Discovering Architectural Mismatch in Distributed Event-based Systems using Software Model Checking*

Technical Report 2006-524

L. Ruhai Cai, Jeremy S. Bradbury, Juergen Dingel

School of Computing, Queen's University
Kingston, Ontario, Canada
{*cai,bradbury, dingel*}*@cs.queensu.ca*

October 2006

Abstract

The success of distributed event-based infrastructures such as SIENA and Elvin is partially due to their ease of use. Even novice users of these infrastructures not versed in distributed programming can quickly comprehend the small and intuitive interfaces that these systems typically feature. However, if these users make incorrect assumptions about how the infrastructure services work, a mismatch between the infrastructure and its client applications occurs, which may manifest itself in erroneous client behaviour. We propose a framework for automatically model checking distributed event-based systems in order to discover mismatch between the infrastructure and its clients. Using the SIENA event service as an example, we implemented and evaluated our framework by customizing the Bandera/Bogor tool pipeline. Two realistic Java applications are implemented to test and evaluate the framework.

1 Introduction

Event-based systems are composed of clients and an event service which are arranged in a publish/subscribe architecture. Information is exchanged between clients by sending events to and receiving events from the service. Event-based services are widely used in applications where asynchronous communication is required among the components. Traditionally, these services have been centralized, however, in recent years, event services have become distributed and used to implement systems such as web services and mobile programming environments.

*This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Despite the complexity of their underlying implementation, distributed event-based infrastructures typically have a small and intuitive interface. Unfortunately, the intuitive nature of the interface can be misleading. An application running on top of an event-based infrastructure will only function correctly if it uses the services of the infrastructure appropriately and does not make any incorrect assumptions on how the service works. This observation is supported by one of the authors of the distributed event-based infrastructure SIENA [Car]:

> *"...people make a lot of assumptions on the order in which they will receive events. In other words, they program their applications with a synchronous communication model in mind, and end up getting weird results when events queue up and get delivered in an unexpected order."*

These kinds of incorrect assumption are an example of architectural mismatch. Mismatch centers "...around the assumptions a reusable part makes about the structure of the application in which (it) is to appear" [GAO95]. In the case of SIENA, we typically observe protocol mismatch in which client components assume, for instance, that events are received in the same order in which they have been sent. In reality, SIENA is considered a best-effort service and does not maintain the order of events. Therefore, clients using SIENA must be designed and implemented accordingly [CRW01]:

> *"...the implementation of SIENA must not introduce unnecessary delays in its processing, but it is not required to prevent race conditions induced by either the external delay or the processing delay. Clients of SIENA must be resilient to such race condition; for instance, they must allow for the possibility of receiving a notification for a canceled subscription."*

The goal of our research is to develop a framework to discover if clients make incorrect assumptions about the event service and if an architectural mismatch between the service and its clients has occurred.

It can be very difficult to discover this kind of architectural mismatch and to determine that, for instance, the clients are not robust enough to handle possible race conditions. Conventional testing methods may be insufficient, because clients are often concurrent. Static approaches relying on behavioural interface specifications may be difficult to apply, because the details of the response of a client to an event may be highly run-time dependent. As an alternative, we suggest to leverage the increasing power, maturity, and availability of software model checkers for the discovery of architectural mismatch. Sophisticated translators and optimizations make software model checking increasingly viable for realistic applications written in languages such as Java or C/C++.

Model checking has already been suggested as an analysis technique for event-based systems using the implicit-invocation architecture [GKK03, BD03]. However, the scope of this previous work was limited to systems with centralized event services and did not attempt to analyze *realistic distributed* event services. Moreover, the systems analyzed by previous work were idealized examples while our work is applied directly to actual implementations. Our approach to analyzing distributed event-based systems is to leverage the architecture to split the analysis into two smaller and more manageable tasks. In one task, we summarize the behaviour of the infrastructure services with a manually created finite-state machine model and verify that the clients function correctly when composed with this model. In the second task, we verify that the implementation of the event

infrastructure conforms to the model. This paper focusses on the first task, and leaves the second for future work.

We propose a general conceptual framework for automatically model checking realistic distributed event-based Java applications. Automatic model extraction and optimization are used as much as possible to ensure that the resulting model is accurate and tractable. To prove the viability of the framework, we implemented it for use with the SIENA event service. A customized version of the Bandera/Bogor tool pipeline is used for model extraction, optimization, and analysis. While the default version of the pipeline provided most of the required functionality, the model extraction phase had to be customized to allow for the automatic integration of different event service/infrastructure models. Moreover, the Indus slicer and the Bogor model checker were customized. Our work thus also provided an opportunity to evaluate the customizability and applicability of the Bandera/Bogor pipeline for domain-specific model checking.

We will first provide a description of distributed event-based systems and the Bandera/Bogor pipeline in Section 2. In Section 3, we outline our conceptual framework before describing an implementation of the framework for the SIENA event service using the Bandera/Bogor model checking pipeline. In Section 4, we evaluate our implementation using a chat program and a peer-to-peer file sharing system. In Section 5, we discuss related work and in Section 6 we provide conclusions and future work.

