Model-Based Testing of a WAP Gateway: an Industrial Case-Study

Anders Hessel and Paul Pettersson

Department of Information Technology, Uppsala University, P.O. Box 337,
SE-751 05 Uppsala, Sweden. E-mail: {hessel,paupet}@it.uu.se.

Abstract. We present experiences from a case study where a model-based approach to black-box testing is applied to verify that a Wireless Application Protocol (WAP) gateway conforms to its specification. The WAP gateway is developed by Ericsson and used in mobile telephone networks to connect mobile phones with the Internet. We focus on testing the software implementing the session (WSP) and transaction (WTP) layers of the WAP protocol. These layers, and their surrounding environment, are described as a network of timed automata. To model the many sequence numbers (from a large domain) used in the protocol, we introduce an abstraction technique. We believe the suggested abstraction technique will prove useful to model and analyse other similar protocols with sequence numbers, in particular in the context of model-based testing.
A complete test bed is presented, which includes generation and execution of test cases. It takes as input a model and a coverage criterion expressed as an observer, and returns a verdict for each test case. The test bed includes existing tools from Ericsson for test-case execution. To generate test suites, we use our own tool CO✓ER— a new test-case generation tool based on the real-time model-checker UPPAAL.

1 Introduction

Testing is the dominating technique used in industry to validate that developed software conforms to its specification. To improve the efficiency of testing, model-based testing has been suggested as an approach to automate the generation of the tests to be performed during testing. In model-based testing, a model is used to specify the desired behavior of the developed software, and the testing efforts aims at finding discrepancies between the behavior of an implementation and that specified by the model. This process can be automated by applying a test generation tool to produce the test to be used during testing, and by automating the execution and validation of the tests using a test-execution tool.

Model-based test generation techniques have been studied thoroughly in the research community [Tre96,HLSU02,LMN05] and several applications to industrial systems have been reported, e.g., [BFG$^+$00,LMNS05]. There is much less literature describing industrial applications of model-based testing techniques for real-time systems, i.e., systems that must react to stimuli and produce output in a timely fashion, i.e., real-time systems including, e.g., clients or servers using protocols with timing.

In this paper, we present experiences from applying a model-based approach to perform black-box conformance testing of a gateway developed by Ericsson. The gateway is used to connect mobile phone clients using the Wireless Application Protocol (WAP) with the Internet. We present how the specification of the transaction layer (WTP) and the session layer (WSP) have been described in the modeling language of timed automata [AD94]. The specific protocol used in the model is a connection oriented version, and the model includes scenarios where several transactions are associated with a session. In addition to the components constituting the WAP stack of the gateway, the model also contains automata modeling abstract behavior and assumption imposed on the components in its environment, such as a web sever and terminals using the gateway.

A specific problem when modeling the WAP protocol is to model the sequence numbers, called Transaction Identifiers (TID), used in the exchanged packages, called Protocol Data Units (PDU). The protocol typically makes use of several TIDs with a domain of size 2^{15} using a sliding window of size 2^{14}. To make automatic analysis feasible, previous models of the protocol, used for model-checking the specification, have introduced a limit on the maximum allowed TID values, assuming that all behaviors of the protocol will be covered with a small maximum TID value [GB00]. We take a different approach and introduce an abstraction technique to handle TID values. It maintains the concrete TID values, so that they can be accessed in the abstract test-cases generated from the model.

To specify how thorough a test suite should test the WAP gateway, we select test cases following some particular coverage criterion, such as coverage of control states or edges in the model. As our model contains the environment of the system under test, a test-case generation tool can find out how the environment should behave to drive the system under test in a desired direction to fulfill a given coverage criterion. To formally specify coverage criteria, we apply results from our previous work [BHJP05], where we have proposed to use observer automaton with parameters as a formal specification language for coverage criteria. We show that the observer language is expressive enough to specify the coverage criteria used to test the WAP gateway.

To perform the actual testing, we have built a complete test bed that supports automated generation and execution of tests. It takes as input a network of timed automata and an observer automaton, and uses our tool UPPAAL CO✓ER to generate an abstract test suite. UPPAAL CO✓ER is a test generation tool based on the UPPAAL model checker [LPY97]. The test suite is compiled, by a tool named tr2mac [Vil05], into a script program that is executed by a test execution environment named TSC2, developed by Ericsson. TSC2 executes a script program by sending PDUs to the WAP gateway and observing the PDUs received in response. If unexpected packages or timing is observed the discrepancy is reported to a log file, and the testing proceeds with the next test case in the suite.

From testing the WAP gateway, we report the effect of executing test suites generated from extended versions of the edge, switch, and projection coverage criteria. In particular, we present two discrepancies between the model and the WAP gateway found during testing, and observe that both these problems were found in the rather small test suites satisfying the edge coverage criterion.

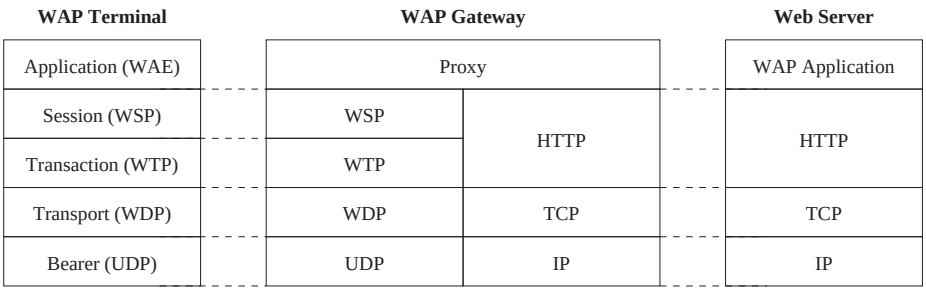

Fig. 1. WAP Gateway Architecture.

