Footprinter: Round-trip Engineering via Scenario and State based Models

Ankit Goel
National Univ. of Singapore
ankit@comp.nus.edu.sg

Bikram Sengupta
IBM Research, India
bsengupt@in.ibm.com

Abhik Roychoudhury
National Univ. of Singapore
abhik@comp.nus.edu.sg

Abstract

In model-driven software development, while scenario-based models are closer to distributed system requirements, state-based models are suitable for code generation. Our tool 'Footprinter' exploits relative strengths of these two modeling styles in a round-trip engineering approach– from requirements, to test-case generation and execution, to tracing implementation defects back to requirements.

1 Introduction

The construction of formal behavioral models lies at the core of model-driven software development. Conventionally, two classes of behavioral models have been studied — (a) state-based models such as Statecharts [8] which show a system as a composition of processes and highlights the behavior of each process via finite state machines (FSMs), and (b) scenario-based models such as High-level Message Sequence Charts (HMSC) which capture the global inter-process communication via interaction snippets called Message Sequence Charts (MSCs) [4]. In some sense, the intra- and inter-process views form two dual views of system behavior — each having its distinct advantages. The intra-process modeling via FSMs highlights the computation steps *inside* each process while suppressing the inter-process communication. It leads more directly to code generation for the individual processes of the system — bringing the model closer to implementation. However, when the designer is trying to get a handle of the system behavior at the very early stages starting from the informal requirements written in English — it is easier for the designer to start by drawing sample scenarios as MSCs. The MSCs highlight inter-process communication while suppressing the computation steps inside each process. These MSCs can then be combined to obtain a MSC-based system model such as HMSC. In other words, the inter-process view of modeling is useful for synthesizing system models from informal requirements, while the intra-process view is useful for generating code from system models.

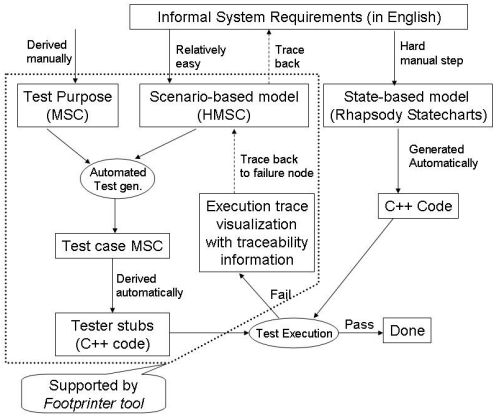


Figure 1. Overall Architecture

In this paper, we *exploit the distinct strengths* of the two modeling styles within a round-trip engineering and validation methodology for distributed systems (see Figure 1). Starting from the initial system requirements, which are generally written in natural language, our first step is to obtain possible scenarios (MSCs) depicting the global interaction patterns among various processes, and then structure the scenarios into a HMSC to capture the intended flow. The traceability information linking requirement snippets with the corresponding MSCs (and events within) is also captured. However, MSCs and HMSCs do not naturally lead to state-based descriptions of individual components. There are technical difficulties (e.g. non-local choice, implied scenarios [13]) associated with the task of automatically moving from global to local views of behavior. Hence, obtaining the state-based per-process model and identifying traceability links with the requirements, still requires considerable manual effort on the part of the designer. For our purpose, we consider the notation of Statecharts for specifying a state-based model using the Rhapsody [2] tool, which also supports automatic code generation. The generated code is, of course, only as sound as the Statechart model, and given the manual effort involved in designing the latter,

testing the final implementation with respect to the original requirements forms a crucial part of our methodology. To enable this, user-defined test purposes are used to derive MSC-based test cases from the HMSC specification, and the test cases are then executed on code generated from statecharts. Execution sequences from unsuccessful test cases are traced back to the HMSC specification and then to the original requirements to aid debugging. This is the essence of our round-trip engineering approach. Based on the above methodology, in this paper we discuss our tool *Footprinter*. The dashed boundary in Figure 1 represents the part of round-trip engineering methodology supported by Footprinter, while we assume existing tool-support for the components lying outside (Rhapsody[2] in our case).

