
Modeling and Validation of Publish/Subscribe Architectures

Luciano Baresi, Carlo Ghezzi, and Luca Zanolin

Politecnico di Milano – Dipartimento di Elettronica e Informazione
Piazza L. da Vinci, 32 – I20133
Milano (Italy)
baresilghezzi|zanolin@elet.polimi.it

Summary. The publish/subscribe component model is an emerging paradigm to support distributed systems composed of highly evolvable and dynamic federations of components. This paradigm eases the design of flexible architectures, but complicates their validation. It is easy to understand what each component does, but it is hard to foresee what the global federation achieves.

This chapter tackles the problem at the architectural level and describes an approach to ease the modeling and validation of such systems. The modeling phase specifies how components react to events. It distinguishes between the *dispatcher* and the other components. The former oversees the communication and is supplied as a predefined parametric component. The latter are specified as UML statechart diagrams. The validation uses model checking (SPIN) to prove properties of the federation defined as *live sequence charts* (LSCs). We do not start from LTL (*linear temporal logic*) formulae, the property language of SPIN, but we render properties as automata. This solution allows us to represent more complex properties and conduct more thorough validation of the modeled systems. The approach is exemplified on a simple application that controls an *eHouse*.

1 Introduction

The *publish/subscribe* [8] component model is an emerging paradigm to support software applications composed of highly evolvable and dynamic federations of components. According to this paradigm, components do not interact directly, but through a special-purpose element called *dispatcher*. The dispatcher receives events and forwards them to subscribers, that is, to all components registered to listen to them.

Publish/subscribe systems decouple the components participating in the communication: a sender does not know the receivers of its messages. Rather, receivers are identified by the dispatcher based on previous subscriptions. New components can dynamically join the federation, become immediately active,

and cooperate with the other components without any kind of reconfiguration or refreshing of the modified application. They must simply notify the dispatcher that they exist by subscribing to particular events.

The gain in flexibility is counterbalanced by the difficulty to the designer in understanding the overall behavior of the application. Components are easy to reason about in isolation, but it is hard to get a picture of how they cooperate, and to understand their global behavior. Although components might work properly when examined in isolation, they can become faulty when put in a cooperative setting.

To solve these problems, we propose an approach, and a supporting environment, to model and validate publish/subscribe systems. We tackle the problem at the architectural level, where each component specifies how it produces and consumes events. The approach analyzes how the different elements cooperate to achieve a common goal, but does not address how they behave in isolation. Individual components are assumed to work as specified.

We point out explicitly that the goal of the paper is to address validation of component-based software through formal analysis at the architectural level. In contrast with other contributions in this book, which focus on component testing, we address model-based validation. There is wide consensus that analysis and testing are complementary rather than competing techniques for validating software systems and improving their quality [27, 20]. This contribution stresses validation by proposing an innovative technique to analyze publish/subscribe systems that helps discover problems early during the design phase, but that does not replace component, integration, and system tests. Contributions to these subjects are presented in other chapters of this book.

Publish/subscribe systems are often implemented on top of a middleware platform that provides all event dispatching features. Implementors have only to configure it. The other components must be designed and coded explicitly. Our approach to modeling and validation exploits this peculiarity. Modeling the dispatcher is a complex task, but the effort is counterbalanced by the fact that the dispatcher is usually reused in different systems. The accurate representation of such a component would clearly impact the whole design phase, but its being application-independent eases the task. We do not ask designers to specify a new dispatcher (middleware platform) for each new application. By following a reuse approach at the modeling stage, we provide a parametric model of the dispatcher. Developers have only to configure it to render the communication paradigms they want to use. In contrast, application-specific components must be specified as UML statechart diagrams [23].

Once all components have been designed in isolation, modeled systems are validated through model checking. The global properties of the architectures, i.e., the federations of components, are rendered as *live sequence charts* (LSCs) [7]; both component models and properties are translated into Promela [13] and passed to the SPIN model checker [14]. We do not code properties as LTL (*linear temporal logic*) formulae, as is usual in the context

of SPIN, since they are not expressive enough. We render them as automata to represent more complex properties and to conduct more thorough validation.

The paper is organized as follows. Section 2 introduces publish/subscribe systems through a simple example. Sections 3 and 4 describe how we model and validate their architectures. Section 5 presents the prototype toolset and Sect. 6 surveys the related work. Finally, Sect. 7 provides conclusions and outlines the possible directions of future work.

2 Our Approach

Our approach addresses publish/subscribe architectures [10, 8]. These architectures are based on loosely coupled components that interact through events. The communication is based on a *dispatcher*. Components *publish* their events and *consume* those received from the dispatcher. They also *subscribe* (*unsubscribe*) to the dispatcher to define the events they want to receive. The dispatcher forwards (*notifies*) published events to all subscribers. This means that connections are not hard-wired in the system, but are implicitly defined by means of published/subscribed events. New components can be added or new connections be established by simply subscribing/unsubscribing to events.

Figure 1 presents the architecture of such a system. It is an excerpt of a control system for an *eHouse*. Since we focus on a simple service that allows users to take baths, the architecture comprises the dispatcher and five application-specific components. When the user requires a bath, the service reacts by warming the bathroom and starting to fill the bathtub. When everything is ready, the user can take the bath.