The rest of this paper is organized as follows: in the next section we give an informal description of the studied WAP gateway. In Section 3 we present the abstraction used to model sequence numbers in the model, presented in Section 4. In Section 5 we present the test generation and execution tools, and results from testing the gateway. We conclude the paper in Section 6, and then presents detailed models in an Appendix.

2 Wireless Application Protocol

The Wireless Application Protocol (WAP)[1] is a global and open standard that specifies an architecture for providing access to Internet services to mobile (hand-held) devices. It is typically used when a mobile phone is used to browse Web pages on the Internet, or when pictures or music are downloaded to a mobile phone. The WAP standard specifies both a protocol and a format, named Wireless Markup Language (WML) being the WAP analogy to HTML used by HTTP. The WML format also has a compressed binary encoding (WML/Binary) that is used during wireless communication to save bandwidth.

An overview of a WAP gateway architecture is shown in Figure 1. A WAP gateway converts between the WML content on the HTTP side, and WML/Binary on the mobile side. It also serves as a proxy for translating WAP requests to Internet protocols (e.g., HTTP). The WAP side of a gateway typically consists of the following protocol layers: Wireless Session Protocol (WSP), Wireless Transaction Protocol (WTP), Wireless Datagram Protocol (WDP), and a bearer layer such as e.g., GSM, CDMA, or UDP. The internet side usually consists of the protocols Hypertext Transfer Protocol (HTTP), Transmission Control Protocol (TCP), and Internet Protocol (IP). The WDP layer and a bearer on the WAP side corresponds to the TCP/IP layers on the Internet side. The security layers Wireless Transport Layer Security (WTLS) on the WAP side and Secure Socket Layer (SSL) on the Internet side are optional and omitted in Figure 1.

The WAP specification defines two roles in the protocol. The part that starts a transaction is called *initiator*, and the other part is called *responder*. For example, a mobile

[1] The Wireless Application Protocol Architecture Specification is available at the web page `http://www.openmobilealliance.org/tech/affiliates/wap/wapindex.html`.

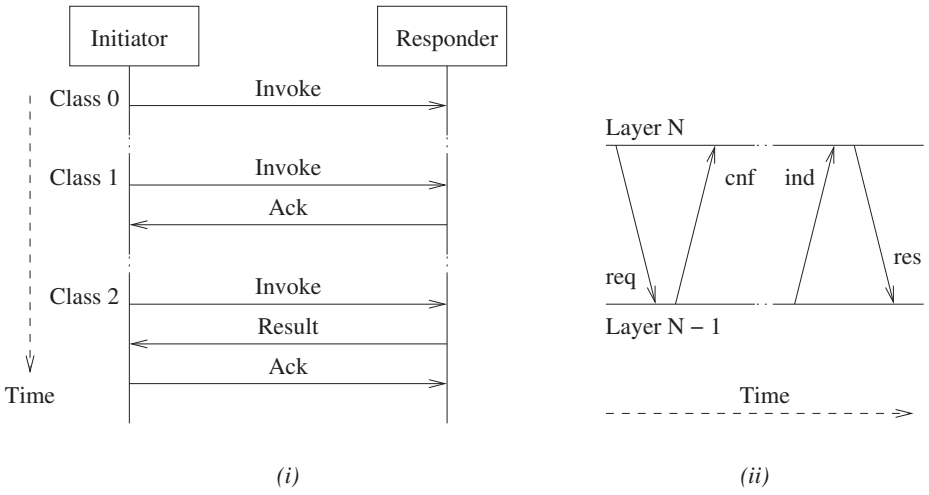

Fig. 2. The three WTP transaction classes (i) and signaling terminology (ii).

device is the initiator when it access data from the Internet, but it can also be the responder if a (push) initiator sends out a message to the mobile device. Communication between initiator and responder is divided in three types of transaction *classes*, ranging from class 0 in which no acknowledgments are used, to class 2 that also send acknowledgments of results. The desired behavior the classes is shown in Figure $2(i)$.

In Figure $2(ii)$ the terminology for message signaling between layers in the WAP stack is illustrated. An upper layer requests (req) a service from the layer below, which then confirms (cnf) that the request has been handled. A message from a peer layer is indicated (ind) by the layer below and the upper layer response (res) to notify that the message is accepted. Some message types do not require response nor confirmation.

The data structures used to and from an upper layer in the WAP stack are called Service Data Units (SDUs). The WTP layer has its own peer messages, e.g. acknowledgment, and it conveys SDUs to and from its upper layers. The behavior of a WTP layer is specified in the WAP specification as a state machine. In practice, every new transaction is a new instance of the WTP state machine, and there can be many simultaneous transactions.

The interfaces of a WAP stack layer are called Service Access Points (SAP). In this paper the Transport SAP (T-SAP), the Transaction SAP (TR-SAP), and the Session SAP (S-SAP) will be referenced.

Session Layer: The WSP layer is responsible for handling sessions in the WAP protocol. A session is a collection of transactions from the same user that can be treated commonly. An example of a case when a session is convenient is when a user logs in to a Web server. When logged in, the session is used to authenticate subsequent requests. If the session is disconnected (or aborted) all the transactions in the session will be aborted.