2 Tool Architecture and Usage

In this section we describe the architecture of our tool Footprinter. The main components of Footprinter are:

1. *A graphical editor* for entering the scenario based model of requirements. It allows user to input– (i) the requirements model as a High-level Message Sequence Chart (HMSC), and (ii) a test-purpose MSC used for guiding the test generation.

2. *A test generation engine* which automatically generates test cases from HMSC model using test purpose MSC, to produce test cases satisfying the test-purpose. Each of these test cases is visualized as an MSC.

3. *A component for generating the tester stubs* (as C++ code) from a given test case MSC. In this step, user first identifies lifelines representing the system under test (SUT) in the test case MSC. Tester stubs are then generated corresponding to the non-SUT lifelines in the test case MSC.

4. *A visualization component* for showing a test execution trace along with the traceability information relating the trace events back to the HMSC model.

We now elaborate on each of the above components.

2.1 Graphical Editor

The graphical editor enables a user to visually input the HMSC requirements model. The description of a HMSC takes two separate inputs. The first input is a directed graph G having nodes N and edges $E \subseteq N \times N$. It consists of a unique start node $s \in N$, such that every other node in N is reachable from s. Further, each node in N corresponds to a MSC. Consequently, the second input required for describing the HMSC is a set of MSC descriptions which correspond to various nodes in the HMSC graph described above. Various paths in the HMSC graph G correspond to different system behaviors. For illustration, the Footprinter screenshot in Figure 4(a) shows a HMSC graph input, while the MSC input corresponding to the graph node labeled *RcvWthr* in Figure 4(a) (encircled in thick boundary) is shown in the Footprinter screenshot in Figure 4(b).

In addition to the HMSC requirements model, the graphical editor is also used to specify the test-purpose MSC. Besides usual MSC events[1], user can also mark some messages as *forbidden* in a test purpose MSC. Visually, a message is classified as forbidden by putting a *cross* over the message arrow. A forbidden message specifies a message that a user does not want to appear in the generated test case(s). For illustration, consider the test-purpose **TP1** shown in a Footrprinter screenshot in Figure 5.

2.2 Test case Generator

Once user has specified a HMSC requirements model and a test purpose MSC, the test generator component of Footprinter can automatically generate test case(s) from the HMSC model, guided by the test purpose. The test generation process also takes as input a depth bound D, which specifies the maximum length of a path in the HMSC graph to be explored for test generation. Our test generator produces all test cases satisfying the test purpose, that can be found within the given depth bound D.

At the core of our test generation process is the comparison of test purpose events with the events appearing in the HMSC requirements model. Test case MSC(s) obtained from the HMSC model contain all the test purpose events (except for the *forbidden* events, corresponding to a forbidden message) according to the partial order specified by test purpose MSC, possibly interspersed with other events appearing in the HMSC model.

Further, during the test generation, Footprinter embeds *traceability* information in the generated test cases. This information is embedded at the event level, mapping an event occurrence in a test case to the HMSC node and the corresponding event within the MSC it represents.

2.3 Test stub Generator

This component of Footprinter generates *tester stubs* from a MSC test case in the form of C++ code, which are actually used for exercising the system under test or SUT. This step requires user to identify which lifelines in a test case MSC represent SUT. The remaining (non-SUT) lifelines then represent the environment of SUT and a tester stub is generated corresponding to each such lifeline. The traceability information contained in a test case MSC event is also captured for the corresponding code snippet generated in the tester stubs — enabling backward traceability of test execution results to the original requirements for debugging failed tests.

[1] A message send, receive or a local action.

We use the Rhapsody tool to (a) construct a Statechart model of the requirements, and (b) automatically generate a C++ implementation of the SUT from the Statechart model. The tester stubs derived by Footprinter (corresponding to a test case) are compiled and executed with SUT code generated from Rhapsody.

2.4 Visualization Component

During test execution, the tester stubs log information regarding the events executed by them. Based on this information, Footprinter reconstructs the (part of) test case MSC covered during test execution, which can then be visualized by the user. Further, using Footprinter user can easily visualize the relationship between events in the test execution trace, and the HMSC requirements model. This is made possible by the traceability information embedded in the test case.

