Proc. of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), pages 78-87, Helsinki, Finland, Sept. 2003

Evaluating and Improving the Automatic Analysis of Implicit Invocation Systems

Jeremy S. Bradbury
Applied Formal Methods Group
School of Computing, Queen's University
Kingston, Ontario, Canada
bradbury@cs.queensu.ca

Juergen Dingel
Applied Formal Methods Group
School of Computing, Queen's University
Kingston, Ontario, Canada
dingel@cs.queensu.ca

ABSTRACT

Model checking and other finite-state analysis techniques have been very successful when used with hardware systems and less successful with software systems. It is especially difficult to analyze software systems developed with the implicit invocation architectural style because the loose coupling of their components increases the size of the finite state model. In this paper we extend an existing approach to model checking implicit invocation to allow for the modeling of larger and more realistic systems. Our focus will be on improving the representation of events, event delivery policies and event-method bindings. We also evaluate our technique on two non-trivial examples. In one of our examples, we will show how with iterative analysis a system parameter can be chosen to meet the appropriate system requirements.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification —*formal methods, model checking*; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Verification

Keywords

architectural styles, implicit invocation, formal methods, linear temporal logic, model checking, publish subscribe, XML

1. INTRODUCTION

Model checking and other finite-state analysis techniques have been very successful when used with hardware systems and less successful with software systems [1]. There are two main reasons why model checking has not been successful when applied to software: a semantic artifact gap and the state explosion problem. First, it is hard to construct a finite-state model of a software system because unlike with hardware, there is a larger gap between the artifacts produced by software developers and the artifacts that are accepted by model checkers. The gap between artifacts typically has to be bridged by humans with little tool support. Thus, there is a possibility for spurious analysis results when the finite-state model does not correspond to the software system. Second, the state spaces of the models constructed for software systems are often infinite or extremely large due to the use of variables ranging over infinite or large domains. The problem is exacerbated by the fact that the state space grows exponentially with the number of parallel processes in the system.

We extend an approach to model checking that was originally developed by Garlan and Khersonsky in [4] and later expanded upon in [5]. This approach focuses on the automatic analysis of implicit invocation (II) or publish-subscribe systems. II systems are event-based and have two primitives. First, components can announce or publish events. Second, other components can listen or subscribe to events that are announced. A centralized message server or dispatcher is used to keep track of which components subscribe to which events. The dispatcher will receive published events and use them to invoke the appropriate subscriber methods. II systems are challenging to model check because the loose coupling of their components increases the size of the finite state model.

The authors of [4] and [5] have made contributions to alleviating both the semantic artifact gap and state explosion problems in the context of II systems. On the one hand, the approach bridges the gap between artifacts by automating parts of the process of generating the finite state model for a software system. Removing the human element in model generation decreases the occurrence of errors due to artifact conversion. On the other hand, optimization techniques such as abstraction are applied to decrease the state space of the software models. Together the automation and optimization allow us to explore ways to make model checking a more effective and efficient method of analysis for non-trivial II software systems.

One limitation of the original approach is the types of II systems that can be represented. Thus, we propose extensions to the approach to allow for the modeling of larger and more realistic systems. Our focus will be on improving the representation of mechanisms for event delivery. Specifically, we improve the representation of events, event delivery

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.

policies and event-method bindings. These improvements allow us to more easily model events with data and to now model II systems that exhibit dynamic behavior in terms of when and how events are delivered (the delivery policy) and where events are delivered (the event-method bindings). In general, there are several reasons why dynamic software architectures are beneficial [8]. One, they reduce the cost and risk associated with architectural change since a system can be modified without being taken offline. Two, they can allow for increased customizations and extensions.

The application areas for dynamic systems are primarily public information systems with high availability and mission- or safety-critical systems. We evaluate our technique on one realistic example from each area. In our example of a safety-critical system we also show how with iterative analysis, a system parameter can be chosen to meet the appropriate system requirements. The use of iteration in model checking is standard practice. Our use of iteration is novel because we use iteration to make design decisions regarding a parameter of an II system, and we use repeated analysis to determine the impact of certain design decisions on the overall system behavior and its conformance to the given requirements.

We will first provide an overview of II in Section 2. The Garlan and Khersonsky approach and our extensions will be described in Section 3. This modeling approach is used to combat the semantic artifact gap. In Section 4 we proceed with the analysis of two non-trivial examples. In Section 5 we provide the optimization techniques used to control state explosion. Finally, we outline the conclusions of our research as well as discuss possible future work in Section 6.

2. II SYSTEMS

Low coupling is an important attribute of systems. In an II system, components never know the identity of other components that are sending or receiving events. In fact, components do not even know if other components exist to receive their events. All information about which components are interested in which events is contained in the dispatcher. Since none of this information is stored in the components, it is easy to see how coupling is decreased and, for instance, modularity and reuse increased. We will now explain in detail the structure and information exchange mechanisms of II (shown in Figure 1).

2.1 II System Structure

An II system is characterized by the following six parameters:

- *Components*: objects that encapsulate data and functionality. Components are usually accessible through well-defined interfaces.
- *Events*: the primary method of communication between components.
- *Event-Method Bindings*: the correspondence between events that are announced and the methods in a component instance that are invoked in response to these announcements [10].
- *Event Delivery Policy*: a set of rules that define the announcement and delivery of events.

