state of the subscribing component at the time of issuing the subscription.*

($notify(S6)$) or a "bus approaching" message, indicating that the bus reached the previous stop along the route ($notify(S5)$). The display then returns to normal operation when the bus is at the stop ($notify(S7)$). In case of a breakdown ($notify(S8)$), the display shows a message to inform travelers about the problem. When the headquarters publishes information on a replacement bus, a proper message is displayed and the system eventually returns to normal operation ($notify(S4)$).

The finite-state models for the remaining components are similarly specified. We omit them for space reasons. The reader can refer to [6] for a complete description.

6.2.2 Properties

To specify this scenario in Loupe, we map each component to a Bogor thread. The threads are grouped into four sets $Buses$, $Stops$, $Personnel$, and $Headquarter$, depending on the actor they model. Because of their dynamic nature, we mark components in $Buses$ and $Personnel$ as susceptible to unannounced disconnections. To express the properties to check, we use LTL[4] and the corresponding Bogor plug-in [10]. Requirement **R1** is expressed as

$$\forall b \in Buses, \Box(Breakdown_b \rightarrow \diamond(\forall s \in Stops | s.route = b.route, DisplayBreakdown_s)). \quad (3)$$

Requirement **R2** is specified as

$$\forall b \in Buses, \Box(Breakdown_b \rightarrow \diamond(\forall s \in Stops | s.route = b.route, DisplayReplacementBus_s)). \quad (4)$$

Requirement **R3** is specified by referring to the state of the in-filed personnel as follows:

4. Loupe is independent of how properties are specified as long as a suitable Bogor plug-in is available. In our experience with Loupe so far, we used LTL. We indeed foresee the integration of our tool with approaches that generate LTL formulas from user-friendly graphical formalisms [39], [62]. This will ultimately provide an easy-to-use and efficient verification tool.

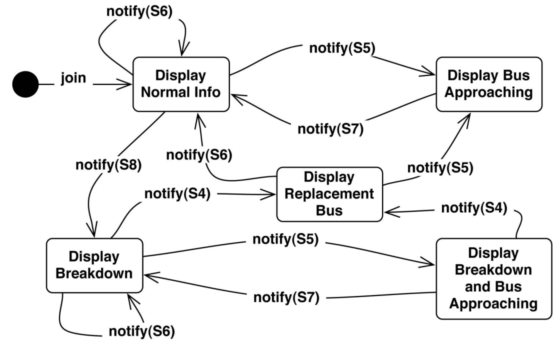

Fig. 8. Finite-state machine modeling a bus stop. Transitions marked with $notify(S_i)$ are enabled when subscription S_i matches and the component receives the corresponding notification.

$$\forall i \in Buses | i.passengers > P, \Box(Breakdown_i \rightarrow \diamond(\forall p \in Personnel | p.route = i.route \wedge p.stop = i.stop \pm T, MovingToBreakdown_p)). \quad (5)$$

Requirement **R4** is expressed by keeping track of an integer time stamp embedded within messages. The sequential condition is enforced by requiring the counter to be monotonically increasing between subsequent notifications:

$$\forall s \in Stops, \Box(\forall b \in Buses | p.route = b.route, CurrentUpdate_{s,b} > LastUpdate_{s,b}), \quad (6)$$

where $CurrentUpdate_{s,i}$ and $LastUpdate_{s,i}$ are the time stamps of the last and current notification at stop s relative to bus b. Note that **R4** does not mandate reliable communication: As long as even a subset of messages is delivered in the correct order, the system satisfies **R4**.

6.3 Verification

Hereafter, we discuss how Loupe supports developers in analyzing the trade-offs of different design decisions, such as those concerning the choice of P/S support and, in particular, the guarantees it offers.

6.3.1 Reliability/Disconnections

In our scenario, a reasonable choice at an initial design phase is to consider a P/S system for mobile scenarios, e.g., assuming the guarantees provided by systems such as the REDS middleware, described in Table 2. The corresponding guarantees already suffice to verify property **R4**.

However, Table 2 shows that P/S systems for mobile scenarios rarely provide reliable communication or support to deal with unannounced disconnections. Should such a design choice be made, Loupe would generate counterexamples that show that the lack of the aforementioned guarantees hinders the verification of some of the required properties. Consider **R1**: If notifications are not guaranteed to reach the subscribers, bus stops may never be notified of a breakdown along the route. Loupe indeed returns that there exists at least one execution in which state *DisplayBreakdown* in Fig. 8 is never reached. Similarly, requirement **R3** cannot be met if notifications addressed to members of the in-field personnel are not delivered due to the corresponding component being temporarily disconnected. In this case,

Loupe shows that the execution never reaches state *MovingToBreakdown*.

Some of our requirements thus ask for reliable communication, a functionality normally delegated to the P/S infrastructure. In practice, this can be achieved using network-level solutions or dedicated protocols [20].

