Specification and Evaluation of Transparent Behavior for SIP Back-to-Back User Agents

Gregory W. Bond
AT&T Labs—Research
Florham Park, NJ, USA
bond@research.att.com

Eric Cheung
AT&T Labs—Research
Florham Park, NJ, USA
cheung@research.att.com

Thomas M. Smith
AT&T Labs—Research
Florham Park, NJ, USA
tsmith@research.att.com

Pamela Zave
AT&T Labs—Research
Florham Park, NJ, USA
pamela@research.att.com

ABSTRACT

A back-to-back user agent (B2BUA) is a powerful mechanism for realizing complex SIP applications. The ability to create, terminate, and modify SIP dialogs allows the creation of arbitrarily complex services. However, B2BUAs must be designed with care so as not to disrupt service interoperability. A commonly-stated goal is for B2BUAs to be as *transparent* as possible while achieving its design goals. Though the notion of transparency is intuitively appealing, it is difficult to define. To address this issue, this paper proposes a definition of transparency and presents a formal model of a transparent B2BUA to serve as the specification of transparency. From this specification, we identify issues with both the realizability and desirability of this behavior, and suggest modifications to the original model. We evaluate the behavior of a number of public B2BUA implementations via testing, using some novel techniques to create test cases based on the formal models.

1. INTRODUCTION

A back-to-back user agent (B2BUA) is a powerful mechanism for realizing complex SIP applications. The ability to create, terminate, and modify SIP dialogs allows the creation of arbitrarily complex services. However, B2BUAs must be designed with care so as not to disrupt service interoperability.

A commonly-stated goal is for B2BUAs to be as *transparent* as possible while achieving its design goals. However, the notion of transparency is not defined by the SIP specification [13]. This specification defines a back-to-back user agent as "a logical entity that receives a request and processes it as a user agent server (UAS). In order to determine how the request should be answered, it acts as a user agent client (UAC) and generates requests." The specification further states that "Since it is a concatenation of a UAC and UAS, no explicit definitions are needed for its behavior."

To date, the behavior of B2BUAs has not been specified, other than that it must comply with the behavior of a UA on each side. This leads to the perception that B2BUAs break transparency of the network, and therefore hinders innovation at the endpoints. On the other hand, a large number of use cases have arisen in real-world deployments of SIP services that require B2BUAs. For example:

- Hide network topology information. This is often performed by Session Border Controllers (SBCs) that interface the networks of two service providers.
- Terminate an existing session, for example by a prepaid application when calling credit has run out, or by an IMS P-CSCF when it detects that the radio linkage with the mobile device has been disconnected.
- Modify the Session Description Protocol (SDP) information in the message body, for example to work through firewalls.
- Perform third party call control by advanced applications, for example to change a direct two-party call to a three-party call by bringing in a mixing media server.

The conflict between common usage and lack of specification is untenable. It is important that the behavior of B2BUAs is specified such that developers can implement them correctly and service providers can test them for compliance. At the same time, any innovations and extensions to the SIP protocol can be designed to work with transparent B2BUAs as intermediaries. While a transparent B2BUA does not provide any useful service, it can serve as the baseline and various B2BUAs that provide services can be defined as deviations from the transparent B2BUA. For example, a prepaid application B2BUA is transparent except when it terminates the session by sending BYE requests on both SIP dialogs.

In 2007, the IETF SIPPING working group started to work on a Best Current Practices document for a transparent B2BUA. Unfortunately, this work has not been continued to completion. The last draft [11] specifies how the Allow, Required and Supported headers should be handled when a request is received. It also specifies that when the

IPTComm 2010, 2-3 August, 2010 Munich, Germany

B2BUA relays a message certain headers should be generated, and the other headers and message body should be copied.

The SIP Servlet API standard [1] provides limited support for B2BUAs by providing methods for creating an outgoing request to be sent out on the UAC side based on an incoming request received on the UAS side. It specifies that the implementation must copy the headers from the incoming request to the outgoing request (with a few exceptions). However, the SIP Servlet API standard does not further specify any transparent B2BUA behavior.

The purpose of this paper is to provide a firmer foundation for B2BUAs in SIP by providing a rigorous and pragmatic specification of transparent behavior. This entails a number of contributions.

First, we show that it is difficult to define what "transparency" means, even on an informal and intuitive basis. After examining the alternatives, we settle on a pragmatic working definition (Section 2).

Second, we formalize our informal definition (Section 3). Message sequencing is formalized in terms of an executable model in Promela, the language of the Spin model checker. Message contents are described in terms of header values. To provide an environment in which the B2BUA model can be analyzed and verified automatically, we also developed new formal models of SIP user agents. Because of the use of model checking, all of the Promela models are guaranteed to be complete, consistent, unambiguous, and correct with respect to well-defined criteria.

Third, we demonstrate the pragmatic use of our specification by evaluating existing B2BUA implementations (Section 4). Because manually generated tests are not sufficient, we generated a suite of 2,408 tests automatically from the formal UA models. We then ran both manually and automatically generated tests, using automated testing tools, on the available implementations. None of the implementations comply fully with the B2BUA specification. This shows that implementing a correct B2BUA is difficult, and that comprehensive testing is necessary to ensure the correctness of implementations.

Overall, this research shows that judicious use of specification, analysis, and testing tools can greatly improve the quality of SIP components and SIP-based applications. Although our research needs to be extended in various ways, further work is amply justified by the initial results.

2. DEFINITION OF TRANSPARENT BEHAVIOR

What does it mean for a B2BUA to behave transparently? Transparency is an appealingly intuitive concept, but it is not easy to give it a rigorous definition.

In general, there are two approaches to definition. An *operational* definition of transparency would focus on the behavior of the B2BUA itself. An *observational* definition would define transparent behavior of a B2BUA as observed by its environment, which consists of the UAs at the far ends of its two dialogs. The advantage of an operational definition is that it is easy to tell whether a specific B2BUA satisfies the definition. One advantage of an observational definition is that it corresponds most closely to the intuitive notion of transparency. Another advantage is that it allows the most freedom in implementing B2BUAs.

In this paper, *pure propagation* refers to the following behavior of a B2BUA: receive a message in one dialog, and send it unchanged in the other dialog. A possible operational definition is, "A transparent B2BUA applies pure propagation to each received message, and does not send any messages that are not propagated."

Pure propagation cannot be correct transparent behavior because a B2BUA must change propagated messages in straightforward ways, such as modifying the `Call-ID` header and `tag` parameters to match the unique identifiers of each dialog. These changes to message content are specified in Section 3.2.