2 Background

2.1 Distributed event-based systems

There are two basic kinds of clients in an event-based system: publishers and subscribers. Publishers publish events or notifications, to the event service, and subscribers subscribe with the event service to the type of events they are interested in. When the event service receives a notification from a publisher, it goes through all subscriptions and dispatches the event to those who have subscribed to it. Publishers announce events without knowing the identity of the subscriber components and do not wait for any response from subscribers. Therefore, event-based systems allow for anonymous, asynchronous communication which in turn provides loose coupling between client components and good maintainability.

There are three main types of distribute event-based systems [MC05]: *collocated middleware* – the event service is in the same address space as the clients (e.g., mSECO [HMN+00]); *single separated middleware* – the event service is located on a single machine while the clients are distributed on other machines (e.g., CORBA); *multiple separated middleware* – clients and event service are distributed and execute on different machines or address spaces (e.g., SIENA). While our approach could be applied to all three types of system, we chose SIENA because both the clients and middleware are distributed making it an interesting and challenging architecture in which to discover mismatch.

2.1.1 SIENA

In SIENA, the event service is implemented with one or multiple servers connected in a hierarchical, acyclic peer-to-peer, generic peer-to-peer or hybrid topology. The event messages in a SIENA system are attribute-value pairs. A client can subscribe to an event by sending a subscription, which

contains the filter patterns that specify the types of events it wants to receive. A filter pattern is a set of (attribute, operator, value) triples. The operator is normally a binary comparison operator, such as "=", ">". Each triple specifies the value range for an attribute and all triples in a filter are combined conjunctively. The event message notifications and filters are used in SIENA to publish events, subscribe to a given filter, unsubscribe from a filter, advertise intent to generate events that match a filter and to unadvertise the publishing of events that will match a filter. In our work we are only interested in the publication and subscription of events and do not handle advertisements.

The most prominent incorrect assumptions made by application developers regarding the behaviour of the SIENA event service appear to be:

1. *"A client will not receive notifications to which it is not subscribed."*
2. *"Notifications will be delivered in the order in which they have been sent."*
3. *"Notifications are never lost."*

Later in this paper, we will describe how to discover mismatch due to the first two assumptions in a chat program and a peer-to-peer file sharing system.

2.2 Bandera/Bogor Tool Pipeline

The Bandera/Bogor tool pipeline is a set of tools for automatically extracting finite-state models from Java source code for model checking [CDH$^+$00, RDH03]. The pipeline has an open structure and the order of the tools in the pipeline is determined in a configuration file by specifying that the output of one tool forms the input of the next tool in the pipeline. Tools can easily be added to or removed from the pipeline by modifying the configuration file.

The main tools in the Bandera/Bogor tool pipeline are:

- *Soot:* translates Java class files into Jimple, an intermediate representation suitable for optimization.
- *Indus:* slices the Jimple code.
- *J2B:* transforms the Jimple code into BIR, the input language for the model checker Bogor.
- *Bogor:* model checks the BIR models.

This pipeline appears ideal for our research, because both the transformation from Java to BIR and the model checking using Bogor are highly flexible and can easily be customized to better support SIENA programs written in Java. The J2B tool, for instance, allows the user to add arbitrary BIR code to the model and to replace portions of the automatically generated BIR code. Bogor, on the other hand, can be extended with new primitive types, expressions and commands to provide better support for the modeling of different domains. Moreover, Bogor has a highly modular, open architecture which allows, for instance, new search algorithms or optimizations to be swapped in. We will discuss our specific customization of the Bandera/Bogor tool pipeline in Section 3.1.

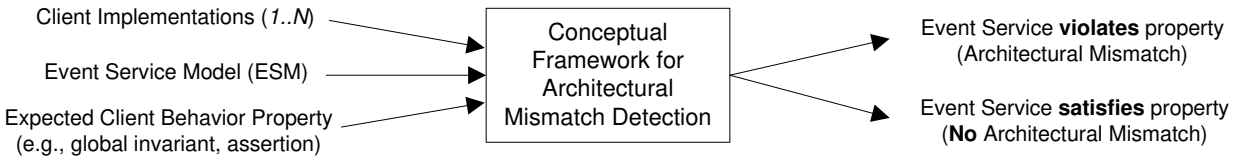

Figure 1: Conceptual Framework

3 Conceptual framework

The input required and the output produced by our conceptual framework are given in Figure 1. Specifically, our framework requires three input artifacts:

1. *Client implementations:* we use actual implementations of client components written in a program language like Java.

2. *Event service model (ESM):* The ESM is assumed to be formulated in the input language of the model checker employed in the framework. Moreover, the ESM is assumed to correctly capture the behaviour of the event infrastructure from a client's perspective. At the moment, the framework offers no validation to ensure the implementation of the infrastructure actually conforms to the ESM.