Thus, in case of a test execution failure[2], using traceability information user can trace back the test execution results back to the original requirements (via the HMSC model). This can help provide a more precise location of failure in the requirements, and aid the system designer to debug the system.

3 System Validation using Footprinter

In this section, we illustrate successful use of Footprinter in testing and debugging a real-life case-study.

CTAS Case Study The CTAS or *Center TRACON Automation System* [1] is a set of tools developed at NASA to aid the air traffic controllers in managing high volume of air traffic flows at large airports. Various processes such as TS (Trajectory Synthesizer), RA (Route Analyzer) etc. that CTAS system comprises of, require latest weather updates for their functioning. The weather updates are provided to these processes by WCP (Weather Control Panel) via the CM (Communications Manager) which is the central controller responsible for communications among these processes. Both WCP and CM are also part of the CTAS system. We refer to various processes requiring the weather updates simply as Clients. *For the purpose of illustration, we consider a CTAS system consisting of 2 Clients*.

Constructing HMSC model The complete scenario-based description of the CTAS requirements is obtained by first deriving MSCs corresponding to various requirements and then composing together these MSCs to form a High-level MSC (HMSC). For illustration, Footprinter screenshots capturing the complete CTAS HMSC description and MSC *RcvWthr* corresponding to the CTAS requirement 2.8.12 (shown in Fig. 2) appear in Figure 4.

Test purpose specification After describing the HMSC model of the CTAS system, user can specify different test purposes using Footprinter. For illustration, consider the test purpose appearing in the Footprinter screenshot Figure 5. The first two sets of *Connect* followed by *Done* messages represent the successful connection of two client objects to controller CM, while the subsequent *Update* message corresponds to a weather update request initiated by WCP. Various forbidden *No* messages from the two clients match with, and hence avoid generating test cases with any *No* message sent by either of the two clients to CM during initial connection setup and subsequent weather update.

Test case generation Footprinter screenshots showing a test case MSC generated by it corresponding to the test purpose described above, appear in Figure 6. The message names appearing in bold italics represent the matching events in the test purpose (Fig. 5). This test case represents the scenario where two clients Client1 and Client2 initially get connected to CM by sending the *Connect* message, and subsequently get updated with the latest weather information via CM.

Test execution For the purpose of our experiments, we considered C++ code as the given system implementation for the CTAS example. The code was generated automatically from the Statechart model of the CTAS requirements using the Rhapsody tool [2]. Note that the Statechart model was derived separately, *by a person other than the authors.* In particular, we focused on testing the central controller (CM) component of CTAS C++ implementation. Thus, for the test case MSC shown in Figure 6, tester stubs corresponding to lifelines Client1, Client2 and WCP were generated by Footprinter. These tester stubs were then used for testing the CM's implementation derived via Rhapsody. However, the execution of this test case was unsuccessful and resulted in a *fail* verdict.

Traceability The (partial) execution trace of the above test case visualized using Footprinter (also as an MSC) is shown in Figures 7 and 8. Note that various events in the execution have been grouped together by Footprinter according to the HMSC nodes they correspond to. The grouping is represented using a rounded box and is labeled with the MSC name in the top right corner. This is achieved by Footprinter using the traceability information embedded in the test case events. The examination of this trace indicated that the tester stubs corresponding to– a) Client1 and Client2 timed-out while waiting for *ClientPostUpd* and

[2] A failure occurs when some tester stub either times out while waiting for a message, or receives a message different from what it is expecting.

The CM should perform the following actions when the Weather Cycle status is "updating" and all connected weather-aware clients have responded Yes to the CTAS_GET_NEW_WTHR messages:

→ it should set the weather status of all connected weather-aware clients to `post-updating'.
→ it should send CTAS_USE_NEW_WTHR messages to all connected weather-aware clients.

Figure 2. CTAS requirement 2.8.12

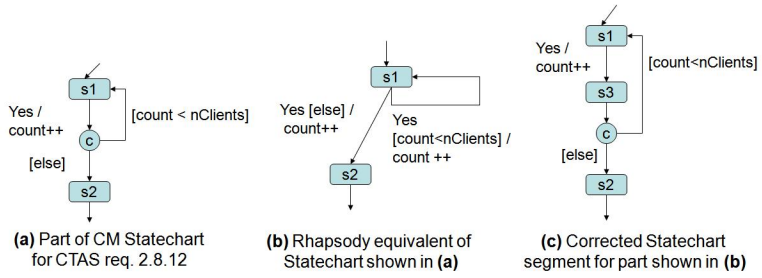


Figure 3. Bug found via traceability.

