Towards a Holistic Architecture for a SIP Test Framework

Teemu Kanstrén, Pekka Aho
VTT, Oulu, Finland
teemu.kanstren@vtt.fi

Abstract— **Testing traditionally focuses on specific aspects of a system separately, such as functional conformance, robustness and performance. In this paper, we present a protocol test automation framework that considers these different viewpoints as a whole. It starts with a common architecture for protocol testing at different scales. A set of test models on top of this architecture are presented for addressing the different types of testing. Each of these models builds on top of another, starting with conformance testing, followed by robustness testing and finally overall performance testing. We apply the framework to session initiation protocol (SIP).**

Keywords- Test automation, framework, sip, protocol testing

I. INTRODUCTION

Testing is a multifaceted discipline. It needs to verify various aspects of system behavior, including performance, robustness, and functional conformance. Different types of testing further have various coverage criteria, many specific to the type of testing and to the domain of the system under test (SUT). Covering these different types of testing and their different coverage criteria extensively can be very expensive. Commonly each system is also different, and creating largely re-usable test automation frameworks and test suites is difficult. In such cases, we commonly choose the most critical pieces of the SUT and target most of our coverage on those parts.

When systems are based on a set of standardized protocols, the test frameworks and test suites for those parts can be more extensively re-used and potential for extensive test automation frameworks is higher. A test framework for a standardized protocol can be applied on many different systems. In fact, protocol testing is an active field of research and various tools for testing different aspects of different protocols exist. A large scale example of this is the protocol conformance testing effort by Microsoft for testing more than 250 protocols [1]. For robustness testing, another example is protocol fuzzers which are a popular type of tool used to test robustness protocol implementations [2] and a popular research topic (e.g., [3, 4]). Fuzzers manipulate the protocol messages and the contained data to evaluate how the implementation can handle malformed inputs.

However, while there exist a wide range of protocol testing research and tools, these traditionally target a narrow part of the overall quality assurance for a protocol and the system built using that protocol. In this paper, we present a holistic test automation framework for protocol testing. It is aimed at supporting testing the protocol implementation at the basic protocol stack level, testing the protocol application to communication between two nodes, and to testing the application of the protocol to the overall communication in a large distributed system. It is also aimed at supporting conformance testing, robustness testing, and performance testing using model-based techniques with each testing type building on top of the previous one.

While some adaptation of the framework architecture is required in going from testing a protocol stack in isolation to testing the protocol use at larger scale, defining the overall common concepts enables us to build reusable components for the overall framework and to systematically build better quality into the different layers. With the different model-based test approaches we gain a diverse coverage for different types of testing, while reducing the costs in building the different models as layers on top of each other.

As our work on this has been practically performed in the context of the session initiation protocol (SIP), we present the application of the framework and its specialization for SIP as a running example throughout the paper.

The rest of the paper is structured as follows. Section II introduces the important background concepts and related works. Section III presents the components of our framework architecture. Section IV presents the set of test models for different types of testing and their composition. Section V discusses the concepts more broadly, and finally conclusions end the paper.

II. BACKGROUND AND RELATED WORKS

In this section, we give a brief introduction to relevant concepts for this paper, and present related works in protocol testing.

A. *Session Initiation Protocol (SIP)*

Throughout the rest of this paper, we will use session initiation protocol (SIP) as a running example to demonstrate the relevant concepts. SIP is a protocol used for signaling in setting up communication sessions. Typical usage scenario is Voice over IP (VoIP), where different endpoints use SIP based communications to signal call control flow. Various other protocols can then be used for the actual call (such as voice and video transport). We focus here only on control flow signaling which is what the SIP protocol is for. Figure 1 illustrates the basic call flow in such scenario.

Beyond this basic call setup shown in Figure 1, there can be various configurations such as the call going through one or more proxies, changing call details in mid-call (e.g., re-negotiating quality parameters), or several parties together negotiating a conference call. Figure 2 illustrates these different configurations from the testing perspective in four different cases (A-D).

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-370-4

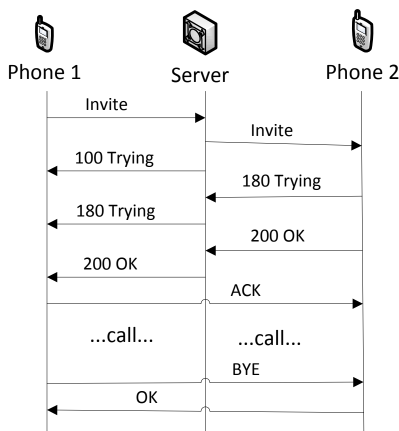

Figure 1. Basic SIP call flow.

