An Approach to Model and Validate Publish/Subscribe Architectures

Luca Zanolin, Carlo Ghezzi, and Luciano Baresi
Politecnico di Milano
Dipartimento di Elettronica ed Informazione
P.za L. da Vinci 32, 20133 Milano (Italy)
{zanolin|ghezzi|baresi}@elet.polimi.it

ABSTRACT

Distributed applications are increasingly built as federations of components that join and leave the cooperation dynamically. Publish/subscribe middleware is a promising infrastructure to support these applications, but unfortunately complicates the understanding and validation of these systems. It is easy to understand what each component does, but it is hard to understand what the global federation achieves.

In this paper, we describe an approach to support the modeling and validation of publish/subscribe architectures. Since the complexity is mainly constrained in the middleware, we supply it as a predefined parametric component. Besides setting these parameters, the designer must provide the other components as UML *statechart diagrams*. The required global properties of the federation are then given in terms of *live sequence charts* (LSCs) and the validation of the whole system is achieved through model checking using SPIN. Instead of using the property language of SPIN (*linear temporal logic*), we render properties as automata; this allows us to represent more complex properties and conduct more thorough validation of our systems.

1. INTRODUCTION

The *publish/subscribe paradigm* has been proposed as a basis for middleware platforms that support software applications composed of highly evolvable and dynamic federations of components. According to this paradigm, components do not interact directly, but their communications are mediated by the middleware. Components declare the events they are interested in and when a component publishes an event, the middleware notifies it to all components which subscribed to it.

Publish/subscribe middleware decouples the communication among components. The sender does not know the receivers of its messages, but it is the middleware that identifies them dynamically. As a consequence, new components can dynamically join the federation, become immediately active, and cooperate with the other components without any reconfiguration of the architecture.

The gain in flexibility is counterbalanced by the difficulty for the designer to understand the overall behavior of the system. It is hard to get a picture of how components cooperate and understand the global data and control flows. Although components might be working correctly when they are examined in isolation, they could provide erroneous services in a cooperative setting.

These problems motivate our approach to model and validate publish/subscribe architectures. Modeling the middleware is the most complex task, since we must consider how components communicate in a distributed environment. The complexity of such a model is partially counterbalanced by the fact that the middleware is not application-specific. We can use the same (model of a) middleware for several different architectures. This is why we provide a ready-to-use parametric model of the middleware. The designer has to configure it and provide the other components as *UML statechart diagrams* [17]. Following this approach, designers trust the middleware and validate the cooperation of components.

The designer describes the global properties of the federation in terms of *live sequence charts* (LSCs) [6] and the validation of the whole system is achieved through model checking using SPIN [10]. LTL (linear temporal logic), the property language of SPIN, is not enough to render what we want to prove. We by-pass this limitation by transforming LSCs in automata. Both the model and properties are then translated into Promela [9] and passed to SPIN.

The paper is organized as follows. Section 2 presents our approach to model publish/subscribe architectures and exemplifies it through a simple example of a hypothetical *eHouse*. Section 3 discusses validation and shows how to model the properties for our case study and how to transform them into automata. Section 4 surveys the related work, and Section 5 concludes the paper.

2. MODELING

In publish/subscribe architectures, components *exchange* events through the middleware. Thus, any model of these architectures must explicitly consider the three main actors: events, middleware, and components. The next three sections describe how to model them using our approach and exemplify all concepts on a fragment of a hypothetical *eHouse*. We focus on a simple service that allows users to

take baths. When the user requires a bath, the service reacts by warming the bathroom and starting to fill the bath tub. When everything is ready, the user can take the bath. Lack of space forbids us to present all models, but interested readers can refer to [13] for the complete set.

2.1 Events

Publish/subscribe architectures do not allow components to exchange events directly. The communication is always mediated by the middleware through the following operations: *subscribe*, *unsubscribe*, *publish*, and *notify*. Components *subscribe* to declare the interest for some specific events, and, similarly, they *unsubscribe* to undo their subscriptions. Components *publish* events to the middleware that *notifies* them to the other components.

Events have a name and a (possibly empty) set of parameters. When components subscribe/unsubscribe to/from events, they can specify them fully or partially. For instance:

subscribe("bath", "ready")

means that the component wants to know only when the bath is ready, but:

subscribe("bath", $)

means that it wants to know all *bath* events.