6.3.2 Subscription Delays

In our scenario, both data consumers (e.g., in-field personnel) and data producers (e.g., buses) may join and leave the system dynamically. Subscription delays may be an issue in the former case, as pointed out in Section 2. As members of the in-field personnel move across routes, their subscriptions must change accordingly since they depend on the current route and location. This is normally implemented by issuing an unsubscribe operation immediately followed by a subscribe with a different filter. Using Loupe, we verified that this behavior may invalidate requirement **R3** if the underlying P/S infrastructure suffers from subscription delays. Indeed, if matching messages are published before subscriptions are active, the personnel components miss the corresponding notifications.

Loupe also showed that subscription delays may be an issue for data producers as well. As discussed in Section 2, subscriptions may experience delays not only when issued, but also when they are already present and must reach newly arrived dispatchers. Consequently, some notifications may not be generated because of the absence of the corresponding filters on dispatchers that just joined. For instance, the counterexamples returned when checking requirement **R1** in the presence of subscription delays show buses publishing messages without generating notifications since the associated dispatchers are unaware of existing subscriptions.

The above aspect was neglected by the initial application model, which was a manifestation of a common design flow that ignores delays caused by the underlying dispatching infrastructure. Recognizing this issue provides insights into the most appropriate architecture and routing protocols for the scenario. It may suggest the use of a centralized dispatching architecture, if possible, to minimize the delays when components join or leave. If this is unfeasible, distributed reliability mechanisms to recover lost messages may alternatively be employed [20].

6.3.3 Message Ordering

Property **R4** mandates pairwise FIFO delivery. A similar requirement can be met either by the communication layer or at the application level. It is less evident, however, that **R2** also requires a specific message ordering, as we realized by inspecting the counterexamples provided by Loupe when checking **R2**. The model in Fig. 8 prevents reaching state *DisplayReplacementBus*—as required by the property to check—without first going through state *DisplayBreakdown*. This entails receiving the breakdown notification before the information about the replacement bus. Because the corresponding messages are published by different components—headquarters and buses, respectively—pairwise FIFO is not sufficient.

To address this issue, the communication infrastructure should provide system-wide FIFO or causal ordering. Both are difficult to implement at the application level; thus, developers may want to push this requirement into the P/S infrastructure. System-wide FIFO and causal ordering subsume pairwise FIFO. Therefore, any P/S system providing the former also provides the latter.

6.3.4 System Dimensioning and Message Delays

We carried out several verification runs by setting different values for the size of input queues, execution rates, and message delays. By exploring the first two dimensions, we could determine bounds on the processing speed of the various components and the size of their input queues. In this context, issues may arise from the variable number of buses involved. For instance, Loupe shows that the *stop* and *headquarter* components must be dimensioned to tolerate a worst-case load determined by situations when a simultaneous (and disastrous) breakdown of all buses occurs. The same situation may be an issue for the dispatcher itself, as messages may overflow its input queue before reaching the subscribers. We also noticed that dimensioning the in-field personnel component essentially depends on the number of buses simultaneously present within $2T$ stops along a given route, T being the parameter in subscription $S2$ of Table 3. This subscription indeed filters out all messages outside the scope determined by T.

Regarding message delays, Loupe shows that ordering guarantees must be assumed on the underlying P/S infrastructure only if message delays are comparable with component execution rates. For instance, if messages travel faster than the time it takes for the headquarters to decide on a replacement bus, requirement **R2** is met regardless of message ordering. In this case, the *stop* component is guaranteed to receive the notification of the breakdown before the one regarding the replacement bus. We also noticed that some message delays may be leveraged to address some requirements without relying on specific P/S guarantees. For instance, as long as message delays are random but the worst-case (highest) delay at the bus component is lower than the best-case (smallest) delay at the headquarters component, requirement **R2** is still met without imposing any message ordering. In the absence of a fine-grained model of the P/S infrastructure, it would have been difficult for developers to grasp similar interactions between P/S guarantees and timing aspects.

7 Empirical Evaluation

This section provides an empirical assessment of Loupe. First, we investigate Loupe's scalability properties. Next, we compare Loupe against a state-of-the-art solution for the verification of P/S architectures [62]. In this solution, both application components and the P/S infrastructure are modeled atop the SPIN model checker. During the discussion, we also briefly compare Loupe's performance against an early prototype [5] to testify its evolution over time. Finally, we run experiments by selectively deactivating some of the abstractions described in Section 4 to study their individual impact on the overall performance.

As performance metrics, we measure the *number of states* generated and the peak *memory consumption* during the verification, and the *time* to complete the verification. Note

TABLE 4
Parameters of Transportation Scenario

Parameter	Value(s)
Stops along a route	[5..50] (step 5)
Bus routes	[5..50] (step 5)
Max bus passengers	50
Buses concurrently on a route	Stops along a route - 1
In-field personnel members	Bus routes - 1

that absolute performance is not indicative per se, given the prototypical nature of our current implementation. Rather, our goal is to assess the improvements w.r.t. state-of-the-art solutions in real-world scenarios [28], [29]. Specifically, we empirically investigated how our techniques advance the current use of model checking in the verification of P/S infrastructures.

We ran all experiments using a Linux desktop PC with a P4 3.2 Ghz CPU and 2 Gb RAM, a standard Sun JVM version 1.5, and the DJProf tool [25] to measure memory consumption.