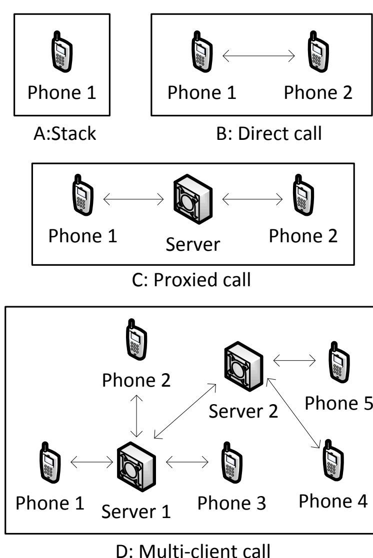

Figure 2. SIP system configurations.

In the rest of the paper, we will refer to these different scenarios in Figure 2 as scenarios A, B, C and D, or more generally scenarios A-D. In scenario A, we test the protocol stack in isolation as a single unit (or module). In scenario B, we test two devices communicating directly with each other. In scenario C, we test two devices communicating with a server connecting them. In scenario D, we have multiple devices all communicating together in a single call session, potentially across several servers as well. Sometimes these clients can also move during the call between servers (mobile clients). We will look at testing these in more detail in Section III with our test framework architecture and in Section IV with our test models.

B. *Model-Based Testing*

Model-based testing (MBT) is a concept we use widely in this paper. We follow the definition of MBT given in [5] as “generation of test cases with oracles from a behavioral model”. That is, the system is described using a behavioral model, in our case as a set of rules and actions, and test cases to exercise the behavior of interest are generated from these models by a test generator tool. SIP conformance testing is a case study quite often used in the MBT literature [5]. However, as we discuss in Section II.C, the existing work is mainly limited to conformance testing and in this paper, we discuss this more broadly with also applications to non-functional testing of robustness and performance.

As mentioned, our use of MBT is based on test models defining a set of rules and actions. In this case, the actions define some functions to be executed on the SUT. In the case of SIP these actions are typically sending SIP request messages. The rules in the test model define when each of these actions are allowed. Following these rules, a test generator can then produce a set of test cases following the test specification (the test model). As the generator follows the rules, it produces valid test cases. These are valid from the test model perspective, and thus we can also define, for example, a robustness test model for producing invalid data and invalid sequences. In such a case, the test model will describe the types of invalid data we are interested in.

In addition to the SIP requests in the test model, SIP responses are handled by the overall test framework as we will discuss in Section III. All these elements are linked to form the overall test framework, including test oracles at chosen detail level.

Figure **3** shows some example rules and actions for testing a single device SIP scenario. The solid boxes are the actions and the attached dashed boxes are the rules for those actions.

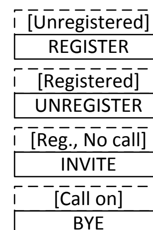

Figure 3. Example rules and actions.

In this case, the names in the action boxes in Figure **3** are different SIP request messages. For example, *invite* is a message used in SIP to establish a call between two parties. However, if a SIP proxy server is used, the parties (SIP user agents) have to *register* with the SIP server to allow the call (the “registered” rule in Figure 3). This is only allowed if not yet registered, and the registration action updates the model state to registered. Once registered, *invite* is then allowed. After a call is established, it can be terminated using the *bye* message.

The model in our case is a form of a *model program,* implemented to send the SIP messages for these actions and to maintain the state of the test client. This state is then used in the rules to define when actions are allowed. This is a common approach to MBT as described in e.g. [5, 1].

In addition to sending request messages, the tester needs to be able to handle other protocol messages as well. For example, the *Trying*, *OK* and *ACK* messages shown in Figure **1** cannot be expressed as actions as they are not actions initiated by the test client (or the test model). Instead,

Copyright (c) IARIA, 2014.

the test framework must be able to process such received messages from the test target(s), update its state (e.g., call established) and to check that required responses are received correctly (such as *Trying* message when expected). That is, the basic functionality of the protocol should be a part of the test framework and the test model is built on top of this. Verification of this test framework then comes from its application to several test targets, each providing a verification of the applied test framework itself.