CTAS_GET_NEW_WTHR messages from CM, and b) WCP timed-out waiting for *WCPEnable* message from CM. Since MSC nodes in the HMSC model of CTAS closely correspond to the original CTAS requirements, using information provided by Footprinter we could trace the test execution results back to the original CTAS requirements. From there, the source of error was discovered in the Statechart description of CM.

Bug found The part of the Statechart where fault was located corresponds to the CTAS requirement 2.8.12 (see Figs. 2 and 4(b)) and is shown in Figure 3(a). In state s_1 the CM waits to receive the message *Yes*, and updates a counter *count* to check if all the clients have replied. However, we found that Rhapsody interprets this as another (supposedly equivalent) structure (shown in Figure 3(b)). Now, in this case the counter is incremented after checking the guard $count < nClients$, instead of doing so before this check. To ensure that the guards of outgoing transitions from the condition node (marked with c) are evaluated after *count* is incremented, a new state s_3 was introduced between state s_1 and the condition node, which corrected this fault. The updated structure is shown in Figure 3(c).

4 Related Work

There exists a large body of work in the domain of model based testing (e.g. see [5] and the chapters therein). Some approaches ([7]) use MSC based scenario notations for specifying the system models as well as test-purposes from which test cases are generated. However, it is difficult to specify complex system requirements using an unstructured collection of MSCs. In [10], the play-engine tool for Live Sequence Charts (LSCs), which are an extension of MSCs, has been extended to support testing of scenario-based requirements. This work, however, deals with testing the requirements themselves and no test-execution of actual system implementations is conducted. The UBET tool [3] also supports test generation from a HMSC model, but the test generation in UBET is driven only by the *edge-coverage* criteria in a HMSC, and not by a user provided test-purpose to elicit specific behaviors.

Many model-based techniques consider state-based models for describing system specifications from which test cases are derived (*e.g.*, see [11, 6, 12, 9]). However, distributed system requirements are in general inter-process in nature, and more naturally expressed as scenarios. In fact, it is often the manual construction of intra-process state-based models from such requirements that serves to introduce implementation errors. Hence, in our approach, we have leveraged the dual strengths: (a) strength of MSC-based models for capturing requirements and (b) strength of state-based models for generating code.

Finally, we feel that the issue of tracing the results of test cases back to requirements have not been adequately studied, and from a tool perspective, it is a novel contribution of our work that can greatly aid debugging.

References

[1] Center-tracon automation system (CTAS) for air traffic control. `http://ctas.arc.nasa.gov/`.

[2] Telelogic rhapsody. `http://modeling.telelogic.com/modeling/products/rhapsody/`.

[3] Ubet tool. `http://cm.bell-labs.com/cm/cs/what/ubet/`.

[4] Message sequence charts (MSC). *ITU-TS Recommendation Z.120*, 1996.

[5] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Prestchner. *Model-Based Testing of Reactive Systems*. Springer, 2005.

[6] D. Clarke et al. STG: A Symbolic Test Generation Tool. *TACAS*, LNCS, volume 2280, 2002.

[7] F. Fraikin and T. Leonhardt. SeDiTeC – Testing Based on Sequence Diagrams. In *ASE*, 2002.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming,8*, 1987.

