Automatic Synthesis of SDL from MSC and its Applications in Forward and Reverse Engineering

Nikolai Mansurov

Department for CASE tools
Institute for System Programming,
25 B. Kommunisticheskaya,
Moscow 109004, Russia
Email: *nick@ispras.ru*

Abstract

Wider adoption of formal specification languages in industry is impeded by the lack of support for early development phases and for integration with older, legacy software. Methodology aimed at improving this situation is presented. The methodology uses Message Sequence Charts (MSC) as a "front-end" specification language and systematically applies an automatic synthesis technique to produce executable specifications in the telecommunications standard Specification and Description Language (SDL). Applications of the automatic synthesis technique for both forward and reverse engineering are demonstrated.

Keywords: requirements engineering, formal specifications, synthesis of programs, use cases, scenarios, MSC, SDL, forward engineering, reverse engineering

1. Introduction

New generation Computer-Aided Software Engineering (CASE) tools based on Formal Description Techniques (FDTs) are aimed at practical improvements of software engineering in the telecommunication industry. CASE-based approaches offer significant improvements in quality, productivity, and time to market [5]. However there exist certain *barriers* for wider adoption of formal specification languages in industry. We identify two major barriers – support of *early development phases* [8,4] and support for integration with *legacy* software [17,2].

Design practice at the early phases of the software development process is not adequately supported by mathematical-based formal methods [8]. Requirements capture is an iterative and exploratory process. At this phase tentative descriptions of the system are suggested and frequently modified. In the initial phases of a design, comprehensive formal specification and verification techniques offer little help to the designer [8]. They appear to require a level of formality and precision that is not available yet. In return, only fairly abstract properties may be established. According to [8] the initial price to be paid is too high, the initial rewards are far too small.

Instead, the so-called *use case based methodologies* are becoming predominant in software development [9,16]. Use case based methodologies share the common way of capturing the customer requirements as *scenarios*. Message Sequence Charts (MSC) [7] or Sequence Diagrams of the Unified Modeling Language (UML) [16] can be used to model use cases. The MSC language is especially attractive as a formal description technique (FDT) for the early phases of the software development process because it is well accepted in the telecommunications industry and has a low learning curve, while at the same time it

has a well-defined formal semantics. We believe that significant improvements of the time-to-market can be gained by expanding the use of FDT-based CASE tools to the early phases of the software development process [5,4,8].

Apart from the support for the early design phases, there is another important issue, which needs to be addressed in order for the formal methods-based CASE tools to become common practice in industrial software engineering. Usually, formal methodologies are only applicable to projects, in which the system is developed completely from scratch. However, most projects in the industrial context involve older, "*legacy*" software. This software is being maintained, ported to new hardware and software platforms, updated by developing new features, or reused in new projects. For formal methods to be adopted in such projects, it is necessary to provide cost-effective methods for integrating CASE-produced components with "legacy" base software [1]. 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 are often very poorly documented. Up to now, this fact has constituted a "*legacy barrier*" to the cost effective use of formal methods-based development technologies and tools [2,17]. In order to overcome the "legacy barrier", there is an increasing demand for developing cost-efficient re-engineering methods, which will significantly reduce the effort involved in creating formal specifications of the base software platforms.

In this paper, we describe our experience in addressing methodological issues for wider adoption of formal methods and the corresponding FDT-based CASE tools in telecommunications industry. We have selected MSC as the "interface" formal method, intended for use by humans. An automatic synthesis technique aimed at producing executable specifications in another telecommunications standard formal language called Specification and Description Language (SDL) [6] was developed. We discuss methodological issues of using MSCs and synthesized SDL specifications at both early and later phases of the development process. Applications of the synthesis techniques for re-engineering formal specifications of base software [2] are discussed in a separate section.

The rest of the paper has the following organization. Section 2 contains a brief introduction into MSC, describes some data extensions and outlines our approach to formal modeling of requirements at the very early phases of software development. Section 3 presents the key concepts of the synthesis algorithm. Section 4 describes applications of synthesis in forward engineering. We describe high-yield requirements validation and architecture validation techniques. We also provide comparison of several related approaches to using synthesis in forward engineering. In Section 5 applications of synthesis in reverse engineering are discussed. We present our dynamic scenario-based approach to re-engineering of formal SDL specifications of legacy telecommunications software. We provide comparison to some related approaches to re-engineering formal specifications. Section 6 concludes the presentation.

2. MSC as a "front-end" formal specification language

We suggest using MSC [7] as a "*front-end*" formal specification language of an FDT-based CASE tool. Requirements for a "front-end" formal specification language include ease of use, low learning curve, very quick turn-around cycle, capability for rapid modification and maintenance of the specification. On the other hand, the CASE tool should support one or several "*back-end*" formal specification languages

amendable for formal verification, generation of test cases, generation of application code, etc. “Back-end” formal specification languages have more comprehensive requirements on the depth, and mathematical precision of description. SDL [6] is an illustration of a “back-end” specification language.

Attempts to apply a “back-end” specification language at the early phases of development will probably create an impediment. We suggest using *automatic synthesis* to produce “back-end” specifications from “front-end” specifications at the early phases of development. We believe that this methodological step is essential to utilize the latent capacity of formal methods to improve time-to-market in an industrial context.

In this paper we explore a combination of MSC as the “front-end” formal specification language and SDL – as it’s "back-end". Obviously, other combinations of “front-end” and “back-end” specification languages are possible. Separation of a “front-end” specification language allows utilizing several “back-end” specification languages in order to increase the capabilities of an integrated set of tools.

2.1. Overview of MSC language

This section provides a brief overview of Message Sequence Charts specification language. Complete definition is contained in [7]. Note, that the MSC language used in this paper is based on the so-called MSC-92 standard, to which certain extensions were added. The current standard, the so-called MSC-2000 language defines several powerful constructs, which are not covered here.

The MSC language has a graphical notation. There are two kinds of MSC diagrams: Basic Message Sequence Charts (bMSC), and High-Level Message Sequence Charts (HMSC). A textual representation of MSC specifications is also available [7].

Let’s consider an illustrative basic MSC (see Figure 1). A basic MSC diagram describes behavior of several *instances*. Each instance is graphically represented as a vertical line (called *instance axis*). Each instance has an *instance head*, which contains the *name* of the instance. An instance axis corresponds to the timeline of the instance. Instances exchange *messages*, which is shown as arrows between two instances, or between an instance and the frame of the diagram. An instance can be *created* by another instance, which is shown as a dashed arrow pointing at the instance head of the child instance.

Message sequence charts can use *timers*. Figure 1 shows how timer T is *set* by instance c, and then expires, resulting in a *timeout*. An instance can also *reset* a timer (not shown at Figure 1).

A basic MSC describes *events* for each instance. Events on each instance axis are *ordered*: if the first event occurs higher on the instance axis than the second one, than the first event occurs “*before*” the second one. Related pairs of message output and message input introduce another order: message output occurs “*before*” the corresponding message input. Semantics of an MSC specification is a (transitive) *partial ordering* of the events for all instances in all basic MSC diagrams.