3. *Expected client behavior properties:* A formal specification of a property that some or all of the clients need to satisfy. Only incorrect client assumptions that cause this property to fail will lead to mismatch that our framework is able detect. While property specifications could be provided using any formalism that the model checker understands (e.g., LTL, CTL, Buechi Automata), in this paper, we will assume that the specification is given as a global system invariant or an assertion.

The conceptual framework will take the client component implementations and transform them into the input language for model checking. During the transformation, common optimizations include slicing and various abstraction techniques such as data and predicate abstraction are used to reduce the state space. The client component models produced via transformation are integrated with the manually created event service model (ESM). The combined system model (client models + ESM) is input to a model checker that verifies the expected client behavior properties and reports any violations together with a counter example. The conceptual framework allows client applications to be checked for different incorrect assumptions through the use of different ESMs. For instance, to see if the correct behaviour of the clients depends on the preservation of the event order, an ESM is built which does not preserve event order. To determine if a client is resilient to message loss, an ESM is built in which messages can get lost.

It is important to note the advantages and disadvantages of using a manually created ESM. A clear disadvantage is that the conformance of the infrastructure to the ESM is not checked. If the ESM does not reflect the behaviour of the infrastructure, our analysis may provide spurious results. Moreover, user effort is required to construct the ESM. However, despite the availability of automatic model extractors such as Bandera's J2B tool, the automatic extraction of an ESM suitable for model checking from the infrastructure code is currently not an option, due to the size, complexity and typically distributed nature of event infrastructures. A manually created ESM,

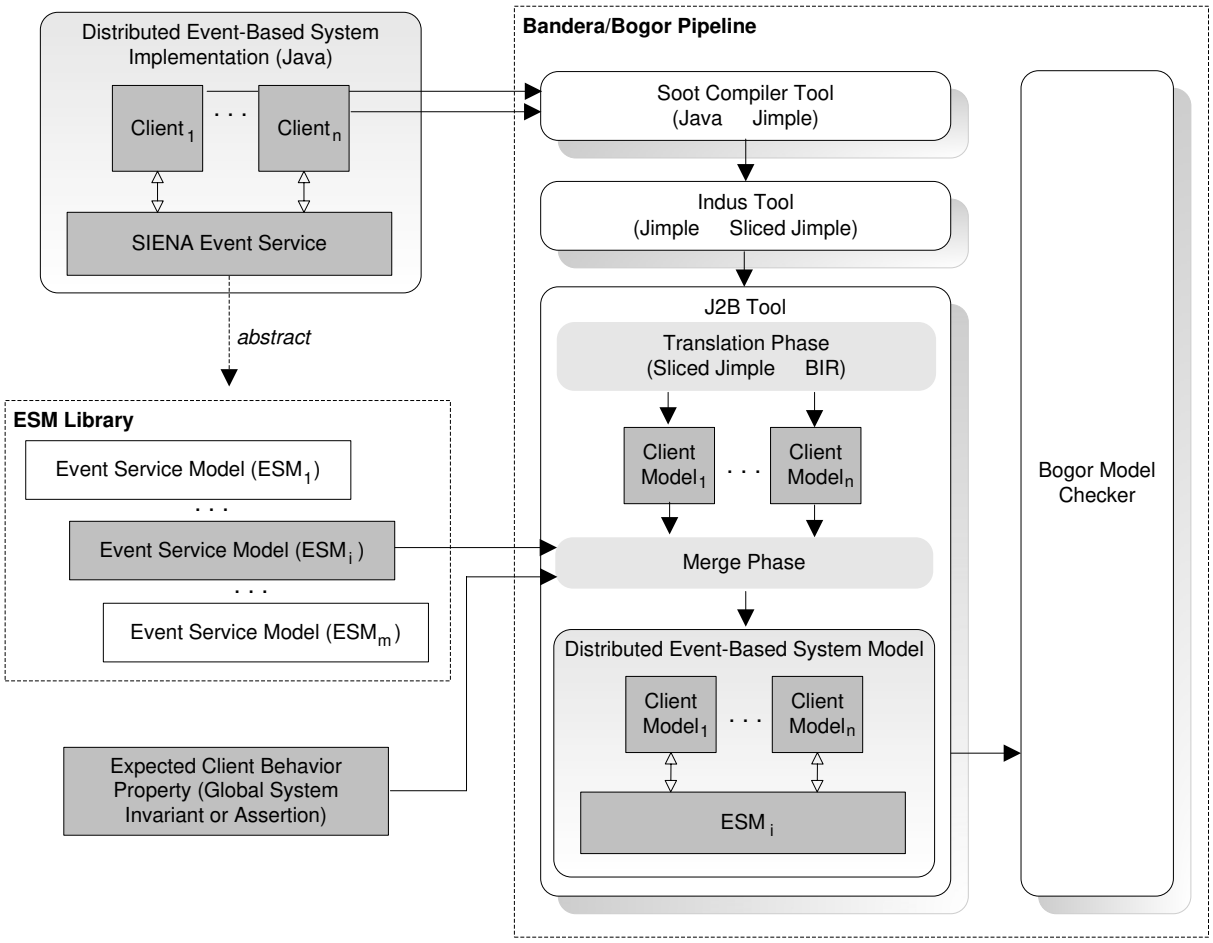

Figure 2: Software model checking framework for SIENA

on the other hand, will be considerably more succinct. Moreover, one ESM could be used for checking several applications so that the cost of building it can be amortized across multiple uses. In conclusion, we feel that a manually created ESM is the best option, and note comformance checking between ESM and the infrastructure implementation as an important direction for future work.