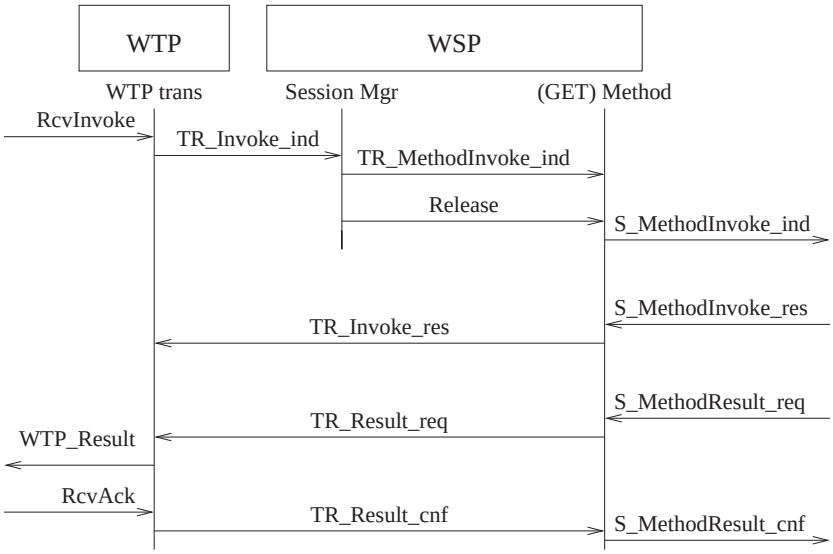

Fig. 3. Messages in the responder during a WSP GET request.

The session layer consists of two parts: a *Session Manager* that handles the connect and disconnect of a session, and a set of processes handling outstanding HTTP requests called *Method*s. For example, at a GET request a *GET-Method* process is spawned off to handle the request. A Method is associated with a WTP transaction and is terminated when the transaction terminates. In Figure 3, a sequence diagram shows WSP, and the underlying WTP layer, in a WAP responder stack during a successful GET request. Note how the Session Manager is only involved in the initialization of the WSP.

Transaction Layer: The WAP transaction layer handles the sending and re-sending of transactions. To separate transactions, each transaction is numbered with a unique sequence number, called *transaction identifier* (TID). New TIDs are created by the initiator by incrementing the last created TID value by one. The initiator can have several ongoing transactions with more than one responder, e.g., a server can push to several terminals. Therefore, a responder cannot be sure that each new transaction has a TID value incremented by exactly one.

The responder of a connection oriented session has a window of 2^{14} TIDs. The last TID value received from an initiator is saved in a variable named *lastTID*. The counter wraps around at $2^{15}-1$. When a new message arrives, it is directly accepted if the TID value is not increased more than 2^{14} times from *lastTID*. We will call such values *greater* than *lastTID*, and other values *less* than *lastTID*, except if the value is equal to *lastTID*.

If the bearer media reorders two messages so that the greater TID value arrives late, the later message is said to be an *out-of-order* message. When an out of order message arrives, the responder invokes a so-called *TID verification procedure* before it contin-

ues. The TID verification is performed by sending a special acknowledge message (with bit *TIDve* set). The initiator acknowledge (with bit *TIDok*) if it has an outstanding transaction with the same TID value.

If an initiator is out of synchronization with *lastTID* (e.g., after a reboot) it can avoid further TID verifications (using bit *TIDnew*). This forces a TID verification that will set *lastTID* to the TID of the *TIDnew* message. During TID verification no new transactions are started by the initiator, and the responder removes any old transactions.

3 Abstraction for Test Case Generation

As described, the TIDs of the messages play an important role in the WAP specification. An instance of the WAP protocol will typically make use of several TIDs from the domain 0 to $2^{15}-1$, and a sliding window of size 2^{14}. Thus, the potential numbers of TID values will be infeasible for exhaustive model-based test-case generation — the generation algorithm will experience the so-called *state-space explosion problem* [Hol97]. To overcome this problem, previous applications of automatic verification techniques to the WAP protocol have limited the analysis to scenarios with only a single transaction [HJ04,GB00]. We will take a different approach and introduce an abstraction. It will allow us to deal with abstract TID values during the analysis of the model, while maintaining the concrete TID values so that concrete model traces can still be generated.

Concrete domain: We assume a set T of TID variables $t_0, \ldots, t_{N-1}$. To describe the semantics we use a *variable assignment* $v : T \to \{n \mid 0 \leq n \leq 2^{15}{-}1\} \cup \{\bot\}$, where $\bot$ represents the unassigned value. Initially all variables are unassigned. The variables can be compared with Boolean combinations of $t_i < t_j$ and $t_i \leq t_j$, and manipulated with the operations

$$\begin{array}{ll} t_i = \textit{free} & v'(t_i) = \bot \\ t_i = t_j & v'(t_i) = v(t_j) \\ t_i = \textit{new}^+ & v'(t_i) = \max(v) + 1 \\ t_i = \textit{new}^- & v'(t_i) = \min(v) - 1 \end{array}$$

where v' is the resulting variable assignment, v the directly preceeding variable assignment, and $\max(v)$ and $\min(v)$ the maximum and minimum assigned integer values of all TIDs, respectively.