2.2 Middleware

As we have already said, the developer does not model the middleware explicitly, but has to configure the *middleware component* that we supply with our approach. Unfortunately, the term middleware is not enough to fully characterize its behavior. The market offers different alternatives: standards (e.g., Java Message Service (JMS) [19]) and implementations from both university (i.e., Siena [1], Jedi [4]) and industry (i.e., TIBCO [20]). All these middleware platforms support the publish/subscribe paradigm, but with different qualities of service.

Given these alternatives, the definition of the parameters that the middleware component should offer to developers is a key issue. On 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. The identification of the minimal set of parameters is essential to verify the application: the simpler the model is, the faster verification would be. Since our goal is the verification of how components cooperate, we can assume that the model must describe what is seen by components and we can get rid of many internal details. For instance, the middleware can be distributed on several hosts, but this is completely transparent to components and provided services are the same. Important differences are on the quality of service, thus we model how services are provided, instead of modeling how the middleware works.

After several attempts, we have selected the following three characteristics as the key elements that concur in the definition of the behavior of a generic middleware:

Delivery Some middleware platforms are designed to cope with mobile environments or with strong performance on the delivery of events. Due to these requirements, the designer can choose to lower the warranties on delivery of events and not to guarantee that all published events are notified to all components. On one hand, this reduces the reliability of the system, but, on the other hand, it increases its performance. Thus, event delivery can be characterized by two alternatives: (a) all events are always delivered, or (b) some events can be lost.

Notification If a middleware behaves *correctly*, it notifies events by keeping the same order in which they were published. Nevertheless, this behavior is not always respected due to the environment in which the middleware executes. 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 the events published by the same component, or (c) there is no relationship and events may be notified randomly.

For instance, if component A publishes events x_1 and x_2, and component B publishes y_1 and y_2, the middleware 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 $x < y$ means that x is notified before y.

Note that the first hypothesis only works in general in a centralized setting. It can be used as an idealized approximation in other practical cases.

Subscription When a component declares the events in which it is interested, that is, it subscribes to these events, it starts receiving them. However, the distributed environment can make the middleware not to react immediately to new subscriptions. Once more, our characterization identifies two alternatives: (a) the middleware immediately reacts to (un)subscriptions, or (b) these operations do not have immediate effects and are delayed.

The actual middleware comes from choosing one option for these three main characteristics. These characterizations cover most of the warranties that a middleware should satisfy. If this is not the case, developers can always get rid of parametric middleware, elaborate their particular model of the middleware, as a UML statechart diagram, integrate it in the architecture, and validate the whole system. They lose the advantages as to modeling, but keep the validation approach.

To increase the confidence in the parametric model of the middleware, we validated it by applying the same approach that we are proposing in this paper. We built a federation of simple – dummy – components to stress all the possible configurations of our middleware. The ease of components allowed us to state that any misbehavior was due to the middleware itself.

Referring to the eHouse example, we assume that the architecture is built on a middleware that delivers all events, keeps the same order of publication and notification, and reacts immediately to all (un)subscriptions.

2.3 Components

The designer provides a UML statechart diagram for each component, where transitions describe how the component reacts upon receipt of an event. Transition labels comprise two parts separated by /. The first part (i.e., the precondition) describes when the transition can fire, while the second part defines the actions associated with the firing of the transition. For instance, the label:

consume("bath", "full") / publish("bath", "ready")

states that the transition can fire when the component is notified that the bath tub is full of water and publishes an event to say that the bath is ready.

Notified events are stored in a *notification queue*. The component retrieves the first event and if it does not trigger any transition exiting the current state, the component discards it and processes the following one. This mechanism allows components to evolve even if they receive events in which they are not interested in their current state.

Moving to our running example, besides the middleware described in the previous section, we have five components that cooperate to provide the service: *User*, *Bathroom*, *Heating*, *Bath*, and *PowerManager*.

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

Finally, *PowerManager* manages the provision of electricity. If there is a blackout, this component notifies this failure and switches from primary to secondary power supplier. The secondary power supplier is less powerful than the primary one and some electric devices must be turned off. 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.

Figure 1 shows the statechart diagram of *Bathroom*. It starts in state *Idle* waiting for some events. At this stage, it is only subscribed to events that ask for a bath. When the user notifies that she/he needs a bath, *Bathroom* evolves 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 about 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 instance, the variable *bathStatus* set to *true* means that the bath tub is full of water.