C. SIP Protocol Testing

One the largest efforts in protocol testing is the Microsoft protocol documentation assurance effort described in [1] for testing more than 250 different protocols to ensure documentation quality and regulatory conformance. The process and tools to perform this validation have been described in detail in [1]. Conformance testing for a protocol includes describing the expected normative behavior of the protocol, meaning the allowed and required messages, their sequences and the data values. In the Microsoft case, both model-based and manual test creation methods were used, which in our experience reflects the general good test automation practices. In such a case, a test model reflects the protocol behavior, and test cases can be generated with a MBT tool from this model. Test execution is built on top of a component based adapter layer, which also forms a basis for creating manual test cases as required. Our general test automation architecture adapts elements from this and includes addressing also larger scale distributed systems and different robustness and performance testing. It thus also enables more complex and realistic test scenarios.

An example test automation framework for SIP conformance testing is presented in [6]. In this case, the framework provides a simulated SIP service environment that can be configured to provide different responses and services for SIP user agents. For example, an emulated user agent can be configured to perform call forwarding for a specific user, allowing the tester to focus on scenarios that make use of such SIP services. The work in [6] is aimed at conformance testing of different services built on top of SIP, and non-functional testing (performance) is left out of scope.

A test automation framework for performance testing is presented in [7]. This framework uses existing SIP platforms and tools such as SailFin [8] and SIPp [9] to generate different types of traffic to test performance of SIP agents. This includes traffic bursts, linearly increasing traffic and other such usage profiles. Mentioned problems include difficulty to implement complex interactions between agents as well as control of generated traffic due to limitations of the third party tools used for generating traffic. That is, the external tools used do not have support for the required level of control in fine grained performance testing. We use similar traffic profiles as part of our performance models, and integrate these with conformance and robustness test models.

A test automation approach based on passive monitoring of operational SIP based systems is presented in [10]. In this case, the idea is to describe the system expected behavior as a set of formalized properties, and use operational monitoring to assess whether the observed system behavior matches this expected specification. A similar approach taken in [11] provides a general specification of a protocol, creating a model of system behavior in different phases starting from observing network traffic to grouping it as transactions and dialogs. Finally, these are compared to the set of rules given in the specification. We do not explicitly do this type of passive monitoring as we also perform active generation of request messages. However, the part of our test framework related to listening for response messages and asserting the overall system behaviour based on those responses and their relation to test model state uses similar concepts.

During performance testing, we have to collect various metrics on system performance to evaluate how the different actions and parameters applied affect the performance. These measurements are collected by deployed probes, which can be located on the different nodes in a distributed system, or measuring the connecting network. For example, [12] describes using numerous measures collected such as CPU load, network load, interrupts and context switches on SUT nodes as a basis for a performance model. These are combined to form a detailed view of how the different components in the system affect the system performance. A high-level, specific, metric for performance testing of networked services is suggested in [13] as throughput. Throughput here means the number of requests (or transactions) performed on the system over a given time unit (such as a minute). In summary, we need both a high-level definition of what system overall performance means for us, as well as means to find the bottlenecks when relevant. In our case, we use the number of SIP messages processed as a basic metric, and focus using more detailed probes where necessary.

For robustness testing, a stateful fuzzer for SIP is presented in [3]. This is based on two components: syntax fuzzer and state evaluator. Besides the traditional data fuzzing and checking of aliveness of SUT, this approach also checks that the correct responses (state transitions) are observed on the SUT and that the data provided in these responses is valid. Checks on the SUT are performed using basic protocol messages to verify the SUT is alive and in correct state. To describe the SUT behavior in [3], a state-machine is learned from observing the protocol implementation. Messages are associated to different simulated clients and separate state-machines are upheld for each, to check responses and transactions against. Different combinations and mutations of messages are used to provide fuzzed messages where different field values are modified using protocol knowledge for invalid messages and where some field can be repeated or otherwise the overall structure fuzzed.

A method coverage analysis for protocol fuzzing is presented in [4]. Constraints for message formats and processing are defined based on protocol specification analysis. The constraints are formally specified and a test generator is used to generate fuzz tests to cover them. Sometimes effectively fuzzing different parts of the protocol and interactions may require accessing deeper parts of the

protocol state-machine, requiring techniques similar to [3] in initiating the protocol to initial phases before fuzzing.

