Scenario-based Approach to Evolution of Communication Software

N. Mansurov a) and R. Probert b)

a) Department for CASE tools, Institute for System Programming (ISP), 25 B. Kommunisticheskaya, Moscow 109004, Russia; email: nick@ispras.ru

b) Telecommunications Software Engineering Group (TSERG), School of Information Technology and Engineering, University of Ottawa, Ottawa, Ontario, Canada K1N 6N5; email: bob@site.uottawa.ca

Abstract

Significant amounts of legacy software create a “barrier” for adoption of advanced software engineering techniques in the communications industry. To overcome this barrier, algorithms, methods and powerful tools for automated extraction of formal descriptions of the legacy communication software are required.

In this paper we present a “scenario-based” approach to re-engineering of legacy telecommunication software into formal specifications. Our approach is iterative and is based on 1) extracting scenarios from the legacy software using a combination of dynamic and static strategies and 2) automatically synthesizing formal specifications from these scenarios. For Step 2) we use the MOST toolkit, which is capable of synthesizing state-machine based formal models from scenarios, formalized as extended Message Sequence Charts (MSCs). The paper provides detailed descriptions of our re-engineering methodology and compares it with related approaches.

A case study is discussed where our scenario-based methodology was applied to re-engineer a small-sized telecommunications-like software system, called the ToolExchange.

1. Introduction and Background

The demand for high-quality, efficient communications systems and the rate of change of requirements for such systems continues to increase rapidly. As a result, development time or “time to market” has become as important to industrial success as product quality, price/performance, and development cost. Though Computer-Aided Software Engineering (CASE) technology and Formal Description Techniques (FDT’s) have offered promising means of delivering better systems sooner, up to now the promises have not been fulfilled. The current situation, however, is showing significant signs of improvement.

First, a small number of scalable, industrial-strength CASE tools have won substantial acceptance in industry. These include TAU and SCADE tools from Telelogic, Statechart from i-LOGIX , and RationalRose from Rational Corp. These tools provide designers with powerful analysis, modeling, and fast-prototyping capabilities, which enable early verification and validation of designs, usually by simulation, with some dynamic state-space exploration features. The tools are stable, and support standard modeling languages and notations for specifying system requirements, designs, and tests.

Secondly, the FDT SDL (Specification and Description Language), used primarily for representing protocols and system design specifications, has now gone through several releases as an international standard language [3], together with the corresponding scenario description language MSC [5]. Moreover, among the FDT’s supported by ITU, SDL is by far the most widely adopted for industrial use. The evolution of widely accepted languages and notations such as SDL, MSC, and Tree and Tabular Notation
(TTCN) [4] and their convergence with the Unified Modeling Language (UML) has been accelerated by the symbiotic evolution of the industrial-strength CASE tools listed above. A number of very successful industrial case studies have been recently completed, claiming improved quality, much lower development cost, and decreases in time to market of 20% to 50%. Some of this increased designer productivity and system quality comes directly from the use of integrated modeling environments - documentation is generated as development proceeds, design decisions are traceable to requirements represented as MSCs (scenarios), code is generated automatically, designs can be simulated before production code is produced, test cases can be produced semi-automatically or automatically in some circumstances, and so on. We believe that more savings in development time actually come from better designs produced with the use of these industrial-strength design and development tools, since better designs will require substantially less rework and redesign. In addition, in a recent pilot study by TSERG and Mitel Electronics [7], it was found that a CASE-based approach could be used to develop functional test specifications for complex telecommunications systems several times faster than current technologies allow. Thus, CASE-based approaches (often using SDL tools) offer significant improvements in quality, productivity, and time to market.

However, in order for CASE-based communication software engineering to become common practice, it is necessary to provide cost-effective methods for integrating CASE-produced components and systems with older, “legacy” base software. Legacy software systems were produced with older development methods, often involving a blend of higher-level code, and system-level code, with heterogeneous languages, architectures, and styles, and were often very poorly documented. Up to now, this fact has constituted a “legacy barrier” to the cost effective use of new development technologies.