3.1 Example Implementation of Framework using Bandera/Bogor

In this section, we will describe an implementation of our conceptual framework using a customization of the Bandera/Bogor tool pipeline (see Figure 2). To test the feasibility of the framework, we chose Java client applications that use SIENA as the underlying distributed event-based infrastructure.

Client application transformation and optimization. The Java source code of the client application is translated into a BIR model for model checking using Soot to translate from Java to Jimple, Indus to slice and optimize the Jimple representation, and J2B to translate the sliced Jimple into the BIR modeling language.

Event service model creation. The behaviour of the SIENA event service is captured by

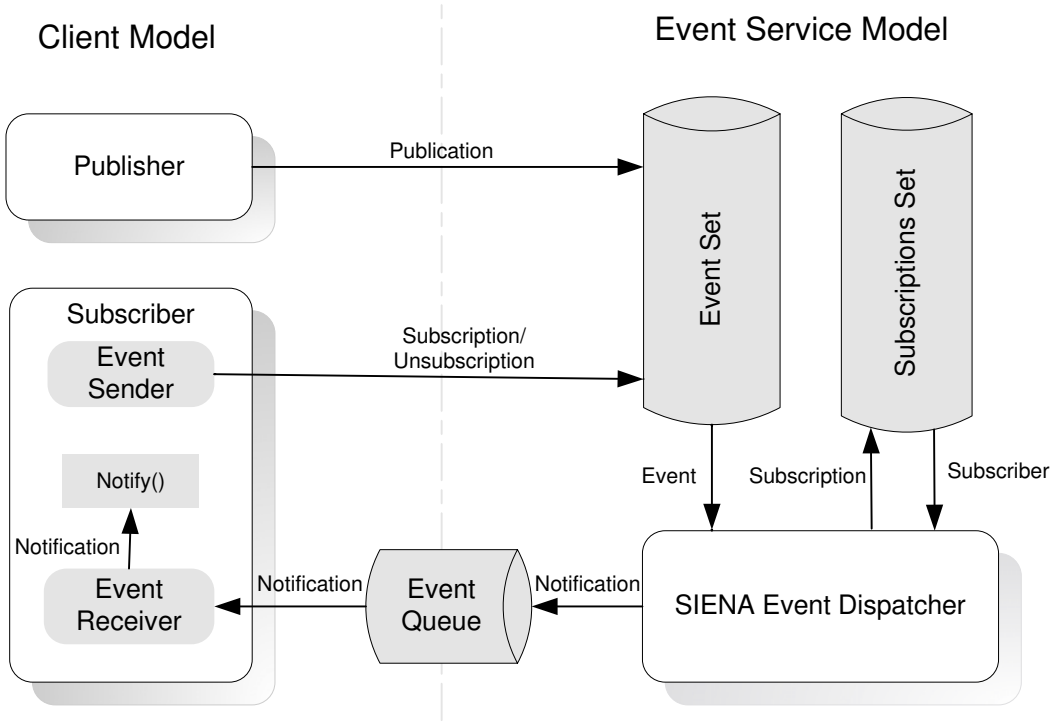

Figure 3: Event Service Model (ESM) and Client Model Interaction for SIENA

manually created BIR models (ESM_i). In the SIENA ESM used for our analyses (see Figure 3) we need two data structures to handle the events, a communication channel between the client and the service model, and an event set to store the events at the server before they are dispatched. As we discussed in Section 1, the SIENA event service does not guarantee the order of dispatching of events. By using an event set, we will be able to exhaustively check all dispatching orders of the events. Since events will be removed from the set in every possible order, a regular FIFO queue is sufficient to simulate the communication channel. Bogor extensions are used to implement the event set and the message queue. The SIENA ESM is developed as an active thread that waits for the arrival of events and handles them based on their types. The current implementation of the model only supports three types of event operations: subscribe, unsubscribe and publish. Note that in our examples, we do not check if clients are resilient to message loss. To do that, an ESM would have to be created in which message can get lost. Recall that the ESM for SIENA is independent of the client applications so the same ESM can be reused to check for mismatch in all SIENA client applications.

Client model and ESM integration. The integration of the automatically generated client application model and the ESM to form a system model happens in the J2B tool. Recall that the main function of the J2B tool is to translate Jimple code into BIR models. After the BIR models are generated for the client application, the J2B tool allows the user to replace methods and threads in the models with user specified methods or threads. It also allows the user to add additional BIR extensions, global variables, methods and threads to the existing BIR model.