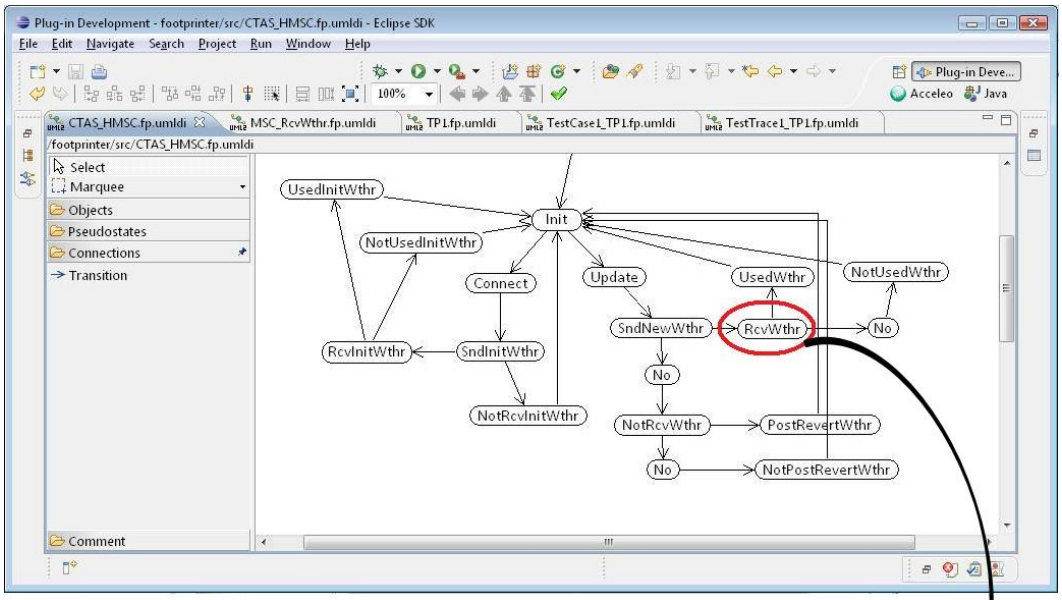
[9] C. Jard and T. Jeron. TGV: theory, principles and algorithms. *Intl. Jnl. on Software Tools for Technology Transfer*, 7, 2005.

[10] H. Kugler, M. Stern, and E. Hubbard. Testing Scenario-Based Models. In *FASE*, 2007.

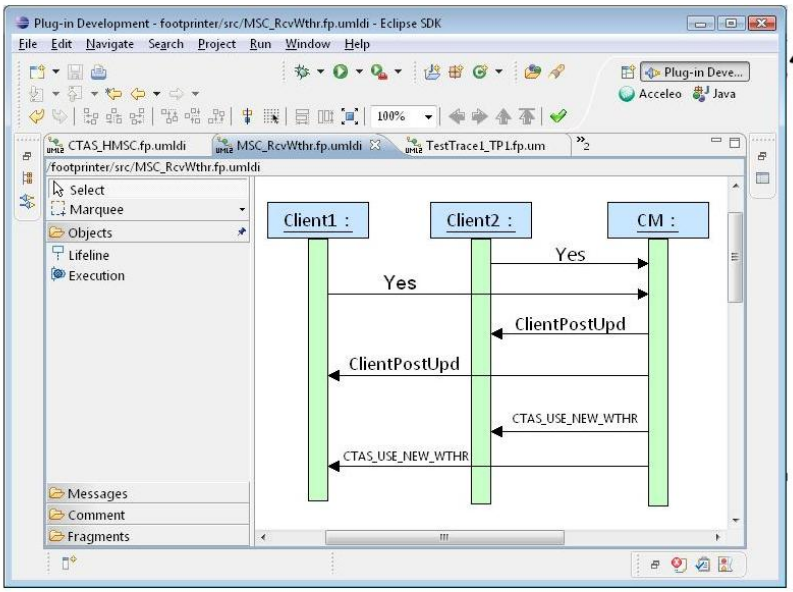
[11] S. Pickin et al. Test Synthesis from UML Models of Distributed Software. *IEEE Trans. Softw. Eng.*, 33(4), 2007.

[12] A. Pretschner et al. One evaluation of model-based testing and its automation. In *ICSE*, 2005.

[13] S. Uchitel, J. Kramer, and J. Magee. Detecting Implied Scenarios in Message Sequence Chart Specifications. In *ESEC-FSE*, 2001.



(a) CTAS HMSC



(b) MSC RcvWthr (corresponds to CTAS requirement 2.8.12 in Fig. 2)

Figure 4. Screenshots capturing CTAS HMSC and an MSC input.