Abstract domain: We use a set A of abstract TID variables $a_0, \ldots, a_{N-1}$, and an abstract variable assignment $v_a : A \to \{n \mid 0 \leq n < N\} \cup \{\bot\}$. We assume that the set of abstract values is *tight* in the following sense: if $v_a(a_i) = k$ then there exists $v_a(a_j) = l$ for all $0 \leq l < k$.

Abstraction of Concrete TID values: We define the *abstraction function* $\alpha : T \to A$ to be the mapping, such that $\alpha(t_i) = 0$ if $\min(v) = v(t_i)$, $\alpha(t_i) < \alpha(t_j)$ if $v(t_i) < v(t_j)$, $\alpha(t_i) = \alpha(t_j)$ if $v(t_i) = v(t_j)$, $\alpha(t_i) = \bot$ if $v(t_i) = \bot$, and v_a is tight. A transition from the abstract state v_a to v'_a is possible if there exists a transition from v to v', $v_a = \alpha(v)$, and $v'_a = \alpha(v')$.

The proposed abstraction is sound in the sense that properties in the abstract state-space also hold in the concrete state-space. That is, if $v_a = \alpha(v)$, then the truth-value of $t_i < t_j$ or $t_i \leq t_j$ is the same for the corresponding abstract TIDs a_i and a_j. It can be shown that the abstract transition relation is *must* abstraction and thus under-approximates the concrete transition relation [LT88,BKY05].

Modeling and Analysis in UPPAAL: When modeling the WAP protocol, we shall use $t_i = \mathit{new}^+$, $t_i = t_j$, and $t_i = \mathit{new}^-$ to model assignment of new *correct* TID values, *existing* values, and values that are *out of order*, respectively. To implement the abstraction, we use UPPAAL's meta variables. Such variables are used to annotate models. They can be refered to in the model, but they are not considered when two states are compared during analysis. We declare the set of concrete TID variables T as a vector of meta variables, the set of abstract TID variables A as vector of ordinary integer variables, and apply the abstraction function to each state explored during state-space exploration[2]. In this way, the analysis will explore concrete states until the reachable abstract state-space is explored, while maintaining the concrete values to support generation of concrete test cases.

4 Testing Model

In this section, we describe our model of the WAP gateway. The model is emphasized on the software layers WTP and WSP. They have been modeled as detailed and close to the WAP specification as possible. Other parts of the gateway are modeled more abstractly, but without loss of externally observable behavior affecting the WTP and WSP layers. We have chosen to model the *connection oriented* version of the WAP protocol, where several outstanding transaction can be held together into a session. The model has been made with the intention to generate real system tests that can be executed over a physical connection. Obviously, the complexity of making this kind of system model and system test is much higher than to test each layer separately.

In Figure 4, an overview of the modeled automata and their conceptual connections is shown as a flow-graph. The nodes represent timed automata [AD94] and the edges synchronization channels or shared data, divided in two groups (with the small arrows indicating the direction of communication). The model is divided in two parts, the *gateway* model, and the *test environment* model. The test environment consists of the two automata Terminal and HTTP Sever. The gateway model is further divided in to a WTP part, a WSP part, and globally shared data and timers[3]. The WTP part consists of the service access point TSAP, two instances WTP0 and WTP1 of the WTP protocol, a WSP Session Manager, two instances Method 0 and Method 1 of the WSP methods, and a session service access point SSAP.

[2] We have implemented this in our UPPAAL CO✓ER tool. The same affect can be achieved by annotating each edge in the model with a simple function, implementing the abstraction function.

[3] To improve the readability of Figure 4, we have omitted many edges to and from the automata Timer and Data Store.

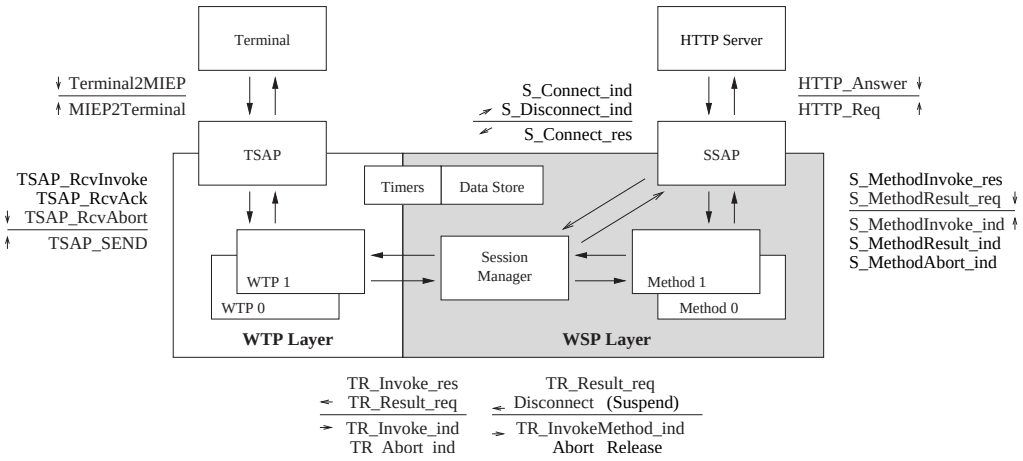

Fig. 4. Overview of the formal model.

The idea of the model is to let the Terminal automaton model a mobile device that non-deterministically stimulates the gateway with input and receives its output. In a typical scenario, the Terminal requests a WML page from a web sever. The request goes through an instance of the WTP and WSP layers and further to a web sever. In case the page exists, it is sent back through the gateway, and is finally received in the Terminal. Such a scenario is depicted in Figure 3.