In order to overcome the “legacy barrier”, there is an increasing demand for developing automatic (or semi-automatic) re-engineering methods, which will significantly reduce the effort involved in creating formal specifications of the base software platforms. Cost-effective methods for producing SDL models of the base software platform will allow the following benefits:

- Better understanding of the operation of the legacy software through dynamic simulation of the SDL model. This often produces more intuitive results and does not involve the costly use of the target hardware;
- Automated generation of regression test cases for the base software platform;
- Automated analysis and validation of the formal specifications of the new features built on top of the SDL model of the base software platform;
- Feature interaction analysis including existing and new features;
- Automated generation of test cases for new features;
- Automatic generation of implementations of the new features. Such implementations are retargetable for different implementation languages as well as for different real-time operating systems.

In this paper we present our methodology of scenario-based re-engineering of legacy telecommunications systems into a system design model expressed in SDL, and illustrate our approach by applying it to a case study, namely a small-sized telecommunications-like software system, called the ToolExchange.
Our approach consists of iteratively
- approximating the user-intended set of scenarios, (the original use cases of the system), using dynamic or static techniques, or a combination of both, and
- synthesizing a formal SDL model from this set of scenarios using the MOST toolkit [2].

This process is repeated until the SDL design model satisfies certain validity constraints. This SDL model is then used to assess and improve the quality and coverage of legacy system tests, including regression tests. The approach may be used to re-engineer and re-test legacy code from a black-box (environment), white-box (source code), or grey-box (collaborations among subsystems) point of view.

The rest of the paper has the following organization. Section 2 provides an overview of our dynamic scenario-based re-engineering methodology. Detailed steps of the methodology are described in Section 3. Section 4 contains a brief presentation of the MOST toolkit. In Section 5 we summarize our experience gained during a case study; we demonstrate how our dynamic scenario-based methodology was used to re-engineer a ToolExchange system. Section 6 contains some comparisons to related approaches and conclusions.

2. Methodology Overview

Scenario-based re-engineering extracts scenarios from the existing system, and then uses automatic synthesis to produce a formal state-machine based model from these scenarios (see Figure 1). The enabling technology for our scenario-based re-engineering process is automatic synthesis of SDL models from a set of MSCs [3]. So far automatic synthesis of SDL models from MSCs was considered only as a forward engineering technology. In our scenario-based re-engineering process we exploit the duality of MSCs as both a requirements capturing language and a trace description language. This allows us to treat dynamic probe traces and/or static source trajectories as requirements for the SDL model.

There are several strategies for extracting scenarios from the system. Scenarios can be extracted dynamically as probe traces, collected by executing the instrumented legacy system (see Figure 1). The execution is driven by a functional test suite. The dynamic strategy has the following steps:
- place semantic probes into the legacy code at strategic locations based on structural analysis of the code,
- select key representative tests from the regression test database and other sources,
- execute the scenarios by the legacy code to generate probe sequences, which are then converted to MSCs with conditions.

Alternatively, scenarios can be extracted statically by manual navigation through the source code. In this static strategy we produce the so-called source trajectories [8].

The static strategy is an exploratory process, which involves
- Selecting a seed location
- inspecting various paths through the seed location to a certain depth (across different procedure boundaries, component boundaries and process boundaries), and
- validating each path as representing some primary use case, an exception to some primary use case, or a some utility functionality.
Let us introduce some terminology for discussing scenarios. We make a distinction between primary scenarios (normal, everything works as expected, success paths) and secondary scenarios (alternative, exceptional, race conditions, collisions, known pathological sequences of client/system interactions, fail paths). All functional scenarios (scenarios which describe how a user achieves a particular service or capability) are primary. Scenarios, which describe how he/she was thwarted, are secondary. In general, scenarios, which are essential and desired by a customer, are primary.

Primary scenarios are denoted “low-yield” since they describe situations and actions, which are generally well understood. The yield (detected or anticipated error count) is therefore low. Secondary scenarios, on the other hand are denoted moderate or high-yield, since they describe situations and interactions, which are generally not well documented, and therefore are not well understood. The associated yield for such scenarios is high because designer choices are likely to differ from client choices, or to be non-deterministic.

Central to our scenario-based approach is the idea of extracting core scenarios, which approximate the intended use cases of the system. The objective of the extraction process is to obtain a minimal set of scenarios, which represent the original business objectives of the system (primary scenarios and key secondary scenarios). Here dynamic and static strategies complement each other. When applying dynamic extraction of scenarios we try to leverage existing functional tests. When applying static extraction we directly analyze the system architecture [10].

