Proceedings of the 10th International SPIN Workshop on
Model Checking of Software (SPIN 03) May, 2003, Portland, Oregon.

Model Checking Publish-Subscribe Systems

David Garlan, Serge Khersonsky, and Jung Soo Kim

Carnegie Mellon University
School of Computer Science
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
+1 412 268-5056
garlan@cs.cmu.edu

Abstract. While publish-subscribe systems have good engineering properties, they are difficult to reason about and to test. Model checking such systems is an attractive alternative. However, in practice coming up with an appropriate state model for a pub-sub system can be a difficult and error-prone task. In this paper we address this problem by describing a generic pub-sub model checking framework. The key feature of this framework is a reusable, parameterized state machine model that captures pub-sub run-time event management and dispatch policy. Generation of models for specific pub-sub systems is then handled by a translation tool that accepts as input a set of pub-sub component descriptions together with a set of pub-sub properties, and maps them into the framework where they can be checked using off-the-shelf model checking tools.

1 Introduction

An increasingly common architectural style for component-based systems is publish-subscribe (pub-sub). In this style components "announce" (or "publish") events, which may be "listened to" (or "subscribed to") by other components. Components can be objects, processes, servers, applications, tools or other kinds of system runtime entities. Events may be simple names or complex structures.

The key feature of such systems is that components do not know the name, or even the existence, of listeners that receive events that they announce. The consequent loose coupling between components in a pub-sub system makes it relatively easy to add or remove components in a system, introduce new events, register new listeners on existing events, or modify the set of announcers of a given event. Thus implicit invocation systems support the ability to compose and evolve large and complex software systems out of independent components [17].

However, there is a downside to pub-sub systems: they are hard to reason about and to test. In particular, given the inherent non-determinism in the order of event receipt, delays in event delivery, and variability in the timing of event announcements, the number of possible system executions becomes combinatorially large. There have been several attempts to develop formal foundations for specifying and reasoning about pub-sub systems [1, 3, 10, 7, 8], and this area remains a fertile one for formal verification. Unfortunately, existing notations and methods are difficult to use in practice by non-formalists, and require considerable proof machinery to carry out.

An attractive alternative to formal reasoning is the use of model checking. A model checker finds bugs in systems by exploring all possible execution states of an approximating finite state model to search for violations of some desired property (often described as a temporal logic formula). Model checking has had great success in hardware verification, and is starting to be used by researchers to find errors in software systems [5].

While model checking is a powerful technique, one of the stumbling blocks to using it is the creation of appropriate finite state models for the systems being checked. Since most software systems are infinite-state, one must first find suitable abstractions that reduce the system to a finite state model, without eliminating the class of errors that one wants to pinpoint.

A related problem is finding a suitable *structure* for the state model so that properties of interest can be easily expressed in terms of the state machine, and further, so that when errors are found, they can be easily related back to the original system. In general, this abstraction and structuring process is highly system- and domain-specific: techniques for deriving models from one class of software system may be completely inappropriate for another. This means that the person creating a model often has to develop a new set of structures from scratch for each system.

One step towards improving this situation would be to provide generic structured models for certain classes of systems that can be easily tailored to the needs of a particular system within class. In this paper we do just that for pub-sub systems. The key idea is to provide a generic, parameterized pub-sub model-checking framework that factors the problem into two parts: (a) reusable model components that capture run-time event management and dispatch, and (b) components that are specific to the pub-sub application being modeled.

Since much of the complexity of modeling pub-sub systems is in the run-time event mechanisms, the cost of building a checkable model is greatly reduced. For example, within our framework event delivery policy becomes a pluggable element in the framework, with a variety of pre-packaged policies that can be used "off-the-shelf." To further reduce costs of using a model checker, we also provide a tool that translates pub-sub application component descriptions (specified in an XML-like input language) and properties into the a lower-level form where they can be checked using standard model checking tools.

2 Related Work

Publish-subscribe systems have received considerable attention in commercial products and standards (e.g., [20, 18]), as well as academic research systems (e.g., [4, 19]). Most of these efforts have focused on the problem of constructing systems that exhibit desirable run time qualities, such as scalability, efficiency, adaptability, and security, rather than the problem of reasoning about the correctness of such systems, as in this paper.