The SIENA ESM is added as a BIR addition. In order to integrate the client and service models, the methods on the client side that handle the communication between the client and the service need to be replaced. The SIENA implementation provides a standard *ThinClient* class as an inter-

```
function ThinClient.subscribe (Pattern p) {
  loc loc0:   invoke initialize()
              goto loc1;
  loc loc1:   do invisible {
                sub := new Event;
                sub.pattern := p.pattern;
                sub.type := EVENT_TYPE.SUBSCRIBE;
                Set.add<Event>(events, sub);
              } return;
}
```

Figure 4: Subscribe method in the event service model

face for the SIENA client to exchange events with the SIENA event service. Thus we only need to replace all the methods in the *ThinClient* class. As the *ThinClient* is standard, the replacement can be reused for different client applications with minor customization. Figure 4 shows the *subscribe* method in the ESM that will be invoked instead of the *subscribe* method in SIENA when Bogor carries out its analysis. A large portion of the code of the *ThinClient* class handles low level socket communication. As all methods of the *ThinClient* class will be replaced, there is no need to translate this code into BIR. Thus only method stubs are kept for the *ThinClient* class.

Model and property integration. A property that a client application is expected to satisfy is provided as an assertion or global invariant. On the one hand, an assertion can be inserted manually into the BIR code at the appropriate place. Typically, we want to check if the behaviour carried out in response to the receipt of a notification is correct. Therefore, the assertion is often placed in the notify() method of a client (see Figure 3) which is called whenever the client receives a notification from the event service. On the other hand, a global invariant can be inserted into an active monitor thread that is added to the integrated client and service models.

Model checking the system model. The combined client model and ESM are checked by the Bogor model checker with respect to the assertion or global property invariant. We will discuss specific model checking results as well as the relationship between property failures and architectural mismatch in the next section during our evaluation of two real Java applications that use the SIENA event service. resulting counter example should be mapped back to the source code.However, this part is still under development.

4 Evaluation

To evaluate the effectiveness of our implementation of the framework two realistic examples are provided – a chat example and a peer-to-peer file sharing system each of which were analyzed with respect to two properties. Table 1 indicates the size of each example implementation as well as the size of the BIR models. For each example and each property, the size of the entire BIR model after optimization is given (column 4). Additionally, column 5 shows the size of the optimized BIR representation of the actual client source code and thus excludes code related to included Java library files.

Example program	# of Java classes	# of Java LOC	# of BIR LOC	# of client BIR LOC
Chat program	11	906	8912, 9036	1752, 1876
Peer-to-peer file sharing system	16	1188	8072, 8194	2365, 2487

Table 1: A comparison of the Java source and BIR model sizes for our examples

4.1 Chat program

4.1.1 Description

In this program, there are an arbitrary number of distributed clients, which can subscribe, unsubscribe, create and close chat rooms, and post messages to and display messages from the chat rooms. The system uses the SIENA event service for message exchange. The basic events in the system are *SubscribeChatRoom*, *UnSubscribeChatRoom*, *CreateChatRoom*, *CloseChatRoom* and *PostMessage*. Each client acts as both a publisher and a subscriber and maintains a list of all active chat rooms.

Our chat program has a GUI interface to display posted messages for each chat room. Unfortunately, the current version of Bandera does not support the Java Swing library, which is used to build the GUI for this program. However, issues of architectural mismatch in SIENA client applications like the chat program require analyzing the interaction between the client and the SIENA event service and not the GUI interface. Therefore, for the purposes of our analysis we separate the GUI from the rest of the application. To facilitate the removal of the GUI code, we assume hat the client has been implemented using the MVC (Model-View-Controller) architecture, which provides a clean separation between the view (GUI) component and the model and control part of the system. Recall that transforming client applications into client BIR models in Bandera requires as input the client application byte code. Thus, the application must compile even with the GUI code removed. A skeleton of the GUI classes needs to be kept with all method bodies and the Java Swing class names removed. This is a manual preprocessing step that is done prior to using our framework. Additionally, some of the skeleton GUI classes are replaced with BIR code during the model integration to simulate any interaction between the GUI and the controller that is required during model checking.

4.1.2 Analysis

For the chat program we consider the analysis of two properties both of which demonstrate architectural mismatch between chat client applications and the SIENA event service.

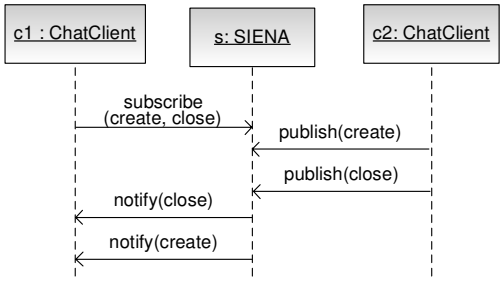

Figure 5: Sequence diagram related to assertion *Chat_Rooms_Close_Correctly*

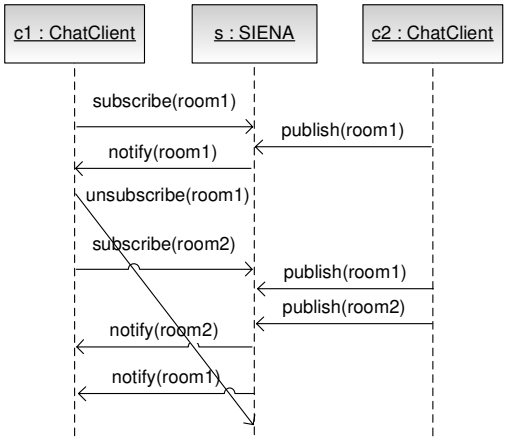

Figure 6: Sequence diagram related to assertion *Displayed_Msgs_Always_for_Current-_Chat_Room*