Our methodology is an iterative process, consisting of the following four phases.
1. Preparation
2. Extraction of scenarios to approximate original intended use cases
3. Synthesis of SDL model
4. Assessment of the SDL model

Each phase involves a few steps. Iterations are controlled by validity criteria, which are checked during the last phase. A detailed description of methodology steps is contained in the next section.
3. Detailed steps in the methodology

3.1. Preparation phase

This aim of this phase is to develop a probe placement strategy and select the set of scenarios which will drive execution of the instrumented system and resulting probe trace capture.

Step 1. Analyze code. This step uses well-known methods of static architecture analysis [10] to select probe placements and locate primary scenarios. Two models of software can be used as guidelines for probe placement - the architectural model of the system (major components and their relationships) and the call graph of the system [10]. The call graph of the system should identify external interfaces of the system (usually - system calls of the target operating system, or assembly inline code).

Step 2. Select modeling viewpoint. Our approach may be used to re-engineer legacy code from a black-box (environment), white-box (core code), or grey-box (collaborations among subsystems) point of view. Viewpoint affects the level of details in traces and therefore determines the structure of the resulting SDL model.

Step 3. Set coverage goal and select probes. At this step we finalize probe placement by selecting particular locations in the source code of the system where probes are to be placed, and define the format of the information generated by each probe. By selecting the coverage goal we control the level of details in traces and thus determine the external interface of the model. This external interface is determined by locations in the architectural model of the system and the call graph, such that probes register desired events and collect desired data.

In order to achieve approximation of the original intended use cases of the system, semantic probing is performed. The coverage requirement is not phrased in terms of syntactic entities such as statements or branches, but in terms of semantic entities, namely equivalence classes of program behavior [9]. These equivalence classes of program behavior are determined solely from the system design. Probe traces obtained by executing instrumented code can be related directly to the system design. Inspection of probe traces may drive modification of semantic probes and thus lead to further iterations of the re-engineering process. Semantic source trajectories can also be build directly from the source code using our static extraction strategy. See Section 5, “ToolExchange case study” for an illustration of probe placement strategy.

Step 4. Collect known primary scenarios + regression tests. The dynamic capture of probe traces is driven by the test suite. We suggest that the (legacy) regression test suite of core functional test cases be used to drive the first iteration of scenario-based methodology.

We start our iterative re-engineering process with existing regression tests. Regression tests consist of a blend of conformance tests (usually success paths and therefore low-yield), primary scenarios (low-yield), and a few known important secondary scenarios (moderate to high yield). We continue with additional functional
(primary) scenarios as required to improve the semantic capture of our SDL model. As our iterations converge, we become more interested in secondary (higher-yield) scenarios.

3.2. Approximating original use cases by traces and trajectories

The aim of this phase is to build an approximation of the original use cases with traces (dynamic strategy) and trajectories (static strategy). A practical re-engineering process should use a combination of both strategies to achieve rapid convergence of iteration.

3.2.1. Dynamic collection of probe traces

The aim of this phase is to capture the set of probe traces, which correspond to the probe placement strategy and selected scenarios.

Step 5. Instrument legacy. Suitable probing infrastructure for generation and collection of probe traces needs to be established. Probes need to be inserted into the source code according to the placement strategy. Discussion of the technical details of instrumenting legacy is outside of the scope of this paper.

Step 6. Run legacy code to generate probe traces. The legacy system needs to be built and executed on a test suite. The target or simulated environment together with the existing testing infrastructure is used. The result of this step is a collection of probe traces. Another result of this step is the measurement of probe coverage of the system by the current test suite.

3.2.2. Static extraction of source trajectories

Step 7. Extract primary source trajectories. There are certain benefits of extracting scenarios directly from the source code of the system without building and executing. For example, the run-time environment may include some costly hardware. It is possible to build scenarios manually, by navigating through the source code and capturing the corresponding events. In [8] we discuss some details of this method and present a cost effective tool support, based on our reverse engineering toolkit inSight [10] and the MOST toolkit [2].

3.3. Synthesis of SDL model

This is the key phase in our methodology. The aim of this phase is to synthesize an SDL model of the legacy system.