Notice that components do not communicate directly. More specifically,

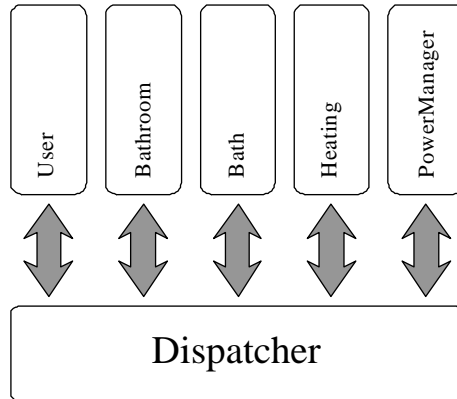


Fig. 1. Our example publish/subscribe architecture

User publishes events to notify that she/he wants to take a bath. *Bathroom* is

in charge of setting the bath and increasing the temperature in the bathroom. The bath and heating systems are described by *Bath* and *Heating*, respectively. When *Bath* receives the event from *Bathroom*, it starts operating and publishes another event when the bathtub is full. At the same time, *Heating* turns on the electric heating and increases the temperature of the bathroom.

PowerManager manages the provision of electricity. If there is a blackout, this component notifies this failure and switches from the primary to the secondary power supplier. The secondary power supplier is less powerful than the primary one, and some electric devices must be turned off for it. For instance, the electric heating turns itself off as soon as there is a blackout. Thus, the user cannot take a bath: the temperature in the bathroom cannot be increased since the electric heating does not work.

After sketching publish/subscribe systems and presenting a simple example, we can anticipate the main steps of our approach to modeling and validation:

1. The designer defines the specific dispatcher by instantiating a parametric component with suitable values for the parameters. Parameters tailor the main characteristics that differentiate the behaviors of the various dispatchers;
2. The designer specifies application-specific components by means of UML statechart diagrams;
3. The designer defines the properties against which the system should be validated through scenarios rendered as live sequence charts (LSCs);
4. Our tool validates the publish/subscribe architecture (against the properties defined above) using model checking techniques.

The following sections describe these three steps in detail and exemplify them on the example of Fig. 1.

3 Modeling

The approach requires that all components be modeled, but it suggests different methods for the dispatcher and the other components. In this section we first discuss the behavior of the parametric dispatcher. Then we show how statecharts can be used to model components. The presentation will be informal and mostly based on examples. A formal description of our approach will be provided in a future paper.

3.1 Dispatcher

The predefined dispatcher is instantiated by providing parameters that customize the dispatcher's behavior to reflect the different peculiarities of existing publish/subscribe middleware platforms. In fact, the market offers different alternatives, ranging from standards (e.g., Java Message Service (JMS) [25])

to research prototypes (e.g., Siena [1], Jedi [5]) to industrial products (e.g., TIBCO [26]). All these middleware platforms support the publish/subscribe paradigm, but their dispatchers offer slightly different characteristics. This means that the definition of the parameters that the dispatcher component should offer to the developer is a key issue. On the one hand, many parameters would allow us to specify the very details of the behavior, but, on the other hand, they would complicate the model unnecessarily. Moreover, we must consider that the target validation environment is based on model checking where state explosion hampers the actual analysis capabilities. Besides smart encoding techniques, the identification of the minimal set of parameters is essential: the simpler the model, the faster the verification.

Since the approach aims at verification at the architectural level, we assume that the model describes the interactions with components but does not address internal details. For example, the dispatcher could be implemented by middleware distributed on several hosts, but this is completely transparent to components and has no impact on functionality. The model describes how events are published and notified; it does not concentrate on how the dispatcher (middleware) works.

After several attempts, we have selected the following three characteristics as the key elements that define the “end-to-end” behavior of our dispatcher:

- **Delivery** Different middleware platforms may satisfy different requirements. For example, a middleware might be used to support cooperation among components in a mobile environment. In another case, it might be used for secure and reliable communication between components in a centralized system. These different cases require different guarantees to be met by the middleware in terms of event delivery. In the former case, we might tolerate the fact that some published events are not delivered to certain subscribers. In the latter, we might insist on the delivery of all published events. Thus, event delivery can be characterized by two alternatives: (a) all events are always delivered, or (b) some events can be lost.
- **Notification** The relationship between the order of event generation and event delivery can vary among different middleware implementations. Ideally, one might expect events to be notified in the same order in which they are published. This ideal behavior, however, can be easily enforced in a centralized setting, but is hard to achieve in a distributed environment. Thus, if we want to relate the order of publication to the order of notification, we can identify three alternatives: (a) they are the same, (b) the order of publication and notification are the same only when we refer to events published by the same component, or (c) there is no relationship, and events may be notified randomly. For example, if component *A* publishes the sequence of events x_1, x_2 , and then component *B* publishes the sequence y_1, y_2 , the dispatcher could notify these events to component *C* as follows:
 - case (a), $x_1 < x_2 < y_1 < y_2$

- case (b), $x_1 < x_2, y_1 < y_2$
- case (c), any permutation,

where $a < b$ means that event a is notified before event b .

- **Subscription** When a component declares the events it is interested in (i.e., *subscribes* to these events), it starts receiving them immediately. Similarly, unsubscribing is also instantaneous. However, the distributed nature of the system can make the dispatcher delay the response to new subscriptions/unsubscriptions. Thus, our parametric characterization identifies two alternatives: (a) the dispatcher immediately reacts to (un)subscriptions, or (b) these operations are not immediate and can be delayed.

The actual dispatcher comes from choosing one option out for these three main characteristics¹ (summarized in Table 1). These options of characterizations cover most of the guarantees that a dispatcher should satisfy. They are a reasonable compromise between balancing the need for light models of the dispatcher and capturing the many nuances of its possible behaviors. More sophisticated models — that is, more parameters to deal with problems like authentication and safety in message delivery — would have made validation heavier with no additional benefits (with respect to the problems we are interested in).