7.1 Scalability

We use the application model illustrated in Section 6 and the numerical parameters in Table 4. Moreover, we set $P = 20$ and $T = 2$ in subscription $S2$ of Table 3. Fig. 9a illustrates the trends in Loupe's performance when verifying requirements **R1** to **R4** with a varying number of stops along a route and 50 total routes. As expected, the number of states examined during the verification increases as the number of stops grows (top figure). Consequently, the time taken to complete the verification increases as well (middle figure). The peak memory consumption during the verification, however, is always within the limit of today's desktop PCs (bottom figure). Note that this metric is determined by the worst-case complexity of the model at hand, independently of how large the model is. Based on these results, future implementations of Loupe may want to trade memory for verification time. The trends in Fig. 9a are exponential. Indeed, more stops along a route make the model more complex to verify, as more combinations of different "local" states at different stops need to be explored.

The trends shown in Fig. 9b with a varying number of routes and 50 stops along each route, however, appear to be linear. We argue that this is due to the nature of the properties we are verifying. Indeed, the properties at hand are essentially specified on a per-route basis. Therefore, adding more routes does not increase the complexity of the model in terms of possible combinations of local states at different components. Rather, more routes simply "extend" the state space with additional states that are examined sequentially w.r.t. those already existing.

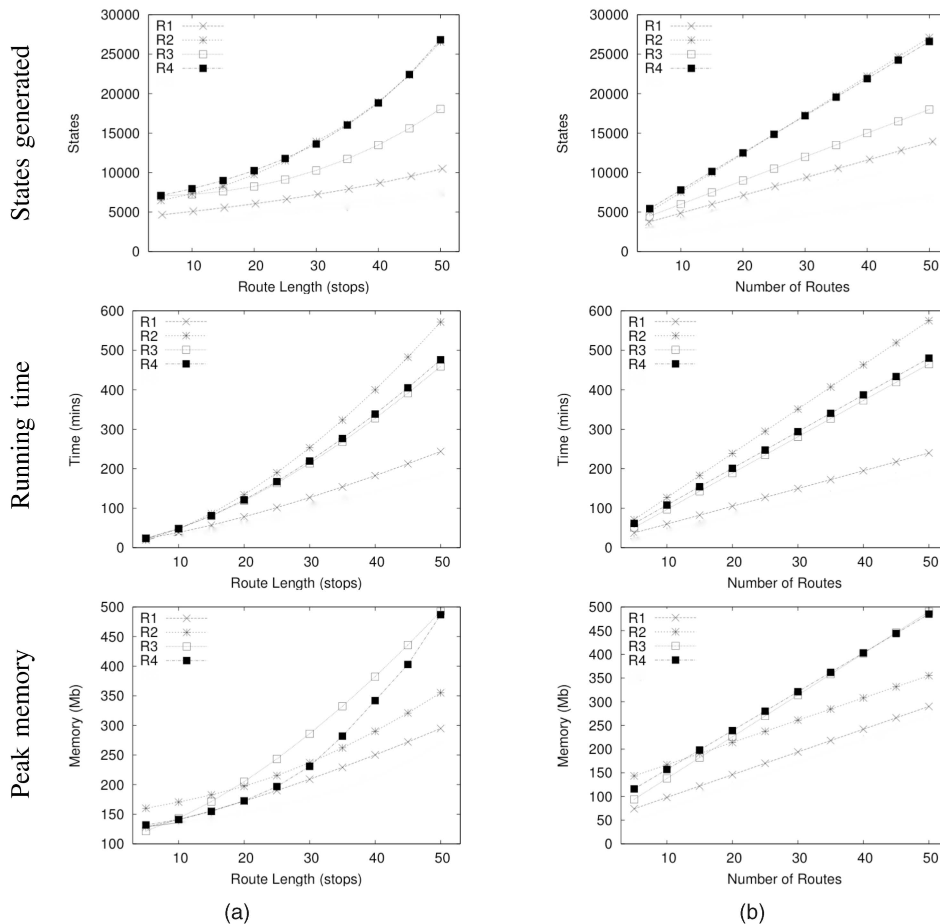

Fig. 9. Loupe performance in transport scenario. (a) Varying stops. (b) Varying routes.

TABLE 5
Loupe Performance against a SPIN-Based Tool [62] in the Transport Scenario

Property	Loupe			SPIN-based [62]		
	Memory (Mb)	States	Time (min)	Memory (Mb)	States	Time (min)
R3 - 50 routes, 2 stops	98.65	4328	7.25	+498.21%	+398.72%	+423.98%
R3 - 50 routes, 10 stops	122.76	6782	12.23	+731.61%	+672.33%	+598.41%
R3 - 50 routes, 15 stops	138.76	7651	14.11	OM	NC	NC
R3 - 2 routes, 50 stops	62.95	3987	7.88	+549.61%	+471.22%	+433.34%
R3 - 10 routes, 50 stops	136.73	6001	12.19	OM	NC	NC
R4 - 50 routes, 2 stops	102.43	4409	8.11	+491.61%	+422.17%	+479.41%
R4 - 50 routes, 10 stops	127.76	6583	12.59	+607.61%	+652.99%	+598.91%
R4 - 50 routes, 15 stops	158.76	8243	15.21	OM	NC	NC
R4 - 2 routes, 50 stops	99.76	4981	10.72	+455.61%	+477.70%	+534.41%
R4 - 10 routes, 50 stops	186.54	7635	17.11	OM	NC	NC