The MSC standard [7] has syntax for some data-related aspects of specifications, but does not describe any semantics for them. Data-related elements of MSC specifications include parameters of messages, parameters of create events, parameters of timers and actions. Section 2.2 contains a brief description of our extensions of the data handling aspects of MSC specifications.

Composition of basic MSCs is described by *conditions*. Composition of basic MSCs is represented graphically using HMSC diagrams. An HMSC diagram contains *references* to basic MSC diagrams and *flowlines* (see Figure 2).

2.2. Data Extensions to MSC

In order to allow more power to specification of scenarios, we have introduced some rather pragmatic extensions to the data handling capabilities of the MSC-92 language [4]. Note, that the current MSC standard, the so-called MSC-2000, defines advanced data concepts. For discussion of data aspects in the new MSC standard see [18].

Our motivation is simple enough: use the semantics of the data handling operations from the "back-end' language (for example SDL [6]) and use the automatic synthesis to transform data handling from the "front-end" MSC specifications into the "back-end" specification. Therefore, the semantics of our data extensions is defined by the automatic synthesis algorithm.

1. **Variable definitions**. We allow to define variables of different types. Variable definition is placed into a text symbol in any MSC diagram. A *local* copy of each variable is created for each actor. Simplified SDL syntax is used for variable definitions:

```
<variable definition> ::= dcl <var_name> <type>;
```

2. **Actions**. We allow MSC action symbols to contain operations on local variables. Simplified SDL syntax is used for actions:

```
<assignment> ::=    <var_name> := <expr>
<function call> ::= <func> (<expr1>,…,<exprn> )
```

3. **Message parameters**. We allow messages to have parameters. We restrict the syntax of message parameters to variable names.
4. **Create parameters**. Actors are allowed to have parameters which are passed from the parent instance to the child instance during the create event. We restrict the syntax of create parameters to variable names.
5. **Local conditions**. We allow to specify local decisions using boolean expressions over instance variables. Syntactically, local decisions are specified as local conditions on the axis of the corresponding instance. The boolean expression is written in a comment box attached to the local condition. Semantics of such condition is that the subsequent events are considered only when the value of the boolean expression is true. Boolean expressions are restricted to the following syntax:

```
<boolean expression> ::= <var_name> <op> <var_name>
                         <var_name> <op> <const>
```

Alternative sequences of events can be specified in a different MSC using a local condition with the same name and a different guard. All guards must be mutually exclusive. Decision is *local* to the instance with a local condition.

6. **Timers**. Subsequent *set* and *timeout* events on an MSC instance axis may be used to specify a delay during use case execution. In an abnormal scenario such delay may specify an expired timeout, which causes an error. Timers with parameters [7] are not supported.

3. Synthesis of Executable SDL Specifications from MSC

In this section we present our technique for synthesis of "back-end" SDL specifications from scenarios formalized in a "front-end" MSC. We describe the key concepts of the synthesis algorithm, and provide an illustrative example. More detailed description is presented in [19].

3.1. Synthesis Algorithm

Our synthesis algorithm is based on the concept of *event automata*. An **event automaton** is a finite automaton corresponding to *an MSC instance*, such that the alphabet of the input symbols for the automaton is the same as the alphabet of *MSC events* of the given MSC instance. Our synthesis algorithm constructs a particular kind of event automata, which we call *MSC slices*. An **MSC slice** (corresponding to an MSC instance *i)* is an event automaton, accepting all valid event sequences for the instance *i*.

In this paper we distinguish between three categories of MSC events: *input*, *active* and *idle* events. Note, that an idle event is not defined in [7]. An idle event is a trivial (empty) event, which was added to simplify algorithm descriptions.

- **input events** require synchronization with other instances, decision about event is taken by another instance:
  - input of message *m* by instance *i*: *in(i,m)*
  - create instance *i* by instance *j*: *create(i,j)*
  - timeout of timer *t*: *timeout(t)*
- **active events** do not require synchronization with other instances, decision is local to the current instance:
  - output of message *m* by instance *i*: *out(i,m);*
  - action *a*: *action(a);*
  - set timer *t* for duration *d*: *set(t,d);*
  - reset timer *t*: *reset(t);*
  - stop action: *stop;*
  - local condition over variable *v* with condition *c*: *check(v, c);*

The following algorithm can be used to construct MSC slices:

1. initial states of the event automaton correspond to symbols at HMSC graph with idle events;
2. for each basic MSC a (sub)sequence of states is created, corresponding to the sequence of events involving the instance *i;*
3. each MSC reference is replaced by the corresponding (sub)sequence of states;
4. the start symbol of the event automaton corresponds to the HMSC start symbol;

Our synthesis algorithm consists of the following steps [4]:

1. Integrate HMSC model
2. Construct *MSC slices*
3. Make MSC slices deterministic

4. Minimize MSC slices
5. Generate SDL behavior
6. Generate SDL structure

3.2. Example

Let's consider a simple MSC model shown in Figure 3. It contains two use cases *Wait* and *Reply* (each of them has only one scenario). We assume sequential composition rule. Instance *R* (*receiver*) corresponds to the system actor and instance *S* (*sender*) is an external actor.

Sender S initiates both use cases. Use case *Wait* is started by sender S sending message X. Receiver R has to wait for an unspecified period of time. Use case *Reply* is started by sender S sending message Y. Receiver R has to respond with message W.

Figure 4 illustrates our algorithm for constructing an MSC slice for instance S. The first step of the algorithm constructs an event automaton from the HMSC graph (step 1). Then an event automaton is constructed from the instance S at MSC *Wait* (step 2). This automaton has only one transition, labeled with an active event *out(x,r).* Then the corresponding transition in the initial event automaton is substituted for the newly created event automaton for instance S from MSC *Wait* (step 3). The following two steps (steps 4 and 5) process instance S from MSC *Reply*. The resulting MSC slice is presented in the bottom right corner of Figure 4. It is labeled non-deterministic event automaton (NEA).

Figure 5 illustrates subsequent steps of our synthesis process. The non-deterministic MSC slice for instance S (NEA S) is made deterministic (in the event automata sense) and minimized (DEA S). An SDL graph is then generated (process S). Figure 6 illustrates the generated SDL structure for our example.

4. Applications in Forward Engineering

4.1. High Yield Requirements Validation

In this section we describe the use of synthesized SDL models for requirements validation. **Requirements validation** is a systematic approach to detect faults in the customer requirements [5]. Requirements validation is a form of testing applied to an early phase. Requirements validation is an iterative process consisting of the following steps (see Figure 7):

1. Formalize requirements in the form of use case scenarios
2. Synthesize executable requirements model from scenarios
3. Create *validation scenarios*
4. Run validation scenarios through the requirements model
5. Validate the execution sequence of each validation scenario to:
   - 5.1. Accept the validation scenario. In this case the validation scenario can be included into requirements use cases
   - 5.2. Reject the validation scenario. In this case the initial customer requirements contain a fault. E.g. the initial requirements can be inconsistent or incomplete. The rejected validation scenario has to be transformed into a use case and the initial requirements need to be updated by including the new use case and removing any existing inconsistencies.