Notice that developers can always get rid of the predefined parametric dispatcher, implemented by a ready-to-use component, elaborate their particular model of the dispatcher as a UML statechart diagram (like any other component), and integrate it with the architecture. They would lose the advantages associated with reuse and well defined communication protocols, but can they could still follow our validation approach.

| | |
|---------------------|--|
| Delivery | (a) All events are always delivered (b) Some events can be lost |
| Notification | (a) Orders of publication and notification are the same (b) Orders of publication and notification are the same only when we refer to the events published by the same component (c) No relationship and events may be notified randomly |
| Subscription | (a) The dispatcher immediately reacts to (un)subscriptions (b) Subscriptions are not immediate and can be delayed |

Table 1. Alternative guarantee policies associated with the *Dispatcher*

¹The tool (Sect. 5) provides a set of check boxes to let the designer select the desired options.

Referring to the *eHouse* example, we assume that the architecture of Fig. 1 uses a dispatcher that (1) delivers all events, (2) keeps the same order of publication and notification, and (3) reacts immediately to all (un)subscriptions.

3.2 Components

The designer provides for each component a UML statechart diagram whose transitions describe how the component reacts to incoming events. Events have a name and a (possibly empty) set of parameters. When components subscribe/unsubscribe to/from events, they can either refer to specific events or use wildcards to address classes of events. For example, *subscribe("bath", "ready")* means that the component wants only to know when the bath is ready, but *subscribe("bath", \$)* means that it wants to subscribe to all *bath* events.

Transitions are labeled by a pair x/y , where x (i.e., the precondition) describes when the transition can fire, while y defines the actions associated with the firing of the transition. Either x or y can be missing. For instance, *consume("bath", "full")/publish("bath", "ready")* states that the transition can fire when the component is notified that the bathtub is full of water and publishes an event to say that the bath is ready.

The events notified to a component are stored in a *notification queue*. The component retrieves the first event from its notification queue, and if it does not trigger any transition exiting the current state, that is, no consume operation "uses" the event, the component discards the event and processes the following one. This mechanism allows components to evolve even if they receive events that cannot be processed in their current states.

| Event | Meaning |
|-------------------|---|
| need,bath | The user needs a bath |
| bath,start | The bath starts running |
| bath,ready | The bath is full of water |
| bath,finished | The user finishes to take the bath |
| bath,notAvailable | The bath is not available |
| bath,full | The bath is full of water |
| bathroom,warm | The bathroom asks to increase the temperature |
| bathroom,freeze | The bathroom asks to decrease the temperature |
| bathroom, hot | The temperature in the bathroom is hot |
| bathroom, cold | The temperature in the bathroom is cold |
| heating, off | The heating is switched off |
| power, alarm | There is a blackout |
| power, ok | The electricity is on |

Table 2. Glossary of events

Figure 2 describes component *Bathroom* of Fig. 1. To facilitate the intuitive understanding of these diagrams, Table 2 provides a glossary of events used. *Bathroom* starts in state *Idle*, waiting for events. At this stage, it is subscribed only to events that ask for a bath. When the user notifies that she/he needs a bath, *Bathroom* changes its state and notifies *Heating* that the temperature should be increased and *Bath* should start to run. At the same time, the component updates its subscriptions by adding those relating to temperature, heating, and bath. This component exploits two variables that are used to store the status of the bath, *bathStatus*, and of the temperature, *temperatureStatus*. For example, the variable *bathStatus* set to *true* means that the bathtub is full of water.

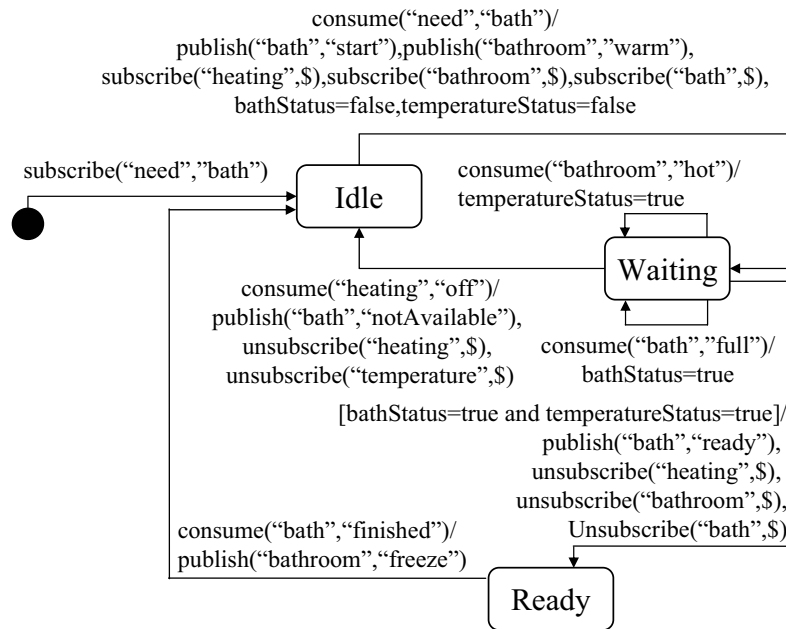


Fig. 2. Bathroom

When the bathtub is full of water and the temperature is hot, the bath is *Ready*. In this state, the component is not interested anymore in events about bath and heating; thus, it unsubscribes from them. Finally, after the user takes the bath, *Bathroom* restores temperature to *cold*².