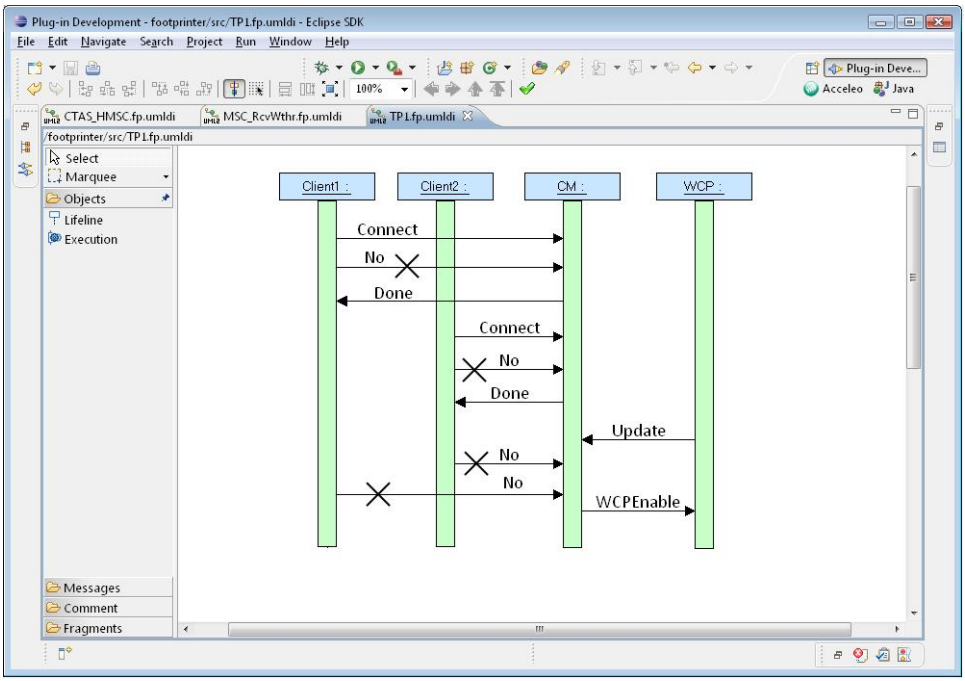
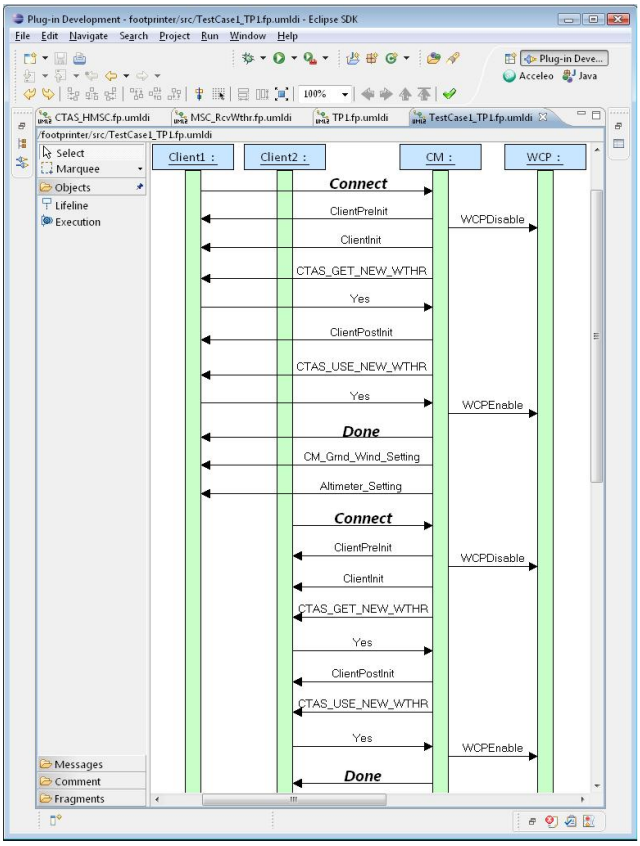
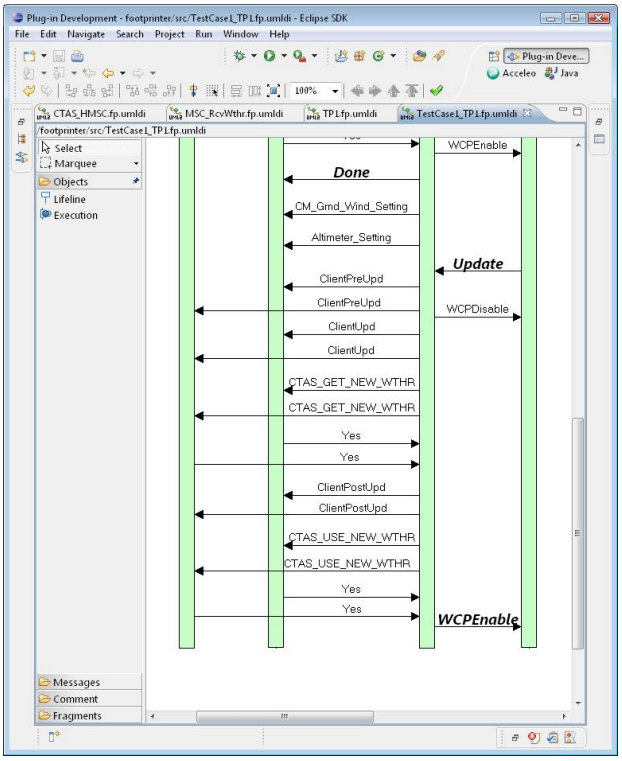