OM = Out of Memory - NC = Not Concluded

7.2 Comparison against a SPIN-Based Tool

The SPIN-based tool we consider [62] is limited to a subset of the P/S guarantees we model in Loupe. It only accounts for subscription delays, message reliability, and message ordering, and yet the modeling of these guarantees is coarser-grained w.r.t. to Loupe. Subscription delays are considered only on the data consumer side, whereas in Section 6, we already discussed how these are also relevant for data producers. Message reliability does not distinguish between publisher and subscriber reliability, and message ordering guarantees do not include total ordering or scrunching policies when messages are prioritized. These features are available in Loupe. Finally, the tool does not embody any notion of time, does not model unannounced disconnections, nor components dynamically joining/leaving the system or dynamic subscribe/unsubscribe operations, as we do in Loupe.

Extending the SPIN-based tool to match the capabilities of Loupe is outside the scope of this work. In some cases, this would not even be possible. Indeed, some of the above limitations are inherited from SPIN itself, e.g., because it prevents on-demand creation of Promela processes and channels. In the following comparison, we use scenarios and properties that can be verified using the standard features and guarantees in the SPIN-based tool.

We use a simplified version of our case study, where we circumvent the limitations of the SPIN-based tool by intentionally ignoring timing aspects and unannounced disconnections, and by submitting all subscriptions at start-up. A variable number of *Bus* and *Personnel* components is hard-wired in the model. We mimic the dynamic join and leave of such components by ignoring their operation until they are artificially "started." This happens based on a fictitious, system-wide logical clock that triggers their operation. We configure the SPIN-based tool to assume reliable delivery of messages and system-wide FIFO ordering. We also ran experiments using causal ordered delivery, obtaining results similar to those described next. We set the scenario-specific parameters as in Section 7.1.

In the following, we focus only on **R3** and **R4**. Indeed, we verified that the SPIN-based tool also correctly verifies both **R1** and **R2** *in the presence* of subscription delays. This is erroneous and stems from the fact that the SPIN-based tool ignores subscription delays on the data producer side. Indeed, in our scenario, buses dynamically enter the system. Their breakdown notifications may not reach the subscribers if the associated dispatcher is not yet aware of the current subscriptions, and thus both **R1** and **R2** should fail. We believe that a comparison based on incorrect assumptions would be pointless, and refrain from considering **R1** and **R2** further.

The SPIN models use `atomic` sections whenever possible. Both bit-state hashing and partial order reduction are also used when running the verification.

Table 5 reports the results of our experiments up to when the SPIN-based tool fails because of memory overflows. This happens quite early compared to the range of values we explore in Section 7.1. When the SPIN-based tool completes the verification, the performance is orders of magnitude worse compared to Loupe. This holds across all metrics we examine, against both a varying number of routes and stops, and regardless of the property to verify. Loupe's performance in this setting is comparable to the figures in Section 7.1. This is a result of embedding the P/S infrastructure within the model checker: The processing/memory overhead is largely independent of the specific guarantees assumed on the underlying P/S infrastructure.

Using the same setting and property **R1-R4**, we also compared Loupe against its earlier prototype [5]. We registered improvements across all metrics: an average gain of 5 percent in peak memory consumption, about 20 percent improvement in the number of states generated during the verification, and a 27 percent gain in time to complete the verification. These improvements come from our continuing experience in the verification of P/S architectures with Loupe, whereby we learned how to further abstract some features of P/S infrastructures. For instance, we recognized that modeling system-wide FIFO ordering may be achieved using a *single* state that represents a specific delivery

TABLE 6
Impact of Individual Abstractions in the Transport Scenario

Abstraction being disabled	Memory	States	Time
Subscription table	+18.12% (± 4.56%)	+24.12% (± 8.98%)	+17.11 (± 7.88%)%
Multi-point communication	+84.75% (± 13.61%)	+119.21% (± 21.01%)	+41.39% (± 4.56%)
Causal ordering	+29.12% (± 11.98%)	+92.12% (± 17.11%)	+39.88% (± 10.54%)

ordering, as opposed to per-subscriber states as in our earlier prototype. This is possible because the ordering is bound to be the same at all subscribers.

7.3 Impact of Individual Abstractions

To complete our discussion, we provide a fine-grained view on the effectiveness of the domain-specific abstractions described in Section 4. To achieve this, we manually "disable" some of them, exposing as explicit states information that would normally be hidden inside Loupe. By doing so, we can isolate the contribution of the individual abstraction to the overall performance. In particular, we examine the impact of our abstractions modeling the subscription table, multipoint communication, and causal ordering of messages. Their implementation can be clearly identified and carved out. We use the model described in Section 6, and the same settings used in Section 7.1, with 50 routes and 50 stops each.