There has been some foundational research on formal reasoning for pub-sub systems [1, 3, 10, 7, 8]. While these efforts provide a formal vehicle for reasoning about pub-sub systems, at present they require an expert in formal specification and theorem proving to use them effectively. Our research builds on those underpinnings, but aims to make reasoning more accessible by leveraging automated property checking afforded by today's model checkers.

Model checking of software systems is an extremely active area of research at present. Like our research, much of this effort aims to make model checking easy for practitioners to use, for example by allowing the input language to be a standard programming language (e.g., [14, 11]), and by providing higher-level languages for specifying properties to check (e.g., [6]). However, to the best of our knowledge, none of these efforts has focused on exploiting the regularities of a particular component integration architecture, as we do for publish-subscribe systems.

Finally, there have been several previous efforts at providing formal, checkable models for software architectures. Some of these even use model checkers (e.g., [2, 12]) to check properties of event-based systems. However, none has been tailored to the specific needs of pub-sub systems development.

3 Modeling Pub-Sub Systems

How should one go about modeling a pub-sub system as a checkable finite state model? Answering this question is difficult in general because pub-sub systems vary considerably in the way they are designed. Although all share the basic principals of loose component coupling and communication via multi-cast events, specific details differ from system to system [15]. However, a typical system can generally be described as consisting of the following structural elements:

- **Components:** Components encapsulate data and functionality, and are accessible via well-defined interfaces. Interface "methods" are invoked by the pub-sub run-time system as a consequence of event announcements, and each method invocation may result in more events being announced in the system.
- **Event types:** The types of events indicate what can be announced in the system. Events may have substructure including a name, parameters, and other attributes such as event priorities, timestamps, etc. In some systems, event types are fixed; in others they can be user-specified.
- **Shared variables:** In addition to event announcement, most event systems support some form of shared variables. For example, in an object-oriented implementation, the methods of a class would have access to the local shared variables of that class.
- **Event bindings:** Bindings define the correspondence between events and components' methods that are to be invoked in response to announcing these events.

– **Event delivery policy:** Delivery policy defines the rules for event announcement and delivery. From an implementation point of view, policies are typically encoded in an event dispatcher, a special run-time component that brokers events between other components in the system according to the event bindings. Event delivery policy dictates such factors as event delivery order (whether or not to deliver events in the order announced), whether events can be lost, the ability to deliver events to recipients in parallel, use of priorities, etc.
– **Concurrency model:** The concurrency model determines which parts are assigned separate threads of control (e.g., one for the whole system, one for each component, or one for each method).

Given this architectural structure, there are two main stumbling blocks to creating a state model for an pub-sub system that is suitable for model checking. One is the construction of finite-state approximations for each of the component behaviors (methods). In the case of our prototype system we have adopted the following restrictions: all data has a finite range; the event alphabet and the set of components and bindings are fixed at runtime; there is a specified limit on the size of the event queue and on the length of event announcement history maintained by the dispatcher; there is a limit on the size of invocation queues (pending method invocations as a result of event delivery).

A second problem is the construction of the run-time apparatus that glues the components together, mediating their interaction via event announcements. This involves developing state machines that maintain pending event queues, enforce dispatch regimes (correctly modeling non-deterministic aspects of the dispatch), and providing shared variable access. In principle, this part of the modeling process could be done afresh for each system using brute force. Unfortunately modeling pub-sub systems involves a fair amount of state machinery, and is not trivial to get right. Moreover, once built, it is hard to experiment with alternative run time mechanisms. For example, one might want to investigate the consequences of using a dispatcher in which events could be lost, or one in which causal ordering for events is guaranteed.

In our research we have factored out this second effort, by providing reusable run-time model checkable infrastructure for the run-time mediation. To account for variability in the dispatch mechanisms we provide pluggable state modules that allow a modeler to choose from one of several possible run-time models.

Figure 1 illustrates the resulting model structure. The user provides a specification of (a) the component methods (as state machines), (b) an optional set of shared variables, (c) the list of events, (d) the event-method bindings, (e) a model of the environment (or a specification of its behavior), and (f) a set of properties to check. In addition, the user picks the specific dispatch policy (from the options listed in Figure 2), and the concurrency (or synchronization) model (from the options also listed in Figure 2). Using a model generation tool that we built, these parts are then translated into a set of interacting state machine descriptions and properties that are checked using a commercial model checker.[1]

[1] We use the Cadence SMV model checker [13].