6. Check termination criteria and start with a new iteration, if necessary (starting from step 2).

High-yield requirements validation approach is being developed by Prof. R. Probert at the Telecommunications Software Engineering Research Group (TSERG) of the University of Ottawa [5]. Same considerations are applicable to requirements validation as to testing: one cannot test a program to guarantee that it is error-free. Since exhaustive testing is out of the question, we must maximize *yield* on the testing investment (i.e., maximize the number of errors found by a finite number of test cases. *Yield* of a validation scenario refers to the number of defects detected at by a particular scenario. An alternative definition of yield can refer to the amount of risk removed by the scenario. *Risk* refers to the "cost" associated with a failure. By definition, Risk R = PF * CF (where PF is the probability of failure; CF is the cost of failure).

Let us introduce some terminology for discussing *validation 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. Essential scenarios, which are desired by a customer, are primary. Primary scenarios are denoted "*low-yield*" since they describe situations and interactions, 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.

The objective of the high-yield requirements validation is to focus the effort on the elements with highest risk. *Low-yield* scenario is not likely to detect a defect by causing an observable failure because such scenarios are in general well understood by both the customers and the developers. On the other hand, *high-yield* scenarios have a high probability of detecting a defect. High-yield scenarios correspond to the secondary scenarios, e.g. exceptional behavior (error-handling behavior path). Usually, these scenarios are less well understood by the developers.

In [4] we suggested to automatically synthesize an SDL requirements model at the requirements analysis phase (Figure 7). Automatic synthesis of SDL requirements models from MSC has the following benefits:

- MSC modeling allows high-yield requirements validation by *simulation* of SDL models using high-yield scenarios
- MSC models can be developed concurrently while architecture integrity can still be maintained via iterative synthesis
- Regression testing is eliminated because accepted validation scenarios are added to the set of validation scenarios and the synthesized model is by construction correct with respect to the previously accepted behavior
- Early fault detection can be performed by the synthesizer
- Different compositions of use cases can be explored (single, concurrent, etc.)
- Slices of the MSC model can be created, explored and reused

The *synthesized requirements model* (SRM) can be used to generate additional scenarios, which are *longer* than the original validation and therefore provide better

understanding of the requirements [14]. Simulation of the synthesized requirements model allows to quickly discover inconsistencies and incompleteness of the requirements because the synthesized model will generate many variations of the original scenarios, including abnormal behavior. Such scenarios are likely to be less well understood by the developers.

4.2. Architecture Validation

Automatic synthesis can also be used at the system analysis phase. During this phase the *architecture* of the system is defined and system scenarios for each architecture component are produced (often by independent development teams) (Figure 8). Formally, the input at this phase is a set of system scenarios. A *system scenario* is a refinement of the corresponding scenario from the requirements analysis phase, capturing the interaction of the architecture components. Structural information available in the system scenarios consists of the set of external actors and the architecture components. In the MSC model architecture components are represented as distinct instances. Behavioral information is available in the form of functional scenarios representing the typical interactions between the architectural components as well as between external actors and the architectural components. Additional behavioral information can be captured in the form of the data flows over the system scenarios. The automatically synthesized model created at this phase is called the *synthesized architecture model* (SAM). SDL tools can be used to explore the SAM in order to validate both the system scenarios and the architecture model (Figure 8). Architecture validation process is a direct continuation of requirements validation process described in section 4.1.

Additionally, automatic synthesis performs *integration* of the model from multiple views produced by independent development teams. Behavior of each process of the SDL model is synthesized by considering all interactions involving the corresponding architecture component in all scenarios. Final integration of the model occurs during synthesis of the SDL structure. Integration process reproduces architectural components of the system and their relations by deriving SDL blocks and channels from system scenarios. Blocks of the SDL system are synthesized from the instances in all system scenarios. Channels between the synthesized blocks are derived from interactions between the corresponding instances in all scenarios. Automatically derived relationships between components can be compared to the intended ones, which are described in the architecture model. In our experience inspection of the synthesized architecture model is helpful in uncovering system analysis faults.

Vertical decomposition of MSC models can be used in conjunction with our synthesis technique to seamlessly refine requirements models into architecture models. Figure 9 demonstrates this approach. In use case *Reply* from example at Figure 3, instance R is decomposed into three instances: R1, R2, R3. Message flow between decomposition instances has to be compatible with the message flow at the parent MSC diagram. Additionally, two alternative behavior paths are now specified for the *Reply* use case. Alternatives are specified using MSC condition with same name. The synthesized architectural SDL model is shown at Figure 10.

4.3. Comparison to related work

Automatic synthesis of executable models from scenarios is an active research field. Much work has been done on the subject of translating MSC to other languages

[14,10]. Synthesizing SDL specifications from MSC is addressed in [10]. Survey of research on a more general subject of protocol synthesis is available in [13].

Methodological issues of using MSC for early development phases are considered in [8]. This paper describes synthesis of formal "back-end" specifications in Promela language aimed at formal requirements verification. The possibility of synthesizing executable specifications in the form of finite state machines was mentioned but not considered as a primary goal of the project.

Methodological issues of generating a formal executable specification from a set of use cases are addressed in [11]. This paper summarizes experience in manually developing a LOTOS specification of an industrial telecommunications standard on the basis of use cases provided by customers. LOTOS tools were used to validate the specification and generate all original use cases as well as additional ones. The main motivation of the project was to use LOTOS tools to analyze and maintain a set of use cases. The benefit of using the formal executable specification for prototyping purposes was emphasized.

The University of Montreal synthesizer [15] translates scenarios with timing constraints into timed automata. The main motivation of the project is to provide formalization of scenarios and ensure the accuracy of requirements analysis.

The Waterloo synthesizer [14] translates MSC models into ROOM specifications. The main motivation of this project is to create an executable architectural model supporting design phases. Firstly, an executable architecture model was considered useful for prototyping purposes. Synthesized ROOM models can be simulated by ObjecTime Developer tool with the possibility to visualize execution sequences as basic MSCs. According to [14], the MSC traces are useful for visualizing execution sequences that are longer that the bMSC scenarios in the original MSC specification and therefore provide a better overview and understanding of the system. Executable architecture models were considered helpful in supporting communication and education of new team members. Secondly, automatic synthesis of architectural models was considered useful in evolutionary prototyping by providing refinements to the model. Designers can modify the synthesized model, execute a number of scenarios, and then feed the results back into the domain of MSC specifications. The possibility of ObjecTime Developer to automatically generate C++ code skeletons was also considered beneficial.

The motivation of the Moscow synthesizer is similar, however we also use automatic synthesis to create executable requirements models. We decided to use SDL as the target language because of the better tool support available for SDL.