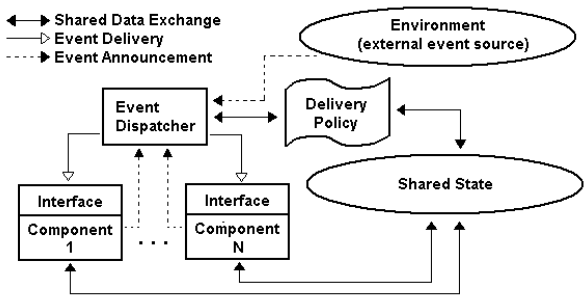

Figure 1: II System Structure

- *Shared State*: an additional method of communication between the elements of a system. Shared variables can help to alleviate possible performance problems. However, a major drawback is that shared variables break the loose coupling of components by creating dependencies between components that share variables.
- *Concurrency Model*: determines for instance if the system has a single thread of control or separate threads of control.

In addition to the above system parameters an environment model is also required. The environment represents the external elements that affect the system and thus is not included as a system parameter.

2.2 Information Exchange

Three main types of information exchange occur between different II system parameters and the environment:

1. *Event Announcement*: The components in the system can announce an event provided that the type of event is one of the event types specified for the system. The environment can also act as an external event source. Data can be exchanged between the components through events with parameters.
2. *Event Delivery*: Upon receiving events from the components, the event dispatcher sends the events out to all subscriber components that have requested to receive a particular event type. Event delivery involves consulting the delivery policy.
3. *Shared Data Exchange*: Data can be exchanged between the components through shared variables. Data access can also occur when the event dispatcher consults with the delivery policy in order to determine how events should be delivered. Finally, data exchange can exist between the delivery policy and the shared state. More precisely, the policy can be dependent on the values of shared variables. However, for simplicity we do not consider this type of exchange.

3. MODELING II SYSTEMS

Below we will outline the Garlan and Khersonsky approach to formally modeling II systems. We will then outline our modifications and enhancements to the original technique.

3.1 Existing Modeling Approach

The existing approach represents the parameters of an II system as follows:

- *Components*: a set of components are represented using a simple imperative programming language. Components consist of local variables, accessible global variables, announced events and methods. Methods allow for conditionals, assignment statements, and the announcement of events.
- *Events*: a list of identifiers.
- *Event-Method Bindings*: a set of pairs (e_i, m_i) where event e_i invokes method m_i.
- *Event Delivery Policy*: one of the following:
  - *Asynchronous*: immediate return from event announcement.
  - *Synchronous*: return from announcement after event is completely processed.
  - *Immediate*: immediate invocation of subscriber methods.
  - *Random Delay*: buffer events before announcement to subscriber components.
- *Shared State*: a set of pairs (id_i, t_i) where a shared variable represented as identifier id_i has type t_i.
- *Concurrency Model*: one of the following:
  - Single thread of control for all components.
  - Separate threads of control.
    - Single thread for each component.
    - Multiple threads for each component with concurrent invocation of different methods.
    - Multiple threads for each component with concurrent invocation of any method.

In addition a behavioral specification of the environment model has to be included. The environment is necessary because it can announce events that are used to initiate the announcement of events by components in an II model. Environment events are represented as a list of identifiers. Each identifer is also associated with a boolean pair of behavior attributes: `always-announced` and `stops-eventually`. These properties are necessary for event announcement because the environment is not modeled in the system. Additional technical details regarding the modeling of II systems and the corresponding environment model can be found in [4].

The above information regarding an II system and the environment is specified in XML. Once the XML input has been created, an automated tool converts the information into a set of state machines. This set of state machines can then be checked using a commercial model checker, such as Cadence SMV [7]. The model checking process relies on building a finite state model of a system and checking that a desired property holds in the model. The check is performed as an exhaustive state space search, which is guaranteed to terminate since the model is required to be finite.

3.2 Modifications and Enhancements

Upon reviewing the existing approach it became obvious that in order to model significant II systems several extensions would have to be made. We extended the approach by improving the event communication. Specifically, we modified the following II parameters: the events, the event delivery policy, and the event-method bindings. Indirectly, this work caused minor alterations to the modeling of components as well.

3.2.1 Enhanced Events

The events parameter was extended to allow for the inclusion of data. More precisely, an event can now have an arbitrary number of parameters. The inclusion of data directly into events resulted in major alterations to the model of the event dispatcher and the component event queues. Previously, shared variables had to be used for the exchange of event data parameters. The main disadvantage of the previous approach, is that it decouples the event-method bindings which govern the announcement flags from the global variable correspondences which govern shared variable access. That is, in order to change a binding we previously had to change the binding itself and individually change the global variable correspondence for each data parameter.

3.2.2 Dynamic Delivery Policies

The delivery policy parameter was also extended from the previous fixed representation of policies. In our approach, delivery policies are user-defined and more expressive. Propositional logic is used to describe these delivery policies. The *Immediate* and *Random Delay* policies, used by Garlan and Khersonsky, are incorporated as delivery rules.

We use propositional logic for representing delivery policies because it is simple, intuitive, and sufficient for representing the delivery policy examples we consider. Propositional logic formulas are built from a set of atomic propositions and using the following connectives: negation, conjunction, disjunction, implication, and equivalence.

It is important to note that not every propositional logic formula represents a delivery policy. The policies we are considering are always of the form:

$((guard_1) \Rightarrow (deliveryExpr_1)),$
$((guard_2) \Rightarrow (deliveryExpr_2)),$
...
$((guard_n) \Rightarrow (deliveryExpr_n))$

More precisely, the representation of a policy is a non-empty list of $(guard_i) \Rightarrow (deliveryExpr_i)$ formulas where $i \in \mathbb{Z}^+$. In an II system, the dispatcher executes the delivery policy by determining the smallest value of i such that $guard_i$ is *true* and then making appropriate changes in the state to make the corresponding delivery expression, $deliveryExpr_i$, *true*.