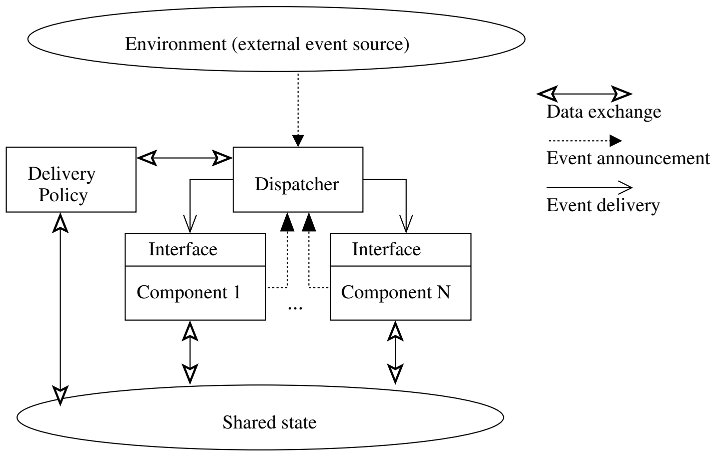

Fig. 1. Structure of a Pub-Sub System Model

In the remaining sections we detail the design of each of these parts and provide examples of their use.

Delivery Policies:
- *Asynchronous:* immediate return from announcement
  - *Immediate:* immediate invocation of destination
  - *Delayed:* accumulate events before announcement
- *Synchronous:* no return until event completely processed

Synchronization Options:
- *Separate threads of control:*
  - Single thread per component
  - Multiple threads per component
    - Concurrent invocation of different methods
    - Concurrent invocation of any method
- *Single thread of control*

Fig. 2. Delivery Policies and Synchronization Options

4 The Run-Time State Model

In our approach the run-time infrastructure supporting pub-sub communication has two parts: (a) the mechanisms that interact with the components of the system to handle event announcement, event buffering, and method invocation; and (b) the mechanisms that implement event dispatch and event delivery policy.

In more detail, the first part must provide state machine structures that faithfully model:

– Event announcements by the system components.
– Storage of event announcements by the run-time infrastructure in preparation for dispatch.
– Event delivery to the system components, after the dispatch mechanism has selected the event(s) for dispatch (described below).
– Invocation of the methods bound to the delivered events.
– Invocation acknowledgement, which indicates that a method has completed its execution.

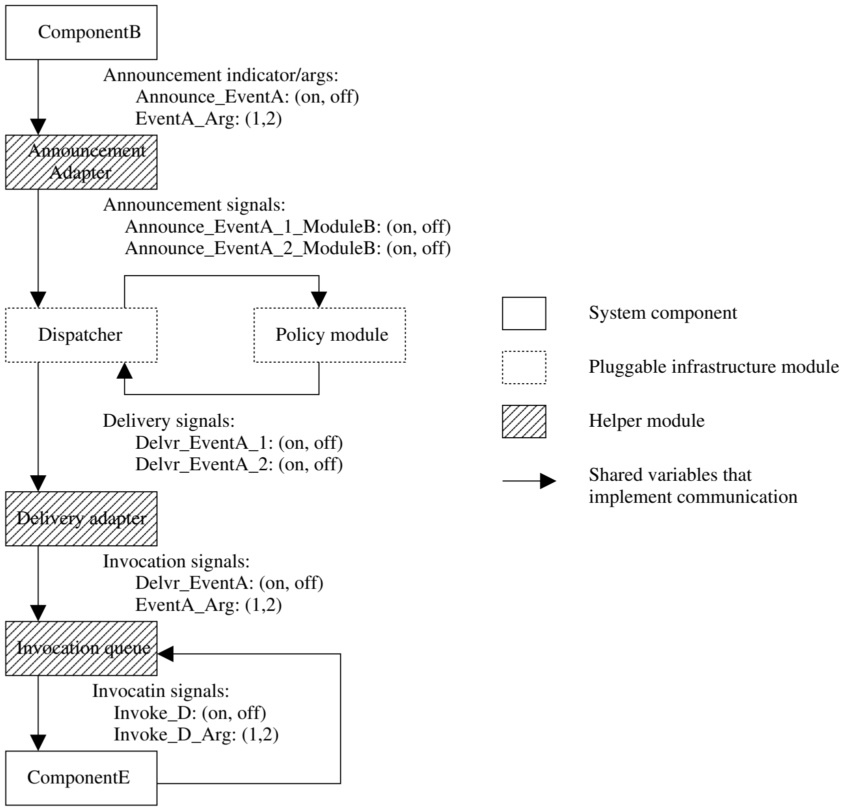

Fig. 3. Event Dispatch Structures

Our state model generation method implements this communication in terms of the following shared state variables (see also Figure 3):