A fuzzing tool for SIP softphones is presented in [14]. This is based on defining templates that identify specific data values to fuzz for different SIP messages. The SUT is then driven to specific states using scripts and these fuzzed messages are injected at these locations in the protocol flow. Additional generic SIP specific fuzzing algorithms are also used, such as re-ordering of SIP headers and defining patterns of specific SIP data vulnerabilities. Finally, the SUT is monitored to evaluate whether it crashes or produces invalid responses.

In [15], performance and robustness testing are combined to evaluate robustness of the system under heavy load. Different types of attacks against SIP based systems are defined, a specific valid load is generated on the system, after which different attack types are launched. System performance is measured before, during, and after the attack. The results indicate system performance in face of attacks under different loads both temporarily and long term.

III. SIP Tester Architecture

To address both the need to test the different configurations (A-D) described in Figure 2, and the different types of testing we are interested in (conformance, performance, robustness) we have to consider both a test framework architecture for executing tests and a set of different test models for generating tests. The architecture needs to support both manual test creation and execution, as well as test generation (and execution) from test models. This means considering the different aspects similar to discussed in [1] but also considering a broader context of testing interacting systems and not just the protocol stack. The following subsections describe our test framework architecture, which is illustrated in Figure 4 (S illustrates the shared state). The test models will be described in Section IV.

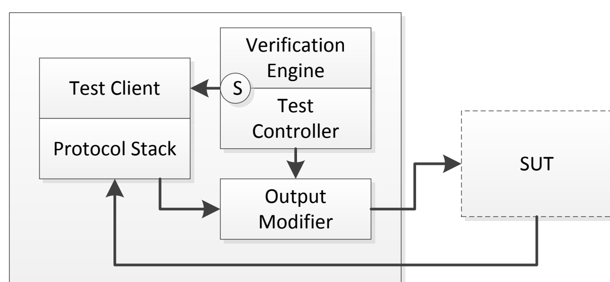

Figure 4. Component Architecture.

A. Protocol Stack

To be able to use the tested protocol and evaluate the SUT against it, the test framework also has to implement the basic protocol stack. This consists of parsing and creating messages, and delivering these across the network. Optimally, existing code or an available stack such as an open-source implementation can be re-used. However, sometimes this is not possible due to issues related to the test framework requiring high degree of control and observability over the protocol, which may not have been factors in building some of the available tools and libraries.

In our case, we have implemented our test environment in Java and used the Jain-SIP protocol stack implementation [16]. In this case, our experience has been that using this type of an open-source stack can save some time initially in getting started, but the control and visibility over protocol details is limited. Also, some of the functionality is limited such as lack for proper SIP authentication support. Lack of control and visibility is due to regular user not requiring detailed access to protocol manipulation, and attempting to abstract some parts of the protocol from the user where possible, such as SIP dialog control. However, in order to build a protocol tester, we need to understand the protocol and be able to verify its specifications in sufficient detail. For this reason, as we need to know and understand the details anyway, we find it better to actually implement the stack, at least for the most parts on our own.

As such, we conclude that to have a robust and powerful base for a protocol tester we need our own highly reliable, configurable and observable stack. In an optimal case, we can write one. In practice, resources for this may not be available especially for more complex protocols. In such cases, we need to look at our options with available stacks or with directly using an existing protocol client (such as a SIP softphone or SIP tester such as SIPp [9]).

B. Test Adapter

On top of the protocol stack, a test adapter capable of performing stateful transactions against a test target has to be implemented. The test adapter should support higher level functions such as initiating and terminating calls, managing responses from the SUT, and maintaining the protocol state for itself according to these actions and responses. In this sense, it is similar to a SIP softphone but does not have to implement a user interface as it is controlled by the test tool.

There are many existing SIP softphones available (such as Twinkle and Linphone), and these can be used if programmatic control over them is available. Their usefulness depends largely on the extent of remote control supported, and the ability to observe details about the results and responses from the SUT. In our experience, most of the actual SIP clients have limited support or no support at all for such control. However, SIPp is a SIP performance test tool that does provide many such features. While it is limited in its support for detailed control and visibility for conformance or robustness testing, it can be a useful starting point for fast test automation prototyping for suitable parts.

Optimally, the test adapter provides a simple and fast network interface to control it, create protocol messages, and receive notifications about SUT responses. Separating the adapter from the rest of the test framework as a separate networked service allows for using any tools available to implement it and to reuse the controllers and output modifier(s), as well as any existing tools and libraries for different platforms to build adapters. For example, SIPp provides a UDP communication and control interface, and similar interfaces are also used by other successful test frameworks, such as Selenium Webdriver for web

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-370-4

applications (which creates and sends JSON requests over the network). This also allows building different controllers on different platforms, using the same adapters, when required.