In the following, we briefly describe how the WAP gateway specification and the components in its environment have been modeled as a network of timed automata. Due to lack of space, several of the automata are not shown in detail in this paper, but can be found in [HP06].

4.1 Test Environment Model

The test environment consists of the two automata Terminal a HTTP Server. Messages from the terminal to the WAP gateway are modeled as the single synchronization Terminal2MIEP, and similar in the other direction (see Figure 4). When the synchronization occurs, a special set of global integer variables are assigned, which corresponds to the fields of the protocol headers, e.g., WTP Type, WTP Class, or WSP Connect. Our model is done so that any state preceding a Terminal2MIEP synchronization (similar in the other direction), contains all values of the variables that corresponds to fields of the modeled message. This is to facilitate the constructions of packets from model traces, which is needed in the later stage when traces are compiled in to concrete test cases.

As mentioned, another important design decision is to let the Terminal model initiate and control the whole interactions. A particular problem is to control the HTTP server. We have solved this by sending control messages encoded into the message content, from the terminal, all the way through the WAP gateway, to the HTTP sever.

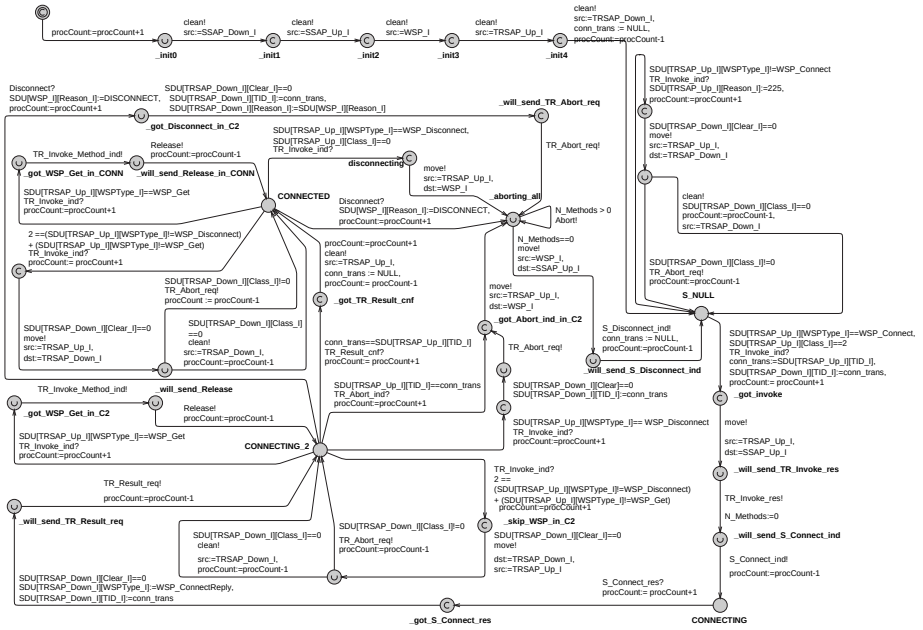

Fig. 5. The Session Manager automaton.

In this way, the HTTP server can be instructed to delay its response message, drop a message, or immediately return.

As the gateway model reacts to stimuli from Terminal, several instances of the WAP layers automata will become active simultaneously. We use a counter to keep track of the number of active automata in the gateway model that are not in a stable state, i.e., a state where it is idle and waiting for new input. The counter is used to restrict the Terminal from sending messages that will not be dealt with immediately. This scheme avoids unnecessary interleavings and reduces the state-space of the model.

How the TID Abstraction is modeled: In the Terminal automaton, TIDs are assigned when new PDUs are created, as described in Section 3. To use an existing TID value (i.e., to perform an assignment), or to free a TID variable (i.e., set it to $\perp$) is straightforward to model in UPPAAL. To model new^+ and new^-, we use two hidden variables MinTID and MaxTID that are initialized to $2^{14}-1$ and 2^{14}, respectively. All TID variables t_i are initially $\perp$. The operation $t_i = new^+$ can now be modeled by assigning a variable ti the value of MaxTID followed by an incrementation of MaxTID, and dually for the operation $t_i = new^-$.

4.2 Gateway Model

The gateway model is a detailed timed automata description made with the intention to comply with the WAP specification as closely as possible. Communications between