– Event announcement: each component announcing a particular event indicates the announcement via a binary *announcement indicator* and a set of corresponding *announcement attributes* such as arguments, priority, etc. For example, if `Announce_EventA` has one argument that can assume values 1 or 2, and `ComponentB` can announce this event, then `ComponentB` announces the event by writing to two shared variables: the 'on/off' flag `EventA` and a $\{1, 2\}$-valued variable `EventA_Arg`.

– Announcement acceptance: for each event type, there is a binary-valued *announcement signal* for each possible combination of the event's attributes, such as argument values, source component, priority, etc. For example, a set of binary announcement signals for `EventA`, explained as above, would be `Announce_EvtA_1_ModuleB`, and `Announce_EvtA_2_ModuleB` (assuming that the event's argument is the only attribute of interest). An adapter mechanism (the *announcement adapter* in Figure 3) is used to assign values to these four flags based on corresponding announcement indicator and announcement attribute values written by `ComponentB`. The announcement signals for each event/attribute combination are then fed directly into the pending event accounting mechanism in the dispatcher (see below).
– Event delivery: implementation of event *delivery signals* is similar to event announcement. To continue the previous example, the set of delivery signal flags generated for `EventA` might be `Delvr_EventA_1` and `Delvr_EventA_2`.
– Method invocation: method invocation is implemented via binary *invocation signals* and (optionally) *invocation arguments* derived from event arguments. If `EventA` is bound to method `MethodD` in `ComponentE`, and `MethodD` takes an argument, then the corresponding generated state variables will be a binary flag `Invoke_D` and a {1, 2}-valued argument variable `Invoke_D_Arg`. There is a translation mechanism (a *delivery adapter*) that sets up the correspondence between event delivery notification and method invocation variables (i.e., makes sure that `Invoke_D` is 'on' whenever either `Delvr_EventA_1` or `Delvr_EventA_2` is 'on', and assigns the appropriate value to `Invoke_D_Arg`).
– Invocation acknowledgment: this is implemented simply by shared binary state variables that are written by component methods and read by invocation queues described above.

The second aspect of the run-time infrastructure is the dispatcher/delivery policy pair that is positioned between event announcement acceptance and event delivery notification (see Figure 3). This portion of the infrastructure is responsible for immediate announcement acceptance and does not keep track of whether the event delivery notification has been properly processed (the invocation queues take care of that).

The dispatcher state machine performs the role of reading the announcement signals, immediately updating the data structure that reflects the set of pending events (the active events history), and assigning delivery signals as directed by the delivery policy. The delivery policy executes by continuously reading the pending event information from the dispatcher and generating another data structure (the delivery directive) that marks events to be delivered during a particular cycle.

The data structures maintained and shared by dispatcher and delivery policy may vary in complexity, depending on how much information about the set of pending events is required by the delivery policy. The most general mechanism maintains all of the attributes of pending events and also keeps track of the temporal announcement information for each event. We have found that this mechanism is most easily modeled with the aid of distinct announcement signals as described above.

The modularization and separation of delivery and synchronization models make it easy to substitute different dispatchers and delivery policies to examine their effects on the system behavior without changing any other portions of the state model; in many cases interesting behavior can be explored by replacing just those modules. For example, one might investigate the effects of moving from a single address space implementation using synchronous delivery and a single thread of control, to a distributed version of the system by changing the delivery policy to asynchronous, and allowing each component to be a separate thread..

5 Example

To illustrate the approach we use a simple example, introduced in [16] and elaborated in [7]. The target system includes two components: a set and a counter. Elements may be added to or removed from the set. The counter may be incremented or decremented. Elements are added/removed from the set upon the receipt of insert(v) or delete(v) events from the environment; when the insertion or deletion is successful (insertion fails if element is in the set already; deletion fails if element is not in the set), the set announces update(ins) or update(del) events which are dispatched to invoke corresponding increment and decrement events in the counter. The goal is to determine if the system preserves the "invariant" that the counter is equal to the size of the set.