Property 1: *Chat rooms are always closed properly*. In this case, the client creates a chat room and then closes it. We use a set to store the list of chat rooms which the client maintains. When a chat room is created, the chat room number is added to the list, and when it is closed, it is removed from the list. In this example, the room list is empty at the beginning and there is only one chat room being created and closed. Therefore, the set will be empty if the chat room is closed properly. The property is expressed as an assertion, which is inserted into the notify() method of the client:

assert allEventsDelivered -> chatRoomList.isEmpty();

The analysis of the chat program with this assertion using our framework shows that the assertion fails because the event service does not preserve event order. The *CreateChatRoom* and *RemoveChatRoom* events are not commutative. If the events are delivered in the right order, a chat room will be created and closed properly. But if the order is reversed, as shown in Figure 5, the chat room remains open after these two events are delivered. In conclusion, the correct functioning of the operation of closing chat room relies on an implicit assumption (preservation of message order) which is not satisfied by the SIENA event service.

Property 2: *Displayed messages are always for the current chat room.* There are two steps involved in switching chat rooms: unsubscription from the current chat room and subscription to a new chat room. This property is expressed as an assertion which is again located in the client's notify() method:

assert (PostMessage.roomName == currentRoomName);

This assertion states that the room name of the incoming message is the current room name and is evaluated whenever a message is received. The analysis using our framework shows that this assertion fails with a counter example as shown in Figure 6. Since it is possible in SIENA for the client to receive unsubscribed events, it is possible for the client to receive messages for the previous chat room after switching to a new chat room. If these messages are not processed properly, as is the case in our example, they might be displayed in the wrong chat room.

4.2 Peer-to-peer file sharing example

4.2.1 Description

In [Hei01], SIENA is used to implement a file-sharing service similar to Gnutella – a well-known peer-to-peer file sharing service. This example was also used in other research on compositional reasoning of descriptions of architectural middleware [CIP04]. Following the ideas in [Hei01], we have implemented a prototype of a peer-to-peer file sharing service as a client application of SIENA. In this prototype, a client can play two roles: file provider (subscriber) and query originator (publisher). There are three message types, which are mapped to the communication events of the underlying event-based system. First, *Offer* messages are sent out by file providers as a subscription of queries. An offer message describes the files located on a host. Second, *Query* messages are publications that a query originator sends to describe the files it is interested in with patterns. A query message publication will be delivered by the event service to all file providers who offer the files matching the patterns. Third, *Response* messages are generated by the file provider and sent back to the query originator via the event service. A response message is actually a notification that contains the detailed description of the files, which match the query as well as a return address, which will be used by the query originator.

Similar to the chat program, the peer-to-peer file sharing example has a GUI interface, that we have implemented using the MVC pattern. Also, since we are mainly concerned with the mismatch between the client and the SIENA event service, the actual file sharing portion of the program is irrelevant and thus not implemented.

4.2.2 Analysis

We evaluate potential mismatch between SIENA and the peer-to-peer client applications by evaluating two properties.

Property 1: *The displayed responses are for the current search.* This property is an assertion located in the notify() method of the query originator:

```
assert (currentQuery.pattern == Notification.pattern);
```

The model checking result shows that this assertion fails. When the query originator starts a query, it sends out the query and subscribes to the response from the file provider. The user of a query originator may choose to stop receiving responses to the current query and start a new query by unsubscribing the old response and sending out a new query (as shown in Figure 7). With SIENA, a query originator may receive unsubscribed responses. However, due to architectural mismatch, the query originator in this example assumes that no unsubscribed responses will be received and that all received responses will be displayed as the responses for the current query.

Property 2: *No queries are received after a file provider revokes the offer.* This property is expressed with the following assertion in the notify() method of the file provider:

```
assert offerRevoked(p) -> (Notification.pattern != p);
```

Model checking using the framework determines that the assertion is violated with the counter example shown in Figure 8. Consider a file provider that stops sharing certain files by sending a *revokeOffer(pattern)* event (i.e., an unsubscription). In our example the SIENA event service sends

out a response every time a query is received assuming no queries for the offer will be received after it is revoked. But since this is not always the case, files can still be shared after being revoked.