Step 8. Translate probe traces into event-oriented MSCs. This step was introduced into the methodology in order to separate two different concerns - dynamically capturing scenarios from legacy and synthesizing SDL models from scenarios. This step performs a (simple) translation between traces and MSC. This step is determined mostly by the
differences between the format of probe traces (as defined at the instrumentation step), and the format of input to the synthesizer tool.

Step 9. **Add conditions to MSCs.** This step was described as “abstraction” in Figure 1. The aim of this step is to identify transaction-like sequences of interactions, corresponding to requirement use cases. Then linear MSCs (corresponding to traces) are converted into an MSC model, which corresponds to requirement use cases. This is done by inserting conditions [5] into places where loops or branching are possible.

![Figure 2. Example of adding conditions to MSC](image)

This process is illustrated in Figure 2. The MSC at the left side does not contain any conditions. This MSC describes a single trace. However, we can identify a transaction-like sequence of interactions such as:

\[
\begin{align*}
&x: \text{out} \text{ a to } y; \\
&y: \text{in} \text{ a from } x; \\
&y: \text{out} \text{ b to } x; \\
&x: \text{in} \text{ b from } y;
\end{align*}
\]

Two such sequences are repeated. Provided we know that this sequence can be arbitrarily repeated, we can add conditions before and after this sequence as demonstrated at the MSC at the right side. The MSC at the right side describes an infinite number of traces (including the one described by the MSC at the left side). Thus adding conditions to MSCs can significantly improve the amount of information, contained in MSCs which will lead to synthesis of models with more interesting behavior.

Step 10. **Synthesize SDL model.** This step is done automatically by importing scenarios into the MOST toolkit and running the MOST-SDL synthesizer tool. Synthesizer technology is briefly described in the next section. A more detailed description is contained in [2].

The outputs of this step are the 1) synthesized SDL model; and some complexity metrics of the model: 2) number of states in SDL model and 3) non-determinism metric of the model. The later metric is an indirect termination criteria for the re-engineering process. A non-deterministic choice is generated each time when two or more input scenarios have different behavior on the same external stimulus. In practice this often means that behavior of the system is determined by the previous history, but the traces captured during the previous steps do not contain enough data. High values of the non-determinism metric should lead to further iterations of the re-engineering process.
3.4. Assessment of SDL model

The aim of this phase is to update the original test suite by using automatic test case generation from the SDL model and to check termination criteria by comparing the probe coverage and the test coverage (as well as the non-determinism metric).

Step 11. Terminating criteria. We need to make sure that the generated model adequately captures the behavior of the legacy system. This may require several iterations of the re-engineering process. Inadequate behavior of the model may be caused by at least two factors: 1) some important primary scenario is not captured in (legacy) regression tests; 2) an abstracted interface of the system is incorrectly selected (missing probe or incorrectly placed probe).

A probe can be incorrectly placed when it a) does not correspond to a desired behavior equivalence class (e.g. two different probes are placed in the same equivalence class); b) probe is placed into correct behavior equivalence class, but is placed in an incorrect syntactical place - into a code location which is not executed when at least some locations of the desired behavior class are executed (e.g. probe is placed into only one branch of a conditional statement).

In our experience, incorrectly placed probes result in errors in probe coverage. Missed probes on input interfaces result in high values of the model non-determinism metric. Missed probes on output interfaces result in errors in generated test coverage. Thus when the probe coverage, non-determinism metric and generated test coverage together are satisfactory the iterations can be terminated.

4. The MOST toolkit

The MOST toolkit is the enabling technology for our re-engineering process. Synthesis methodology involves using extended Message Sequence Charts to formalize use cases. The input for the synthesis is a set of MSC-92 with states extended with data operations. The synthesis algorithm produces a flexible object-oriented SDL-92 model. The model is syntactically and semantically correct, and complete with respect to the input set of MSCs. The model extensively uses SDL-92 structured types. The model includes one SDL package consisting of a process type for the system and a separate process type for each actor in the model. The package also contains the block type showing instantiations of processes and collaboration between the system and its environment. The synthesized SDL model is ready to be simulated and validated by commercial SDL tools. More complete presentation of MOST-SDL tool can be found in [2].