In this paper, *propagation* as a B2BUA behavior is the same as pure propagation, except with necessary header changes. A revised operational definition is, "A transparent B2BUA applies propagation to each received message, and does not send any messages that are not propagated." This definition does not work either, because it sometimes violates the SIP standard in the individual dialogs. Section 3.1.3 describes these situations.

Unfortunately, a rigorous observational definition is even harder to find than an operational definition. It could require that the presence of the B2BUA be undetectable by the far endpoints, but that is not achievable, even when real-time delays and header changes are excepted (see Section 3.1.3).

An observational definition should require that the media sessions between the far endpoints be the same whether there is a B2BUA present or not, because controlling media sessions is the primary purpose of SIP. To formalize this successfully, it would be necessary to define "the same" so that it generalizes over the nondeterministic behavior of the network between the far endpoints, which can affect endpoint behavior and media sessions even when there is no B2BUA. Also, a definition of transparency based on media sessions would be necessary but not sufficient—SIP signaling accomplishes more than just controlling media sessions.

In this paper we use a pragmatic definition of transparency that lies somewhere between the two extremes. A B2BUA is *transparent* if and only if:

- It acts as a standards-compliant UA in both dialogs.
- Its behavior within the two SIP dialogs is to propagate each message, and to not send any messages that are not propagated, except when this behavior would violate the protocol in either dialog.
- When its behavior is an exception to the basic rule, its behavior minimizes the effect of its presence between the far endpoint UAs.

The first two points are operational and precise. The third point is observational and rather vague.

We feel that this definition, despite its flaws, has enabled us to make progress toward understanding transparency. We regard it as an interim result, to be replaced in the future by a more precise observational definition.

3. SPECIFICATION OF TRANSPARENT BEHAVIOR

Our study covers the basic version of SIP defined in RFC 3261 [13], plus *info* [3] requests. *Info* requests allow application-level mid-call signaling without affecting dialog state, and

are used extensively for PSTN–SIP interworking and media server control.

3.1 Message Sequencing

3.1.1 Method of Study

Message sequencing is an aspect of behavior. It is concerned with when a user agent can or must send a message, and what messages a user agent might receive at any given time. We study message sequencing by means of formal modeling and analysis.

In the sequencing view a message is identified primarily by a type, which is a member of an enumerated set. The request types within our scope are *invite, ack, cancel, info,* and *bye*. The possible responses to these requests are categorized in an enumerated set according to the level of detail needed. For example, the possible responses to an *info* request are categorized as *infoDVR* or *infoRsp*.

The *infoDVR* category consists of 408 (Request Timeout) and 481 (Call/Transaction Does Not Exist) messages in response to an *info*. The name stands for Dialog Vanished Response, because both of these indicate that the dialog is gone. The *infoRsp* category consists of all other responses, whether successful (200) or failing (3xx-6xx). In the models, there is a need to distinguish between DVR responses and all other responses, because they are handled differently by the models. There is no need to distinguish between successful responses and other failing responses, because (from the perspective of our models, see Section 3.1.2) they are simply passed to the user.

Secondarily, messages that can carry SDP are categorized as carrying *offer, answer,* or *none* in their SDP fields. All other aspects of message content are discussed in Section 3.2, and are not included in the sequencing models.

In a previous study [18], we used formal modeling in the Promela language and verification with the Spin model checker [7] to investigate invite dialogs in SIP. We wrote nondeterministic models documenting all possible behaviors of the two user agents (caller UA and callee UA) during an invite dialog. To validate the models with respect to the RFCs, we included pointers to those documents. We used a suite of formal analysis and verification techniques to ensure that the models were complete and consistent according to specific definitions of those terms. We also wrote a large number of in-line assertions expressing our assumptions and understanding of the protocol, and verified automatically that the model was correct according to those assertions. These validation and verification techniques are described in detail in [18].

Our study of B2BUAs builds on this previous work. First, we improved our UA models in various ways. The endpoint UAs are the environment of a B2BUA, so they must be understood as well as possible. Most importantly, we added UA failures as manifested by 408 and 481 messages.

The new models are described in Section 3.1.2, and are available on the Web [5]. Some readers may be surprised at their complexity—the original intent was for SIP to be a "simple" protocol, but simplicity is long gone, even for the basic version studied here. The important point is that, faced with this unavoidable complexity, we must take advantage of available technology such as model checking to help us deal with it.

Our specification of transparent behavior of a B2BUA also takes the form of a Promela model. Unlike the UA models, it is a deterministic program, prescribing exactly what the B2BUA should do in each circumstance. It has been subjected to all the same analysis and verification activities as the UA models. This means that it is guaranteed to be complete, consistent, and unambiguous. It is also guaranteed to preserve a large number of correctness assertions evaluated at control points within the UA and B2BUA code.

The B2BUA model (in two versions) is described in Sections 3.1.3 and 3.1.4, and is available on the Web [5].

3.1.2 The User-Agent Models

We assume that message delivery is reliable and FIFO in each direction, because without this assumption a number of significant new problems arise [18].

The UA models are more complete with respect to RFC 3261 than our previous models. They include early media, 408 and 481 messages, and timeouts in the callee UA waiting for an *ack* to a successful initial *invite*. In the modeled behavior, failure of one UA is detected when the failed UA does not respond to a request from the live UA. Simultaneous failure of both UAs is not represented, however.

Because our primary goal is to help people program B2BUAs, SIP is modeled from the viewpoint of the *transaction user* in RFC 3261. According to RFC 3261, 100 (Trying) messages, retransmissions, and acknowledgments after *invite* failures are all handled exclusively by a lower-level *transaction* layer of the protocol stack. This means that they need not be present in our models.

As mentioned previously, the UA models are highly nondeterministic. There are four major causes of nondeterminism. First, nondeterminism can reflect user choice. For example, after sending an initial *invite*, a caller UA can choose to send a *cancel* message or wait for the response to the *invite*. Second, nondeterminism can represent the possibility of failure. Whenever a UA is due to respond to a request, the UA model can send the request or else fail. Third, nondeterminism can reflect concurrency. The two UAs and message channels between them are distributed and largely independent, so their events can be interleaved in arbitrary ways. Fourth, nondeterminism can reflect implementation freedom. For example, on receiving a *cancel* message when it has not yet responded to the initial *invite*, a callee UA must send both a 200 response to the *cancel* and a failure response to the *invite*. The order is not specified, however, so the model has a nondeterministic choice between the two orders.

We have made every effort to read RFC 3261 closely and interpret it correctly, but this is difficult to do because the RFC is informal, incomplete, and vague in many places. Our formal models have precise semantics and are guaranteed to be complete; they are organized so that a specific answer to a specific question is always easy to find. With the help of the SIP community they can be improved until they are declared correct by consensus, at which time they can serve as valuable appendices to the RFCs.