Table 6 shows how Loupe's figures in Fig. 9 change in the absence of some domain-specific abstractions. The values are averages and deviations over all properties in our case study, specifically obtained by selecting causal ordering of messages to verify property **R2**. It appears that abstractions over communication functionality have the greatest impact. In particular, exposing as explicit states the information needed to model multipoint communication yields more than twice as many states, and the verification time consequently suffers. Instead, abstracting the subscription table seems to contribute the least to performance. Note that the effectiveness of this feature becomes more relevant as the total number of subscriptions grows. In the transport scenario, however, communication dominates over the filtering functionality. The limited deviation around the average value confirms that our reasoning is general and not an effect of the specific property being verified.

8 Related Work

Loupe is the main result of the research carried out by the authors on the formal analysis of P/S architectures. The first results of this activity [62] contributed the idea of modeling the P/S infrastructure as an implicit, parametric component, and the use of live sequence charts to express user properties. They also demonstrated the inadequacy of traditional model checkers, such as SPIN, to analyze realistic models of these applications. In this case, the trade-offs between accuracy and performance imposed a model of the P/S infrastructure with only a few of the guarantees presented in Section 2. This is why, in [4], we thought of a radically different solution and started conceiving the approach described in this paper. The ideas behind Loupe have been further elaborated in [5], where we gave a first description of the approach, and in [3], where we demonstrated how to embed a notion of time in our approach. In parallel, we also investigated probabilistic models by using a stochastic model checker to account for the variability of the network infrastructure [34].

Other research efforts focus on issues that are related to our work. Cadena, an Eclipse-based extensible modeling and development framework for component-based systems, proposes an approach to deal specifically with the CORBA Component Model (CCM), and exploits an early version of Bogor to focus on its real-time features [24], [33]. This work directly exploits the Bandera Specification Language [18] to specify the properties to verify. Holzmann and Joshi [37] propose a solution to augment Promela models with fragments of C code. The aim is to provide means to express complex data structures and user-defined functions to guide the state generation algorithm. Both this approach and ours provide means to customize the way the model-checker works, but in the former case, the definition of the code fragments is up to the user, while in our case, they are predefined and readily available.

As for the verification of P/S architectures, Garlan and Khersonsky [28], [29] use a custom language to define the behavior of application components and provide different variants for the middleware infrastructure. These alternatives are far from fully capturing the different characteristics of existing P/S systems. The approach is extended in [11] by adding more expressive events, dynamic delivery policies, and dynamic event-method bindings. These features are then used in a framework that produces both specifications amenable for model checking and executable artifacts for testing [63], [64]. The resulting approach only deals with a few delivery policies and does not capture finer-grained guarantees.

Beek et al. [7], [8] concentrate on augmenting an existing groupware protocol with a P/S notification service, and report on the improvements in user awareness of the development status. Caporuscio et al. [12] propose a compositional reasoning technique based on an assume-guarantee approach and apply it to the development of a file sharing system on top of Siena [13]. The customization of the verification engine is the fundamental difference between the aforementioned approaches and Loupe.

In the broader field of verification of software architectures, Colangelo et al. [15] overcome the state explosion problem by means of *slices* and *abstractions*. The approach works at the architectural level, and the property of interest represents the slicing criterion [42], along with the set of to-be-observed events and the relationships among them. Abstraction rules reduce the state machines without compromising their significance.

Proposals that exploit model checking to verify application models, often rendered as UML state machine diagrams (or state charts), also exist. Among others, vUML [47], veriUML [16], JACK [31], and HUGO [55] provide generic frameworks for the verification of distributed systems where components are rendered as state charts diagrams, but do not support any sophisticated communication paradigm. These approaches provide general-purpose solutions implemented on top of existing model checkers (SPIN is often the target verification engine). JACK and HUGO only support broadcast communication, where the events produced by a component are notified to all others. vUML and veriUML provide explicitly declared channels to let different components communicate, but their characteristics remain hidden in the model. They do not support dynamic creation/destruction of components, and the communication topology must be fixed a priori. As for properties, vUML simply uses SPIN to detect deadlocks, live-locks, and other similar properties, while veriUML, JACK, and HUGO allow users to exploit temporal logic, like CTL, ACTL, and LTL, to state the properties of interests. Inverardi et al. [39] and Kaveh and Emmerich [41] exploit model checking to verify the cooperation among distributed automata. The former uses SPIN to verify properties expressed in LTL, Büchi automata, or *property sequence charts*, an extension of UML 2.0 sequence diagrams. The latter studies distributed applications based on remote method invocation, and only addresses potential deadlocks.

9 Conclusions

This paper presented an in-depth description, discussion, and evaluation of the most mature incarnation of Loupe. With our tool, we flip the traditional approach to the verification of P/S architectures by embedding the communication infrastructure within the checking engine. By virtue of this, our domain-specific abstraction techniques allow for accurate, parametric models of P/S infrastructures while reducing state explosion problems. Our evaluation assessed the effectiveness of Loupe in verifying nontrivial distributed applications based on the P/S paradigm, giving application designers a powerful tool to explore the trade-offs involved in assuming different guarantees on the communication infrastructure.