The Waterloo synthesizer produces architectural models with both structural and behavioral components [14]. The Waterloo synthesizer derives static process structure based on the instances in basic MSC. Similar approach is taken in the Moscow synthesizer. Additionally, the Moscow synthesizer derives *dynamic process structure* by considering basic MSCs with instance creation and deletion. When synthesizing behavior components, the Waterloo synthesizer considers only message input and output events. The Moscow synthesizer additionally considers timer events and supports *data flow extensions* to the MSC language (variables, message and create parameters, actions and local conditions with guards).

The Concordia University synthesizer [10] translates MSC models into SDL specifications. The main motivation of the project is to eliminate validation of SDL specifications against the set of MSCs by ensuring consistency between the SDL specification and the MSC specification through automatic synthesis [10]. The main characteristic of the Concordia synthesizer is that the architecture of the target SDL

specification is required as an input to the synthesis algorithm and the question of implementability of the given set of MSCs within the given SDL architecture is addressed [10]. Thus the Concordia synthesizer produces only behavioral components. Composition of basic MSCs using HMSC was not addressed in [10] although it was considered as a direction for future work.

Although the Moscow synthesizer was developed independently, some of the technical decisions are similar, e.g. the use of SDL save statement to avoid deadlocks in the synthesized SDL models. However the motivation of the Moscow synthesizer is somewhat different. The Moscow synthesizer produces both the behavioral and the structural components (similar to [14]) which allows to synthesize executable requirements models (similar to [11]) as well as executable architecture models [14]. Consideration of data flows in the Moscow synthesizer allows more accurate capture of the functional requirements as well as more accurate capture of the architectural issues.

5. Applications in Reverse Engineering

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 often very poorly documented. Up to now, this fact has constituted a "*legacy barrier*" to the cost effective use of new development technologies [5,2].

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 (Figure 11):

- better understanding of the operation of the legacy software through dynamic simulation of the SDL model, which 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;

Additional benefits can be obtained for using formal methods *for new feature development* (Figure 12):

- 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.

5.1. Dynamic scenario-based approach to re-engineering

In this section we describe our methodology of *dynamic scenario-based* re-engineering of legacy telecommunications systems into a system design model expressed in SDL [2].

Our approach consists of

- placing semantic probes [2] into the legacy code at strategic locations based on structural analysis of the code,
- selecting key representative scenarios from the regression test database and other sources,
- executing the scenarios by the legacy code to generate probe sequences, which are then converted to MSCs with conditions and
- synthesizing an SDL model from this set of MSCs using the Moscow Synthesizer Tool [4].

This process is repeated until the SDL design model satisfies certain validity constraints [2]. 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 [2].

5.1.1. Overview

Dynamic scenario-based re-engineering of legacy software into SDL models is a process, where an SDL model is synthesized from *probe traces* [2], collected from *dynamically* executing the *instrumented* legacy system (see Figure 13,14). More specifically, in the process of scenario-based re-engineering, the SDL model is synthesized from a higher-level representation - *MSC model* which is abstracted from probe traces. The execution is driven by a *test suite* [2].

The enabling technology for our dynamic scenario-based re-engineering process is automatic synthesis of SDL models from a set of MSCs [4]. So far automatic synthesis of SDL models from MSC was considered only as a forward engineering technology (see Section 3). In our dynamic scenario-based re-engineering process we exploit the duality of MSCs as both a requirements capturing language and a trace description language which allows us to treat probe traces as requirements for the SDL model.

Our re-engineering methodology is an *iterative* process, consisting of the following four *phases*.

1. Preparation
2. Dynamic collection of probe traces
3. Synthesis of SDL model
4. Investigation of the SDL model

Each phase involves a few *steps.* Iterations are controlled by *validity criteria*, which are checked during the last phase. An overview of all steps of the methodology is shown in Figure 13,14. In Figure 14 the methodology is presented as a dataflow diagram. Important *artifacts* are represented as rectangles; methodology steps (sub-processes) are represented by ovals. The main artifacts of our re-engineering process are highlighted. Lines in Figure 16 represent flows of data, which determine the

sequence of methodology steps. A detailed description of methodology steps is contained in the next section.

5.1.2. 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 structural analysis to select probe placements. 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 [3]. 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 and re-test legacy code from a *black-box* (environment), *white-box* (core code), or *grey-box* (collaborations among subsystems) point of view. Viewpoint 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 defining 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. The external interface of the model is determined in terms of locations on the architectural model of the system and the call graph, such that probes register desired events and collect desired data.

Semantic probing [2] is assumed. 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 [2]. 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.

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 be used to drive the first iteration of scenario-based methodology.

We start our iterative re-engineering process with 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 are more interested in secondary higher-yield scenarios. Discussion of the yield of scenarios with respect to requirements validation was presented in Section 4.1.

5.1.3. 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.

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 are 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.

5.1.4. 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 7. **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 8. **Add conditions to MSCs.** 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 [7] into places where loops or branching are possible. We are using an extended event-oriented MSC-92 notation as the input to the MOST-SDL tool [4]. In MSC-96 this corresponds to creating an HMSC.

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 9. **Synthesize SDL model.** This step is done automatically by applying the Moscow Synthesizer Tool (MOST-SDL). Synthesis technique was described in section 3. A more detailed description is contained in [4].

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.

5.1.5. Investigation of the synthesized SDL model

The aim of this phase is to check termination criteria by investigating the probe coverage and complexity metrics of the synthesized model, including a very important non-determinism metric.

Step 10. **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.

5.2. Comparison to related approaches

In this section we compare our dynamic scenario-based approach to the so-called direct re-engineering [1] and the so-called partial re-engineering [17]. Schematic representation of the transformations performed by these approaches is shown in Figure 15.

Direct re-engineering approach derives SDL model statically from the source code by performing semantic-preserving translation [1]. 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*, as demonstrated in [1]. In contrast, SDL models which are synthesized according to our dynamic scenario-based approach always contains *less* information than the implementation.

The so-called *partial re-engineering* [17] is another static approach. It provides an interesting alternative to direct re-engineering. According to this approach, only the framework of the model is extracted automatically (in [17] a state machine model was extracted from a program in C). Extraction of any details of the legacy is controlled by the so-called statements map. The statements map contains all different source statements (in some canonical form) and their translation into the model statements within the automatically generated framework. The statements map is inspected and filled-in manually. By default, the source statements are simply skipped, thus

resulting in quite abstract models. Thus the statements map controls the precision of the extracted model. The statements map is relatively stable to the changes in the source code, which makes this approach suitable for evolutionary re-engineering.

Static approaches have certain advantages over dynamic ones since they are independent of (legacy) regression tests, and they usually easier to achieve complete semantic coverage of the legacy. Another important advantage is that static approaches are independent of the target platform. Static re-engineering techniques in general require considerably deeper analysis of source code, and thus are much more expensive. The disadvantage of the direct mapping is that it has to handle large volumes of base software platform source code, therefore - SDL tools need to handle larger SDL models. Partial re-engineering seems quite promising, since it provides a balance of effort between expensive automatic static analysis and manual modification of the statements map. However, the automatic extraction of the state machine framework can be also quite expensive. In [17] cost-efficient extraction of the finite state machine framework was made possible because of the incidental use of a special notation in the code. This notation was introduced for an unrelated purpose, but made extraction of states fairly trivial [17].