5. ToolExchange case study

We have applied our methodology to re-engineer a small-sized telecommunications-like system, called the ToolExchange. The ToolExchange was developed at the Department for CASE Tools, Institute for System Programming, Moscow.
5.1. Description of the ToolExchange system

The ToolExchange implements an integration mechanism for extensible multi-component CASE environments. ToolExchange provides interoperability between loosely connected interactive tools by allowing them to perform remote services. Thus the ToolExchange implements a simple lightweight text-based protocol for communication between tools.

The ToolExchange supports the following functionality. Each tool has a unique symbolic name. When a tool dynamically connects to the ToolExchange it is registered at the ToolExchange as a “subscriber” with unique identifier. A service can be requested either based on a symbolic name or a unique identifier. When the service is requested via symbolic name, the ToolExchange first checks, if there is any active subscriber with such name. If an active subscriber exists, the ToolExchange sends the service request to it. If no active subscriber exists, the ToolExchange launches the tool using the “tool command line” for a particular symbolic name. When the service is requested via a unique identifier, the ToolExchange checks if the particular subscriber is still connected and sends the service request to it. The ToolExchange establishes a temporary connection between the service requestor and the service provider, by sending the unique identifier of the service requestor to the service provider, so that the later can send back the reply.

5.2. Analysis of the system and probe placement strategy

The conceptual view of the ToolExchange is presented in Figure 3. Rectangular boxes represent functional components. Lines represent relationships between components (usually implemented via function calls; each line can represent several function calls; direction of lines represent the direction of function calls). Lines between components and the frame of the picture represent external interfaces of the system. Each external interface has a name (to the left of the corresponding line). Components are arranged according to their logical level (low-level components are at the bottom, higher-level components are at the top). Such architecture diagrams can be produced e.g. using inSight reverse engineering tool [10] from the source code of the system.

The Driver component implements the low-level protocol based on UNIX sockets. The Driver contains the main event loop, which handles messages from already opened sockets (interface FromSocket) as well as new connections (interface Connection). It also encapsulates sending messages to sockets (interface ToSocket). The Driver component consists of several modules: low-level socket communication, socket connections, interpreter of the low-level message format, etc. The Initialization component allows reading an initial sequence of requests from a file (interface File). The ToolLaunch component implements the protocol for launching new tools (interface Launch).

The ToolList component represents the key abstraction of the ToolExchange - the list of current subscribers. This list is used to search for an active subscriber with a given name. Elements of this list can be marked as active or locked based on requests from Control component, which implements the logic of processing requests.
performed internally by the ToolExchange. The Connection component represents the logic of adding new subscribers. The Communication component represents the logic of handling service requests. The Extensions component represents the logic of handling events and customizable menus (outside of the scope of this case study).

Implementation of the ToolExchange system consists of 14 modules in C++ language. The implementation contains 110 functions. The total size of the implementation is 1128 LOC. There exists a small regression test suite, consisting of 15 test cases.

### 5.3. Deriving scenarios

The external interface of the ToolExchange consists of `recv` and `send` UNIX system calls (interfaces `ToSocket` and `FromSocket`).

The ToolExchange waits for service requests and connections in an endless loop in function `MainLoop` (Driver component). The service request is analyzed by `processMessage` function of the Driver component. The `processMessage` function calls one of the higher-level functions in components Communication, Control, Connection or Extensions. Most service requests will be forwarded to some other tool. Sending requests via tool name is done by the Communication component, which will later again call some functions of the Driver component to send the message. All call sequences originate in the Driver component. Most call sequences go into one of the higher-level components (Communication, Connection, Control or Extensions), then visit the ToolList component and then go back to the Driver component. Some call sequences after going into ToolList component will go into ToolLauncher component and terminate by issuing a “launch request”. A special call sequence originates in the Driver component, then goes into the

---

**Figure 3. Architecture of the ToolExchange**

Implementation of the ToolExchange system consists of 14 modules in C++ language. The implementation contains 110 functions. The total size of the implementation is 1128 LOC. There exists a small regression test suite, consisting of 15 test cases.

### 5.3. Deriving scenarios

The external interface of the ToolExchange consists of `recv` and `send` UNIX system calls (interfaces `ToSocket` and `FromSocket`).