To summarize, optimally the test adapter provides a stateless control and observation interface to the underlying protocol. This allows the test controller to create various types of protocol messages and observe the results at a selected level of detail.

C. *Test Controller*

To produce actual, executable test cases, a test controller is required. In the case of manually scripted test cases, this executes the given scripts using the test adapter. In the case of using a test generator, this generates the scripts based on a test model and executes them using the test adapter. In a distributed multi-client scenario similar to scenario D in Figure **2**, the controller manages several adapters in parallel.

The controller upholds the test state. When testing scenarios B and C, this means keeping track of the current state of the test client. When testing scenario D, this means tracking all the different test adapters and their connections. It shares this state with the verification engine, which makes assertions about the correct responses received from the SUT based on the controller actions. The adapters can be distributed across the network or on a single machine.

D. *Verification Engine*

A central part of testing is the test oracle, which is a component used to verify that the expected properties hold at the selected points of time in the testing process. For different types of tests, different types of verification engines are required. In conformance testing, the received responses are typically checked after specific actions (such as initiating a call) have been performed. In performance testing, we are interested in measuring the response times to the messages and collecting various metrics on the SUT to assess the impact of test load. In robustness testing, we are interested in observing the state of the target system and using this information to make assertions about how invalid inputs impact the SUT state and responses.

The verification engine performs these various checks to evaluate test results during system operation. The checks performed can be split into passive and active checks. Active checks are performed as specific checks at specific points in the test execution, e.g., to establish that response messages such as TRYING, ACK, and OK are received when required and contain the correct data related to their associated requests. Examples of passive checks would be to track that messages that require a dialog should not be received outside dialogs, or to continuously ping the SUT to ensure it is alive.

For active verification, the verification engine has to share state with the test controller to be able to make the required assertions. For some forms of passive verification such as pinging the SUT this is not required but the verification engine still has to communicate back to the test controller to notify of any failed checks. This then fails the executed test and reports the results back to the user.

E. *Output Modifier*

The output modifier is used by the test controller during robustness testing to invoke specific modifications on the input messages produced by the test client. Test data is passed through the output modifier and forwarded to the SUT. During robustness testing, the test controller can enable different types of fuzzing patterns to be applied to the data, while during other types of testing the output can be forwarded as is.

IV. TEST MODELS

This section discusses different types of test models we use for testing conformance, performance and robustness and how these relate to the architecture. The basis is the conformance test model, and the other models specialize and extend it in different ways. These models are also different depending on the type of scenario addressed. For scenario A, the models target the protocol stack functionality, performance and robustness. For scenarios B and C, the models target the interactions of a single client with other nodes in the network. For scenario D, the models target the overall system behaviour. In our testing, we have focused at the level of scenarios B-D, and assume that the stack will be tested sufficiently as part of these test cases and separately at unit and component testing level by the developers.

While describing these as test models implies our preference towards model-based test generation, it is equally possible to use the information in the models to create test cases manually. The test model is used by the test controller as a part of the overall test framework.

A. *Conformance Test Models*

The conformance model describes the expected functional behavior of the SUT as described by its specification. In the case of scenarios B and C, this is the SIP RFC 3261 [17]. In scenario D, the specification describes the expected behaviour of the overall system and its interactions. Table I lists the basic rules and actions for the scenario B and C model. Table II lists the basic rules and actions for the scenario D model.

TABLE I. SCENARIO B AND C MODEL ELEMENTS.

ID	Rules	Action
S_R	Unregistered	Register
S_U	Registered	Unregister
S_I	Registered, No Call	Invite
S_C	Registered, Calling	Cancel
S_O	Registered	Options
S_B	Registered, Call On	Bye
S_M	Registered	Message

The actions in Table I are basically the SIP request messages as described in [17]. For scenario B and C, the test model is focused on generating requests to interact with the SUT. In addition to this, the test framework must handle the responses from the SUT, such as failures, errors, and successes. It must also provide its own responses to such messages when required, such as the ACK message to the OK for INVITE. In our test framework architecture, the test

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-370-4

controller executes the tests and maintains the relevant state information to manage the responses. The state information describes the protocol interaction state including registration status, call invite status, and ongoing call status.

Additionally, the model has to define the valid data values in order to properly evaluate the conformance of the system and to expect the correct responses.