When the bath tub is full of water and temperature is hot, the bath is *Ready*. In this state, the component is not interested in the events about bath and heating anymore, thus it unsubscribes from them. Finally, after the user takes the bath, *Bathroom* restores temperature to *cold*[1].

Figure 2 shows the statechart diagram of *Heating*. For simplicity, we suppose that, when this component starts, the power supplier is working correctly 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*.

[1]Temperature can only assume two values: cold and hot.

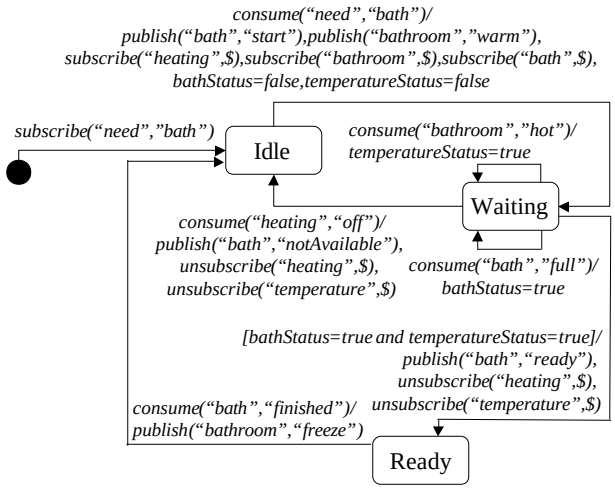

Figure 1: Bathroom

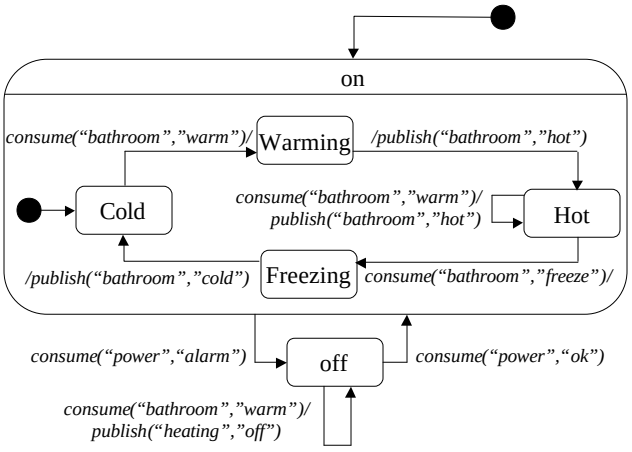

Figure 2: Heating

3. VALIDATION

Validation comprises two main aspects: the definition of the properties to validate the cooperation of components and the transformation of both the model and properties into automata (i.e., Promela).

3.1 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 these reasons, we did not use any temporal logic formalisms, like *linear temporal logic* [16] (LTL), since they work at a different level of abstraction and, thus, developers can find it difficult to use. We chose *live sequence charts* (LSCs) [6] since they are a graphical formalism powerful enough to describe how entities exchange messages, that is, the properties that we want to analyze.