The ToolExchange waits for service requests and connections in an endless loop in function `MainLoop` (Driver component). The service request is analyzed by `processMessage` function of the Driver component. The `processMessage` function calls one of the higher-level functions in components Communication, Control, Connection or Extensions. Most service requests will be forwarded to some other tool. Sending requests via tool name is done by the Communication component, which will later again call some functions of the Driver component to send the message. All call sequences originate in the Driver component. Most call sequences go into one of the higher-level components (Communication, Connection, Control or Extensions), then visit the ToolList component and then go back to the Driver component. Some call sequences after going into ToolList component will go into ToolLauncher component and terminate by issuing a “launch request”. A special call sequence originates in the Driver component, then goes into the
ToolLauncher component and then goes back into the ToolList, one of the higher-level components and then back into the Driver component. This allows the ToolExchange to launch a new tool, wait until it connects and then send the request to it.

We have experimented with three different probe placement strategies using the same test suite (see Table 1).

**Table 1. Probe placement strategies**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Name</th>
<th>Description</th>
<th>MSC (LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Medium</td>
<td>probes inserted inside the Driver component (at functions, encapsulating message send and receive)</td>
<td>610</td>
</tr>
<tr>
<td>1</td>
<td>High</td>
<td>probes inserted at high-level interface of the Driver component (SendMessage, ProcessMessage functions)</td>
<td>408</td>
</tr>
<tr>
<td>2</td>
<td>Low</td>
<td>probes inserted at the operating system interface</td>
<td>44852</td>
</tr>
<tr>
<td>2</td>
<td>Medium</td>
<td>probes inserted at complete medium level (12 more functions added)</td>
<td>10000</td>
</tr>
<tr>
<td>2</td>
<td>High</td>
<td>probes inserted at complete high level (2 more functions added)</td>
<td>1600</td>
</tr>
</tbody>
</table>

Table 1 gives brief explanation of each probe placement strategy and shows the volume of the generated MSC models (converted from probe traces) for each probe placement strategy. The volume of the trace is roughly the product of the number of events and the amount of data handled by each event. As expected, traces of the higher-level protocol have less volume than the lower level protocol. Investigation of the probe coverage after the first iteration allowed us to place probes on some additional functions, which significantly improved the coverage (and hence the volume of the trace).

We applied static extraction strategy. Considering the size of the system it was very easy to extract 11 scenarios, which exactly correspond to the original use cases of the system. These scenarios consist of 493 LOC MSC, which is quite close to scenarios, extracted dynamically according to the high-level probe placement strategy.

Table 2 shows the resulting probe coverage of components. Most of the probe hits are inside the Driver component. It is interesting to observe that moving to a low-level placement strategy from a medium-level one did not increase the coverage of the high-level components.

**Table 2. Probe hits per component**

<table>
<thead>
<tr>
<th>Component</th>
<th>High</th>
<th>Medium</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver</td>
<td>286</td>
<td>1553</td>
<td>9840</td>
</tr>
<tr>
<td>Communication</td>
<td>37</td>
<td>74</td>
<td>74</td>
</tr>
<tr>
<td>Connection</td>
<td>14</td>
<td>186</td>
<td>186</td>
</tr>
<tr>
<td>Control</td>
<td>2</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Extensions</td>
<td>47</td>
<td>89</td>
<td>89</td>
</tr>
<tr>
<td>Initialization</td>
<td>0</td>
<td>122</td>
<td>186</td>
</tr>
<tr>
<td>ToolList</td>
<td>0</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>ToolLauncher</td>
<td>0</td>
<td>71</td>
<td>71</td>
</tr>
</tbody>
</table>
5.4. Synthesizing SDL models from scenarios using the MOST toolkit

We applied the MOST-SDL Synthesizer Tool to MSC models, captured at the previous step. We also examined MSC models and added conditions. Table 3 summarizes synthesized SDL models. Table 3 shows the volume of the synthesized SDL models, the number of states in each SDL model and the non-determinism metric (# any) of each SDL model (see step 10 of the methodology).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>MSC model</th>
<th>SDL (LOC)</th>
<th># states</th>
<th># any</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Medium-level probes, no conditions</td>
<td>554</td>
<td>68</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>Medium-level probes, simple loop conditions</td>
<td>120</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>High-level probes, no conditions</td>
<td>250</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>High-level probes, loop conditions</td>
<td>155</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Medium-level probes, no conditions</td>
<td>9784</td>
<td>1391</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>High-level probes, no conditions</td>
<td>1653</td>
<td>227</td>
<td>10</td>
</tr>
</tbody>
</table>

An analysis of Table 3 shows that high level probes cause less non-determinism compared to medium level probes (lines 1 and 3, also lines 5 and 6). Adding conditions to the medium level probes resulted in folding loops which leads to a dramatic reduction in the size of the synthesized model (lines 1 and 2 and also lines 3 and 4).