Figure 5. Screenshot– test purpose TP1 for CTAS example.



(a) First half of test case **TC1**



(b) Second half of test case **TC1**

Figure 6. Screenshots capturing test case TC1 generated for test purpose TP1.

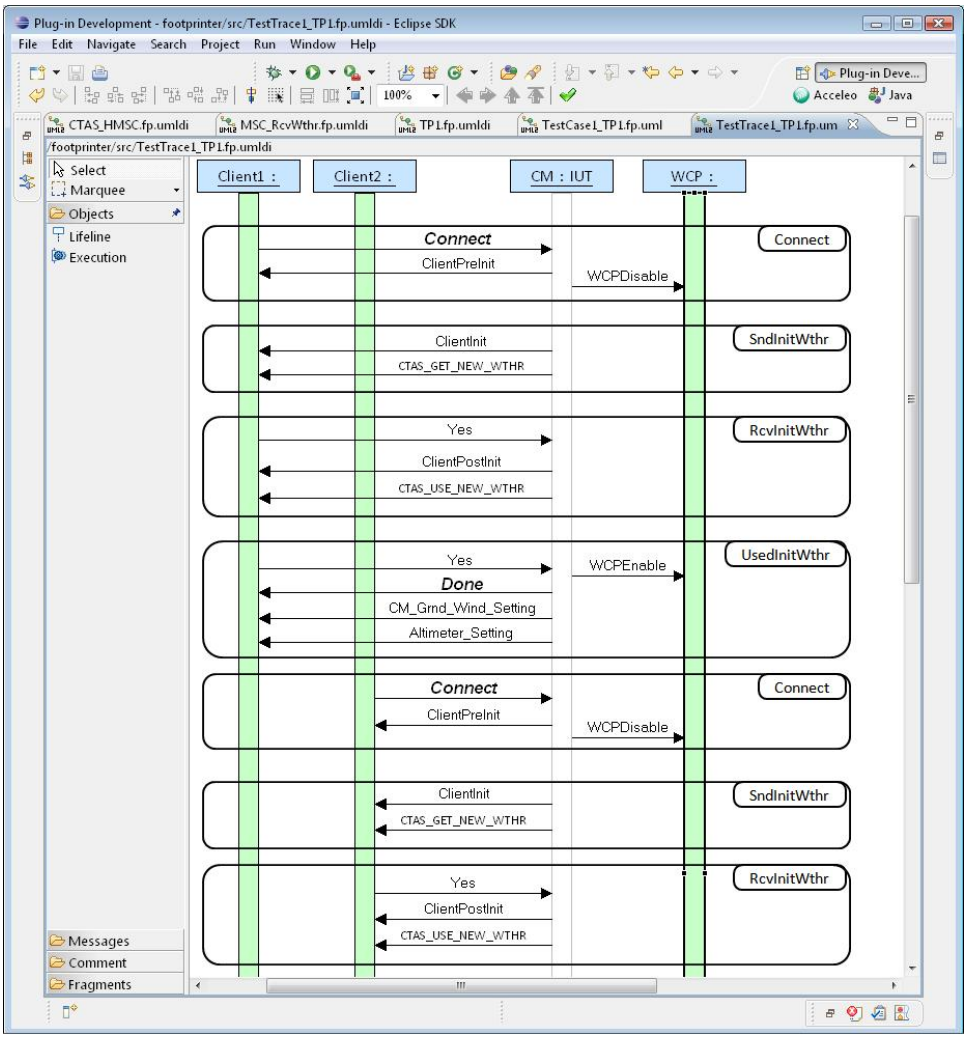


Figure 7. Screenshot– first half of execution trace for test case TC1.

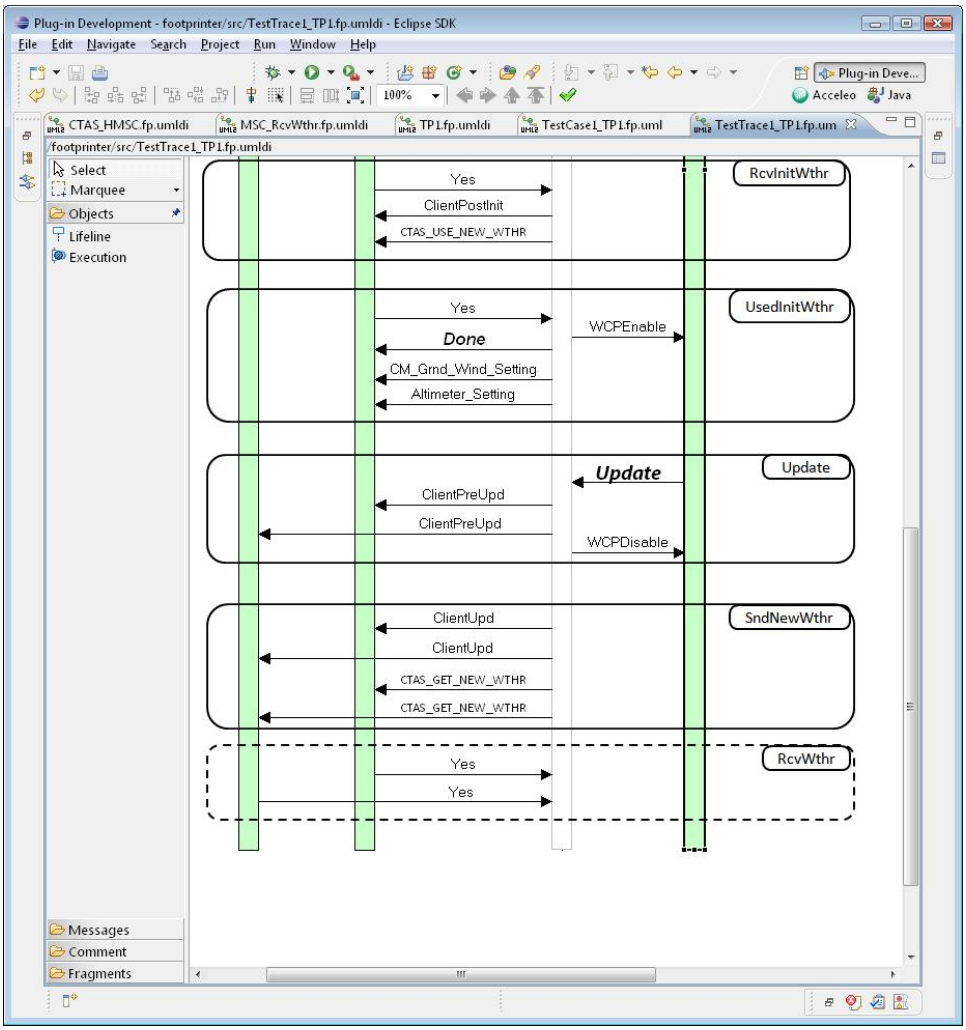


Figure 8. Screenshot– second half of execution trace for test case TC1.