In the remainder of this section we discuss some specific aspects of UA behavior that are important for B2BUA behavior.

During a confirmed dialog, either UA can send an *invite* message to alter the session description (specification of the media channels). Because there is only supposed to be one such re-invite transaction at a time, a *re-invite race* occurs

if both UAs re-invite at about the same time.

A typical re-invite race is shown in Figure 1. Each UA knows there is a race as soon as it receives *invite* after sending *invite*. Each UA responds with *inv491* (a 491 message in response to an *invite*), so that both re-invite requests fail. Although each UA is free to try again at a later (and different) time, our models do not show any relationship between the earlier and later re-invites.

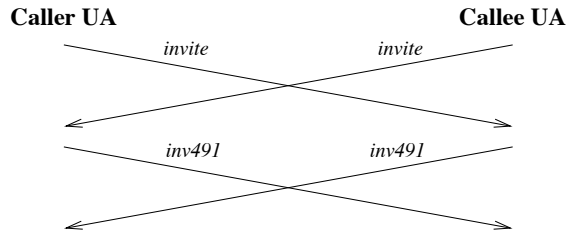

Figure 1: A re-invite race.

On receiving any *invite* (initial or re-invite) message, a UA need not respond immediately. This provides time for the UA to get instructions from a human user if necessary. In the models, a UA receiving an *invite* goes into an *invited* or *reInvited* state. In these states the UA can send or receive other messages. At any time, however, it has the choice to send a final response to the *invite*.

An invite transaction can take two forms with respect to the offer/answer exchange [12]. These two forms are illustrated in Figure 2 by re-invites from the callee UA. On the left, the *invite* message carries an offer and the *inv200* carries an answer. On the right, the *invite* message does not carry an offer, but rather solicits an offer from the other UA. In this form the *inv200* carries the offer, and the *ack* message carries the answer.

On the left, the caller UA leaves the *reInvited* state after sending *inv200*, even though it has not yet received the *ack*. Because the offer/answer exchange is complete, *even before receiving the ack* it can send a new *invite* message to begin a new re-invite transaction [12].

Any time after sending the initial *invite* and before receiving a final response to it, a caller UA can send *cancel* to cancel the transaction and abort the dialog. A *cancel race* occurs if the *cancel* message arrives at the callee UA after the callee UA has sent a final, successful response to the *invite*.

A typical cancel race is shown in Figure 3. The caller UA knows there is a race as soon as it receives *inv200* (a 200 message in response to an *invite*) after sending *cancel*. Having failed to cancel the initial transaction, it ends the dialog by sending a *bye* instead. Later it receives *canc200* sent by the callee UA.

For all requests, a *DVR* response in the model corresponds to either a 408 or 481 response. In the models, a failing UA sends a DVR response and then enters a state in which it no longer communicates except to send additional DVR responses.

This modeled behavior corresponds quite closely to the actual behavior of a UA that fails and restarts, having lost dialog state. The restarted UA will respond to all subsequent requests for that dialog with 481 messages.

The modeled behavior corresponds more loosely to the

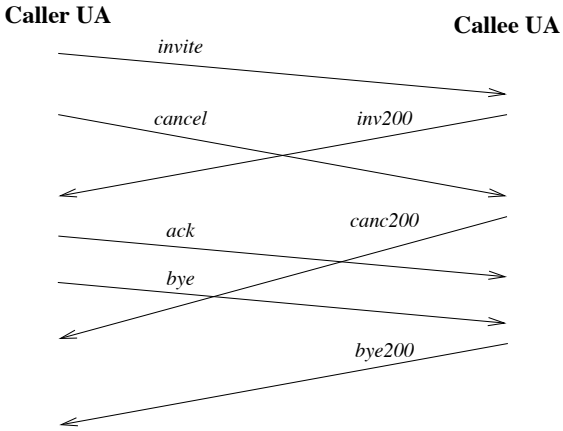

Figure 3: A cancel race.

actual behavior of a UA that fails and does not restart. In this case, obviously, the dead UA does not send any messages. Rather, the transaction layer of the live UA generates a 408 response for the transaction user to see. Thus a UA's sending 408 messages at and after failure is a modeling trick ensuring that one UA gets 408 responses when and only when the other UA has failed.

On receiving a DVR response, a UA that has not already sent a *bye* is supposed to send a *bye*. This causes two difficulties in the callee UA. First, the callee UA can receive a 408 or 481 response to an *info* message when it is still in the *invited* state and cannot legally send a *bye*. In this case we have the callee UA send a failure response to the initial *invite*.

Second, the callee UA can receive these responses to *info* requests when it has already sent an *inv200* for the initial *invite* but has not yet received the corresponding *ack*. It cannot legally send a *bye* in this case, either. It becomes blocked until it receives an *ack* or *ack* timeout, at which time it sends the *bye*.

Modeling reveals that a queue of messages in transit from one UA to the other can grow to size 7 (even though the model allows only one provisional response and only one outstanding *info* request). In this unusual scenario, one UA generates the message sequence *inv200, invite, info, bye* and then is suspended for a long interval. During this interval the other UA receives the 4 messages and processes them to generate the following sequence: *ack, inv200, infoRsp, info, invite, bye, bye200*. Of these 7 messages, 4 are responses to the 4 queued messages, and 3 are new requests.

3.1.3 The Back-to-Back User Agent Model

Our model of a back-to-back user agent is a deterministic Promela program that acts as a callee UA in one dialog and a caller UA in another. It is proposed as a specification of correct transparent behavior.

Whenever possible, the transparent B2BUA reacts to receiving a message from one dialog simply by propagating it. The remainder of this section discusses the situations in which this is not possible, and how the B2BUA can deal with the situation safely.

A typical re-invite race is shown in Figure 4. When the B2BUA receives an *invite* from the right, it cannot forward