Figure 4 shows the state model structure and the shared state variables used for communication. Note that in this case no announcement/delivery adapters are required for Delete and Insert events because they have no arguments and can simply be represented by single binary signals. An appendix to this paper contains additional details of the SMV model, and in particular shows two different delivery policies.

To check the model for properties of interest we use a model checker. In the case of Cadence SMV we can exploit a feature that allows us to assert certain properties of the model, and then use those assumptions in verifying other properties. For example, to avoid buffer overflow in the finite event queues, we assert that the environment will not generate an overflow. We also assert that the environment will eventually stop, in order to verify that even though Counter may get out of sync with the Set contents, it will eventually catch up).[2]

```
ConsiderateEnvironment :
  assert (G (~disp.evtBuffOverflow & ~updateInvQueue.error));
```

[2] In this example, properties are expressed in the logical notation of LTL, using "F" to represent "eventually," "G" to represent "globally," and tilda to represent "not."

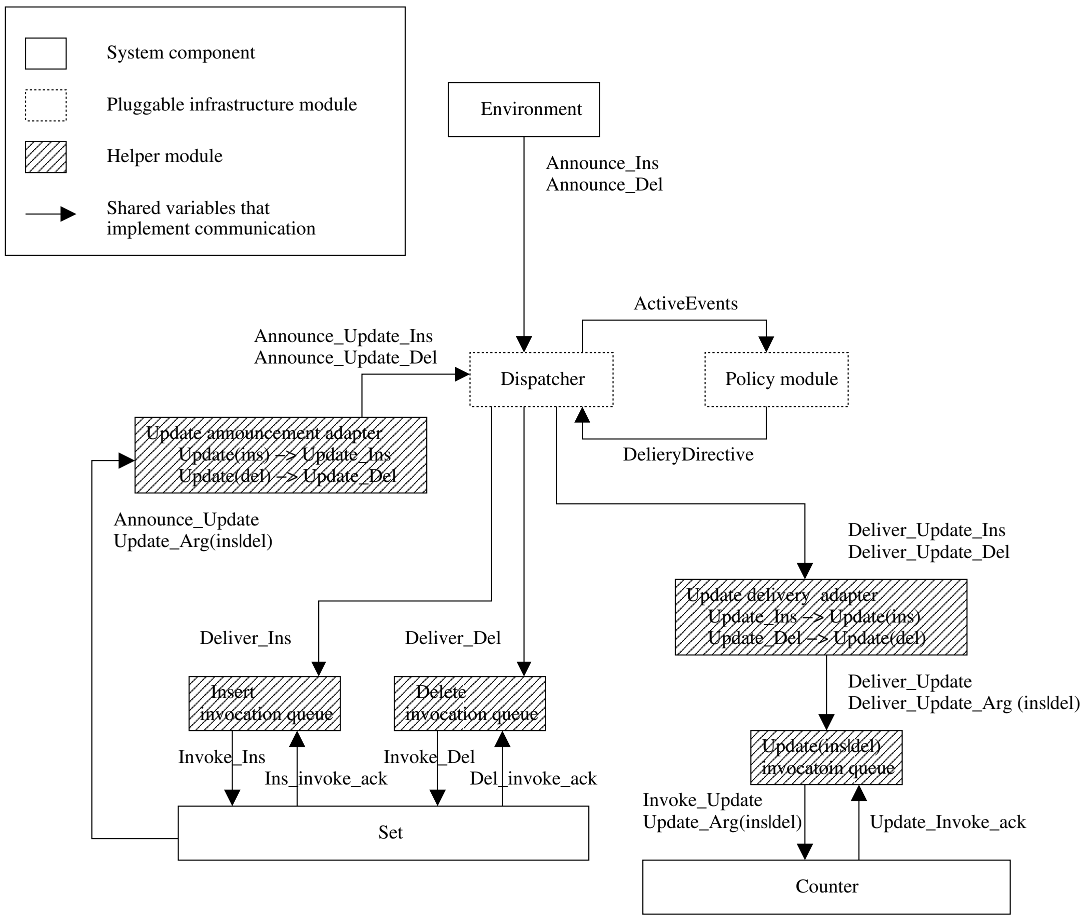

Fig. 4. Generated Set-and-Counter Model

```
StoppingEnvironment :
  assert(F G (~announceUpdt));
```