Acknowledgments

The authors wish to thank Luca Zanolin for his early efforts in this research activity, Natasha Sharygina for the insightful discussion on how to apply partial order reductions to model random message delays, Giorgio Gerosa for the implementation of the time extension of Loupe, and the anonymous reviewers for their insightful comments on the first versions of the manuscript. This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom, and by MIUR Italy, FIRB Project RBNE05C3AH- D-ASAP. It has also been partially supported by SSF, the Swedish Foundation for Strategic Research, and by CONET, the Cooperating Objects Network of Excellence, under EU-FP7 contract number FP7-2007-2-224053.

References

[1] ActiveMQ, Home Page, activemq.apache.org, 2010.

[2] R. Alur, C. Courcoubetis, and D. Dill, "Model Checking for Real-Time Systems," *Proc. Fifth Int'l Symp. Logic in Computer Science*, 1990.

[3] L. Baresi, G. Gerosa, C. Ghezzi, and L. Mottola, "Playing with Time in Publish-Subscribe Using a Domain-Specific Model Checker," *Proc. Specification and Verification of Component-Based Systems Workshop*, 2007.

[4] L. Baresi, C. Ghezzi, and L. Mottola, "Towards Fine-Grained Automated Verification of Publish-Subscribe Architectures," *Proc. 26th Int'l Conf. Formal Methods for Networked and Distributed Systems*, 2006.

[5] L. Baresi, C. Ghezzi, and L. Mottola, "On Accurate Automatic Verification of Publish-Subscribe Architectures," *Proc. 29th Int'l Conf. Software Eng.*, 2007.

[6] L. Baresi, C. Ghezzi, and L. Mottola, "Accurate Verification of Publish-Subscribe Architectures," technical report, Politecnico di Milano, 2008.

[7] M.-H. Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, and M. Sebastianis, "Model Checking Publish-Subscribe Notification for Thinkteam," *Proc. Ninth Int'l Workshop Formal Methods for Industrial Critical Systems*, 2004.

[8] M.-H. Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, and M. Sebastianis, "A Case Study on the Automated Verification of Groupware Protocols," *Proc. 27th Int'l Conf. Software Eng.*, 2005.

[9] S. Bhola, R.E. Strom, S. Bagchi, Y. Zhao, and J.S. Auerbach, "Exactly-Once Delivery in a Content-Based Publish-Subscribe System," *Proc. Int'l Conf. Dependable Systems and Networks*, 2002.

[10] Bogor Project, Extensions for LTL Checking, projects.cis.ksu.edu/projects/gudangbogor, 2010.

[11] J.-S. Bradbury and J. Dingel, "Evaluating and Improving the Automatic Analysis of Implicit Invocation Systems," *Proc. Ninth European Software Eng. Conf.*, 2003.

[12] M. Caporuscio, P. Inverardi, and P. Pelliccione, "Compositional Verification of Middleware-Based Software Architecture Descriptions," *Proc. 19th Int'l Conf. Software Eng.*, 2004.

[13] A. Carzaniga, D.-S. Rosenblum, and A.-L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Trans. Computer Systems*, vol. 19, no. 3, pp. 332-383, 2001.

[14] E. Clarke, O. Grumberg, and D. Peled, *Model Checking.* MIT Press, 1999.

[15] D. Colangelo, D. Compare, P. Inverardi, and P. Pelliccione, "Reducing Software Architecture Models Complexity: A Slicing and Abstraction Approach," *Proc. 26th Int'l Conf. Formal Methods for Networked and Distributed Systems*, 2006.

[16] K. Compton, Y. Gurevich, J. Huggins, and W. Shen, "An Automatic Verification Tool for UML," Technical Report CSE-TR-423-00, Univ. of Michigan, 2000.

[17] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S.P. Robby, and H. Zheng, "Bandera: Extracting Finite-State Models from Java Source Code," *Proc. 22nd Int'l Conf. Software Eng.*, 2000.

[18] J.C. Corbett, M.B. Dwyer, J. Hatcliff, and Robby, "A Language Framework for Expressing Checkable Properties of Dynamic Software," *Proc. Seventh SPIN Workshop*, 2000.

[19] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G.P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis, "The RUNES Middleware for Networked Embedded Systems and Its Application in a Disaster Management Scenario," *Proc. Fifth Int'l Conf. Pervasive Comm.*, 2007.

[20] P. Costa, M. Migliavacca, G.P. Picco, and G. Cugola, "Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation," *Proc. 24th Int'l Conf. Distributed Computing Systems*, 2003.