TABLE II. SCENARIO D MODEL ELEMENTS.

ID	Rules	Action
SD_R	Phones < MAX_P	Register Phone
SD_U	Phones > 0	Unregister Phone
SD_I	Free Phones >= 2	Initiate Call
SD_T	Busy Phones > 0	Terminate Call
SD_S	Servers < MAX_S	Start Server
SD_X	Servers > 0	Stop Server

For scenario D, our model is focused on testing high-level interactions of different communicating entities in the overall system. As illustrated in Figure **2**-D, there may be several servers and phones or other SIP user agents in the system, connected in various ways. The actions shown in Table II allow dynamic creation of such configurations and to establish and terminate connections between the nodes. Test framework/model state in this case consists of the nodes and their status. The state for SIP clients is the same as for scenario B and C, including the connected nodes in a call. The SUT in this case is the overall system interactions. The model of a SIP phone described in Table I can be used to represent a phone, which is controlled by an overall system test model.

The test oracles for the verification engine in the conformance model are checks of the test model state against the SUT state. This means we will check that when the SUT should accept a call, the *invite* message passes and the call is established. Similarly, when registration should succeed, the response is expected to be a success. After an *invite* has been performed but before the call is started, a *cancel* message should stop the call from starting. Once a call is started, a *bye* message should stop the call and allow re-starting a new call with another *invite*. Similar checks are performed at every point during the execution of a generated test case with regards to every request message performed, and every response message received. As the test controller maintains the system state according to performed actions, it can automatically verify all these properties with minimal effort.

B. *Robustness Test Models*

We define robustness according to the IEEE glossary as "The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions." [18]. In our robustness models, we consider both invalid interaction sequences as well as invalid input data. We represent the interaction sequences with similar models to the conformance model, and the input data using the output modifier configured with message modification patterns.

Defining invalid inputs and their expected responses explicitly can be challenging as many protocol definitions have required parts (must) and optional parts (may). In many cases, also ambiguities exist and different implementations have taken different interpretations of these. Due to this, we classify any issues observed either as warnings or errors. Warnings are issues related to potential issues, while errors indicate clear problems in the implementation.

Our robustness models specialize the conformance models by changing specific parts of the models to produce invalid interaction sequences and data. We call these specializations *robustness model patterns*. To do this, several mechanisms of our MBT tool [19] are used. These are *model composition*, *startup-sequences*, and *model slicing*.

Model composition refers to combining several separate model objects together. The normative model elements are expressed using the conformance models as shown in Table I and Table II. The invalid sequences are in a separate model object that is combined with the conformance model by the test generator. If we call the conformance model C and the robustness model R, the actual base test model T is their union, $T = C \cup R$. Example robustness rules and actions for the model in Table I are shown in Table III.

TABLE III. ROBUSTNESS MODEL ELEMENTS.

ID	Rules	Action
SR_R	Registered	Register
SR_U	Unregistered	Unregister
SR_I	SUT in call	Invite
SR_C	SUT not called	Cancel
SR_O	Not registered	Options
SR_B	No call with SUT	Bye
SR_M	Not registered	Message

Startup-sequences are used to initiate the SUT into a state of interest for the robustness test pattern. These can be used to drive the initial test generation to a specific state by making the generator take a specific set of steps before starting its own algorithmic generation. For example, we can define one from the model in Table II as "Register, Invite", which means the generator will start all test cases with this sequence and thus the tests will start from a valid registered state with a valid initiated call started with the SUT.

Model slicing allows us to define which parts of the model are to be used for generation and how much they are used. If we take the test model $T = C \cup R$, the sliced model S is then a subset of T, $S \subset T$. The slice can either remove a step from T completely or limit the number of times it can appear in T. The slice does not affect the startup sequence and the startup sequence does not affect the slice, allowing these to independently define different elements of the robustness test scenario. For example, the slice may forbid any *invite* messages but the startup sequence can still use them as the slice only affects parts after startup.

As an example, let us show a pattern for robustness testing registration handling for a single node during an ongoing call. This is illustrated in Table IV. In this case our model is $T = C \cup R$ as discussed, where C equals the model shown in Table I and R equals the model shown in Table III. Using this pattern configuration, the generator will generate sequences that always start with valid *register* and *invite*

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-370-4

messages. This is then followed by any allowed messages except *bye*, which is forbidden by the slice. Notice that this pattern also includes and allows all steps in R.

TABLE IV. EXAMPLE ROBUSTNESS PATTERN.