The fact that the environment does eventually stop is used to verify that the counter indeed eventually catches up with the set contents:

```
CounterCatchesUp :
  assert(G F (set.setSize = counter.count));

using StoppingEnvironment, ConsiderateEnvironment
prove CounterCatchesUp;
```

Other interesting properties of the model are obtained via counter examples. For example the following property demonstrates that the counter can actually be negative. (This can happen if, by the whim of the delivery policy, insert events headed for the counter are held up while delete events are allowed through for a number of cycles.)

```
CounterNeverNegative :
  assert(G (~counter.count = -1));
```

The generated SMV code is about 184 lines (excluding comments). Of this, 147 lines (about 80%) is automatically-generated run-time structure, produced by our tool. While this percentage would go down for larger examples, nonetheless it represents a significant reduction in modeling effort, and one that is nontrivial to get right.

6 A More Interesting Example

To investigate the applicability of the approach to more realistic situations, we applied it to the problem of reasoning about dynamic resource allocation for tasks in a mobile computing environment.

The system in question contains components of three types: resources, tasks, and schedulers. Task components represent units of meaningful computation for mobile device users, such as voice call, ftp download, and mp3 play. Resource components represent assignable system resources, such as battery, CPU, memory, and network bandwidth. They are targets for competition between task components. Scheduler components lie between these two groups and arbitrate competition between tasks for available resources.

Communication between components uses publish-subscribe. As illustrated in Figure 5, task components are triggered by environment events requesting their execution. These tasks then announce resource request events, and wait for either grant or reject events. At the same time, resource components announce their resource levels and prices. Resources can also receive events notifying them that some of their capacity has been consumed or replenished. Scheduler components mediate resource assignment, listening for resource requests and resource status, then making decisions about which tasks are to be granted which resources, and how much they are granted.

The primary goal of the modeling exercise was to verify that the loosely coupled behavior of publish-subscribe was consistent with the need for a scheduler to be able to accurately assign resources. To test this we initially restricted the system to use a single resource, a single task, and a single scheduler. We also abstracted away the application-specific behavior of the tasks, modeling their behavior simply in terms of units of work yet to be accomplished. A representative sample of the kinds of properties that can be checked include:

1. When started, a task component will eventually finish its work.
   `assert(G ((TASK.Work > 0) -> F (TASK.Work = 0)));`
2. The scheduler will always eventually have an accurate representation of resource levels.
   `assert(G F (RESOURCE.Level = SCHEDULER.Level));`

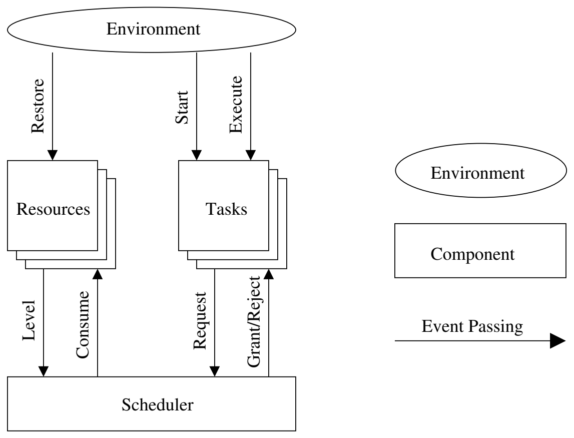

Fig. 5. Mobile Resource Allocation System

3. The scheduler must be able to know when resources are exhausted.
   `assert(G((RESOURCE.Level = 0) -> F (SCHEDULER.Level = 0)));`

As a result of verification, the second property turned out to be false. Because there is no assumption of a stopping environment, similar to the one used in set-counter example, it is possible for the level of a resource component to continuously change in response to incoming consume and restore events, leading to a situation in which the scheduler will never catch up with resource components. The first and third properties proved to hold.

7 Discussion and Future Work

We have outlined an approach to model checking pub-sub systems that factors out and parameterizes the run-time mechanisms that support component integration. We then illustrated the idea for two examples, showing how interesting properties could be checked, and the fact that the tool automatically generates a significant body of model checking code.