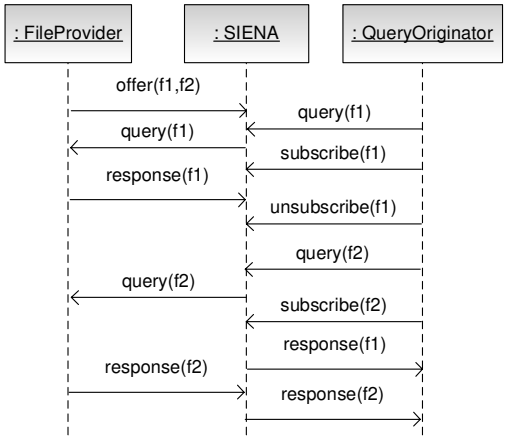

Figure 7: Sequence diagram related to assertion *Displayed_Responses_For_Current_Search*

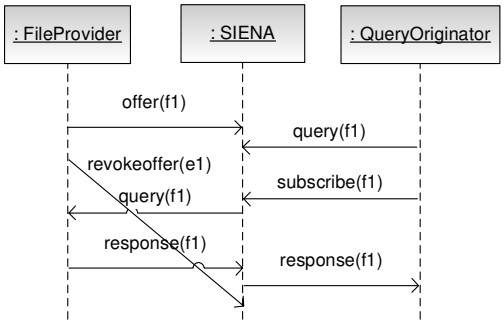

Figure 8: Sequence diagram related to assertion *No_Queries_After_Revoke_Offer*

4.3 Summary

We have successfully used our framework to identify architectural mismatch between two realistic (but small) client applications and the SIENA event service. Table 2 summarizes the results of our analysis together with some relevant metrics. All of the timing results were achieved on a shared system running Linux with 5 GB of memory and four 3 GHz processors.

One drawback of our approach is the modification of the code required when applying it to GUI-based event systems. For both of our GUI examples, a moderate amount of manual modification of either the input Java code or the generated BIR model was required (however, it appears that these modifications could be automated using standard source code analysis and transformation techniques). Moreover, our SIENA model does currently not support the *advertise* and *unadvertise* operations and complex filters available in SIENA. Finally, state space explosion only allowed minimal configurations to be analyzed. Nonetheless, we believe that we have presented a viable approach to the discovery of architectural mismatch in distributed event-based system implementations and that most of the limitations mentioned above can be mitigated or even removed with further research.

5 Related work

Previous work on discovering architectural mismatch has primarily focused on specification-based approaches. For example, discovering mismatch using process algebra [BCD01] and architecture description languages (ADLs) [UY00, ZDL01]. One of these previous approaches, XADL, is designed for event-based architectures and includes the specification of message orderings through interaction protocols [ZDL01]. Our approach differs from this work in that we allow for the discovery of mismatch in existing applications and do not require the manual specification of all com-

Global system invariant or assertion	Result	Time (h:m:s:)	# of states	Reason for mismatch
Chat program				
Chat_Rooms_Close_Correctly	False	00:25:53	67291	Reordering
Displayed_Msgs_Always_For_Curr_Room	False	01:23:23	222566	Unsubscription
Peer-to-peer file sharing system				
Received_Responses_For_Current_Search	False	00:00:43	2379	Unsubscription
No_Queries_After_Revoke_Offer	False	00:01:08	3008	Unsubscription

Table 2: Anaylsis results for all global system invariants and assertions

ponent and connector assumptions. Instead, we require only a global system invariant or assertion and the manual construction of the ESM.

Several existing projects have focused on model checking event-based systems using publish/subscribe architectures [GKK03, BD03, HDD+03, ZBCD06]. The work started by Garlan, Khersonsky and Kim [GKK03] and later extended by Bradbury and Dingel [BD03] focuses on model checking systems with a centralized event services, not distributed. Further extensions to this work have allowed for the model checking of event systems written in a special purpose language, IIL [ZBCD06]. The Cadena project uses an approach to model checking systems that use CORBA [HDD+03]. Similar to our project, Cadena uses Bogor as a model checker. However, unlike our work, Cadena requires manual specification of client behavior. The work by Caporuscio et al. in [CIP04] also uses an implementation of Gnutella on top of SIENA to illustrate an approach to compositional model checking of middleware specifications. However, it also assumes specifications of the client behaviour based on state machines and does not work on source code directly. Other related work in the area of event-based systems includes a semi-automatic approach to the analysis of GUI systems using Bandera/Bogor [DRTV04], and several approaches to the analysis of distributed Java that use remote method invocation [SL01, CCDD05]. Another related project in the area of model checking software architectures is CHARMY which allows for the specification of UML-like diagrams for system design and verification [IMP05].

6 Conclusions and future work

In this paper, we have proposed an approach for the semi-automatic discovery of architectural mismatch between an event-based system and its clients based on invalid assumptions the client makes about the behaviour of the event service. We have described a proof-of-concept implementation of this approach that targets the Java version of the SIENA event service and uses a customized version of the Bandera/Bogor tool pipeline. Finally, we have demonstrated the viability of the approach by evaluating our implementation on several case studies including a chat program and a file-sharing application. Our implementation leverages the increasing maturity of software model checking tools in general and the customizability and power of the Bandera/Bogor tool pipeline in particular. We found the Bandera/Bogor tool pipeline a usable, powerful platform for the analysis of domain-specific Java applications.