The biggest advantages of dynamic scenario-based approach as compared to direct approach, is the flexibility to produce a broad range of distinct models by varying input scenarios and probe placement strategies. In general, scenario-based approach yields more abstract models, which are free from implementation detail. Thus SDL tools could be easier applied to such models. Both kinds of SDL models are *trace-preserving* 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 two SDL models have different *levels of detail*. Traces produced by directly generated SDL model contain all implementation detail, plus some additional detail, introduced by the mapping [1]. The level of detail of directly generated SDL models can be controlled by selecting external interface of the implementation. Traces, produced by 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*.

6. Conclusions

Support for early phases of the development process and support for integration with older, legacy software are, in our opinion, two major barriers for wider adoption of formal methods in industry. At the early phases, there is “too little” to formalize, while on the other hand, at the later phases there is often “too much” to formalize. However, early formalization is required because it can enable tool-aided feedback and thus allow rapid iterative development. Requirements for an early formalization technique include ease of use, low learning curve, very quick turn-around cycle, maintainability. It is also beneficial to be able to re-engineer formal models of legacy in a cost-efficient way, because it allows to use formal methods for subsequent development of new features, as well as to use tools for better validation of the base software platform.

We presented the methodology in which MSC is used as a “front-end” specification language and the automatic synthesis technique is applied to hide more complex formal specification languages from direct manipulation by users, while still

allowing full benefits of formal methods-based CASE tools. MSC language proved to be suitable both for the forward and reverse engineering purposes. As a formal technique for capturing requirements, MSC satisfies all usability criteria for early development phases. On the other hand, MSC are suitable to capture "real" scenarios of legacy through collecting probe traces from suitably instrumented source code.

We presented our approach to synthesizing executable SDL models from scenarios formalized in MSC. As demonstrated in this paper, it turns out that this technique provides adequate support for both forward and reverse engineering.

We have given a broad overview of our accelerated development methodology, based on MSC and SDL, which can be used to significantly improve time-to-market in an industrial software development context. In our experience, the use of this accelerated development methodology combined with the use of SDL tools allows between 20 and 30% speedup in time-to-market for a typical telecommunication system. The use of tools in a related project was found to yield a 20-25% improvement in time-to-market; therefore the estimate above is likely quite conservative.

7. References

[1] Probert R., Mansurov N. Improving time-to-market using SDL tools and techniques (tutorial), Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999.

[2] Holzmann G. Formal Methods for Early Fault Detection. Invited paper in 4th Int. School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems. September 1996. Uppsala, Sweden.

[3] Mansurov N., Zhukov D. Automatic synthesis of SDL models in Use Case Methodology. In Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999, Elsevier Science Publishers B.V. (North-Holland).

[4] Holzmann G., Smith M.H. A practical method for the verification of event-driven software. In Proc. ICSE'99, pp.597-607, Los Angeles CA USA, May 1999.

[5] Mansurov N., Probert R. Dynamic scenario-based approach to re-engineering of legacy telecommunication software. In Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999, Elsevier Science Publishers B.V. (North-Holland).

[6] Jacobson I., Christerson M., Jonsson P., Overgaard G. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Reading, MA, 1992.

[7] Rumbaugh J., Jacobson I., Booch G. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[8] Z.120 (1996) *CCITT Message Sequence Charts (MSC)*, ITU-T, June 1992.

[9] Mansurov N., Laskavaya E., Ragozin A., Chernov A. On one approach to using SDL-92 and MSC for reverse engineering. In Voprosy kibernetiki: System Programming Applications, 3, Moscow, 1997 (in Russian).

[10] ITU-T (1993) *CCITT Specification and Description Language (SDL)*, ITU-T, June 1994.

[11] Engels A.G., Feijs L.M.G, Mauw, S., MSC and data: dynamic variables, In Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999, Elsevier Science Publishers,B.V.(North-Holland).

[12] Mansurov N., Vasura D., Approximation of (H)MSC semantics by Event Automata, in Proc. SAM'2000 workshop, Grenoble, France, 2000

[13] Leue S., Mehrmann L., Rezai M. *Synthesizing ROOM Models from Message Sequence Chart Specifications*. University of Waterloo, Technical Report 98-06, 1998.
[14] Robert G., Khendek F., Grogono P. Deriving an SDL specification with a given architecture from a set of MSCs. In Proc. of the 8-th SDL Forum, Evry, France, 23-26 September, 1997, Elsevier Science Publishers B.V. (North-Holland), pp. 197-212.
[15] Probert R., Saleh K. Synthesis of communication protocols: survey and assessment. IEEE Transactions on Computers, 40(4), pp. 468-475, April 1991.
[16] Tuok R., Logrippo L. Formal specification and use case generation for a mobile telephony system. Computer Networks and ISDN Systems, 30 (1998), pp. 1045-1063.
[17] Some S., Dssouli R., and Vaucher J. From scenarios to timed automata: Building specifications from user requirements. In Proc. 2nd Asia Pacific Software Engineering Conference, IEEE, December 1995.
[18] Rajala N., Campara D., Mansurov N. inSight Reverse Engineering CASE Tool. In Proc. of the ICSE'99, Los Angeles, USA, 1998.