Figure 3 shows the statechart diagram of *Heating*. For simplicity, we suppose that, when this component starts, the power supplier is working cor-

²For simplicity, we assume here that the temperature can assume only two values: cold and hot.

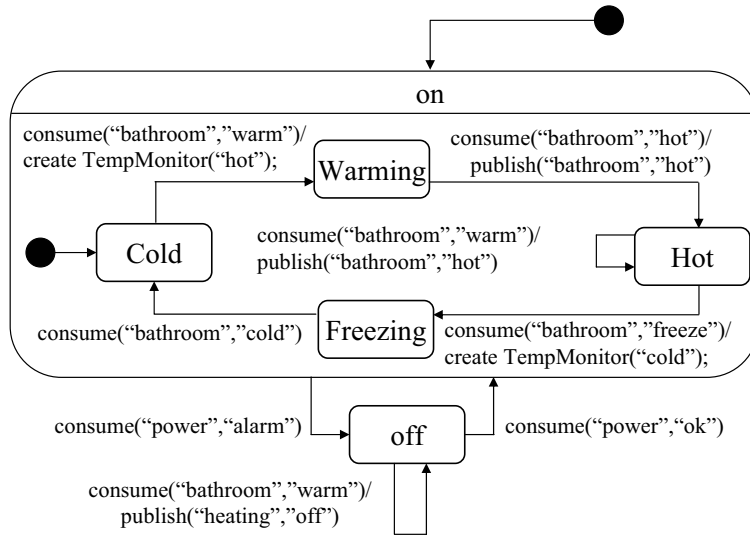


Fig. 3. Heating

rectly and temperature is *cold*. When *Heating* receives an event that asks for increasing the temperature, it moves to an intermediate state to say that the bathroom is warming. When the temperature in the bathroom becomes *hot*, *Heating* moves to the next state, i.e., *Hot*.

4 Validation

Validation comprises two main aspects: the definition of the properties that we want to prove on the federation of components and the transformation of both the model and properties into automata (i.e., Promela).

4.1 Properties

Our goal was to provide an easy-to-use graphical language to specify properties, which would allow designers to work at the same level of abstraction as statechart diagrams. For this reason, we did not use any temporal logic formalisms like *linear temporal logic* [21] (LTL), since they work at a different level of abstraction and, thus, developers would find them difficult to use. We chose *live sequence charts* (LSCs) [7] since they are a graphical formalism powerful enough to describe how entities exchange messages, which are a key aspect of the properties we wish to analyze.

Briefly, a basic LSC diagram describes a scenario of how the architecture behaves. LSCs allow us to render both existential and universal properties,

that is, scenarios that must be verified in at least one or all the evolutions of the architecture.

Entities are drawn as white rectangles with names above them. The life cycle of an entity is rendered as a vertical line which ends in a black rectangle. The white and black rectangles denote the entity's birth and death, respectively. Messages exchanged between entities are drawn as arrows and are asynchronous by default. Each message has a label that describes the message contents.

In our approach, we assume publish/subscribe as the underlying communication policy, and we omit the middleware in the charts. When we draw an arrow between two entities, what we actually mean is that the message is first sent to the middleware and then routed to the other entity. The arrow means that the target entity receives the notification of the message and that the message triggers a transition inside the entity (i.e., inside its statechart).

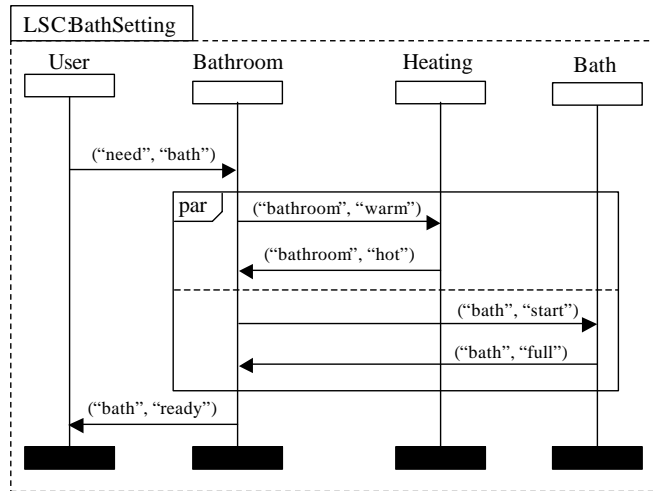


Fig. 4. A basic LSC: Bath setting

Let us define some properties on our bath service. For example, we want to state that, when the user requires a bath, the temperature in the bathroom increases and the bathtub starts to fill. These two tasks are done in parallel and, after the termination of both, the user is notified that the bath is ready. These properties are described in Fig. 4, which shows a basic LSC scenario. *User* issues his/her request for a bath and *Bathroom* reacts by requesting that *Heating* must start to warm the bathroom and *Bath* to fill the bathtub. The two tasks are performed in parallel without any constraint on their order. This parallelism is described through the *par* operator, which states that its two scenarios (i.e., warming the bathroom and filling the bathtub) evolve in

parallel without any particular order among the events they contain. When the bathtub is full and the bathroom is warm, *User* is notified that the bath is ready.

This chart describes only a possible evolution since we cannot be sure that *Bathroom* always notifies *User* that the bath is ready. In fact, if we had a blackout, *User* would receive a notification that the bath cannot be set. Thus, we do not require that the application always complies with this scenario. Rather, we request that there be some possible evolutions that are compliant with it. In LSCs, these scenarios are called *provisional* or *cold* scenarios, and are depicted as dashed rectangles, as shown in Fig. 4.