The guards in a delivery policy must be exhaustive. The guards have to be exhaustive otherwise a situation could arise for which no delivery expression exists. The guards do not have to be disjoint because we choose only the first guard that is *true*. The guards within a delivery policy can overlap but only one delivery expression will be used. Additionally, consistency is required within the delivery expression and with the corresponding guard, that is, $guard_i \wedge deliveryExpr_i$ must be satisfiable.

The guards in a delivery policy consist of event variables and relational expressions. Boolean event variables are used to represent the presence of an event. If an event variable is *true* then that event is waiting to be delivered by the dispatcher and a $false$ means that the event is not waiting to be delivered. Furthermore, relational expressions consisting of integer event variables are used as a form of event filters. The relational expressions are acquired from comparing two integers using one of the following relational operators $=$, $<$, $>$, $\leq$, or $\geq$. For example, we can use relational expressions to compare the number of pending events.

The delivery expressions consists of delivery variables. If a delivery variable is *true* then the corresponding event will be delivered in the next state, otherwise the corresponding event will be delivered at some later unspecified time. To indicate that a variable value within a delivery expression is in the next state we use primed variables. Unprimed variables are used to indicate in the current state.

Suppose, for instance, we have a system with event types: x, y, z and we want the delivery policy for this system to have the following two sub-policies. One, an immediate policy where events are delivered immediately in a state where the `envImmediate` event is announced. In this policy multiple events can be delivered in a single state. Two, a priority-based policy that uses a priority queue based on event type with random delay is used when the environment event, `envImmediate`, is not announced. This policy assumes no two events have equivalent priorities and that at most one event will be delivered in each state of the model. The priority values of the event types are $x = 1$, $y = 2$, $z = 3$.

The above dynamic delivery policy can be represented in propositional logic as follows:
$((pending\_envImmediate > 0)$
$\Rightarrow deliver\_x' \wedge deliver\_y' \wedge deliver\_z')),$
$((\neg(pending\_envImmediate > 0) \wedge (pending\_x > 0))$
$\Rightarrow ((deliver\_x' \vee \neg deliver\_x') \wedge \neg deliver\_y' \wedge \neg deliver\_z')),$
$((\neg(pending\_envImmediate > 0) \wedge (pending\_y > 0))$
$\Rightarrow (\neg deliver\_x' \wedge (deliver\_y' \vee \neg deliver\_y') \wedge \neg deliver\_z')),$
$((\neg(pending\_envImmediate > 0) \wedge (pending\_z > 0))$
$\Rightarrow ((\neg deliver\_x') \wedge \neg deliver\_y' \wedge (deliver\_z' \vee \neg deliver\_z'))$

In the above example we show how our approach to delivery policy representation supports dynamic delivery policy selection while a system is running. Specifically, we allow multiple policies to be specified prior to runtime, each of which can be used depending on the state of the overall model.

3.2.3 Dynamic Event-Method Bindings

Dynamic event-method bindings are created by adding a boolean attribute to each pair, (e_i, m_i), representing an event-method binding. A boolean attribute specifies whether the corresponding binding is active or inactive. Thus, we can have a set of static bindings that can be dynamically activated and deactivated at run-time. To change the status of a binding the component whose method is invoked can directly set the status of the boolean attribute. The addition of parameters to events, discussed earlier, makes the extension to dynamic event-method bindings possible. We are now able to allow for support the four main types of dynamic change in an II system [12]:

1. *Component Addition:* a component receiving no events activates bindings to receive events. If a component receives no events it will also announce no events since method invocation occurs only by the delivery of an event.

2. *Component Removal:* a component deactivates the bindings for all of the events to which it subscribes.

3. *Connector Addition:* a component activates the binding for an event.

4. *Connector Removal:* a component deactivates the binding for an event.

4. ANALYZING II SYSTEMS

In general, the II architectural style is considered difficult to reason about and test. For instance, there is no specific way to determine the effects of certain event announcements or of policy changes on the overall system. Moreover, it may be difficult to determine the effects of adding or removing components [2], [3]. In our research, we use the exhaustive state space exploration that is part of model checking to answer these kinds of questions.

The main reason why II systems are hard to analyze using model checking is the size of the state machine corresponding to a given II system. For a specific system the number of possible system executions maybe extremely large because of the possibly large degree of parallelism and the fact that system executions are not only affected by the events being sent and received, but also by the timing of these events. Timing has to be taken into consideration because when an event is announced and eventually received will affect what the listener component does. In fact, the timing of events is just as important as the type of events. Some of the factors that affect timing are delays in the announcement of events, delays in delivery of events, ordering of events on delivery, and events that are simply not received at all.

The original Garlan and Khersonsky approach was evaluated in [4], but only on a small example. It was later evaluated on a more significant example in [5]. In an effort to demonstrate our extended approach on larger, more realistic systems, we conducted an evaluation on two significant examples. We modeled the Active Badge Location System (ABLS) [11], an electronic tagging alternative to pagers, and the Unmanned Vehicle Control System (UVCS) [9], designed for a major European port. II models were generated for each system based on the requirements. For each model we verified a set of properties expressed in Linear Temporal Logic (LTL). The LTL operators used in the expression of properties are: X ϕ (in the next state ϕ holds), G ϕ (ϕ holds globally), F ϕ (ϕ holds eventually), ϕ_1 U ϕ_2 (ϕ_1 holds at least until ϕ_2 does).

The focus of the examples is on evaluating the modeling of II systems that utilize our event and delivery policy extensions. Due to space restrictions the following examples due not make use of dynamic event-method bindings.

4.1 ABLS Example

The ABLS system can be thought of as an example of a public information system with high availability. Specifically, it is an electronic tagging system for locating people in an office setting. Active Badges are considered an innovative solution to pagers because they allow for the location of an Active Badge wearer to be transmitted.