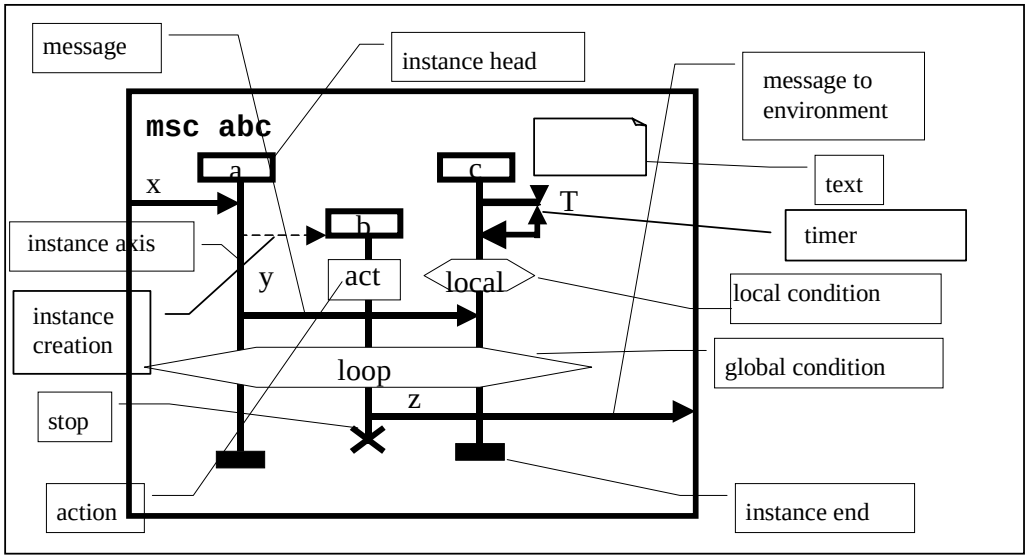

Figure 1. Elements of a basic Message Sequence Chart diagram

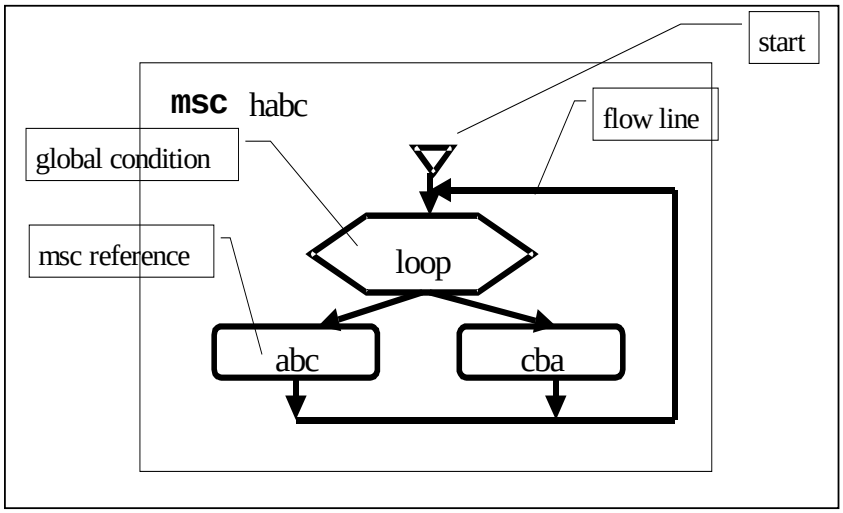

Figure 2. Elements of a High-level Message Sequence Chart

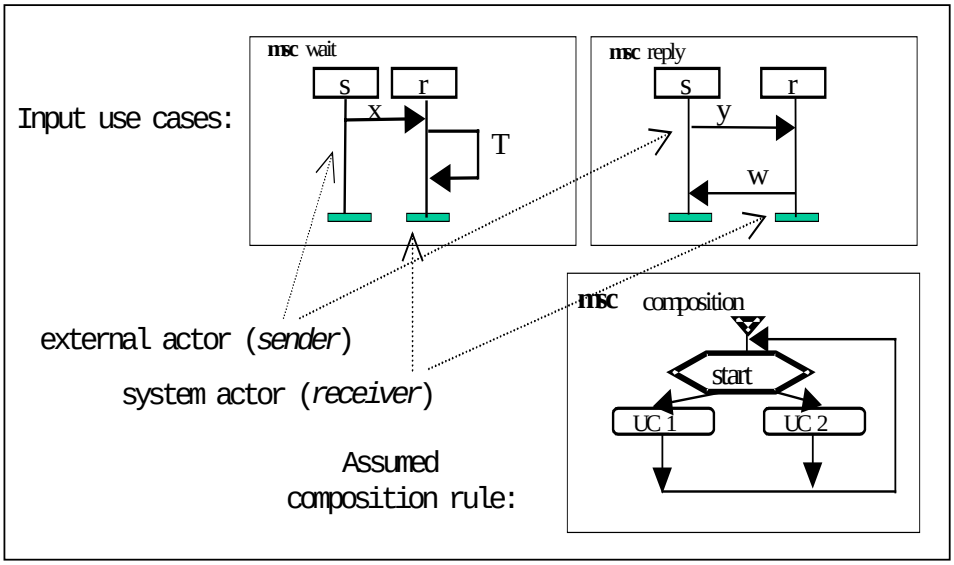

Figure 3. Example

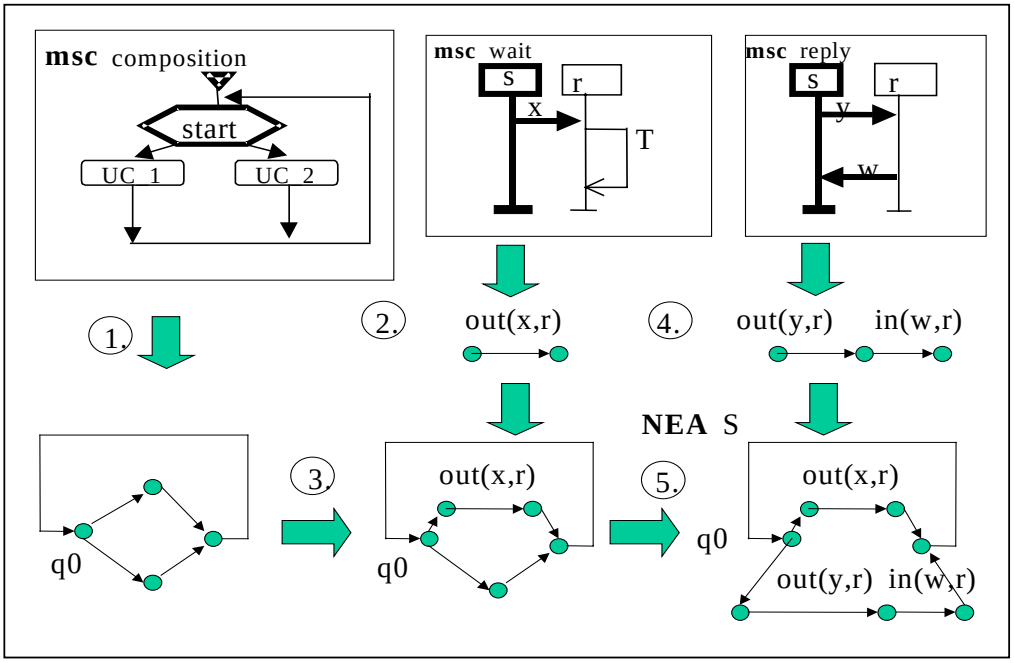

Figure 5. Steps of the synthesis algorithm

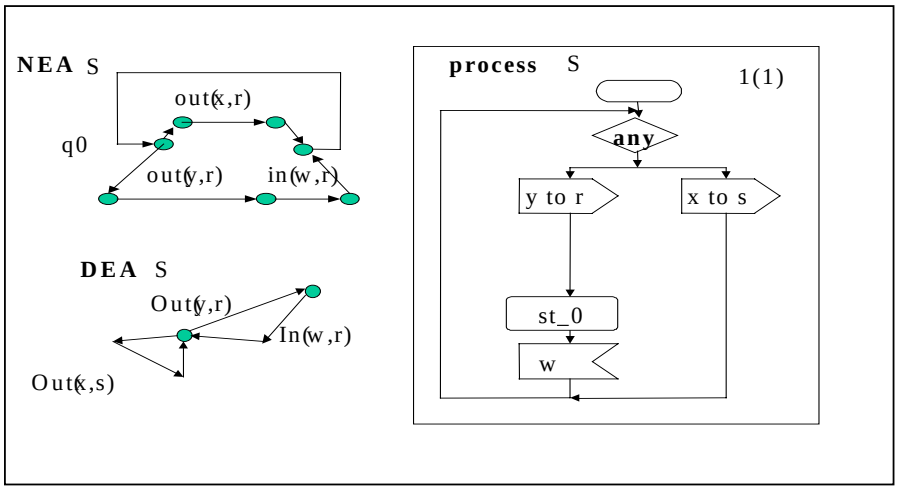

Figure 4. Generating SDL process from Event Automaton