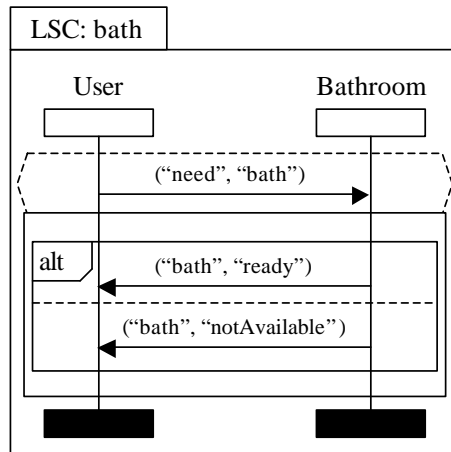


Fig. 5. LSC: Bath ready

To fully specify the bath service, the designer also wants to describe the possible evolutions of the service, that is, when the user requires a bath, she/he always receives a positive or negative answer. This property is shown in Fig. 5. LSCs allow us to define such a property through a *mandatory* or *hot* scenario (depicted as a solid rectangle).

In general, it is difficult to identify global properties that must be satisfied in all evolutions. For this reason, LSCs support the definition of preconditions, that is, the property must hold in all the evolutions for which the precondition holds. Preconditions are drawn as dashed polygons, while the hot part of the scenario is depicted as a solid rectangle. For clarification, we can say that the precondition implies the hot scenario. The chart in Fig. 5 states that, for all the evolutions in which *User* requires a bath, *Bathroom* notifies two possible events, that is, either the bath is ready or it is not available. In this chart, we exploit *alt* (alternative), which is another operator supported by LSCs. This operator says that one of its two scenarios must hold. Thus, Fig. 5 describes

that, after requesting a bath, an event will notify when the bath is ready, unless the bath notifies that it is unavailable.

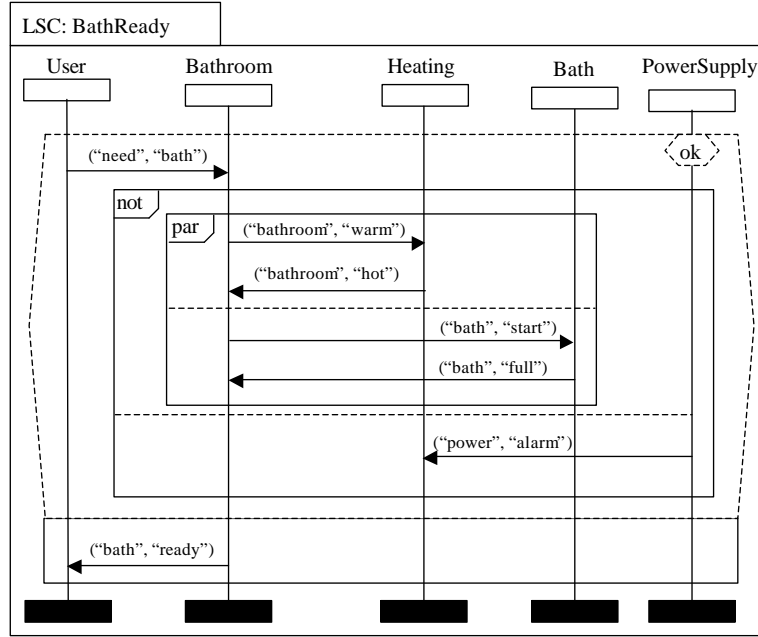


Fig. 6. LSC: Bath ready

Finally, we can redefine the previous property (Fig. 4) to state when the bath becomes ready: the bath must always become ready if no blackout occurs. This property is described in Fig. 6. If no blackout occurs while *Heating* warms the bathroom, the bath must always become available. In this chart, we introduce the *not* operator, which is not part of standard LSCs. This operator has two scenarios as parameters and states that while the first evolves, the second cannot happen simultaneously: if no blackout occurs during the setup of the bath, *User* always receives a positive answer (i.e., the bath becomes ready).

4.2 Transformation

So far, we have shown how to model the architecture and define the properties. After these steps, we can start the analysis. We have decided to use the SPIN model checker [14] as verifier, but, as we already explained, we do not use LTL to describe the properties that we want to prove. Everything is transformed into automata and then translated into Promela [13].

After tailoring the dispatcher according to the parameters set by the developer, we can render it with Promela. Each alternative corresponds to a Promela package and the tool selects the right packages and assembles the model of the dispatcher directly. The translation of statechart diagrams into Promela is straightforward. We do not describe this translation since it has been done by others before (e.g., vUML [17] and veriUML [3]), and we have borrowed from these approaches a method to implement our translation.

Properties are translated in two different ways: cold scenarios are described through plain automata; hot scenarios need auxiliary LTL formulae also. This translation is rather complex since SPIN does not support natively the verification of existential properties. It can verify LTL formulae, which define universal properties but not existential ones.