4.1.1 ABLS Model

An II model of the ABLS will consist of three types of components: request workstation(s), a main workstation, and sensor workstation(s). The Active Badges, carried by all people in the system, communicate directly with the sensor workstations via sensors and are thus not considered part of the II model.

The request workstations randomly issue the following commands: *Find* an Active Badge's location, determine which other badges an Active Badge is *With*, *Look* at one location and return all of the badges that are there, *Notify* an Active Badge holder via an audible or vibration notification, and obtain a *History* location report of an Active Badge. The badge and location information used in the commands is chosen at random within a request workstation. In reality, the request workstation could be any sensor workstation. However, for simplicity we have chosen to model it separate from sensor workstations. This is an acceptable assumption since it does not affect the integrity of the system.

The main workstation has three primary responsibilities: information retrieval, information storage, and command execution. Information on the location of Active Badges is received by polling the sensor workstations. Information received from sensor workstations on the current and previous locations of Active Badges is stored in a database. Commands from the request workstations are received and fulfilled using information in the database.

The sensor workstations are responsible for sending polling results to the main workstation whenever a polling event is received. The information regarding the Active Badge locations is determined randomly. Since the addition of components in an II system causes exponential growth in the size of the model, we have chosen to model the sensor workstations in one component. This optimization preserves correctness and is discussed in Section 5. A generalized representation of the ABLS II system that does not contain this optimization is shown in Figure 2.

Event types for communication between the main workstation component and the sensor workstations component are needed in the context of polling:

- *Poll Event:* The `Poll` event is invoked by the announcement of an environment event, `EnvPoll`, which is guaranteed to always be announced and guaranteed to always stop being announced. The `Poll` event is delivered from the master workstation to the component representing the sensor workstations. It has no event data and its delivery is used to invoke the announcement of the `PollResult` event.

- *PollResult Event:* This event is announced by the sensor workstations component and is delivered by the event dispatcher to the master workstation. A `PollResult` event is assumed to contain information regarding the location identification number of the sensor workstation sending the event and the Active Badges that are "close" to the specified sensor workstation. The term "close" means that the Active Badge is communicating directly via a sensor with the sensor workstation. When a `PollResult` is delivered to the master workstation it causes the database in the component to be updated. It does not invoke the announcement of any additional events.

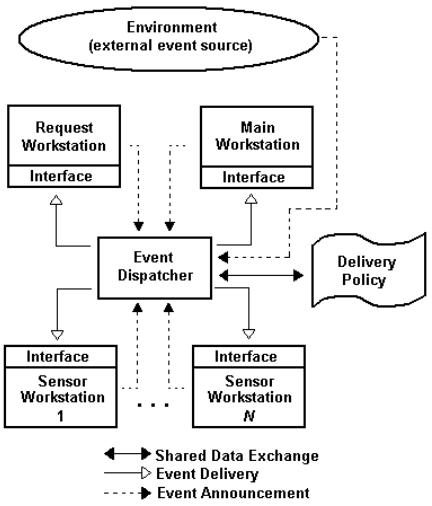

Figure 2: System Structure for ABLS System

Event types are also needed for the execution of the commands such as *Find* and *Look*. This communication takes place between a request workstation and the main workstation. Below we list the event types required for the *Find* command. Similar events exist for the *With*, *Look*, *Notify*, and *History* commands.

- *Find Event:* A `Find` event is invoked by an environment announced event, `EnvFind`. The `Find` event is sent from a request workstation to the main workstation. This request command contains only one data item, the name of the holder associated with the Active Badge being located.

- *FindResult Event:* A `FindResult` event is invoked via the delivery of a `Find` event. The master workstation determines the last known location in the database of the name given in the `Find` event. This name and the corresponding location are returned in the `FindResult` event.

The concurrency model for the ABLS system is a single thread of control for all components allowing for concurrent execution of methods in different threads. We do not discuss the shared state parameter of an II system in the context of this model because no shared variables are used in the system. The final II parameter, the delivery policy will be discussed in Section 4.1.2 where we will consider two possible policies. A simplified example of an ABLS II system is given in Figure 3. The example is not complete and is provided primarily to show the XML input language.

4.1.2 ABLS Analysis

The properties that we will be verifying for the ABLS system are as follows:

- *Poll Event Announcement: PollEventAlwaysAnnounced* is a property that holds if eventually in all future states a `Poll` event will be announced, that is, if the announcement flag will eventually be set to 1.

```
<event-system name = "ABLS">
 <env-event name = "EnvPoll"/>
 <event name = "Poll"/>
 <event name = "PollResult">
  <event-data name="locationID" type="array 0..2 of array -1..1" />
 </event>

 <delivery-policy>
  <policy-statement>
   <delivery-statement rule="Immediate">Poll, PollResult
   </delivery-statement>
  </policy-statement>
 </delivery-policy>

 <component name = "MasterWorkstation">
  <event-announced name = "Poll"/>
  <local-var name = "database" type = "array 0..2 of array 0..2"/>
  <method name = "pollSensors">
   <statement>
    <announcement>Poll</announcement>
   </statement>
  </method>
  <method name = "receivePollResult">
   ...
   <statement>
    <assignment var-name = "database[0][0]">
    PollResult.locationID[0]</assignment>
   </statement>
   ...
  </method>
 </component>
 <component name="SensorWorkstations">
  ...
 </component>

 <component-instance component-name="MasterWorkstation"
                     instance-name="Master">
 </component-instance>
 <component-instance component-name="SensorWorkstations"
                     instance-name="Sensors">
 </component-instance>

 <event-binding event-name="EnvPoll">
  <method-binding instance-name="Master"
                  method-name="pollSensors"/>
 </event-binding>
 <event-binding event-name="Poll">
  ...
 </event-binding>
 <event-bindingevent-name="PollResult">
  ...
 </event-binding>

 <property name="PollEventAlwaysAnnounced">
    F G(Master.announcePoll.flag=1)
 </property>

</event-system>
```