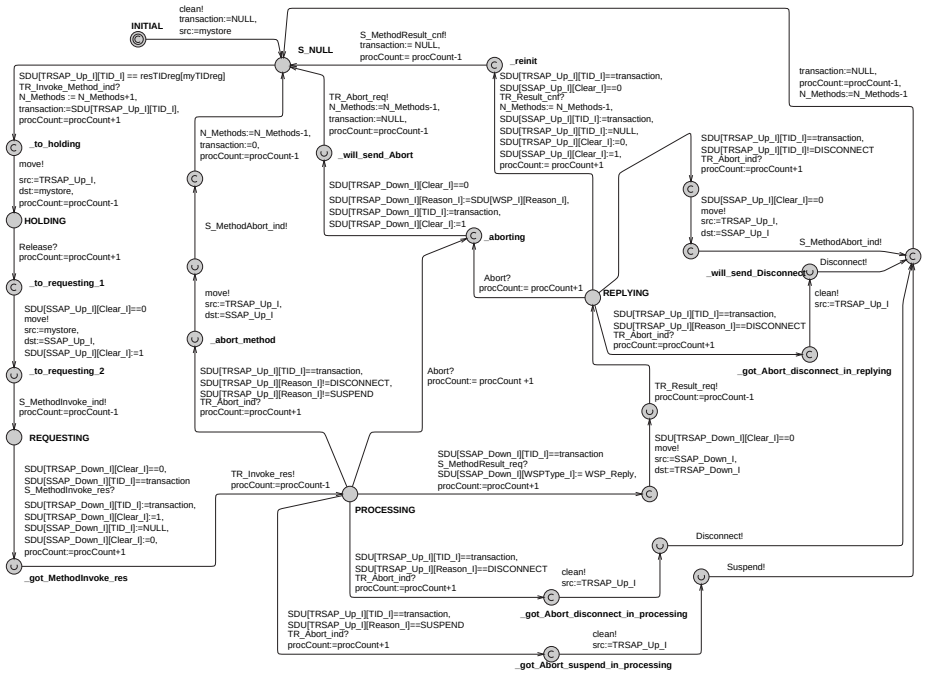

Fig. 6. The Method automaton.

two layers are modeled as synchronization labels and an array of data representing the modeled fields values. All communications to or from WTP or WSP go via SAPs to mimic the real protocol.

TSAP: As illustrated in Figure 4 the SAP below the WTP layer, called T-SAP, is modeled by automaton TSAP that converts the raw data fields sent over the Terminal2MIEP channel into signals that mimics a Transport SAP. In the upward direction, TSAP converts the WTP layer data into signals, e.g., RcvInvoke, RcvAck, and RcvAbort. In the downward direction TSAP merely copies the data to the environment (i.e., no headers to be added). The TSAP automaton also inspects the TID value and decides if the message should be delivered or dropped.

WTP layer: Two instances of the WTP layer are modeled, i.e., there can be two transactions active at the same time. An instance is activated when a message arrives with a TID that does not already exist in the layer. Successive messages with the same TID are directed to the activated instance. The WTP automata are named WTP0 and WTP1 and are instances of the same automaton template in UPPAAL. All messages from the WTP state machine of the WAP specification are modeled, including all types of aborts. The timers are also modeled, with the two intervals *acknowledge interval* A and *retry interval* R.

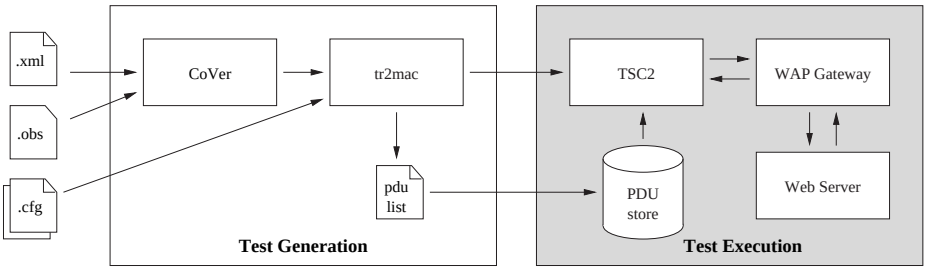

Fig. 7. Overview of the setup used for testing the WAP gateway.

WSP layer: The WSP layer consists of two types of automata: session manager Session Manager shown in Figure 5, and two methods automata Method0 and Method1, shown in Figure 6. The Session Manager is responsible for connections and disconnections of the session. It forwards incoming method invokes, e.g., when a WML page is requested. We model the GET method that, on the HTTP side, becomes a HTTP GET request. When a session is disconnected all methods are aborted. Each method has a corresponding outstanding transaction that it aborts. It is also possible to abort a individual method transaction without terminating the whole session.

SSAP: Above the WSP layer is the Session SAP. We model an automaton SSAP that mimics the gateway from the S-SAP to the communication with the HTTP server.

Timer: Timers are modeled by four instances of automaton Timer, two for each WTP layer automaton. A timer can be activated and deactivated by the WTP automaton. If a timer expires it sends a message to its WTP automaton.

Datastore: The automaton named Datastore manages data. Its memory is modeled as an array where the rows are "owned" by different automata. The columns represent fields in the PDUs. The Datastore automaton implements three convenient sub routines that can be used by the other automata in the gateway: copy, clear, and move. To not over-write data in an unintended way, the rows also include a Clear bit that is set when new data is allowed to be written.

5 Test Generation and Execution

The tool setup used for generating and executing tests at Ericsson is shown in Figure 7. The setup is divided in two parts, a test *generation* part for generating and transforming test cases into executable format, and a test *execution* part that executes the tests on the WAP gateway in a controlled computer network. In the following, we first describe the test criteria used as input to our test generation tool, then the test generation, and last how the tests were executed and some experiences.

5.1 Test Criteria

To specify how thorough a test suite should test a system, we select test cases following some given coverage criteria. Before presenting the criteria, we (informally) characterize a stability property that will be used in all testing criteria. We say that the gateway model is in a *stable state* if all automata are in locations modeling idling states from which they need an (input) synchronization to proceed. In the system under test, this corresponds to a situation where the whole gateway is idle and waiting for some input from the environment, which implies that there are no transactions active in the gateway, and no other ongoing activity. We shall use a predicate named *stableState()* that is true only if the gateway model is in a stable state.

In Figure 8, the three coverage criteria used in this case study are formally specified as *observers with parameters* [BHJP05] [4].