[21] G. Cugola, M. Migliavacca, and A. Monguzzi, "On Adding Replies to Publish-Subscribe," *Proc. First Int'l Conf. Distributed Event-Based Systems,* 2007.
[22] G. Cugola and G.P. Picco, "REDS: A Reconfigurable Dispatching System," *Proc. Sixth Int'l Workshop Software Eng. and Middleware,* 2006.
[23] D. Dailey, M. Haselkorn, and D. Meyers, "A Structured Approach to Developing Real-Time Distributed Network Applications for ITS Deployment," *The ITS J.,* vol. 3, no. 1, pp. 163-180, 1997.
[24] X. Deng, M.-B. Dwyer, J. Hatcliff, and G. Jung, "Model Checking Middleware-Based Event-Driven Real-Time Embedded Software," *Proc. First Int'l Symp. Formal Methods for Components and Objects,* 2002.
[25] DJProf, Java Memory Profiler, www.mcs.vuw.ac.nz/djp/djprof/, 2010.
[26] B.S. Doerr and D.C. Sharp, "Freeing Product Line Architectures from Execution Dependencies," *Proc. First Conf. Software Product Lines,* 2000.
[27] P.-T. Eugster, P.-A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys,* vol. 35, no. 2, pp. 114-131, 2003.
[28] D. Garlan and S. Khersonsky, "Model Checking Implicit-Invocation Systems," *Proc. 10th Int'l Workshop Software Specification and Design,* 2000.
[29] D. Garlan, S. Khersonsky, and J. Kim, "Model Checking Publish-Subscribe Systems," *Proc. 10th Int'l SPIN Workshop,* 2002.
[30] D. Gelernter, "Generative Communication in Linda," *ACM Computing Surveys,* vol. 7, no. 1, pp. 80-112, 1985.
[31] S. Gnesi, D. Latella, and M. Massink, "Model Checking UML Statecharts Diagrams Using JACK," *Proc. Fourth Int'l Symp. High Assurance Systems Eng.,* 1999.
[32] T.H. Harrison, D.L. Levine, and D.C. Schmidt, "The Design and Performance of a Real-Time Corba Event Service," *Proc. 12th Conf. Object-Oriented Programming, Systems, Languages, and Applications,* 1997.
[33] J. Hatcliff, X. Deng, M.-B. Dwyer, G. Jung, and V. Ranganath, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-Based Systems," *Proc. 25th Int'l Conf. Software Eng.,* 2003.
[34] F. He, L. Baresi, C. Ghezzi, and P. Spoletini, "Formal Analysis of Publish-Subscribe Systems by Probabilistic Timed Automata," *Proc. 27th Int'l Conf. Formal Methods for Networked and Distributed Systems,* 2007.
[35] D. Heimbigner, "Adapting Publish/Subscribe Middleware to Achieve Gnutella-Like Functionality," *Proc. Eighth ACM Symp. Applied Computing,* 2001.
[36] G.J. Holzmann, "The Model Checker Spin," *IEEE Trans. Software Eng.,* vol. 23, no. 5, pp. 279-295, May 1997.
[37] G.J. Holzmann and R. Joshi, "Model-Driven Software Verification," *Proc. 11th Int'l SPIN Workshop,* 2004.
[38] IBM Research, The Gryphon Middleware, www.research.ibm.com/gryphon, 2010.
[39] P. Inverardi, H. Muccini, and P. Pelliccione, "Charmy: An Extensible Tool for Architectural Analysis," *Proc. 10th European Software Eng. Conf.,* pp. 111-114, 2005.
[40] S. Kalasapur, K. Senthivel, and M. Kumar, "Service Oriented Pervasive Computing for Emergency Response Systems," *Proc. Fourth IEEE Workshop Ubiquitous and Pervasive Health Care,* 2006.
[41] N. Kaveh and W. Emmerich, "Deadlock Detection in Distributed Object Systems," *Proc. Eighth European Software Eng. Conf.,* 2001.
[42] T. Kim, Y.-T. Song, L. Chung, and D.T. Huynh, "Software Architecture Analysis: A Dynamic Slicing Approach," *ACIS Int'l J. Computer and Information Science,* vol. 1, no. 2, pp. 91-103, 2000.
[43] A.D. Kshemkalyani, "The Power of Logical Clock Abstractions," *Distributed Computing,* vol. 17, no. 2, pp. 131-150, 2004.
[44] A.D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge Univ. Press, 2008.
[45] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, 1978.
[46] S. Li, Y. Lin, S.H. Son, J.A. Stankovic, and Y. Wei, "Event Detection Services Using Data Service Middleware in Distributed Sensor Networks," *Telecomm. Systems,* vol. 26, no. 2, pp. 351-368, 2004.
[47] J. Lilius and I.P. Paltor, "vUML: A Tool for Verifying UML Models," *Proc. 14th Int'l Conf. Automated Software Eng.,* 1999.
[48] H. Liu and H.-A. Jacobsen, "A-TOPSS: A Publish/Subscribe System Supporting Approximate Matching," *Proc. 28th Int'l Conf. Very Large Data Bases,* 2002.
[49] Loupe, Home Page, loupe.sf.net, 2010.
[50] P. Merino and J.M. Troya, "Modelling and Verification of the Itu-t Multipoint Communication Service with Spin," *Proc. Second Int'l Workshop SPIN Verification,* 1996.
[51] G. Muhl, L. Fiege, and P. Pietzuch, *Distributed Event-Based Systems.* Springer, 2006.
[52] OpenJMS, Home Page, openjms.sourceforge.net, 2010.
[53] A. Ouksel, O. Jurca, I. Podnar, and K. Aberer, "Efficient Probabilistic Subsumption Checking for Content-Based Publish-Subscribe Systems," *Proc. Seventh ACM/USENIX Int'l Middleware Conf.,* 2006.
[54] Robby, M.-B. Dwyer, and J. Hatcliff, "Bogor: An Extensible and Highly-Modular Software Model Checking Framework," *Proc. Ninth European Software Eng. Conf.,* 2003.
[55] T. Schäfer, A. Knapp, and S. Merz, "Model Checking UML State Machines and Collaborations," *Electronic Notes in Theoretical Computer Science,* vol. 55, no. 3, pp. 357-369, 2001.
[56] J. Schiefer, S. Rozsnyai, C. Rauscher, and G. Saurer, "Event-Driven Rules for Sensing and Responding to Business Situations," *Proc. First Int'l Conf. Distributed Event-Based Systems,* 2007.
[57] E. Souto, G. Guimares, G. Vasconcelos, M. Vieira, N. Rosa, and C. Ferraz, "A Message-Oriented Middleware for Sensor Networks," *Proc. Second Workshop Middleware for Pervasive and Ad-Hoc Computing,* 2004.
[58] Sputnic Project, Strategies for Public Transport in Cities, www.sputnicproject.eu, 2010.
[59] Sun Microsystems, JMS Specifications and Reference Implementation, java.sun.com/products/jms/docs.html, 2010.
[60] A. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2006.
[61] TIBCO, TIBCO Rendezvous Home Page, www.tibco.com/software/messaging/rendezvous/default.jsp, 2010.
[62] L. Zanolin, C. Ghezzi, and L. Baresi, "An Approach to Model and Validate Publish/Subscribe Architectures," *Proc. Int'l Workshop Specification and Validation of Component-Based Systems,* 2003.
[63] H. Zhang, J.S. Bradbury, J.R. Cordy, and J. Dingel, "A Transformational Framework for Testing and Model Checking Implicit Invocation Systems," *Proc. Int'l Workshop Distributed Event-Based Systems,* 2004.
[64] H. Zhang, J.S. Bradbury, J.R. Cordy, and J. Dingel, "Implementation and Verification of Implicit-Invocation Systems Using Source Transformation," *Proc. Fifth Int'l Workshop Source Code Analysis and Manipulation,* 2005.