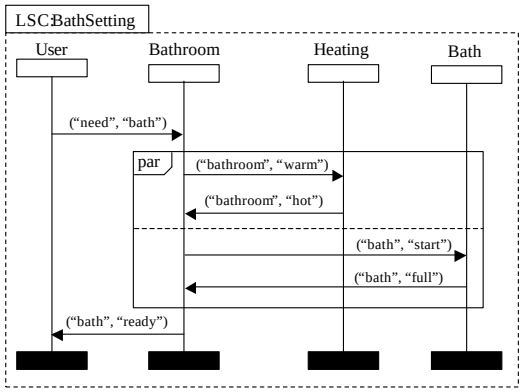

Figure 3: A basic LSC: Bath Setting

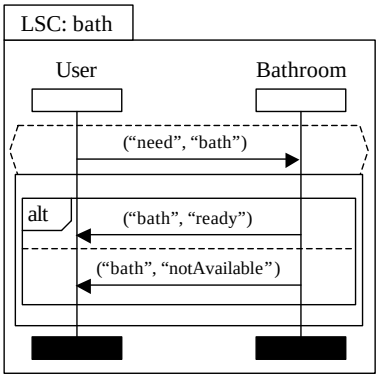

Figure 4: LSC: Bath Ready

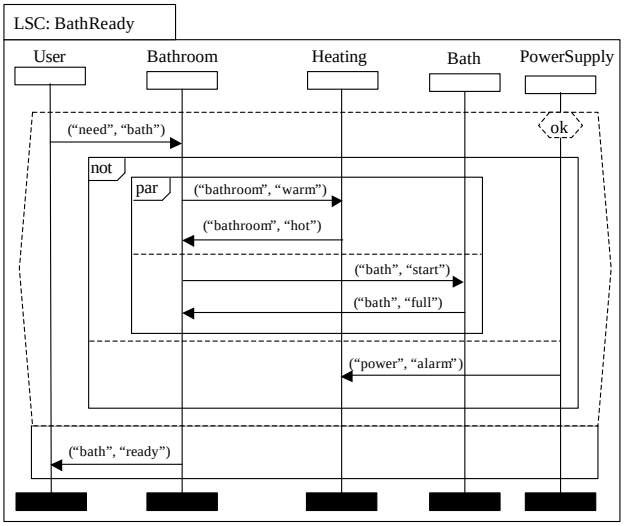

Figure 5: LSC: Bath Ready (revised)

Briefly, a basic LSC diagram describes a scenario on 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. The life-cycle of an entity is rendered as a vertical line and a black rectangle: The entity is created when we draw the white rectangle and dies when we draw the black one. Messages exchanged between entities are drawn as arrows and are asynchronous by default. Each message is decorated with a label that describes the message itself.

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, we implicitly assume 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).

After introducing LSCs, let us define some properties on how our bath service should behave. For instance, we want to state that, when the user requires the bath, the temperature in the bathroom increases and the bath tub starts to fill. This two tasks are done in parallel and after the termination of both, the user is notified that the bath is ready. This property is described in Figure 3, which shows a basic LSC scenario. *User* requires the bath and *Bathroom* reacts notifying that *Heating* must start to warm the bathroom and *Bath* to fill the bath tub. The two tasks are performed in parallel without any constraint on their order. This parallelism is described through the *par* operator that states that its two scenarios (i.e., warming the bathroom and filling the bath tub) evolve in parallel without any particular order among the events they contain. When the bath tub 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 that the bath is ready. In fact, if we had a blackout, *User* would receive an event to inform her/him that the bath cannot be set. We do not require that the application always complies with this scenario, but, that there are some possible evolutions that are compliant with it. In LSCs, these scenarios are called *provisional* or *cold* scenarios and are depicted in dashed rectangles, as shown in Figure 3.

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 Figure 4. LSCs allow us to define such a property through a *mandatory* or *hot* scenario. In general, it is difficult to identify global properties that must be satisfied in all evolutions. For this reason, LSCs support the definition of pre-conditions, that is, the property must hold in all the evolutions for which the precondition holds. Preconditions are drawn in dashed polygons, while the hot part of the scenario is depicted in a solid rectangle. For clarification, we can say that the precondition implies the hot scenario. The chart in Figure 4 states that, for all the evolutions in which *User* requires a bath, *Bathroom* notifies two possible events, that is, the bath is ready or the bath is not available. In this chart, we exploit *alt* (alternate), which is another operator supported by LSCs. This operator says that one of its two scenarios must hold. Thus, Figure 4 describes that, when we require a bath, we must receive an event to know if the bath is ready or not.

Finally, we can redefine the previous property (Figure 3) to define when the bath must be available: The bath must always become ready if in the meanwhile we do not have

blackouts. This property is described in Figure 5. If we have no blackouts while *Heating* warms the bathroom, the bath must always become available. In this chart, we introduce the *not* operator that 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 we have no blackouts while the bath sets itself, *User* always receives a positive answer, i.e., the bath is ready.

3.2 Transformation

So far, we have shown how to model the architecture and define the properties. After these steps, we can start the analysis. We decided to use the SPIN model checker [10] as verifier, but as we have 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 [9].

We customize the middleware according to the parameters set by the developer. Each alternative corresponds to a Promela package and the tool selects the right packages and assembles the model of the middleware directly.

Translating of statechart diagrams in Promela is straightforward. We do not need to describe this translation since it has been done by others before (e.g., vUML [14] and veriUML [2]) and to implement our translation we have borrowed from these approaches.

Properties are translated in two ways: They are described through automata and, if necessary, through auxiliary LTL formulae. This translation is rather complex since SPIN does not support the verification of existential properties natively. 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 several times for each property that we want to verify. Instead of using LTL formulae and their translation into Büchi automata, we investigate a different solution based on simple automata rendered as Promela processes.