Figure 3: XML Representation of Active Badge Location System

```
F G(Master.announce_Poll.flag = 1)
```

PollEventAlwaysEventuallyAnnounced is a liveness property that verifies if a `Poll` event will be announced infinitely often.

```
G F(Master.announce_Poll.flag = 1)
```

- *Badge Location: BadgeCanBeLocated* is a reachability property that verifies if a badge will eventually be located in the master workstation database. For example, `database[2][0]` refers to the location of *Badge 2*, zero polls ago. Note, a badge is not in the system if its location is -1. For optimization reasons we only verify that *Badge 2* will eventually be located in the database. This property could easily be repeated for all badges in the system.

```
F(Master.database[2][0] ~= -1)
```

OnceLocatedBadgeAlwaysLocated is a property that verifies that *Badge 2* will eventually be located in the master workstation database and then always be able to be located in the database. That is, once an Active Badge communicates with a sensor workstation it will always communicate with some sensor workstation in the system.

```
F G(Master.database[2][0] ~= -1)
```

- *Event Delivery Guarantees:* We want to determine the number of states required for the request workstation to receive a result from a command such as *Find*. The *FindCorrectnessImmediate* property checks if the value in the master workstation database is equivalent to the value received by the request workstation in a `FindResult` event. In the optimized version of the above general property, we verify the case where the `Find` event has requested the location of *Badge 2*.

```
G((((Master.state=sendFindResults)
& X(Master.database[2][0]=1))
->X(Request.invoke_getFindResult.location=1))
&(((Master.state=sendFindResults)
& X(Master.database[2][0]=0))
->X(Request.invoke_getFindResult.location=0))
&(((Master.state=sendFindResults)
& X(Master.database[2][0]=-1))
->X(Request.invoke_getFindResult.location=-1)))
```

Additionally, *FindCorrectnessInNextState* and *FindCorrectnessInTwoStates* test to see if the value currently in the master workstation database is equivalent to the value received by the request workstation in the next state and in two states.

- *Multiple Event Correctness:* The *FindLookCorrectness* property examines the situation where the master workstation component sends the results of a *Find* command in one state and then sends the results of a *Look* command in the next state. Our property involves a *Find* on *Badge 2* and a *Look* on *Location 1*. We verify two specific cases to determine correctness: *Badge 2* is at *Location 1* and no badge is at *Location 1*. In the case where *Badge 2* is at *Location 1* we verify that the `location` returned in the `FindResult` event will be 1 and `badgesAtLocation` in the `LookResult` event will return *true* indicated that at least one badge is in *Location 1*. In the case where no badge is at *Location 1* we verify that the `FindResult` event will not return 1 as the `location` for *Badge 2* and `badgesAtLocation` will be $false$.

```
G((((Master.state=sendFindResults)
& X(Master.state=sendLookResults)
&(Master.database[2][0]=1))
->(XX(Request.invoke_getFindResult.location=1)
& XXX(Request.invoke_getLookResult
                .badgesAtLocation)))
&(((Master.state=sendFindResults)
& X(Master.state=sendLookResults)
&(Master.database[2][0]~=1)
&(Master.database[1][0]~=1)
&(Master.database[0][0]~=1))
->(XX(Request.invoke_getFindResult.location~=1)
& XXX(Request.invoke_getLookResult
              .badgesAtLocation = 0))))
```

We will now present analysis results for the above properties using two different delivery policies. These policies demonstrate the granularity of our delivery policy representation. Previously, a delivery policy such as *Immediate* or *Random Delay* was applied at a system level meaning all events would be subject to the same policy. We apply these policies as delivery rules at an event level. Thus, different events can be delivered using different rules. Policy 1 involves using *Immediate* delivery for all events related to polling (`Poll`, `PollResult`) and *Random Delay* delivery for all events related to command requests (`Find`, `FindResult`, `Look`, `LookResult`. Policy 2 is similar except polling events are delivered randomly and command requests are delivered immediately. We are interested in these two policies to determine a priority for event types. In more general terms, the goal of this analysis is to determine which of the delivery policies, when used, will satisfy the system properties. We use iterative model checking to determine an appropriate delivery policy. Table 1 shows the results of our analysis using partial ABLS models.

When using Policy 1, we determine that the system will always keep polling eventually based on the results of the *PollEventAlwaysAnnounced* and *PollEventAlwaysEventuallyAnnounced* properties. Additionally, the *BadgeCanBeLocated* and the *OnceLocatedBadgeAlwaysLocated* properties show that we cannot guarantee the location of Active Badges in relation to the system. The reason that a badge's location can not be guaranteed is that it is possible in our model for a person to move to a location outside of the area covered by the sensors. For example, if the ABLS system is used in an office setting, Active Badge holders will no longer be in the system once they leave the physical location of the office. In terms of command event correctness, we can make no guarantees of the correctness of command results. This is because random delivery is used and events can take an arbitrarily long time to be delivered.