Luciano Baresi received the PhD degree in computer science from the Politecnico di Milano, where he is an associate professor in the Department of Electronics and Information. He was also a visiting researcher at the University of Oregon at Eugene and the University of Paderborn, Germany. He is a member of the editorial board of *Service Oriented Computing and Applications*. He was the program cochair of ICECCS, FASE, ICWE, and ICSOC. He has authored some 100 papers in international journals and conferences on various aspects of software engineering. His present research interests are in dynamic software systems, service-oriented applications, and software architectures.

Carlo Ghezzi is a professor and the chair of software engineering in the Department of Electronics and Information at the Politecnico di Milano. He is the Rector's delegate for research, a past member of the Academic Senate and the Board of Governors, and a past department chair. He is a fellow of the ACM, a fellow of the IEEE, and a member of the Italian Academy of Sciences (Istituto Lombardo). He received the SIGSOFT Distinguished Service Award. He is a member-at-large of the ACM Council. He is a member of the editorial board of the *IEEE Transactions on Software Engineering*, *Communications of the ACM*, *Science of Computer Programming*, *Service Oriented Computing and Applications*, and *Software Process Improvement and Practice*. He was the program chair of ESEC, the program cochair of ICSE, and the general chair of ICSE and ICSOC. He has been a keynote at ESEC, ICSE, ETAPS, and ICSOC. He is a member of IFIP WG 2.9 on Requirements Engineering. He has authored more than 150 papers in international journals and conferences on various aspects of programming languages and software engineering, and three books. His present research interests are in rigorous approaches to the design and evolution of software for pervasive distributed systems.

Luca Mottola received the PhD degree from the Politecnico di Milano, Italy, in 2008, with a thesis on programming abstractions for wireless sensor networks (WSNs). He is a postdoctoral researcher at the Swedish Institute of Computer Science (SICS). Previously, he was a postdoctoral researcher at the University of Trento, Italy, and a research scholar at the University of Southern California (USC). His PhD work, which was extensively published at major WSN and closely related conferences, was awarded the 2009 EWSN/CONET Best PhD Thesis Award. His software systems are used in real-world deployments, e.g., as described in a paperwhich appeared in 2009 at IPSN/SPOTS, for which he received the Best Paper Award. His expertise in building WSN software is also demonstrated by the Best Demonstration Award received at ACM SenSys in 2007. His research interests include programming abstractions and distributed computing on sensor networks, and automatic verification of distributed software architectures. More information about his research can be found at www.sics.se/~luca.

▹ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.