Element	Value
Startup	S_R, S_I
Slice	!S_B

Additionally, the output modifier patterns change the created messages in various ways:

- Duplicate headers and message parameters
- Remove headers and message parameters
- Modify headers and message parameters

When running robustness tests, the test oracle definitions require some special attention. We can define how each of the invalid input producing steps should impact the SUT operation and state. Typically this would be to ignore the input with invalid data or sequences. In other cases we can also define specific impacts and update state accordingly. For example, if we consider security vulnerability scanning as part of robustness, some specially crafted input for such tests can be considered valid but should have no unwanted side-effects. In these cases, we should update the state in those steps, and evaluate the oracles accordingly.

However, due to different specification interpretations or desire to provide flexibility in communicating with other endpoints of varying quality, the responses to some of the robustness input may differ, and the SUT may accept some of them as valid. For example, duplicate headers produced by the output modifier may be interpreted as an issue or not in the SUT. In such cases, we can choose to disable some of the more strict oracles for those models and tests generated from them and focus on the more generic ones to check the system for generic properties such as not crashing or ending up in a bad state for any node, or consume excess resources over time.

For such cases, the test oracles that make such assertions can also be represented as their own model object(s). The test model T then becomes $T = C \cup R \cup O$, where O is the model object holding these oracles. By removing O, or parts of it, from the equation, the oracles can be disabled as required. In any case, as mentioned the generic test oracles are always valid, such as pinging the SUT and checking error codes. These can also be configured to run at specific intervals to check for general properties in, e.g., long running performance tests.

C. *Performance Test Models*

Our performance test models are combinations of different configurations of the conformance and robustness test models. They are intended to explore the performance limits of the SUT under different environment and load conditions. They represent different usage scenarios for different types of user profiles in the system. Basically we use our conformance test models as valid client type and the robustness model instances as another (invalid) client types.

Similar to [7], we use different types of traffic patterns for specific user profiles, such as traffic bursts and linear increase in traffic. We start with our conformance test model clients as the reference set of providing the system performance under these different types of varying load. Once we have this model, we apply our different types of robustness test patterns as clients to represent invalid data, similar to attacks discussed in [15]. Finally, we re-run the initial reference test set with the conformance test clients for valid data and compare the results with the initial run before invalid data was used. We then use these results to give us a model for the overall system performance under different types of load.

V. DISCUSSION

While we have described a composition of model objects as one for the conformance test model and another for the robustness model, and using model composition and scenario slicing to create robustness patters, it is possible to further decompose these models as much as desired. The actual composition we support is not limited in the number of model objects and thus the operation can be seen as $T = C_N \cup R_N \cup O_N$, where N refers to having any number of these in the end result. However, in practice we have found that a smaller number of model objects makes it much easier to manage the overall set of patterns. For a protocol such as SIP, where there is a relatively small set of potential messages having one C, R, and O has worked well for us. For more complex protocols it may be necessary to split these further, for example, to make model composition and slicing for different test purposes and patterns easier.

As it is, in the work presented in this paper we have so far focused on the SIP protocol. More generally, we see the approach applicable more widely to different protocols and networked systems. The architecture, conformance and robustness models, and robustness pattern definitions simply need to be adapted to the new specifications. This means creating suitable protocol adapters, and defining the valid and invalid sequences to be used for test generation. That is, the overall framework and modelling approach is intended to be easily specialized for a variety of protocols.

So far our test execution and generation has focused on a single host environment. For now we have found this to be sufficient for our testing needs, as modern systems can run numerous clients in parallel even on a single multi-core system. However, more distributed systems are needed to address more realistic usage scenarios as well as to scale to very large scale testing. This would also include modelling different concurrent users more realistically in terms of latencies, burst traffic, occasional robustness scenarios interleaved with conformance scenarios, and other similar attributes. In our previous work, we have investigated distributed model-based test generation [20]. In the future we hope to extend also our test framework to make use of this type of strategies, including distributed (cloud) deployments.

Another interesting point of extension for this work is to include actual specific security related attacks to the robustness patterns. Currently we mainly use fuzzing related patterns, which change the inputs in different ways and evaluate the SUT robustness. Additionally, specific inputs to target specific vulnerabilities in underlying backend systems

Copyright (c) IARIA, 2014.

could be of interest. Generally, we are also looking at extending our work to cover more aspects as well, such as quality of service for the call under different conditions.