To state an existential property through LTL, we could negate the LTL formula and verify that the system violates it: this means that there is at least one evolution in which the LTL formula is satisfied (i.e., its negation is violated). However, this approach would require that SPIN be run once for each property that we want to verify. We cannot compose the properties to eliminate executions: we would not find out if there are evolutions that satisfy our properties, but, rather if a single evolution satisfies all of them. Thus, instead of using LTL formulae or their translation into Büchi automata, we investigate a different solution that is based on representing properties (LTLs) as Promela processes. Reasoning on the state reachability feature provided by SPIN, we can compose properties, that is, we have some automata in parallel, and we verify if LSCs hold. However, automata are not enough to describe all LSC features, and, when required, we introduce local LTL formulae to overcome this problem. The transformation of an LSC into an automaton (and LTL) goes through the following four steps:

1. We simplify the property by downgrading all the hot scenarios to cold ones. This means that an automaton can describe the property.
2. We translate the simplified property into the automaton that recognizes the sequence of events described by the LSC. This task is quite easy since the structure of the automaton replicates the structure of the LSC.
3. We reintroduce the fact that the scenario is *hot* by identifying the states in the automaton in which the *hot* scenario starts and ends.
4. We describe the *hot* scenario through a constraint expressed as an LTL formula. The constraint states that if an automaton has reached the state that corresponds to the first message of the *hot* scenario, it must always reach the state that corresponds to the last message of the *hot* scenario. In other words, if the automaton recognizes the first message of the *hot* scenario, it must always recognize all the messages that belong to the same *hot* scenario.

The combination of the automaton and LTL formulae allows us to translate any LSC into Promela and verify it through SPIN.

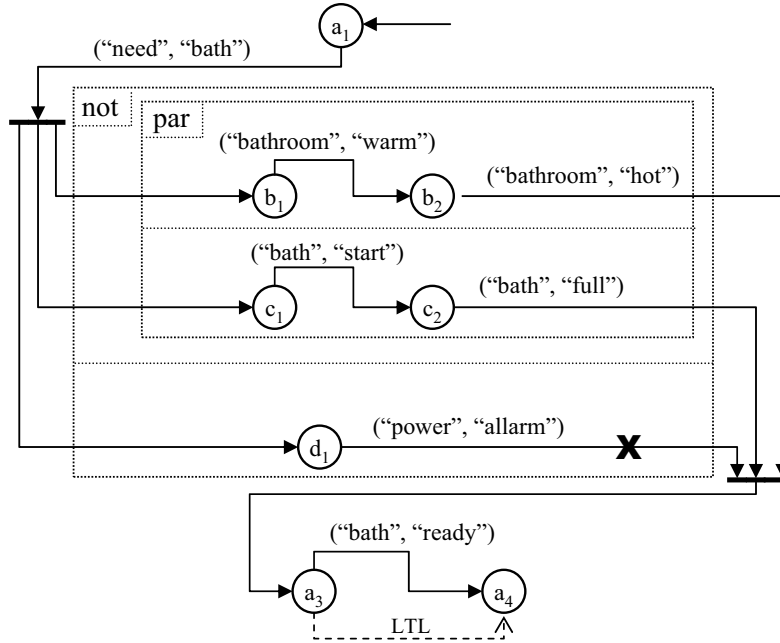


Fig. 7. The automaton that corresponds to the LSC of Figure 6

For example, let us consider the LSC of Fig. 6 and the corresponding automaton of Fig. 7. This automaton has three types of arrows: solid, solid with a cross, and dashed. Solid arrows describe *standard* transitions and have labels that describe recognized events. For instance, if the automaton evolves from state b_1 to state b_2 , this means that the event (“bathroom”, “warm”) has been published and consumed³. This type of arrow can end in a state or in a fork/join bar. Besides recognizing events, solid arrows with a cross disable the join bar in which they end. For instance, when the transition leaving d_1 fires, the join bar in the right hand side of the figure is disabled. Finally, dashed arrows do not define transitions between states, but constrain the evolution of the automaton. The constraint — described by an LTL formula — is always of the same kind: if the automaton reaches a state (i.e., the source state of the arrow), it must always reach the other state (i.e., the target state of the arrow).

³For the sake of clarity, in Fig. 7 we do not describe who publishes or consumes events.

The automaton of Fig. 7⁴ has the same structure as the LSC of Fig. 6. This means that when the dispatcher notifies the first event, that is, (“need”, “bath”), the fork bar is enabled and the automaton splits its evolution into three different threads. Moving top-down, the first thread describes the warming of the bathroom, the second thread depicts the filling of the bathtub, and the last thread corresponds to the blackout. If the first two threads evolve completely while the third thread does not, the join bar is enabled and the automaton evolves to state a_3 . This means that we do not have a blackout while the bathroom is warming and the bath tub is filling. Then, if the dispatcher notifies the last event (i.e., (“bath”, “ready”)), the automaton reaches state a_4 . Reasoning on state reachability, we can argue that if state a_4 is reachable, then there is at least one evolution that complies with the simplified property (i.e., the cold scenario). The property described by this automaton — with no LTL formulae — states that there exists an evolution in which no blackout occurs and, after set-up, the bath becomes available to the user. But this does not match the property of Fig. 6, which states that, if no blackout occurs, the bath must become available. This is why we must refine the automaton and add the dashed edge. Notice that, in this example, the hot scenario comprises only states a_3 and a_4 . In fact, the scenario has only a single message ((“bath”, “ready”). All previous states define the precondition associated with the scenario, that is, the dashed polygon of Fig. 6. The hot constraint means that, if the precondition holds (i.e., all the previous events have already happened), then this event must always occur. This constraint is described by the following LTL formula:

$$\Box(In(a_3) \Rightarrow \Diamond In(a_4)),$$

where we require that when the automaton is in state a_3 (i.e., $In(a_3)$ holds), it must always reach state a_4 .

We can verify this property by reasoning on the reachability of states. In particular, we require that the final state a_4 be reachable; thus, there is at least one evolution that complies with this property. If the model checker does not highlight any evolution in which the LTL formula is violated, we can say that when the precondition is verified, it is always the case that the post-condition is verified in the same evolution.