Edge Coverage Observer: It is shown in Figure $8(i)$. Assume that P is a set of automata. The expression $edge(P)$ returns a value only if an automaton (in the set P) is active in a transition. The parameter E is then assigned to the edge of the active process in P. The observer then reaches state *gotEdge(E)*, where E is the assigned edge. The *gotEdge(E)* location has a true loop which allows it to stay in the location forever. When the *stableState()* macro becomes true, the observer reaches location *done(E)*, indicating that the edge E is covered.

Intuitively, the edge coverage observer specifies that a test suite should cover as many edges E of the automata in P as possible, given that after every E a state satisfying *stableState()* is reached. If the set P includes two or more automata from the same automaton template, we assume that $edge(P)$ is the same identifier for both automata if the same edge is traversed. That is, the edge is considered to be covered if it is traversed by any instance of the template.

Switch Coverage Observer: The observer in Figure $8(ii)$ is similar to the edge coverage observer, but it specifies that any two adjacent edges in the same automaton instance should be covered. In this case, it is crucial that the edges are from the same automaton. Therefore, we require that the automaton P that takes the first edge E, must also take the second edge $E2$.

Projection Coverage Observer: Figure $8(iii)$ shows an observer that specifies a projection criterion. It specifies that a pair of locations from the WTP layer, and the WSP layer should be covered. The macro *stackProj(WTP,WSP)* returns a pair of locations $(L, L1)$, where L is from a WTP automaton, and $L1$ is from a WSP automaton. It is further required that L and $L1$ are associated with the same transaction.

[4] Due to lack of space, we refer the reader to [BHJP05] for a detailed description of the observer language.

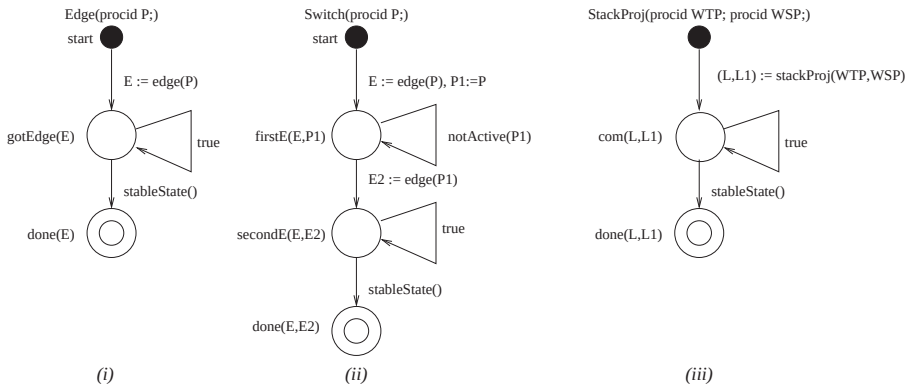

Fig. 8. The three observers used in the case study.

5.2 Test Generation

The problem of generating test cases is solved by the UPPAAL CO✓ER tool, which extends the model-checking tool UPPAAL with capabilities for generating test suites [5]. It takes as input the timed automata model of the WAP gateway described in the previous section, and a coverage criterion specified as a parameterized observer (.xml and .obs in Figure 7, respectively). The output of UPPAAL CO✓ER is a set of abstract test cases (or test specifications) represented as timed traces, i.e., alternating sequences of states, and delays or discrete transitions, in the output format of the UPPAAL tool.

Results: Table 8 shows the result of the test suite generation. Each row of the table gives the numbers for a given coverage criteria and the automata it covers (used as input). For example, WTP denotes WTP0 and WTP1 for which the tool has found 63 coverage items, i.e., edges in the WTP template. To cover the WTP edges a test suite of 16 test cases is produced. The number of transitions of the test suite is 1562. The test suite interacts with the system 92 times, i.e., 92 PDUs are communicated. We will discuss the rightmost column in the next subsection.

The table shows the result of the other test criteria as well. We note that, as expected, the switch coverage criterion requires many more test cases to be executed than edge coverage. We also note that it is more efficient to execute the test suites covering all templates at once, i.e., WTP, Session Manager, and Method, than to execute all the individual test suites. For example, the test suite with edge coverage in all templates sends 142 PDUs, whereas the sum of sent PDUs in the individual suites is 225. For switch coverage the numbers are 467 compared to 555 PDUs.

5.3 Test Execution

The timed traces representing abstract test cases are converted to executable script programs by the tr2mac tool [Vil05], which also takes two configuration files as input (.cfg

[5] For more information about the UPPAAL CO✓ER tool, see the web page `http://user.-it.uu.se/~hessel/CoVer/`.

Criteria		Items	Test suite		Test script	Failed tests
Observer	Templates		cases	trans	PDUs	
Edge	WTP	63	16	1562	92	1
	Session Manager	46	12	1058	57	1
	Method	31	10	1497	76	0
	All	140	28	2548	142	2
Switch	WTP	109	44	5082	313	2
	Session Manager	76	28	3020	166	7
	Method	37	9	1495	76	0
	All	222	74	8129	467	10
StackProj	All	101	21	2129	114	0

Table 8. Test generation and execution results.

in Figure 7). In a trace, each action label and combination of variable values in the associated state, represents the parameters of a PDU to be sent or received, or a null operation (all internal actions are mapped to null operations). The files .cfg describe how to perform the translation for a given UPPAAL model, i.e., which labels to consider as external and where to put the state variable values in the PDUs. Each delay of a timed trace naturally represents a delay to be performed by the test program. The tr2mac program accumulates the delays between non-null operations and inserts the result in the script program.