Property	**Results Policy 1**	**Results Policy 2**
PollEventAlwaysAnnounced	False	False
PollEventAlways-EventuallyAnnounced	True	True
BadgeCanBeLocated	False	False
OnceLocatedBadge-AlwaysLocated	False	False
FindCorrectnessImmediate	False	False
FindCorrectnessInNextState	False	True
FindCorrectnessInTwoStates	False	False
FindLookCorrectness	False	True

Table 1: Analysis Results of ABLS System

When using Policy 2, we achieve the same verification results regarding polling and the location of badges in relation to the system. The main benefit of Policy 2, is that we can now provide guarantees on command result correctness. First, the property *FindCorrectnessInNextState* is the only property that is verified as *true*. Thus, in our model and using the specified policy it takes one state transition for the request workstation to receive the *Find* command results from the dispatcher. In other words, we can guarantee that the location information received in a `FindResult` event is equal to the location in the master workstation database exactly one state ago. Second, for the particular case we examined in the *FindLookCorrectness* property we were able to verify the correctness of two command results received back-to-back.

In conclusion, Policy 2 will satisfy all of the specified properties and thus is the better choice for the ABLS system. However, we have not discussed guarantees on the correctness of polling results which may or may not be important. On the one hand, in a system where badge movement is frequent this should be included as a property to verify. On the other hand, if the ABLS system was used in a hospital situation where badges were associated with patient's beds then guarantees on the correctness of polling results would be less important. Analysis performance for the model checking of the ABLS system will be discussed in general terms in Section 4.3.

4.2 UVCS Example

The control system for unmanned vehicles developed by Stuurman and van Katwijk [9] is a safety-critical system that has "real-world" applications since it is designed for the Maasvlakte port system located to the west of Rotterdam.

4.2.1 UVCS Model

The architecture presented in [9] uses distributed processes communicating via the subscription model. The subscription model is similar to II except that processes subscribe to channels not events. We will now explain how we modeled the UVCS as an II system containing four types of components: vehicle, region, visualizer, and rules injector.

Vehicle components contain information including the identity, long-term goal, rule version, and position of the corresponding vehicle. A vehicle component is also responsible for transmitting the identity, position, rule version, and short-term plan to other interested components. For the purpose of keeping the model small we will not send additional information such as vehicle speed and size in the event data. As a result of this information exclusion, all vehicles are assumed to occupy only one "square" in a region grid and to move at a constant speed. Conversely, a vehicle component receives the same information from other vehicles in appropriate regions.

Region components collect data on all vehicles in their region. This information is compiled to create a summary of information.

The visualizer component gathers the summary information from all regions and displays it externally. The visualizer also has the ability to zoom in on a region process by receiving events from vehicle components.

To avoid collisions the system supports a set of traffic rules

to govern the movement of vehicles. The version of the rules being used can be updated by the rules injector component, which publishes new rules via event announcement. Vehicle components usually subscribe to these events and listen for new versions of the traffic rules. To avoid conflict in the event of a possible collision, vehicles announce the version of the rules they are using along with their short-term plan. In the case of a possible collision the traffic rules are used to decide which vehicle should wait. In the event that vehicles are using different rule versions a default rule is used instead.

In our II model there will be only one visualizer component, only one rules injector component, at least one region component, and possibly multiple vehicle components. Figure 4 is a generalized representation of the UVCS II system that shows the components of the system.

Events and event-method bindings in the UVCS system are listed below:

- *RegionInfo Event:* The `RegionInfo` event is announced by a region component and contains a summary of data collected on all vehicles in the publisher region. Additionally, a region identification number is also included as event data. The visualizer component is the only subscriber to the `RegionInfo` event.

- *VehicleInfo Event:* The `VehicleInfo` event is announced by a vehicle component. Other vehicles as well as the region that the vehicle is occupying subscribe to the `VehicleInfo` event.

- *RulesInfo Event:* The rules injector component announces the `RulesInfo` event. All vehicle components subscribe to this event and it is used to announce new traffic rules that are used in the case of possible collisions. The version number of the traffic rules is also announced with the `RulesInfo` event.

The concurrency model for the ABLS system is a single thread of control for all components allowing for concurrent execution of methods in different threads. No shared variables exist in the UVCS system. The final II parameter, the delivery policy parameter will be discussed in Section 4.2.2. We will consider several variations on delivery policies that have the implementation constraint that only *one* event can be delivered in any given state by the dispatcher.

4.2.2 UVCS Analysis

The properties that we will be verifying for the UVCS system are as follows:

- *ruleVersionCurrentandConsistentAlways:* This property verifies that within a 5 vehicle UVCS system that all of the rules versions used by the vehicles are equivalent to each other and to the `currVersion` variable in the `RulesInjector` component instance.

```
G ((Vehicle1.ruleVersion =
theRulesInjector.currVersion)
& (Vehicle2.ruleVersion =
theRulesInjector.currVersion)
...
&(Vehicle5.ruleVersion =
theRulesInjector.currVersion))
```

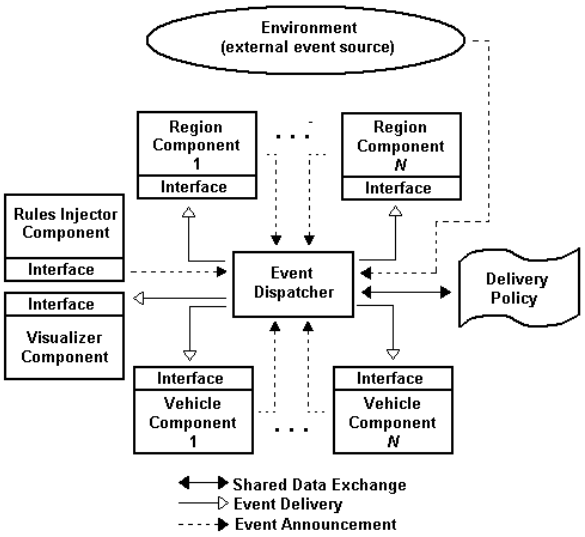

Figure 4: System Structure for UVCS System

- *vehicleMovementCorrectness:* This property is used to ensure that when a vehicle moves from one region to another the change is handled correctly. Each location in a region grid is denoted by an (x,y) pair with the pair (1,1) being in the bottom left hand corner of a region. Additionally, the vehicle is moved by modifying the direction and then applying the direction to the current location. The directions are 2 for East, 3 for North, 4 for West, and 5 for South. The vehicle movement does not follow a preset path and is generated based on a comparison between the current location and the destination location. Specifically, this property verifies that the current region and (x,y) location information is correct as the vehicle moves within and between regions.