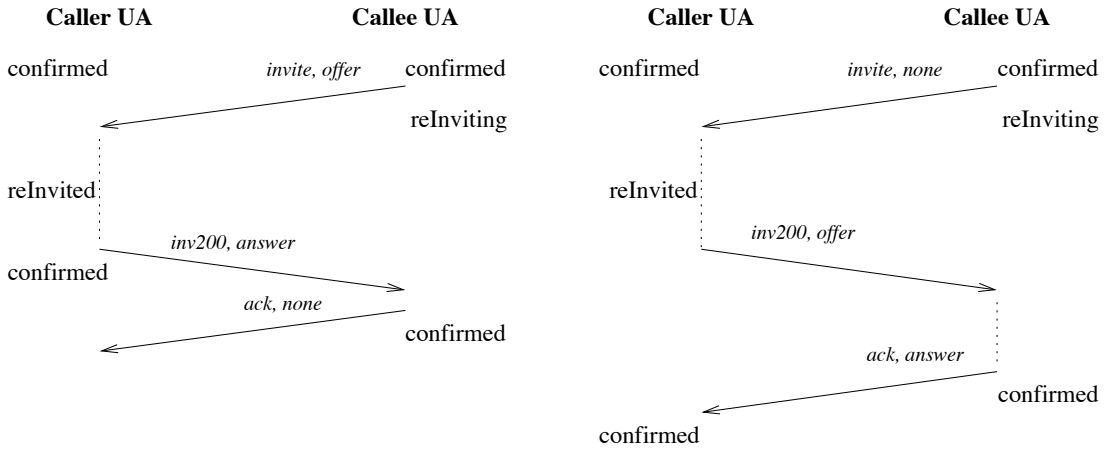

Figure 2: Two ways the callee UA can re-invite, with local states of the UAs shown. These transactions can also be initiated symmetrically by the caller UA.

it to the left, because it would violate the SIP protocol in the leftmost dialog by knowingly creating a re-invite race. Because it knows that there is a race, it generates an *inv491* response instead. After that point it can resume propagation of messages.

As a result of the presence of the B2BUA, the endpoint caller UA on the left receives *inv491* without having received a racing *invite*—something that could never happen in a simple dialog. This illustrates the point that "transparent behavior" cannot be defined as "undetectable behavior."

In Figure 4 the B2BUA absorbs a request (rather than propagating it) and generates its own response to the request. This deviation is benign for two reasons: (1) if the B2BUA had propagated the request, the request would not have changed the state of the UA that received it; and (2) both endpoint UAs receive the same responses to their requests as they would have received without the B2BUA.

A B2BUA can never simply propagate *cancel* requests, because *cancel* requests are "hop-by-hop". On receiving a *cancel* from the left, a B2BUA must immediately generate a response in the leftmost dialog. If the B2BUA also sends a *cancel* to the right and receives a response from the right, then the B2BUA must absorb the response. This behavior is shown in Figure 5.

In one form of cancel race, *cancel* and *inv200* messages cross in the left dialog. This means that the B2BUA receives the *cancel* after it has already propagated *inv200* to the left, so that the dialog on the left of the B2BUA looks like Figure 3. There is no point to propagating *cancel* to the right, and it would also be illegal to do so because the dialog to the right has been confirmed. The B2BUA must simply absorb the *cancel* and generate *canc200* as a response.

Cancel races detected on the right of the B2BUA do not require a transparent B2BUA to behave differently than in Figure 5. If the *cancel* arrives at the callee UA too late, then the calle UA may have already sent *inv200*. As with *invFail* in Figure 5, the B2BUA simply propagates *inv200*.

The B2BUA can receive a request (for example a re-invite) from a dialog after it has received a *bye* from the other dialog and propagated it to the requesting dialog, as shown in Figure 6. If the B2BUA were to propagate the new request (in Figure 6, to the right), it would be sending a new request in a dialog that has already seen a *bye*. In our opinion this is clearly wrong and should be illegal, although we cannot find a specific prohibition in RFC 3261. Instead of propagating the *invite*, the B2BUA should absorb it and generate *invFail*. If the B2BUA had propagated the *invite*, it would have had no effect on the state of the callee UA.

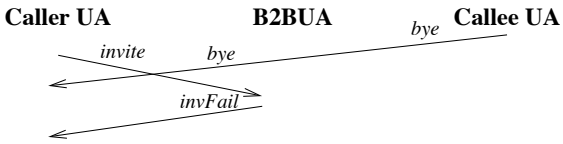

Figure 6: A late request arrives at a B2BUA.

Similarly, the B2BUA cannot propagate a provisional response by sending it in a dialog that has already seen a *bye*. In this case the B2BUA absorbs the message without generating any other message.

3.1.4 A Modified B2BUA

The pure B2BUA described in Section 3.1.3 propagates messages whenever propagation is legal. Unfortunately, it is a specification that cannot be implemented in a SIP Servlet container. The reason is that the SIP Servlet standard [1] mandates handling *cancel* requests in a different and less transparent way.

Because of the importance of the SIP Servlet containers as platforms for SIP applications, we provide a modified model to serve as an alternative specification of a transparent B2BUA. The modified specification is compatible with the SIP Servlet standard.

Figure 7 shows how the modified B2BUA handles a *cancel*. Provided that the *cancel* is not too late in arriving at the B2BUA (see Figure 3), the B2BUA immediately generates *invFail* to the left, ending that dialog. It also sends the *cancel* to the right.

In the scenario shown in Figure 7, there is a cancel race on the right, so that the B2BUA receives *inv200* from the right and generates *bye* to the right. If there were no race, it would receive *invFail* from the right and simply absorb it.

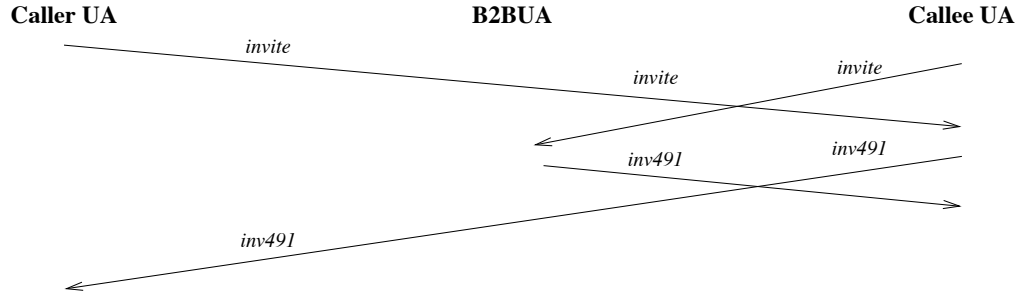

Figure 4: A re-invite race in the presence of a B2BUA.

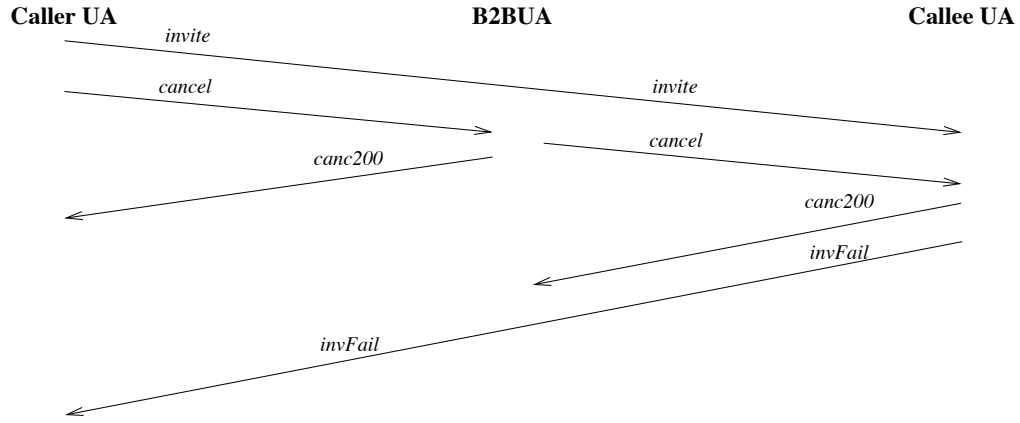

Figure 5: Canceling in the presence of a B2BUA.