The validation of the bath services is performed by assuming that the available dispatcher provides the following guarantees: (1) events are always delivered, (2) the order of publication is preserved in the notification, and (3) the dispatcher immediately reacts to subscriptions and unsubscriptions. The validation process shows that the service is incorrectly designed since it violates the property shown in Fig. 5. This becomes clear if we consider the

⁴Readers can interpret this automaton as a statechart extended with inhibitor arcs (i.e., negative conditions). The diagram contains three parallel threads: The first two flows identify “positive” evolutions, while the third flow states a negative condition.

following scenario: *User* asks for a bath and *Bath* starts to fill the bathtub. At the same time *Heating* increases the temperature, but before its becoming *hot* we have a blackout that turns the heating off. At this point, *Bathroom* and *User* wait forever, since *Heating* notifies neither that it is switched off nor that the temperature is *hot*. This problem can be avoided by modifying the arrows between states *on* and *off* in *Heating* (Fig. 2) to introduce the publication of an event to notify that *Heating* fails.

5 Tool Support

Figure 8 shows the architecture of our prototype environment. Components, that is, UML statecharts, are translated directly into Promela. We do not need an intermediate step since Promela allows for easy coding of automata. Properties are translated as explained in the previous section. Both statecharts and LSCs are designed using UML-like CASE tools and supplied to the translator as XMI (XML Metadata Interchange [19]) files. In contrast, the dispatcher is specified by selecting the check boxes that correspond to the properties guaranteed by the middleware (as specified in Table 1). Each alternative is implemented as a package: the tool selects the appropriate parts and composes them in Promela directly.

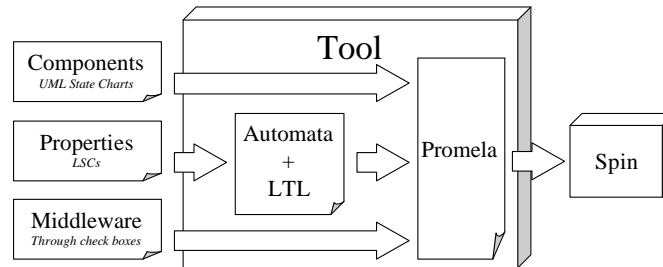


Fig. 8. The prototype toolset

The whole translation — dispatcher, components, and properties — creates a specification that can be analyzed by SPIN. The initial version of the toolset does not support backward translation of analysis results. This feature, which will be part of the next version, visualizes results in different ways. It will represent a misbehavior either as a trace that highlights the error or as a warning that the scenario does not hold. This is the case, for example, of a violated *cold* scenario, that is, no evolution complies with the scenario. Execution traces would be meaningless; we can only say that the property is not satisfied.

6 Related Work

Finite state automata have been used for many years to model and validate complex software systems. In this chapter we investigated their use in the context of components and component composition. In this particular context, most previous work concentrated on validation via testing. These approaches identify the states through which a component can evolve, along with the meaningful transitions between them. They then identify test cases by applying suitable coverage criteria on the automata that mimic the behavior of the components [?, 9].

Our approach differs in the way we validate component-based software, which is based on model checking rather than testing. Several previous efforts applied this technique to the validation of software models, often specified as UML statechart diagrams, and the study of coordination among the components of distributed applications based on well defined communication paradigms.

vUML [17], veriUML [3], JACK [11], and HUGO [24] provide generic frameworks for model checking statecharts. All of these approaches support the validation of distributed systems, where each statechart describes a component, but do not support any complex communication paradigm. JACK and HUGO only support communication based on broadcasting, where the events produced by a component are notified to all the other components. vUML and veriUML support the concept of a channel, that is, each component writes and reads messages to or from a channel. These proposals aim at general purpose applications and can cover different domains, but are not always suitable when we need a specific communication paradigm. In fact, if we want to use them with the publish/subscribe paradigm, we must model the middleware as any other component. Moreover, the communication between components, and thus between the middleware and the other components, is fixed: it depends on how automata are rendered in the analysis language. For instance, vUML would not allow us to model middleware which guarantees that the order of publication be kept while notifying the events. These approaches also impose that channels between components be explicitly declared: vUML and veriUML do not allow software designers to specify architectures where components are created and destroyed at runtime and the topology of the communication is fixed.

The proposals presented so far do not support a friendly language to define properties. With vUML one can state only reachability properties, while with veriUML, JACK, and HUGO one can also define complex properties on how the application evolves; but, in all cases, the properties must be declared directly in the formalism supported by the model checker, that is, CTL, ACTL and LTL, respectively. All these formalisms are based on temporal logic and are difficult to use and understand by designers with no specific background in mathematical logic.

Two other projects try to overcome these limitations (i.e., the definition of properties and the communication paradigm). Inverardi et al. [15] apply model checking techniques to automata that communicate through channels. In this approach, properties are specified graphically through MSCs. They support two kinds of properties: (a) the application behaves at least once as the MSC, or (b) the application must always comply with the MSC. MSCs are directly translated into LTL, the property language supported by SPIN. Kaveh and Emmerich [16] exploit model checking techniques to verify distributed applications based on remote method invocation. Components are described through statechart diagrams where, if a transition fires, some remote methods are invoked. Only potential deadlocks can be discovered using this approach.