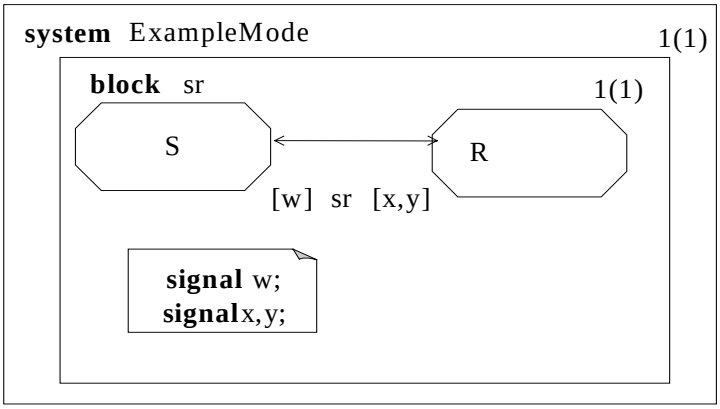

Figure 6. Generated SDL structure

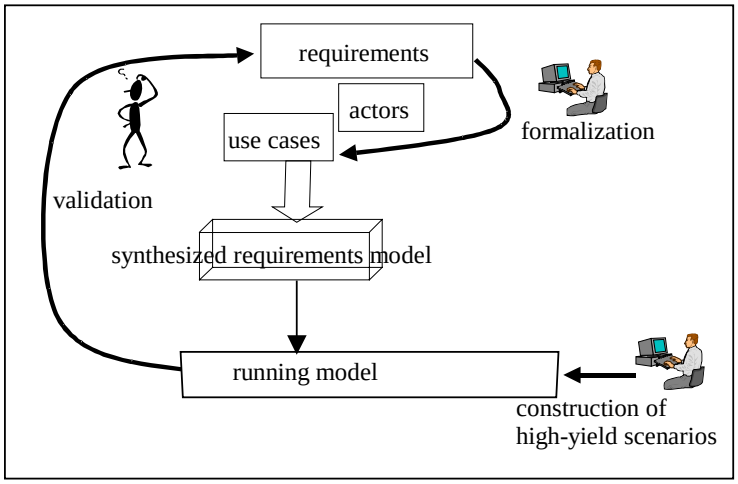

Figure 7. High-yield requirements validation

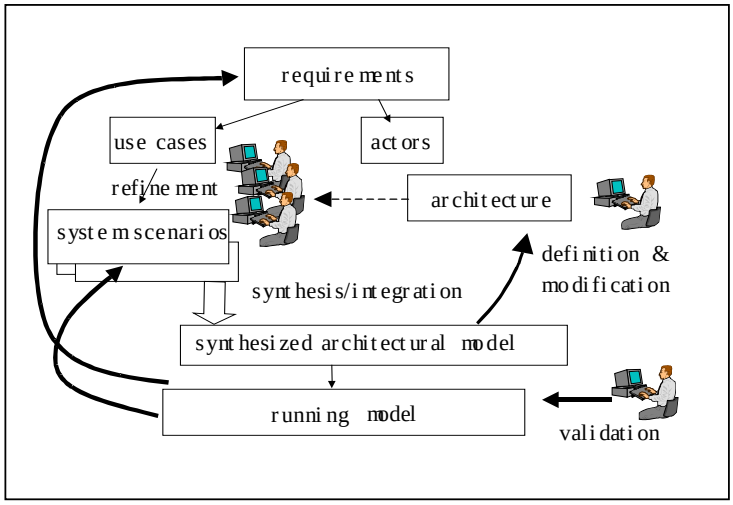

Figure 8. Refinement of original scenarios

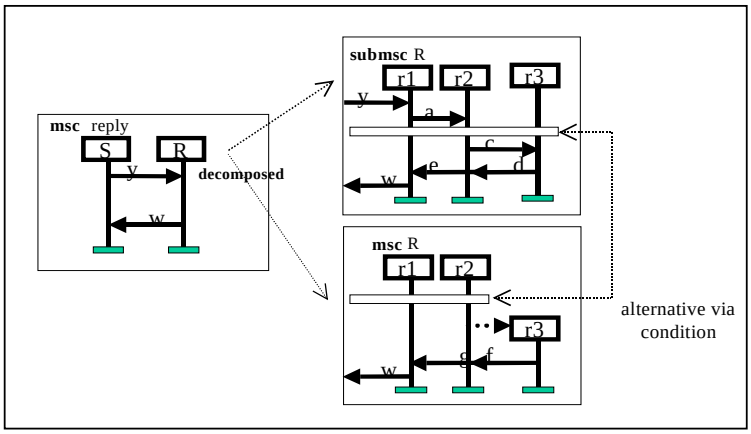

Figure 9. Refined scenario

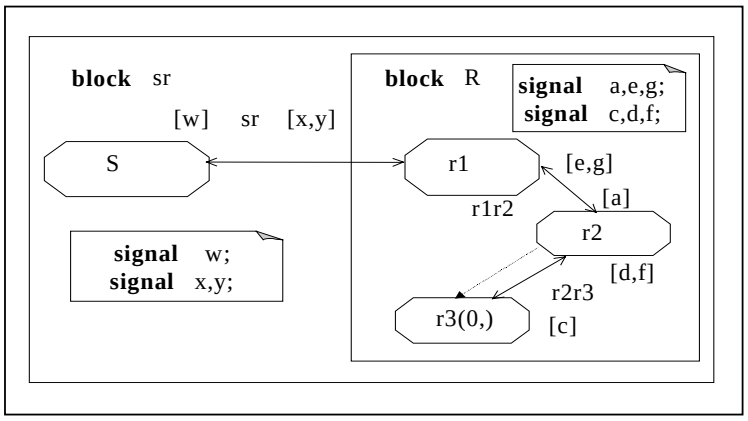

Figure 11. SDL model generated from the refined scenario

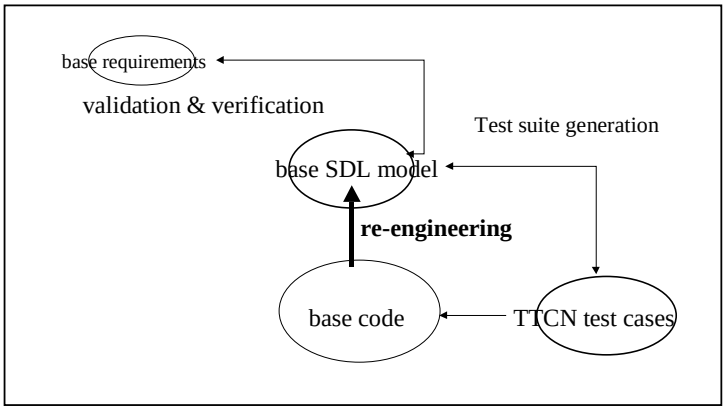