- *collisonAvoidanceGuaranteed:* This safety property verifies that two vehicles moving in the same region will never crash. In this context, crash is defined as both vehicles occupying the same x and y position on the grid. Specifically, `Vehicle1` and `Vehicle2` will never both be in the same region with the same x and y position.

```
G (~(Vehicle1.currRegion = Vehicle2.currRegion)
 | ~(Vehicle1.xpos = Vehicle2.xpos)
 | ~(Vehicle1.ypos = Vehicle2.ypos))
```

- *longTermGoalAchieved:* This liveness property verifies that a vehicle will reach its long term goal.

```
F ((Vehicle1.currRegion = Vehicle1.destRegion)
 & (Vehicle1.xpos = Vehicle1.destxpos)
 & (Vehicle1.ypos = Vehicle1.destypos))
```

We will now present verification for the above properties using three delivery policies. The goal of this analysis is to

Property	Results Policy 1	Results Policy 2	Results Policy 3
ruleVersionCurrentandConsistentAlways	False	False	True
vehicleMovementCorrectness	True	True	True
collisonAvoidanceGuaranteed	False	True	False
longTermGoalAchieved	True	True	True

Table 2: Analysis Results of UVCS System

determine which of the delivery policies, when used, will satisfy the system properties. We use iterative model checking to determine an appropriate delivery policy.

If we know all of the parameters of the II architecture, excluding the delivery policy, and we know a set of global temporal logic properties, then we can use model checking to gain insight into the appropriate delivery policy to use. That is, we can hopefully determine an acceptable delivery policy for the UVCS system.

Policy 1 involves immediate delivery of events where we try and deliver the event type with the most events in queue first. In order to handle tie-breaking of the event queue size, we prioritize based on type. Policy 1 is described in our propositional notation as follows:

$(((pending\_VehicleInfo >= pending\_RulesInfo)$
$\wedge(pending\_VehicleInfo >= pending\_RegionInfo))$
$\Rightarrow (deliver\_VehicleInfo' \wedge \neg deliver\_RulesInfo'$
$\wedge \neg deliver\_RegionInfo')),$
$((pending\_RulesInfo) >= pending\_RegionInfo)$
$\Rightarrow (\neg deliver\_VehicleInfo' \wedge deliver\_RulesInfo'$
$\wedge \neg deliver\_RegionInfo')),$
$((true) \Rightarrow (\neg deliver\_VehicleInfo'$
$\wedge \neg deliver\_RulesInfo' \wedge deliver\_RegionInfo')),$

After adding this policy to the UVCS model and generating the model checking artifacts we verify using the Cadence SMV model checker. The results, summarized in Table 2, are promising, on our first try we have satisfied two of the four requirements. We still have to modify the delivery policy to satisfy collision avoidance and rule version consistency since our policy cannot guarantee the delivery of `VehicleInfo` and `RulesInfo` events in a sufficient time period.

In an effort to satisfy collision avoidance, we use our intuition and try an alternate delivery policy, Policy 2. This policy gives priority to `VehicleInfo` events with immediate delivery.
$((pending\_VehicleInfo > 0) \Rightarrow (deliver\_VehicleInfo'$
$\wedge \neg deliver\_RulesInfo' \wedge \neg deliver\_RegionInfo')),$
$((pending\_RulesInfo > 0) \Rightarrow (\neg deliver\_VehicleInfo'$
$\wedge deliver\_RulesInfo' \wedge \neg deliver\_RegionInfo')),$
$((pending\_RegionInfo > 0) \Rightarrow (\neg deliver\_VehicleInfo'$
$\wedge \neg deliver\_RulesInfo' \wedge deliver\_RegionInfo')),$

Using the results from Policy 2 we conclude that we can guarantee collision avoidance by giving `VehicleInfo` events top priority. However, in this policy we can not guarantee that the rule version will always be consistent.

Policy 3 attempts to guarantee our final property by giving priority to `RulesInfo` events with immediate delivery. Policy 3 allows the satisfaction of rules consistency but removes the satisfaction of collision avoidance.

An interesting result of verifying the properties for the different delivery policies is that it appears that the collision avoidance and rules consistency properties conflict. We now have to revisit the II system and the requirements to determine if there is a compromise. For example, we could conclude that collision avoidance is of higher priority than rule version consistency. As a result, we could change the system property for rule version consistency from always true to always eventually true. Verification of the new weakened rule version consistency property with Policy 2 results in the property verifying to *true*. Alternatively, another possible solution is to change the implementation restriction on the UVCS delivery policy to allow for the delivery of multiple events in a given state. Changing the delivery policy to delivery each event type simultaneously and immediately will allow us to satisfy all of the other system requirements.