The more general idea behind the approach is to exploit regularities and known variabilities in architectural structure so that common checking infrastructure can be built once and then used by anyone designing systems in the corresponding architectural style. Not only does this approach reduce the cost of building the models to be checked, but it greatly simplifies the technology since the modeler need only worry about the application-specific aspects of the problem (in this case, the behavior of the components). Naturally, a similar approach could be applied to other architectural styles, with similar gains in cost reduction and ease of use — promising avenue for future research.

One of the drawbacks to such an approach is that because the models are generic and machine-generated, they may be less efficient than hand-crafted models. By "efficient," we mean that they can be represented by a more compact state representation. In some cases, given today's model checking technology, this may be the difference between a tractable and intractable model. Understanding when such fine-tuning and extra effort is required represents another important area of future work.

Additionally, the approach is partially hampered by the need to use temporal logic to represent properties of interest, in many cases requiring the modeler to have a detailed understanding of the generated structures in order to specify those properties. An attractive avenue of research would be to develop a set of common pub-sub specific properties that could be easily specified and automatically checked. We are currently working on this, using an extension of the Bandera Specification Language [6]. We are also developing a component specification language that provides a closer link to code than the current SMV-oriented XML-based input language.

Finally, for a complete solution to the problems of checking pub-sub systems, there are several other complementary advances that must be made. Given the translation step, there must be mechanisms to relate counterexamples back to the input model. A hard problem in general, it is even more difficult for us, since the details of the run-time mechanism are hidden from the user of the tool, and so explaining how a series of event announcements leads to an anomalous result is problematic. Another place requiring additional attention is the general problem of developing component models from source code, ensuring that the models are adequate approximations of the actual implementation.

Acknowledgements

This paper is an expanded version of a previously-published, short workshop paper [9]. The research grew out of work on formal reasoning about pub-sub systems, conducted jointly with Juergen Dingel, David Notkin, and Somesh Jha. Goran Momiroski developed an early prototype of the translation tool. This paper also has benefited from discussions with Bill Griswold, Matthew Dwyer, Kevin Sullivan, and Mary Shaw. Partial funding for this research was provided by NSF (CCR-9633532), DARPA (F30602-97-2-0031 and F30602-96-1-0299) and the Army Research Office (ARO) (DAAD19-01-1-04). Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsor institution or the US Government, NASA, or the Department of Defense.

References

1. G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 18(5), pages 9–20. ACM Press, December 1993.

2. R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.
3. D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, October 1996.
4. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving expressiveness and scalability in an internet-scale event notification service. *Proc. 19th ACM Symposium on Principles of Distributed Computing*, July 2000.
5. E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 24(4), December 1996.
6. J. Corbett, M. Dwyer, and J. Hatcliff. Bandera : Extracting finite-state models from java source code. *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
7. J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about Implicit Invocation. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Buena Vista, Florida, November 1998. ACM.
8. J. Dingel, D. Garlan, S. Jha, and D. Notkin. Towards a formal treatment of implicit invocation. *Formal Aspects of Computing*, 10:193–213, 1998.
9. D. Garlan and S. Khersonsky. Model checking implicit invocation systems. In *Proceedings of the 10th International Workshop on Software Specification and Design*, San Diego, CA, November 2000.
10. D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag, LNCS 551.
11. K. Havelund and T. Pressburger. Model checking java programs using Java Pathfinder. *International Journal on Software Tools for Technology Transfer, STTT*, 2(4), April 2000.
12. J. Magee and J. Kramer. *Concurrency: state models & JAVA programs.* John Wiley & Son, April 1999.
13. K. McMillan. *Cadence SMV.* http://www-cad.eecs.berkeley.edu/ kenmcmil/smv/.
14. Microsoft. Slam. http://research.microsoft.com/slam.
15. D. Notkin, D. Garlan, W. G. Griswold, and K. Sullivan. Adding implicit invocation to languages: Three approaches. In S. Nishio and A. Yonezawa, editors, *Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software*, pages 489–510. Springer-Verlag LNCS, no. 742, November 1993.
16. K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. In *Proceedings of SIGSOFT '90: Fourth Symposium on Software Development Environments*, Irvine, December 1990.
17. K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.
18. Sun Microsystems. JavaBeans. http://java.sun.com/products/javabeans.
19. R. N. Taylor, N. Medvidovic, K. M. Anderson, J. E. James Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996.
20. TIBCO The Power of Now. Tibco hawk. http://www.tibco.com/solutions/.