We verify if LSC hold by reasoning on the state reachability feature provided by SPIN. However, automata are not enough to describe all the LSC features, and, when required, we introduce LTL formulae to overcome this problem. The transformation of an LSC into an automaton (and LTL) goes through these four steps:

1. We simplify the property by downgrading all the hot scenarios to cold ones;

2. We translate the simplified property into an automaton that recognizes the sequence of events described by the LSC. This task is quite easy since the structure of the automaton replicates the one 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 constraints expressed as LTL formulae. These constraints state 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.

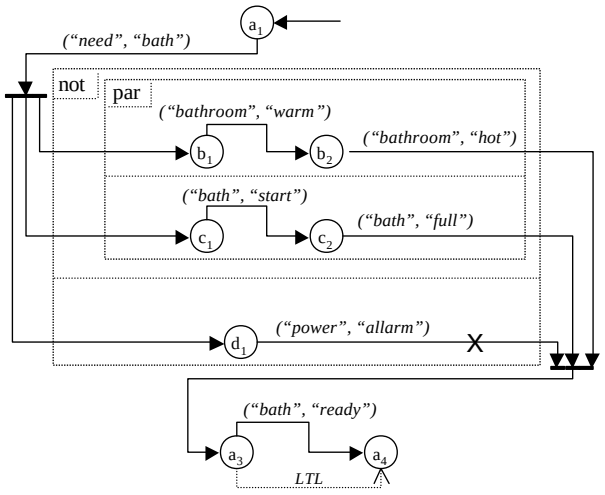

Figure 6: The automaton that corresponds to the LSC of Figure 5

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

For space reasons, we omit the details of this algorithm and we illustrate the translation informally through an example. Let us consider the LSC of Figure 5 and the corresponding automaton of Figure 6. This automaton has three types of arrows: solid, solid with a cross, and dashed. Solid arrows describe *standard* transitions and are decorated with 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 consumed by the component[2]. This type of arrow can end in a state or in a fork/join bar. Besides recognizing events as the previous kind, 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 describe constraints on the evolution of the automaton. The constraint – described by an LTL formula – is always the same: 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).

The automaton of Figure 6 has the same structure as the LSC of Figure 5. This means that when the middleware notifies the first event (i.e.,*("need", "bath")*), the fork bar is enabled and the automaton splits its evolution in three different threads. Moving top-down, the first thread describes the warming of the bathroom, the second thread the filling of the bath tub, and the last thread corresponds to the blackout.

[2]For the sake of clarity, in Figure 6 we do not describe who publishes or consumes events.

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 middleware notifies the last event, 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 formula – states that there exists an evolution in which we have no blackout, after setting, the bath becomes available to the user.

Nevertheless, the property of Figure 5 states that, if we have no blackout, the bath must be always available. This is why we must refine the automaton and add the last arrow. In the example, the hot scenario only concerns the last event, that is the transition between states a_3 and a_4. 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 an 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 an evolution that is compliant with this property. Finally, 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 whole translation – middleware, components, and properties – creates a specification that can be analyzed by SPIN. If it finds misbehaviors in the way components cooperate, they are rendered back to the designer as an evolution of the trace that highlights the error. Unfortunately, this is not always the case: Some misbehaviors cannot be simply described through execution traces. For instance, if a *cold* scenario is not respected, that is, no evolution is compliant with the scenario, it is meaningless to show an execution of the system.

The validation of the bath service has been performed by assuming a middleware that delivers all events, keeps the same order of publication and notification, and reacts immediately to all (un)subscriptions. The validation process shows that the service is incorrectly designed since it violates the property shown in Figure 4. For example, if we consider the following scenario: *User* asks for a bath and *Bath* starts to fill the bath tub. At the same time *Heating* increases the temperature, but before becoming hot, we have a blackout that turns the heating off. At this point, *Bathroom* and *User* wait forever since *Heating* does notify neither that it is switched off nor that temperature is hot.

This misbehavior can be avoided by modifying the arrows between states *on* and *off* in *Heating* (Figure 2) to introduce the publication of an event to notify the failure of *Heating*.