When the modified B2BUA generates *invFail*, it creates a situation in which its two dialogs are in different and incompatible states. From that point on, there is *no* message propagation, and the B2BUA handles the two dialogs separately.

Although the modified B2BUA does not satisfy our interim definition of transparency, it has other advantages, such as responding faster overall to a *cancel* request. This points to a potential benefit of finding a better definition of transparency. If the definition were more observational, it would allow more freedom in the implementation of B2BUAs. This would give application and platform developers the room to design better B2BUAs, with improved efficiency and possibly other desirable properties.

3.2 Message Content

In order to achieve transparency, the message *content*, as well as sequence, must be preserved. Message content includes the *headers* of a SIP message as well as the *body*. A minimal number of headers are mandated by RFC 3261; other headers are specified in a variety of RFCs. Finally, it is always possible for a SIP UA to include so-called *private* or *extension headers*. Message bodies convey information in a wide variety of contexts, for example, descriptions of media connectivity, conveyed via SDP; carriage of instant messages (IMs); or carriage of commands and associated responses between a UA and a media server.

As discussed in Section 1, [11] begins to address the issues of transparency with respect to message contents. The recommendations in this section generally accord with that document; however that document also discusses issues that are outside the scope of this paper.

For correct propagation of the message, the body must be copied from the incoming message to the outgoing message. Furthermore, with the exceptions noted below, all headers in the incoming message should be copied to the outgoing message.

Part or all of three headers are used to provide a unique dialog identifier: the value of the `Call-ID` header along with the values of the `tag` parameter of the `From` and `To` headers. Due to requirements for global uniqueness, these values cannot be re-used in a new dialog; the B2BUA must generate its own unique values.

The `Via` and `Contact` headers are used for hop-by-hop message routing, and thus should not be copied. Similarly, if the topmost `Route` header in an incoming request targets the B2BUA, it should not be copied. The `Record-Route` header applies to a dialog; since the B2BUA terminates two dialogs, it is responsible for adhering to any routing requirements of this header in the two dialogs, but the header should not be copied between dialogs.

The B2BUA must inspect `Allow`, `Supported`, and `Required` headers and modify them accordingly to reflect the capabilities of the B2BUA. The `Max-Forwards` header is used to detect routing loops. If the value of the header in an incoming request is greater than 0, the B2BUA should decrement the value of the header by 1 for the propagated request; otherwise the B2BUA should reject the request.

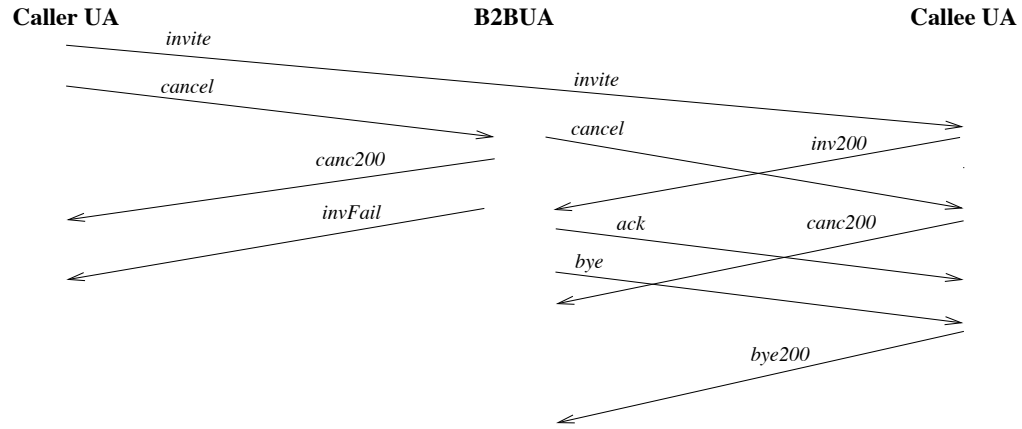

Figure 7: A cancel race in the presence of a modified B2BUA.

4. EVALUATION OF IMPLEMENTATIONS

After we completed the specification of the correct behavior of a transparent B2BUA and wrote a Promela program to formally model the behavior, we undertook the task of testing existing B2BUA implementations to evaluate if they comply with the specification.

4.1 Systems Under Test

Our evaluation is restricted to implementations that (1) are built on the SIP Servlet API, because it is the dominant standard for SIP application development; and (2) have source code freely available for inspection and use, so that users of these implementations may make use of these results to make any necessary correction to the source code if desired.

The SIP Servlet API provides limited support for B2BUA applications in the form of the `B2buaHelper` class with several convenience methods. A B2BUA application can use this class to manage linkage of its two dialogs. As well, upon receiving a request on the first dialog, the application can call one of the convenience methods to create an outgoing request on the second dialog. The SIP Servlet container is responsible for modifying and copying various headers correctly. The SIP Servlet container also handles certain requests on behalf of the application, for example the *cancel* request. Therefore, a B2BUA implemented using the SIP Servlet API relies on both correct application programming and correct container behavior.

The B2BUA implementations we evaluated are listed below:

B2bTerminator (BT) This is a complete example application to illustrate the use of `B2buaHelper` given in [2]. This application behaves transparently except it tears down the call after a certain time. We test this application as a transparent B2BUA by setting a very large timeout value.[1]

ECharts for SIP Servlets (E4SS) E4SS is an open source framework that allows the use of the finite state machine paradigm to program SIP servlets at a higher level of abstraction [16]. It also includes reusable features, amongst them a transparent B2BUA application called `B2buaSafe`. The version tested is SVN version 1578.

SailFin Converged Application Framework (CAFE) This is another open source framework that provides a higher level of programming abstraction for SIP applications [15]. By default, a CAFE application acts as a transparent B2BUA. The programmer can override the transparent behavior at different events to implement the specific logic of the application. The version tested is sailfin-cafe-v1-b24.