APPENDIX: Set-counter Details

In the Set-and-Counter model, information about pending events is kept in the SMV data structure shown in Figure 6. Note that the events announced in the system are (a) update with an 'insert' or 'delete' argument, (b) insert, and (c) delete.

The dispatcher module (not shown here) maintains the above event counts by correctly updating them during every execution cycle. The dispatcher also receives directives from the delivery policy and generates delivery signals.

```
/*****************************************************************/
/* ActiveEventsHistory holds information about pending events.     */
/* For each event count number of steps ago that it was announced, */
/* and if events of this kind are still pending.                   */
/*****************************************************************/
typedef ActiveEventsHistory struct {
 Update_Ins_Pending: boolean;
 Update_Ins_Recent : array 1..EVENT_Q_SIZE+1 of boolean;
 Update_Ins_Oldest : 0..EVENT_Q_SIZE+1;
 Update_Del_Pending: boolean;
 Update_Del_Recent : array 1..EVENT_Q_SIZE+1 of boolean;
 Update_Del_Oldest : 0..EVENT_Q_SIZE+1;
 Insert_Pending: boolean;
 Insert_Recent : array 1..EVENT_Q_SIZE+1 of boolean;
 Insert_Oldest : 0..EVENT_Q_SIZE+1;
 Delete_Pending: 0..boolean;
 Delete_Recent : array 1..EVENT_Q_SIZE+1 of boolean;
 Delete_Oldest : 0..EVENT_Q_SIZE + 1;
}
```

Fig. 6. Pending Events

Figure 7 shows an example of a simple delivery policy that instructs the dispatcher to deliver all events immediately. With this delivery policy, the Counter component always remains in sync with the Set. (In fact, the counterNeverNegative property described before is true.)

A more interesting delivery policy, while making use of the same communication infrastructure, may decide to randomly delay the events as long as the event buffers do not overflow. This is shown in Figure 8. With this delivery policy, the counterNeverNegative property is false and model checker generates a counterexample illustrating how the Counter may get out of sync with the Set.

```
module EventDeliveryPolicy(activeEvents, delivery) {
    input activeEvents  : ActiveEventsHistory;
    output delivery     : EventsDeliveryDirective;

    /* Simple policy: just deliver events as they arrive */
    delivery.Deliver_Update_Ins := activeEvents.Update_Ins_Pending;
    delivery.Deliver_Update_Del := activeEvents.Update_Pending;
    delivery.Deliver_Insert := activeEvents.Insert_Pending;
    delivery.Deliver_Delete := activeEvents.Delete_Pending;
}
```

Fig. 7. Event Delivery Policy 1

```
module EventDeliveryPolicy(activeEvents, delivery) {
    input activeEvents: ActiveEventsHistory;
    output delivery: EventsDeliveryDirective;
    /* Deliver events with random delays, as long as events are not
       delayed forever */
    /* If no pending events, or pending events are old ==>> -1
       (i.e., deliver oldest pending event if any)
       Else randomly decide to deliver oldest pending event (if any)
       or stall delivery */
    delivery.Deliver_Update_Ins :=
        (activeEvents.Update_Ins_Oldest = 0 |
         activeEvents.Update_Ins_Oldest = EVENT_Q_SIZE + 1 ? -1 : { 0, -1 });
    delivery.Deliver_Update_Del :=
        (activeEvents.Update_Del_Oldest = 0 |
         activeEvents.Update_Del_Oldest = EVENT_Q_SIZE + 1 ? -1 : { 0, -1 });
    delivery.Deliver_Insert :=
        (activeEvents.Insert_Oldest = 0 |
         activeEvents.Insert_Oldest = EVENT_Q_SIZE + 1 ? -1 : { 0, -1 });
    delivery.Deliver_Delete :=
        (activeEvents.Delete_Oldest = 0 |
         activeEvents.Delete_Oldest = EVENT_Q_SIZE + 1 ? -1 : { 0, -1 });
}
```

Fig. 8. Event Delivery Policy 2