When executing large scale test cases, the biggest issue we observe is making reliable overall assertions about the system state. For this reason we have at large scale mostly focused on observing overall performance across large sets of users. However, when issues are observed from such large scale tests, debugging them for root cause analysis can be very challenging due to large numbers of different types of interacting clients and servers. While these issues are not specific to our approach but to large scale testing in general, in the future we hope to explore better solutions to these issues as well and integrate these into our approach as easily applicable solutions.

VI. CONCLUSIONS

In this paper, we have presented an architecture, a set of test models and ways to compose and slice these to form a holistic test framework for the SIP protocol. Our framework supports conformance testing, robustness testing, and performance testing, with each part building on top of the previous, allowing for an effective and extensive implementation. We are currently extending the work by collecting a wider set of patterns building on top of the framework presented in this paper, as well as applying these in industry case studies. In the future, we are interested in refining this work based on practical applications and experiences, and extending it to more diverse set of protocols.

VII. REFERENCES

[1] W. Grieskamp, N. Kicillof, K. Stobie and V. Braberman, "Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology," *Journal of Software Testing, Verification and Reliability,* vol. 21, no. 1, pp. 55-71, 2011.

[2] M. Sutton, A. Greene and P. Amini, Fuzzing: Brute Force Vulnerability Discovery, Addison-Wesley, 2007.

[3] H. J. Abdelnur, R. State and O. Festor, "KiF: A Stateful SIP Fuzzer," in *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications (IPTComm2007)*, 2007.

[4] P. Tsankov, M. T. Dashti and D. Bashin, "Semi-Valid Input Coverage for Fuzz Testing," in *International Symposium on Software Testing and Analysis (ISSTA2013)*, 2013.

[5] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufman, 2006.

[6] C. Caba and J. Soler, "An IMS Testbed for SIP Applications," in *Principles, Systems and Applications on IP Telecommunications - IPTComm '13*, 2013.

[7] L. Roly and L. Schumacher, "SIP Overload Control Testbed: Design, Building and Validation Tests," in *IEEE Consumer Communications and Networking Conference*, 2011.

[8] SailFin, "SailFin Project," [Online]. Available: https://sailfin.java.net/. [Accessed 23 4 2014].

[9] SIPp, "SIPp," [Online]. Available: http://sipp.sourceforge.net. [Accessed 23 4 2014].

[10] F. Lalanne and S. Maag, "A Formal Data-Centric Approach for Passive Testing of Communication Protocols," *IEEE/ACM Transactions on Networking,* vol. 21, no. 3, pp. 788-801, 2013.

[11] D. Bao, D. C. Carnì, L. D. Vito and L. Tomaciello, "Session Initiation Protocol Automatic Debugger," *IEEE Transactions on Instrumentation and Measurement,* vol. 58, no. 6, pp. 1869-1877, 2009.

[12] P. Xiong, C. Pu, X. Zhu and R. Griffith, "vPerfGuard : an Automated Model-Driven Framework for Application Performance Diagnosis in Consolidated Cloud Environments," in *International conference on performance engineering (ICPE '13)*, 2013.

[13] M. H. Sørensen, "Use Case-Driven Performance Engineering without "Concurrent Users"," in *International Conference on Performance Engineering (ICPE '13)*, 2013.

[14] S. Taber, C. Schanes, C. Hlauschek, F. Fankhauser and T. Grechenig, "Automated Security Test Approach for SIP-Based VoIP Softphones," in *Internation Conference on Advances in System Testing and Validation Lifecycle (VALID2010)*, 2010.

[15] P. Steinbacher, F. Fankhauser, C. Schanes and T. Grechenig, "Work in Progress : Black-Box Approach for Testing Quality of Service in Case of Security Incidents on the Example of a SIP-based VoIP Service," in *Principles, Systems and Applications of IP Telecommunications (IPTComm2010)*, 2010.

[16] "JSIP: Java API for SIP Signaling," [Online]. Available: https://jsip.java.net/. [Accessed 24 4 2014].

[17] IETF, "RFC 3261 SIP: Session Initiation Protocol," IETF, 2005.

[18] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990,* 1990.

[19] T. Kanstrén, "OSMO Tester Home Page," April 2014. [Online]. Available: http://code.google.com/p/osmo. [Accessed April 2014].

[20] T. Kanstren and T. Kekkonen, "Distributed Online Test Generation for Model-Based Testing," in *Asia Pacific Software Engineering Conference*, 2013.