The containers we evaluated are SailFin [14] version sailfin-v2-b31g and Oracle Communication Converged Application Server (OCCAS) version 4.0. BT and E4SS can be deployed and tested on both containers. CAFE currently only supports the SailFin container and thus is not tested on OCCAS. Thus in total there are five systems under test (SUTs): BT/SailFin, BT/OCCAS, E4SS/SailFin, E4SS/OCCAS, and CAFE/SailFin.

4.2 Manual Test Generation

We first utilized KitCAT [17] to test the above B2BUA implementations. KitCAT is a test tool for performing functional testing of converged (SIP and HTTP) applications. For this testing, KitCAT acts as both the caller and callee user agents. Drawing on experience at the SIP Interoperability Test events and call flow documents [9, 6], we wrote 12 test cases including race conditions where the two endpoints send messages at the same time (e.g. *cancel* and *inv200*, *bye* and *bye*). The test cases include assertions to check that the SUT sends the correct messages and that the message headers and contents are passed correctly according to Sections 3.1 and 3.2 respectively. The test results are discussed in Section 4.4.1.

However, writing these KitCAT test programs manually proved to be time-consuming. Moreover, KitCAT imposes a call state machine on its test agents which precludes the generation of certain scenarios such as the re-invite race shown in Figure 1. We concluded that we require a lower level test tool for this kind of protocol testing, and also automatically generated tests for better coverage. This approach is

[1]This code change, together with changes necessary for BT to run on OCCAS, is available at [5].

discussed in the following section.

4.3 Model-Based Test Generation

Given the complexity of the SIP protocol, and the possible interactions that may occur between agents and a B2BUA, one might infer that the universe of possible behaviors is very large indeed. Verification confirms this: the Spin model-checker discovers 48,966,575 unique states for our combined agent-B2BUA model. In the context of testing, this immense state space indicates that a hand-crafted test suite of 10, 20, or even 100 tests cannot possibly provide adequate coverage. The testing challenge then is to improve upon what can achieved by hand-crafting a test suite.

The approach we've chosen is to generate tests using the same model we use for verification. One advantage of this approach is that generated tests are guaranteed to conform to behaviors specified by the model. Another advantage is that it is possible, in principle, to generate a test suite that satisfies a notion of complete coverage. However, as we have seen, the tremendous size of the state space makes this latter goal impractical for any obvious notion of completeness. For this reason we've identified a series of test criteria that allow us to intuitively partition the universe of possible tests into practically sized test suites. For each test we identify:

- the length: the total number of messages sent or received by the user agents – the greater the number of messages sent, the more complex the interaction is between agents;
- the maximum queue size: the maximum number of messages that are enqueued at any time on the agent and B2BUA queues – more enqueued messages corresponds to greater channel latency or scarcity of processing resources
- the "weather profile" which is determined by the messages present in the test:
  - a "sunny day" test excludes *invFail*, *cancel*, *DVR*, and *ackTimeout* messages;
  - a "cloudy day" test includes at least one *invFail* or *cancel* message but no *DVR* or *ackTimeout* messages;
  - a "stormy day" test includes at least one *DVR* or *ackTimeout* message.

We can now identify tests that meet a particular criteria, for example: "all" sunny day tests with queue size 1 and length less than or equal to 12 (the meaning of the "all" will be qualified shortly). This test suite would correspond to moderately complex but normal behavioral interaction between agents via a B2BUA.

We use a two phase approach to automatically generate tests as shown in Figure 8. The first phase generates "test traces" from the model. A test trace is a high-level symbolic representation of a sequence of messages sent or received by the user agents via the B2BUA. The second phase translates a set of test traces to an executable test suite. Each executable test ensures that messages are sent and received in a timely fashion and in the expected order.

An example of a test trace is:

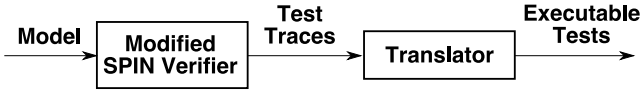

Figure 8: Two phase model-based test generation.

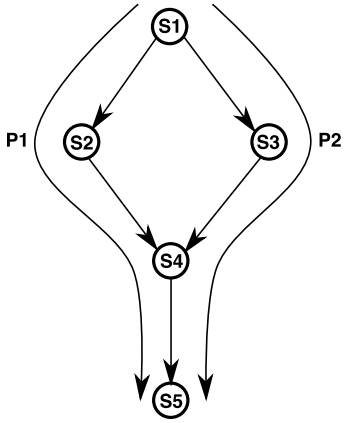

Figure 9: A model's states and paths.

```
8
(caller)[out!invite,none]:(callee)[in?invite,sdp]:
(callee)[out!inv200,offer]:(caller)[in?inv200,sdp]:
(caller)[out!bye,none]:(callee)[in?bye,sdp]:
(callee)[out!bye200,none]:(caller)[in?bye200,sdp]:
1:1:1:ALL:1
```

where the first number indicates the test length, the final five colon-delimited fields indicate the individual and overall maximum queue lengths for the agents and the B2BUA (here "ALL" indicates that both agents and the B2BUA had the same maximum queue size of 1) and the remaining colon-delimited fields represent messages sent or received by the agents. Thus, a test trace contains all the information required for selecting a test that meets the criteria described in the previous section.

Since the Spin model checker parses our model and traverses its state space to perform verification we chose to harness this machinery in order to generate tests. However, trace-based test generation involves recording the paths interconnecting a model's states but, being a model checker, Spin endeavors only to visit all of a model's states. Considering the example model shown in Figure 9, Spin would visit states S1 through S5 of the model shown. However, utilizing its depth-first search algorithm, Spin would completely traverse path P1 or P2 but it wouldn't completely traverse both. This is because state S4 is common to both paths so the second path would be truncated when the depth first algorithm arrives at state S4 a second time. To bridge the divide between state and path traversal we augmented our model and Spin's verifier. The agent model is augmented to maintain a record of messages sent and received by the agents. This way, each reachable state of the augmented model will include a record of the path traversed to reach the state. Spin's depth-first verifier algorithm is augmented to output a complete path (a path from the initial state to a valid end state) when it encounters one.

It was also necessary to refine the modified B2BUA model presented in Section 3.1.4 in order to exclude tests that reflected unachievable container behavior. Containers normally serve requests in FIFO order but, using a B2BUA model that dedicates a separate request queue to each agent, tests were generated that required a container to serve one agent's requests while unfairly neglecting the other agent's requests. To eliminate this unfair behavior we replaced the B2BUA's two input queues with a single queue that is shared by the two agents. This modification ensures that the B2BUA serves requests in FIFO order the same way a container does.