4.3 Analysis Performance

As with any model checking technique the size of the models and the time required for analysis are both important to the usability of the approach. In this paper we independently verify 28 properties, giving us 28 different sets of model checking data. All of the properties were run on a time-shared system with 8 GB of memory and four 750 MHz processors. We will now try and give a general perspective of the performance issues. The verification time for all properties in both models was under 2 hours. However, this result is slightly misleading since the majority of properties, with the exception of three in the ABLS system, could actually be verified in under 5 minutes. In terms of the number of states reached, the results also varied. On the one hand, using Policy 1 in the ABLS system the Poll event properties involved approximately $5x10^5$ reached states and the badge location properties involved $1.7x10^7$. On the other hand, using Policy 2 in the UVCS system for analyzing collision avoidance involved $2.6x10^{24}$ states reached and determining rule version consistency with Policy 3 involved $5.4x10^{21}$ states. Unfortunately, due to space restrictions we can not provide more details regarding the analysis performance.

5. OPTIMIZATIONS

During the evaluation of our approach it became clear that in order to model significant II systems, such as the ABLS and UVCS, optimization would have to be used. If optimization techniques were not used many of our models would have been too large to verify in a reasonable amount of time. By using optimization techniques we were able to avoid any serious problems as a result of the state explosion problem . For the two system discussed in this paper we used standard optimization techniques such as cone of influence reduction, data abstraction, and reduction of nondeterminism. We also examined architectural style-specific optimizations. All of the optimizations were implemented by hand, requiring both intuition and experience. Models were optimized independently for each property being analyzed. No approach to automating the application of optimization

techniques has been developed in the context of this work.

Cone of influence reduction involves the removal of variables that do not influence variables in the properties or specifications currently being verified. Another standard optimization technique, data abstraction, relies on the observation that there usually exists a simple relationship among data values within system specifications that involve data paths. Data abstraction is a process in which a mapping is found between actual data values and a smaller abstract set of data values. The system with the abstract data values, in place of the actual data values, is then modeled. The final standard optimization technique used was the reduction of non-determinism in cases where the reduction does not compromise system integrity.

In addition to standard techniques, we also considered architectural style specific optimizations for II systems. Recall the six parameters that compose an II system discussed in Section 2.1. The removal and combination of two of the parameters, components and events, are both ways that a model can be optimized. Note that these system-level optimizations are event dependent and can not always be utilized.

6. CONCLUSIONS

We have summarized our extensions to the approach in [4]. Specifically, we modified the representation of event data, allowed for the expression of more complex user-defined delivery policies, and added dynamic bindings to support the addition and removal of both components and connectors at run-time. These enhancements have allowed for the modeling and analysis of more significant II systems then was previously possible. Our evaluation of the ABLS and UVCS systems demonstrate that our extended approach can be used to analyze systems with "real-world" significance.

However, even with optimization, accurate models of realistic systems are often extremely large. The size of these models requires state-of-the-art computers and a large quantity of patience. While model checking the ABLS and UVCS systems, it became obvious that it is not feasible to model some larger systems even when exhaustively using model optimizations. Although some larger systems are not feasible to model check in their entirety, there is hope that our approach can also be applied to partial models of larger systems. Although our approach is only explored in the context of II systems, we are optimistic that a similar compositional approach could be developed to model check more general forms of publish subscribe systems. For example, we could include support for multiple distributed dispatchers and multicast event announcement.

In our analysis of the UVCS system we have also shown how our analysis can be used to uncover and fix conflicts between the requirements and the delivery policy. Although our results were promising, we need to explore further the use of iterative analysis in determining the appropriate definition of system parameters, such as delivery policies, in the context of overall system requirements.

The work started in [4] and [5] and now extended here aides in alleviating the semantic artifact gap and state explosion, however more work needs to be done. A major drawback to the current approach in comparison to other work such as [1] and [6] is that it is only a partial solution and is not completely automatic since user interaction is required in developing the XML representation. Until this gap is bridged, the possibility of spurious results, although reduced, is still present since the mark-up of source code and parameterization of the system occur by hand. Currently, work being done within our research group at Queen's University is extending this approach to be completely automatic.

7. ACKNOWLEDGEMENTS

We would like to thank David Garlan, Serge Khersonsky and Jung Soo Kim for providing us with information and software that served as the basis for our research.

8. REFERENCES

[1] J. Corbett, M. Dwyer, J. Hatcliff, et al. Bandera: extracting finite-state models from Java source code. In *Proc. of the Int'l Conference on Software Engineering*, pages 439–448, Jun. 2000.

[2] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. In *Proc. of the ACM SIGSOFT Int'l Symposium on the Foundations of Software Engineering*, pages 209–221, Nov. 1998.

[3] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Towards a formal treatment of implicit invocation using rely/guarantee reasoning. *Formal Aspects of Computing*, 10:193–213, 1998.

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

[5] D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *Proc. of the* 10^{th} *Int'l SPIN Workshop on Model Checking of Software*, May 2003.

[6] G. J. Holzmann and M. H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. on Software Engineering*, 28(4):364–377, Apr. 2002.

[7] K. L. McMillan. *The SMV Language*. Cadence Berkeley Labs, Mar. 1999.

[8] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proc. of the Int'l Conference on Software Engineering*, pages 177–186, Apr. 1998.

[9] S. Stuurman and J. van Katwijk. On-line change mechanisms: the software architectural level. In *Proc. of the ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering*, pages 80–86, Nov. 1998.

[10] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Trans. on Software Engineering and Methodology*, 1(3):229–268, 1992.

[11] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. *ACM Trans. on Information Systems*, 10(1):91–102, Jan. 1992.

[12] M. Wermelinger and J. L. Fiadeiro. Algebraic software architecture reconfiguration. In *Proc. of the European Software Engineering Conference and the ACM SIGSOFT Int'l Symposium on the Foundations of Software Engineering*, pages 393–409, Sept. 1999.