The output of tr2mac is a script program that can be executed by the TSC2 test environment, and a list of partially instantiated PDUs that will be needed. The PDUs are fully instantiated at the time the script is executed in the test harness. The TID values and information about the specific test environment, e.g., the IP addresses, are filled in at execution time. In this way, many PDUs can be reused between different test cases (and the set of needed PDUs will eventually become stable).

When TSC2 executes a script, all listed PDUs must be available in the PDU store[6]. TSC2 will send PDUs to the WAP gateway and check that the expected response appear at the right time points. If this is not the case, TSC2 will report the discrepancy to a log file, and proceed with the next test script. During testing, TSC2 acts in place of the mobile device (i.e. the terminal). As described in the previous sections, the mobile device (and thus TSC2 when executing the generated test cases) thus controls the behavior of the surrounding computer network. The behavior of the web server is controlled by sending parameters in the PDUs that are interpreted as commands by a php script running on the web server.

Results: The test cases presented in the Table 8 have been executed on an in-house version of the WAP gateway at Ericsson. As shown in the rightmost column of Table 8 most of the test case went well. A few tests failed due to two discrepancies — one in the WTP automata and one in the Session Manager automaton.

[6] Currently, non-existing listed PDUs must be manually created. It is possible to automate also this step.

The first discrepancy is in the WSP layer. The session manager is modeled to not accept any new *Connect* messages. Reading the WAP specification carefully, after finding this discrepancy, we conclude that it is allowed to accept new *Connect* messages and replace the current session if the layer above agrees. This problem in the model explains the discrepancy found with the test suite covering the edges of Session Manager, and the seven discrepancies found when executing the test suite with switch coverage in the Session Manager.

The second discrepancy is a behavior present in our model of the WTP layer but not in the tested WAP gateway. We found that no acknowledge is sent from the WTP state, RESULT_WAIT, when an (WTP) invoke is retransmitted and an acknowledgment has already been sent. The retransmission is required in the WTP specification [For01] but not performed by the implementation. This discrepancy was found both when running test suites covering the edge and switch criteria of the WTP template.

We also observe that the two discrepancies were both found when executing the edge covering test suites — one in the test suite for WTP, and the other in the test suite for Session Manager. The test suite with switch coverage finds the same discrepancies, but many times (as many as the erroneous edges appear in some switch). The suite with projection coverage did not find any discrepancies.

6 Conclusion

We have presented a complete test bed where test cases are automatically produced and executed, from a formal model and coverage criteria formally described as observers. The validity of the tests has been proven in a case study where test cases have been executed in a real test environment at Ericsson. The test generation techniques and the coverage criteria used have industrial strength as complete test suites have been generated for an industrial application, and discrepancies have been found between the model and the real system.

We have also presented an abstraction technique that can be used in models making use of sequence numbers with large domains. It preserves the relations needed when comparing sequence numbers in the WAP protocol, while the size of the analyzed state space is significantly reduced. We believe that the abstraction will be useful for specifying and analyzing models of other protocols.

Acknowledgment

We thank the other members of the ASTEC AuToWay project at Ericsson and Uppsala University: Tomas Aurell, Anders Axelssson, Johan Blom, Joel Dutt, Bengt Jonsson, Natalie Jost, John Orre, Payman Tavanaye Rashid, and Per Vilhelmsson.

References

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BFG+00] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, , Claude Jard, Thierry Jron, Alain Kerbrat, Pierre Morel, and Laurent Mounier. Verification and test generation for the sscop protocol. *Science of Computer Programming*, 36(1):27–52, 2000.

[BHJP05] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson. Specifying and generating test cases using observer automata. In J. Gabowski and B. Nielsen, editors, *Proc. 4^{th} International Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 125–139. Springer–Verlag, 2005.

[BKY05] Thomas Ball, Orna Kupferman, and Greta Yorsh. Abstraction for falsification. Technical Report MSR-TR-2005-50, Microsoft Research, June 2005.

[For01] WAP Forum. Wireless transaction protocol, version 10-jul-2001. online, 2001. http://www.wapforum.org/.

[GB00] S. Gordon and J. Billington. Analysing th wap class 2 wireless transaction protocol using colored petri nets. In M. Nielsen and D. Simpson, editors, *ICATPN 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 207–226. Springer–Verlag, 2000.

[HJ04] Yu-Tong He and R. Janicki. Verification of the wap transaction layer. In *Software Engineering and Formal Methods*, pages 366–375, 2004.

[HLSU02] H.S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems : 8^{th} International Conference, (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer–Verlag, 2002.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.

[HP06] Anders Hessel and Paul Pettersson. Model-based testing of a wap gateway: an industrial case-study. Technical Report 2006-045, Department of Information Technology, Uppsala University, 2006.

[LMN05] K. G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using uppaal. In J. Gabowski and B. Nielsen, editors, *Proc. 4^{th} International Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer–Verlag, 2005.

[LMNS05] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron - an industrial case study. In *Proc. of the 5th ACM International Conference on Embedded Software*, 2005.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LT88] K.G. Larsen and G.B. Thomsen. A modal process logic. In *Proc. 3^{rd} Int. Symp. on Logic in Computer Science*, 1988.

[Tre96] J. Tretmans. Test generation with inputs, outputs, and quiescence. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 2^{nd} Int. Workshop (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer–Verlag, 1996.

[Vil05] Per Vilhelmsson. A test case translation tool - from abstract test sequences to concrete test programs. Technical report, Department of Information Technology, Uppsala University, 2005.