Spin supports limiting a search to a pre-defined depth, where depth is defined in terms of the number of transitions traversed between model states. We use this depth limiting facility to limit the number of generated tests. For example, for a depth limit of 71 we generate 361,737 unique tests ranging from length 4 to 17, and overall maximum queue size of 5. Generating a test set this way is memory and CPU intensive. For example, generating the aforementioned tests required 105 GB of preallocated RAM and 40 CPU minutes of a single 2.40 GHz Intel Xeon processor core.

Given the enormous number of generated tests, one might expect reasonable coverage of system behavior. To confirm this intuition we inspected the tests for instances of example call flows. We confirmed that there were many representative test cases corresponding to the hand-crafted test suite described in Section 4.2. We also identified the four call flows from the "Example Call Flows of Race Conditions" RFC [6] that conform to the scope of our model and confirmed that representative tests existed for each.

Although inspection of the generated tests cases reveals excellent test coverage, inspection also reveals that the set of generated tests is not complete for the specified depth limit. For example, inspection of our tests reveals that not all message interleavings are present for all call flows. We assume that the underlying reason is that Spin's verification algorithm is not designed for traversing all paths of a model, rather it is designed for traversing all states of the model. Determining which particular aspect of the verification algorithm is responsible for compromising completeness is something we are currently investigating.

Figure 10 shows the test application architecture. The test driver is responsible for administering the generated tests and recording test results. We use the JUnit unit test framework [10] for executing a test suite and reporting test results. In addition to JUnit the tests also use the ECharts for JAIN-SIP (E4JS) API for sending and receiving SIP messages. JAIN-SIP [8] provides a transaction-user API for SIP and E4JS [4] is an abstraction layer on top of JAIN-SIP that provides facilities for managing multiple agents, in our case a caller and callee agent, sharing a SIP stack instance. The system under test is a B2BUA SIP Servlet application running in a SIP Servlet container.

4.4 Test Evaluation

The following presents the results of applying the manually and automatically generated tests to the systems under test.

4.4.1 Results of Manually Generated Tests

Table 1 shows the results of using KitCAT and hand-crafted test cases to test the five SUTs listed in Section 4.1.

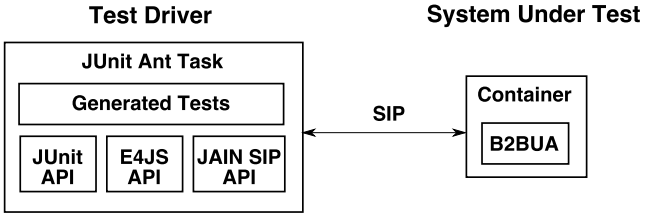

Figure 10: The test application architecture.

The results reveal two problems related to message sequencing. First, when faced with the cancel race shown in Figure 7, BT and CAFE do not send *bye* to terminate the right dialog even though the left dialog has been terminated.

Second, in the scenario where callee receives re-*invite*, sends *inv200*, but before receiving *ack* the callee sends *bye*, all three SUTs on SailFin fail. This reveals a bug in the SailFin implementation where it throws an exception if the application attempts to send a mid-dialog request before receiving the *ack* request, even though this is allowed by [13].

In terms of message content transparency, BT and CAFE rely on `B2buaHelper` class to create outgoing requests based on incoming requests. OCCAS copies unknown extension headers in this operation, but SailFin does not. However, in forwarding responses the application must copy unknown headers. E4SS uses its own code to copy headers from incoming to outgoing messages. However, this testing reveals a bug in the E4SS implementation where headers in the *ack* request are not copied.

4.4.2 Results of Automatically Generated Tests

We used only the OCCAS container for evaluating B2BUAs with automatically generated tests. This is because of the SailFin bug uncovered using manual testing described in the previous section. Since we did not use SailFin then we could not test CAFE. Thus in total there are two SUTs for testing with automatically generated tests: BT/OCCAS, E4SS/OCCAS. Table 2 shows the results of our testing.

Of the over 360,000 tests generated, we used the criteria described in Section 4.3 to select a manageable test suite of 2,408 tests: 257 sunny day, 1,335 cloudy day and 816 stormy day. We confirmed that these tests included the scenarios covered by our manually generated tests. Furthermore, we made sure that the cloudy day tests included *cancel*/*invite200* races, re-*invite* races, and common failure scenarios. In general, we selected tests with a maximum queue size of 1 except for cases that required queue sizes of 2, such as in some *cancel* scenarios where two response messages can be sent in a row by an agent. Using a small queue size represents normal environmental conditions, with minimal channel latency and minimal competition for processing resources. Test length for *cancel* scenarios and stormy day scenarios were limited to length 11 and 13, respectively.

The test results, shown in Table 2, reveal failures in both the B2BUA applications and in the underlying OCCAS container (container failures are indicated by a * superscript).

For the cloudy day tests, neither SUT was capable of negotiating the complexities of certain re-*invite* races. Testing also uncovered the same problem with BT that we uncovered using manual testing, namely the inability to properly handle a *cancel*/*inv200* race. The E4SS B2BUA does not

SUT	Message Sequence		Message Content
	Tests passed	Failed cases	
BT/OCCAS	11/12	*cancel* and *inv200* race	Partially fail: responses
BT/SailFin	10/12	*cancel* and *inv200* race Callee sends *bye* before receiving *ack*	Fail: requests, responses
E4SS/OCCAS	12/12		Partially fail: *ack*
E4SS/SailFin	11/12	Callee sends *bye* before receiving *ack*	Partially fail: *ack*
CAFE/SailFin	10/12	*cancel* and *inv200* race Callee sends *bye* before receiving *ack*	Fail - requests, responses

Table 1: Results of Manually Generated Tests

Category	SUT			
	BT/OCCAS		E4SS/OCCAS	
	Tests passed	Failed cases	Tests passed	Failed cases
Sunny Day	257/257		257/257	
Cloudy Day	830/1,335	56 re-*invite* race 98 *cancel*/*inv200* race 217 request after *bye* 134 504s after dialog terminated*	888/1,335	56 re-*invite* race 40 outstanding requests after *invFail* 217 request after *bye* 134 504s after dialog terminated*
Stormy Day	568/816	32 *canc200* instead of *cancDVR** 196 *cancel*/*inv200* race 15 *bye* after DVR* 5 create final response after DVR	760/816	32 *canc200* instead of *cancDVR** 5 outstanding requests after *invFail* 15 *bye* after DVR* 4 *bye* after *ackTimeout*

Table 2: Results of Automatic Testing

propagate responses to outstanding requests after receving an *invFail*. Both B2BUAs continue to propagate requests after receiving a bye. Finally, the tests revealed an OCCAS container bug, where the container sends 504 responses to outstanding requests after a dialog has terminated.