Figure 10. Re-engineering

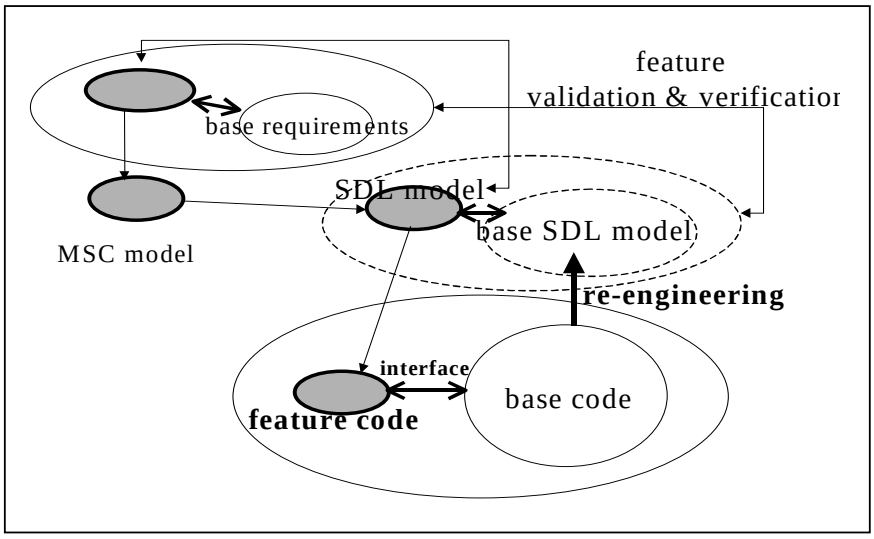

Figure 12. Using synthesized SDL models with re-engineered models

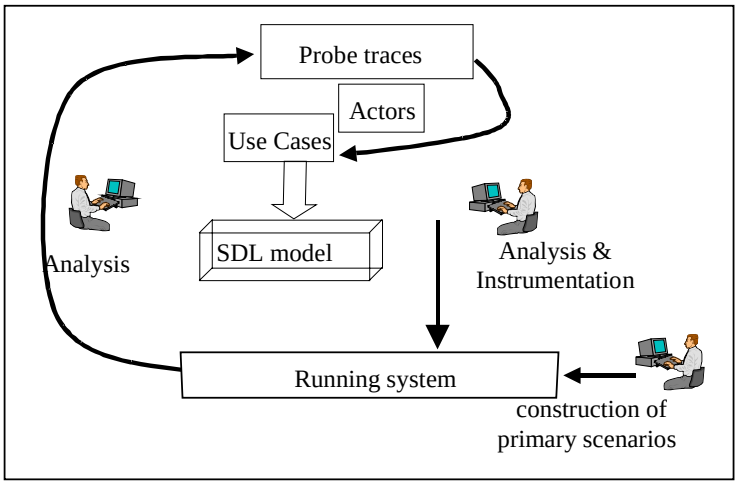

Figure 13. Dynamic scenario-based re-engineering process

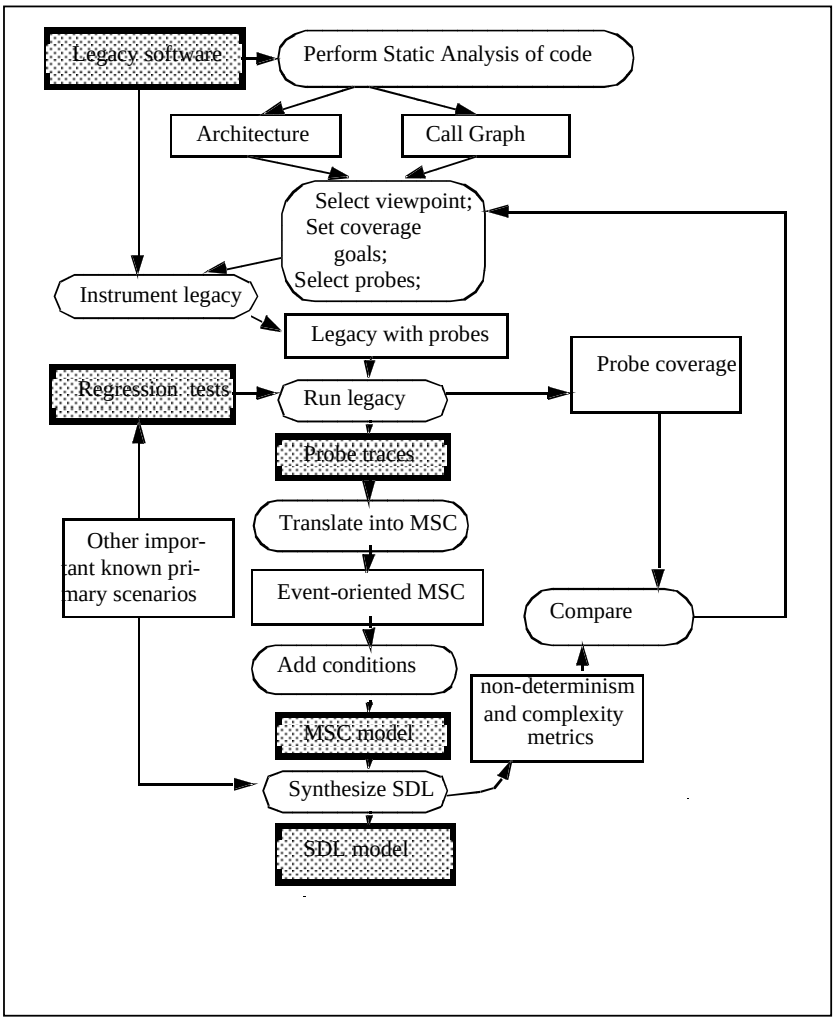

Figure 14. Overview of the dynamic scenario-based re-engineering

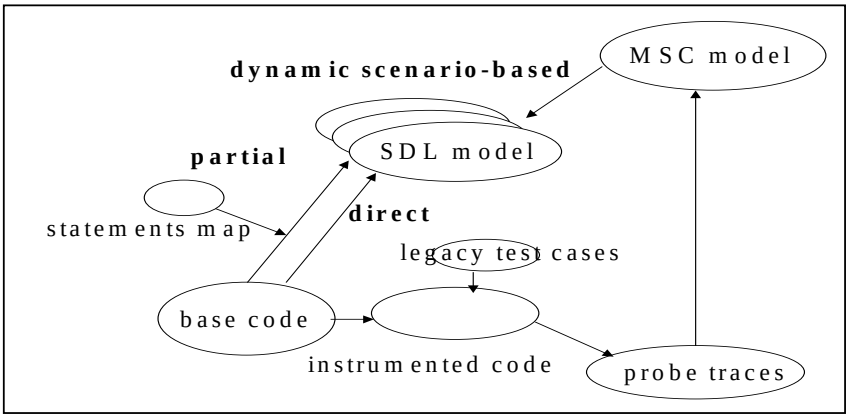

Figure 15. Comparison of re-engineering approaches