The biggest drawback of our approach is its reliance on a manually created event service model

that correctly captures the relevant aspects of the behaviour of the event service. Currently, the conformance of the event service to its model is not checked, but it is conceivable that ideas from model-based testing (e.g., [CGN+05]) or conformance checking (e.g., [FHRR04]) could be used to address this issue. Additional directions for future work include: increasing the degree of automation of the framework, and applying the framework on other event-based infrastructures such as CORBA.

7 Acknowledgments

We would like to thank the members of the SAnToS Laboratory at Kansas State University for support in customizing the Bandera/Bogor tool pipeline.

References

[BCD01] M. Bernardo, P. Ciancarini, and L. Donatiello. Detecting architectural mismatches in process algebraic descriptions of software systems. In *Proc. of WICSA 2001*, pages 77–86, Aug. 2001.

[BD03] J. S. Bradbury and J. Dingel. Evaluating and improving the automatic analysis of implicit invocation systems. In *Proc. of ESEC/FSE 2003*, pages 78–87, Sept. 2003.

[Car] Antonio Carzaniga. Personal e-mail correspondance with J. Dingel. Feb. 9, 2005.

[CCDD05] T. Cassidy, J.R. Cordy, T. Dean, and J. Dingel. Source transformation for concurrency analysis. In *Proc. of the Int. Work. on Language Descriptions, Tools and Applications (LDTA 2005)*, Apr. 2005.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, et al. Bandera: extracting finite-state models from java source code. In *Proc. of ICSE '00*, pages 439–448, 2000.

[CGN+05] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Model-based testing of object-oriented reactive systems with Spec Explorer. Technical report, Microsoft Research, 2005.

[CIP04] Mauro Caporuscio, Paola Inverardi, and Patrizio Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *Proc. of ICSE 2004*, pages 221–230, 2004.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Comp. Sys.*, 19(3):332–383, Aug. 2001.

[DRTV04] Matthew B. Dwyer, Robby, Oksana Tkachuk, and Willem Visser. Analyzing interaction orderings with model checking. In *Proc. of ASE 2004*, pages 154–163, 2004.

[FHRR04] C. Fournet, C.A.R. Hoare, S.K. Rajamani, and J. Rehof. Stuck-free conformance. In *Proc. of the Int. Conf. on Computer Aided Verification (CAV 2004)*, Jul. 2004.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or, why it's hard to build systems out of existing parts. *IEEE Software*, 12(6):17–26, Nov. 1995.

[GKK03] D. Garlan, S. Khersonsky, and J.S. Kim. Model checking publish-subscribe systems. In *The Int. SPIN Work. on Model Checking of Software (SPIN 2003)*, May 2003.

[HDD+03] John Hatcliff, Xinghua Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. Cadena: an integrated development, analysis, and verification environment for component-based systems. In *Proc. of ICSE 2003*, pages 160–173, May 2003.

[Hei01] Dennis Heimbigner. Adapting publish/subscribe middleware to achieve Gnutella-like functionality. In *Proc. of the ACM Symp. on Applied Computing (SAC '01)*, pages 176–181, 2001.

[HMN+00] Mads Haahr, René Meier, Paddy Nixon, Vinny Cahill, and Eric Jul. Filtering and scalability in the ECO distributed event model. In *Proc. of the Int. Symp. on Soft. Eng. for Parallel and Distributed Systems (PDSE '00)*, page 83, 2000.

[IMP05] Paola Inverardi, Henry Muccini, and Patrizio Pelliccione. Charmy: an extensible tool for architectural analysis. In *Proc. of ESEC/FSE-13*, pages 111–114, 2005.

[MC05] René Meier and Vinny Cahill. Taxonomy of distributed event-based programming systems. *The Computer Journal*, 48(5):602–626, 2005.

[RDH03] Robby, M.B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. of ESEC/FSE-11*, pages 267–276, Sept. 2003.

[SL01] Scott D. Stoller and Yanhong A. Liu. Transformations for model checking distributed Java programs. In *Proc. of Int. SPIN Workshop on Model Checking of Software (SPIN'01)*, 2001.

[UY00] Sebastian Uchitel and Daniel Yankelevich. Enhancing architectural mismatch detection with assumptions. In *Proc. of the Int. Conf. and Work. on the Engineering of Computer Based Systems*, pages 138–146, Apr. 2000.

[ZBCD06] Hongyu Zhang, Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Using source transformation to test and model check implicit-invocation systems. *Special Issue on Source Code Analysis and Manipulation, Science of Computer Programming*, 62(3):209–227, Oct. 2006.

[ZDL01] Bo Zhang, Ke Ding, and Jing Li. An XML-message based architecture description language and architectural mismatch checking. In *Proc. of Comp. Soft. and Applications Conf. (COMPSAC 2001)*, pages 561–566, Oct. 2001.