4. RELATED WORK

Software model checking is an active research area. A lot of effort has been devoted to applying this technique to the validation of application models, often designed as UML statechart diagrams, and the coordination and validation of the components of distributed applications based on well-defined communication paradigms.

vUML [14], veriUML [2], JACK [7], and HUGO [18] provide a generic framework for model checking statecharts. All of these works 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 broadcast communication, that is, the events produced by a component are notified to all the others. vUML and veriUML support the concept of channel, that is, each component writes and reads messages on/from a channel. These proposals aim at general-purpose applications and can cover different domains. However, they 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, 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 a middleware which guarantees that the order of publication is kept also during notification. These approaches also impose that channels between components be explicitly declared: vUML and veriUML do not allow us to create or destroy components at run-time and the topology of the communication is fixed.

The proposals presented so far do not support a friendly language to define properties. With vUML we can only state reachability properties, while with veriUML, JACK, and HUGO we can also define complex properties on how the application evolves, but in all cases they 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.

Two other projects try to overcome these limitations (i.e., definition of properties and communication paradigm). Inverardi et al. [11] 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 be always complaint with the MSC. MSCs are directly translated into LTL, the property language supported by SPIN.

Kaveh and Emmerich [12] 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 by this tool.

Garlan et al. [5] and the Cadena project [8] apply model checking techniques on distributed publish/subscribe architectures. Both of these proposals do not deal with UML diagrams, but define the behavior of components through a proprietary language [5] or using the CORBA IDL-like specification language [8].

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 [15] 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 that does not have to deal with any particular textual and logical formalism, (b) the set of warranties supported by our middleware is wider (e.g., [5] does not deal with subscriptions), and (c) LSCs provide the operators for describing the communication among components in a graphical and natural way, while CTL is a general-purpose temporal logic.

Cadena, which is the other proposal that supports publish/subscribe architectures, only deals with the CORBA Component Model (CCM). 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 [3] to specify properties against which the system must be validated.

5. CONCLUSIONS AND FUTURE WORK

In this paper we present 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 middleware is supplied as a configurable predefined component. As to validation, properties are described with *live sequence charts* (LSCs) and transformed into automata. Components, middleware, and properties are translated into Promela and then passed to SPIN to validate the architecture.

Our future work is headed to different directions. We are extending the approach to model time and probabilities associated with publication/notification of events. But we are also trying to understand how these analyses can be performed in an incremental way: do properties remain valid? What about results?

Finally, we are studying how to better support the designer while modeling applications, the adoption of different model checkers to understand the impact they have on obtained results, and the possibility of automatically coding the infrastructure of these architectures by means of the analysis models.

6. 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] K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An automatic verification tool for UML. Technical Report CSE-TR-423-00, 2000.

[3] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proceedings of the SPIN Software Model Checking Workshop*, volume 1885 of *LNCS*, August 2000.

[4] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transaction on Software Engineerings*, 27(9):827–850, September 2001.

[5] 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.

[6] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[7] S. Gnesi, D. Latella, and M. Massink. Model Checking UML Statecharts Diagrams using JACK. In *Proc. Fourth IEEE International Symposium on High Assuarance Systems Enginering (HASE)*, pages 46–55. IEEE Press, 1999.

[8] J. Hatcliff, W. Deng, M.D. Dwyer, G. Jung, and V. Ranganath. Cadena: An integrated development, analysys, and verification environment for component-based systems. To appear in Proc. of the International Conference on Software Engineering (ICSE 2003),IEEE Press,2003.

[9] G.J. Holzmann. *Design and Validation of Network Protocols*. Prentice Hall, 1991.

[10] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[11] P. Inverardi, H. Muccini, and P. Pelliccione. Automated check of architectural models consistency using SPIN. In *Proc. Automated Software Engineering conference (ASE2001)*, pages 349–349. IEEE Press, 2001.

[12] N. Kaveh and W. Emmerich. Deadlock detection in distributed object systems. In *Proc. of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 44–51, Vienna, Austria, 2001. ACM Press.

[13] L. Zanolin and C. Ghezzi and L. Baresi. Model Checking Pusblish/Subscribe Architectures. Technical report, 2003.

[14] J. Lilius and I.P. Paltor. vUML: a tool for verifying UML models. In *Proc. 14th IEEE International Conference on Automated Software Engineering (ASE)*, pages 255–258, Cocoa Beach, Florida, October 1999.

[15] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[16] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science(FOCS-77)*, pages 46–57, Providence, Rhode Island, October 1977.

[17] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Lognman, 1999.

[18] 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.

[19] Sun Microsystem. Java Message Service Specification. Technical report, Sun Microsystem Technical Report.

[20] TIBOC. The Power of now. Tibco hawk. http://www.tibco.com/solutions/.