For the stormy day tests we discovered that OCCAS prevents sending a *bye* after receiving a *DVR* response, even though sending a *bye* is specified by RFC 3261. As for the cloudy day tests, E4SS failed to propagate responses to outstanding requests after receiving an *invFail* and BT failed to handle *cancel*/*inv200* races. Another OCCAS container problem is that it would sometimes send a *canc200* instead of the expected *cancDVR* in some DVR scenarios. BT fails to propagate a message because one of BT's SipSessions no longer exists. It isn't clear if this is due to a bug in BT, the SIP Servlet specification or the OCCAS container. Finally, E4SS failed to propagate *bye* messages after an *ackTimeout* event.

4.4.3 Discussion of Results

Our testing, using both manually and automatically generated tests, reveals problems with every application and container we looked at. From this we conclude that implementing a correct B2BUA is difficult and, moreover, that comprehensive testing is necessary to validate B2BUA behavior. Our results support efforts like SailFin CAFE and E4SS whose goals include providing a reusable, correctly implemented B2BUA that hides the inherent complexity from the programmer. Furthermore, our results indicate that comprehensive application-level testing supports validating container behavior and reveals ambiguities in the SIP Servlet specification. Finally, our results support our approach to model-based test generation. Not only do our automatically generated tests uncover the same B2BUA failures that our hand-crafted tests do, but they also uncover new failures resulting from unusual stormy day call flows.

5. DISCUSSION AND FUTURE WORK

SIP is becoming increasingly important as the dominant protocol for IP-based telecommunications and multimedia systems. The specification of SIP is informal and in some places incomplete, inconsistent, or ambiguous. SIP is complex already and its complexity is increasing, as the protocol is extended for a variety of reasons.

This study and our previous work [18] show that this situation is both dangerous and unnecessary. With judicious use of formal specification and automated analysis, the SIP protocol can be documented in a way that is guaranteed complete, consistent, unambiguous, and correct with respect to a variety of assertions. Critical SIP components such as B2BUAs can be defined with an equivalent level of quality. These models can be exploited to generate large, comprehensive test suites for real implementations. Given the number of bugs and other problems that our work has uncovered, it is safe to say that important goals such as interoperability and reliability cannot be achieved without formal methods.

Important future work is to continue to extend the scope of the model such that commonly used extensions to the SIP protocol are included.

The B2BUA models presented here prescribe deterministic behavior. However, in some cases, we made a choice from multiple legal alternatives. For example, in the cancel race of Figure 7, the B2BUA sends *ack* before *bye*, even though it is legal to send the *bye* without sending a previous *ack*. Further study is required in order to determine the criteria used to resolve such ambiguous situations.

It is our intention to expand the scope of our testing to include tests with greater lengths and maximum queue sizes. The machine we use for generating traces has 128 GB of RAM which limits us to a Spin verification depth limit of 71. Using the current model this results in a maximum test length of 17. By simplifying the model, for example, by constraining certain transition sequences to execute atomically, we should be able to greatly reduce the state space without compromising completeness, thereby permitting deeper searches and generation of longer test traces.

Another challenge we faced was analyzing test results. While our test platform unambigiously indicates how many tests pass and how many fail, it does not provide any insight into why tests fail. To do this we resorted to manually inspecting failure signatures extracted from log files, their associated test cases and the associated application code. Naturally, this becomes tedious and error-prone as the number of failure cases increases. This process would benefit from automated post-processing where similar failure signatures could be grouped thereby reducing the number of failures requiring investigation.

A goal of the SIP Servlet specification is to simplify life for the application developer. To this end, a SIP Servlet container presents an abstraction of the SIP stack to the programmer. This abstraction intersects with that of a SIP transaction-user but, in some cases, also presents a higher-level abstraction. The problem, as revealed by our testing, is that this abstraction is incompletely specified and has led container vendors to make their own, independent implementation decisions without fully understanding their implications. The result is that the SIP Servlet standard, whose goal is to support interoperability of applications across containers, does not achieve that goal. To address this situation, a topic for future research is to formally specify SIP Servlet container behavior and integrate the resulting model with our B2BUA models.

6. ACKNOWLEDGMENTS

We gratefully acknowledge Kristoffer Gronowski for supplying the BT example code, as well as the SailFin CAFE team for their publicly-available B2BUA implementation. Finally, we acknowledge, with gratitude and fondness, the great contributions and lasting memories of our late colleague Venkita Subramonian.

7. REFERENCES

[1] BEA. SIP Servlet API version 1.1, 2008. Java Community Process JSR 289. http://jcp.org/en/jsr/detail?id=289.

[2] C. Boulton and K. Gronowski. *Understanding SIP Servlets 1.1*. Artech House, April 2009.

[3] S. Donovan. The SIP INFO method, October 2000. IETF RFC 2976.

[4] ECharts for JAIN SIP (E4JS). http://echarts.org/.

[5] Formal models of SIP, 2010. `http://www.research.att.com/~pamela/sip.html`.

[6] M. Hasebe, J. Koshiko, Y. Suzuki, T. Yoshikawa, and P. Kyzivat. Example call flows of race conditions in the session initiation protocol (SIP). IETF RFC 5407, December 2008.

[7] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[8] *JAIN(tm) SIP Specification*. Java Community Process, 2003. Available from: `http://jcp.org/aboutJava/communityprocess/final/jsr032/`.

[9] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers. Session Initiation Protocol (SIP) basic call flow examples. IETF RFC 3665, December 2003.

[10] JUnit. http://www.junit.org/.

[11] X. Marjou, I. Elz, and P. Musgrave. Best current practices for a session initiation protocol (SIP) transparent back-to-back user-agent (B2BUA). IETF Internet-Draft draft-marjou-sipping-b2bua-01, July 2007.

[12] J. Rosenberg and H. Schulzrinne. An offer/answer model with the session description protocol (SDP), June 2002. IETF RFC 3264.

[13] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session initiation protocol, June 2002. IETF RFC 3261.

[14] Project SailFin. https://sailfin.dev.java.net/.

[15] SailFin CAFE project. https://sailfin-cafe.dev.java.net/.

[16] T. M. Smith and G. W. Bond. ECharts for SIP Servlets: a state-machine programming environment for VoIP applications. In *IPTComm '07: Proceedings of the 1st International Conference on Principles, Systems and Applications of IP telecommunications*, pages 89–98. ACM, 2007.

[17] V. Subramonian. Towards automated functional testing of converged applications. In *IPTComm '09: Proceedings of the 3rd International Conference on Principles, Systems and Applications of IP Telecommunications*, pages 1–12, New York, NY, USA, 2009. ACM.

[18] P. Zave. Understanding SIP through model-checking. In *Proceedings of the Second International Conference on Principles, Systems and Applications of IP Telecommunications*, pages 256–279. Springer-Verlag LNCS 5310, 2008.