Garlan et al. [6] and the researchers involved in the Cadena project [12] apply model checking techniques to distributed publish/subscribe architectures. Neither of these approaches is not based on UML diagrams, but both define the behavior of components through their own specific language [6] or through an IDL-like specification language [12]. Garlan et al. provide different middleware specifications that can be integrated into the validation tool. The properties are specified in CTL, which is the formalism provided by the SMV [18] model checker. Although the ideas in this proposal and in our approach are similar, there are some differences: (a) we provide a complete graphical front-end for the designer who does not have to deal with any particular textual and logical formalism, (b) the set of guarantees supported by our middleware is richer (e.g., [6] does not deal with subscriptions), and (c) LSCs provide the operators for describing the communication among components in a graphical and natural way, whereas CTL is a general-purpose temporal logic.

Cadena is the other proposal that supports publish/subscribe architectures. It deals directly with the CORBA Component Model (CCM), and both, the description of component interfaces and the topologies of components, are given directly in CORBA. Cadena supports only the CORBA Component Model (CCM) as middleware, while we support a wider set of models: the CORBA dispatcher is one possible instantiation of our parametric component. In Cadena, the communication is established explicitly, that is, each component declares the components from which it desires to receive events. This particular implementation of the publish/subscribe paradigm does not allow the use of Cadena with other middleware platforms. Cadena supports the Bandera Specification Language [4] to specify properties against which the system must be validated.

The same research group which proposed Bandera and Cadena has also proposed Bogor [22], a special purpose modular model checker for analyzing software architectures. The main difference with respect to our approach is that they use their own modeling languages — instead of well known diagrammatic notations — to specify both components and properties. An interesting aspect of Bogoeer is that it allows users to change its modules: the optimization phase is not in the encoding of models, but is shifted to within the model

checker. The specification language is richer than those offered by the other model checkers, and the adoption of different modules allows users to investigate different alternatives during the checking phase. For example, in our approach we cannot change the algorithms embodied in SPIN. By using Bogor, we would instead be able to modify how the system works by both using its library of components and implementing our own strategies. This is an important aspect that changes the perspective in model checking software architectures and makes Bogor an interesting alternative as a support model checker for our approach.

7 Conclusions and Future Work

In this chapter we have presented an approach to model and validate distributed architectures based on the publish/subscribe paradigm. Application-specific components are modeled as UML statechart diagrams while the dispatcher is supplied as a configurable predefined component. Developers do not handle its internals directly, but only identify the desirable features by selecting some check boxes. We have validated this component by applying the same approach proposed here. We have built a federation of simple, dummy, components to stress all its possible configurations. The simplicity of components has allowed us to state that any misbehavior was due to the dispatcher itself.

Validation properties are described with *live sequence charts* (LSCs) and transformed to automata. Components, middleware, and properties are bundled together, translated into Promela, and then passed to SPIN to validate the architecture.

Our future work is heading in different directions. We would like to extend the approach to model time and probabilities associated with publication/notification of events. But we are also trying to understand how analysis can be performed in an incremental way. The concept is twofold and addresses both evolution of models and composition of properties. In the first case, we are interested in “adapting” properties to evolved systems. In the second case, we are trying to understand how to prove new properties incrementally by composing fragments of properties already verified.

We are also studying how to better support the designer while modeling applications. We are evaluating how the adoption of different model checkers may impact the quality of the results of analysis, and we are exploring the possibility of automatically generating code from these models.

References

1. A. Carzaniga and D. S. Rosenblum and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug 2001.

2. R.V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.
3. K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An automatic verification tool for UML. Technical Report CSE-TR-423-00, 2000.
4. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proceedings of the 7th SPIN Workshop*, volume 1885 of *LNCS*, August 2000.
5. G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineerings*, 27(9):827–850, September 2001.
6. D. Garlan and S.Khersonsky and J.S. Kim. Model checking publish-subscribe systems. In *Proceedings of the 10th SPIN Workshop*, volume 2648 of *LNCS*, May 2003.
7. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
8. P.T. Eugster, P.A. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.
9. S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison Wesley, 1995.
11. S. Gnesi, D. Latella, and M. Massink. Model checking UML statecharts diagrams using JACK. In *Proceedings of the 4th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, pages 46–55. IEEE Press, 1999.
12. J. Hatcliff, W. Deng, M.B. Dwyer, G. Jung, and V. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering*, pages 160–172, May 2003.
13. G.J. Holzmann. *Design and Validation of Network Protocols*. Prentice Hall, 1991.
14. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
15. P. Inverardi, H. Muccini, and P. Pelliccione. Automated check of architectural models consistency using SPIN. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering conference (ASE)*, pages 349–349, 2001.
16. N. Kaveh and W. Emmerich. Deadlock detection in distributed object systems. In *Proceedings of the joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 44–51, 2001.
17. J. Lilius and I.P. Paltor. vUML: a tool for verifying UML models. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE)*, pages 255–258, October 1999.
18. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.
19. Object Management Group. *XMI: XML Metadata Interchange, v.1.2*, 2002. <http://www.omg.org/>.
20. M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley, 2004. To appear.

21. A. Pnueli. The temporal logic of programs. In *Proceedings of 18th IEEE Symposium Foundations of Computer Science(FOCS)*, pages 46–57, October 1977.
22. Robby, M.B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the joint 9th European Software Engineering Conference (ESEC) and 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 267–276, 2003.
23. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Lognman, 1999.
24. T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13 pages, 2001.
25. Sun Microsystem. Java Message Service Specification. Technical report, Sun Microsystem Technical Report.
26. TIBCO. The power of now. TIBCO hawk. www.tibco.com/solutions/.
27. M. Young and R.N. Taylor. Rethinking the taxonomy of fault detection techniques. In *Proceedings of the 11th International Conference on Software Engineering*, pages 53–62, May 1989.