We observed that higher-level probes result in more meaningful models. This happens because more logical information becomes available as probes are inserted higher in the call graph of the system: low-level functions send, recv operate with socket identifiers; sioSend, sioReceive functions operate with connection identifiers; processMessage, sendMessage functions operate with tool identifiers. It is thus easier to relate high-level information to the architectural and logical models of the system.

6. Relation to other approaches and conclusions

We presented our scenario-based approach to re-engineering legacy telecommunications software. Our approach consists of
1. Extracting scenarios from the legacy software using a combination of dynamic and static strategies to approximate the original use cases of the system and
2. synthesizing an SDL model from this set of Message Sequence Charts (MSCs) using the MOST toolkit.
This process is repeated until the SDL design model satisfies certain validity constraints. This SDL model is then used to assess and improve the quality and coverage of legacy system tests, including core functional tests. The approach may be used to re-engineer and potentially also re-test legacy code from a black-box (environment), white-box (core code), or grey-box (collaborations among subsystems) point of view.

The alternative approach to re-engineering of legacy software into SDL models is the so-called direct automatic re-engineering [6]. The direct re-engineering approach derives an SDL model statically from the source code by performing semantic-preserving translation. Thus the direct SDL model contains at least the same amount of information as the implementation itself. In fact, directly generated SDL models contain on average 8-12 times more information than the implementation, because the mapping from a conventional language to SDL is divergent. In contrast, SDL models, which are synthesized according to our dynamic scenario-based approach always contain less information than the implementation.

Both kinds of SDL models are trace-equivalent with respect to the traces, produced by the test suite. However, a directly generated SDL model is capable of producing more traces, than those produced by the original test suite, while a scenario-based SDL model is fully defined by the original test suite.

On the other hand, traces produced by the SDL models have different levels of detail. Traces produced by a directly generated SDL model contain all implementation detail, plus some additional detail, introduced by the mapping. The level of detail of directly generated SDL models can be controlled by selecting an external interface of the implementation. Traces produced by a scenario-based SDL model are expected to contain much less detail. As demonstrated above, the level of detail of the scenario-based model is controlled by the probe placement strategy.

The direct approach has certain advantages: it is independent of (legacy) regression tests, and it is usually easier to achieve complete semantic coverage of the legacy. However, there are some disadvantages as well: direct mapping has to handle larger volumes of base software platform source code, therefore - SDL tools need to handle larger SDL models. The biggest advantages of scenario-based approach compared to direct approach, is the ability to produce a broad range of distinct models by varying input scenarios and probe placement strategies. In general, a scenario-based approach yields more abstract models, which are free from implementation detail. Thus SDL tools could be easier applied to such models.

In our experience, the use of dynamic scenario-based re-engineering methodology combined with subsequent use of SDL tools allows between 20 and 25 % speedup in time-to-market for a typical telecommunication system. The use of tools alone in a related project was found to yield a 20-30% improvement in time-to-market; therefore the estimate above is likely quite conservative.

As compared to a direct static re-engineering approach, the dynamic scenario-based approach has greater potential for creating abstract SDL models thus avoiding the confusion and complexity of existing “spaghetti code” in legacy systems. Dynamic derivation of scenarios is a cost-effective way of capturing data manipulations within the base software platform.

Our methodology provides an efficient means of improving understanding of both legacy code and regression (core functional) test suites. Automatic re-engineering of
legacy telecommunication software into SDL models is a key prerequisite for adoption of SDL methodology in industry. Thus, our approach appears to be a cost-effective means of removing barriers to full adoption of SDL in industry. Such an approach as ours appears to be a highly effective and efficient means of capturing core user intentions and therefore core system behaviours. This provides a necessary starting point and the necessary guidance for the orderly evolution of communication software.

7. References

9. R. Probert, Grey-Box (design-based) testing techniques, in Proc. of 15th Hawaii Int. Conf. on System Science, 1982, pp. 94-102.