Formal methods for accelerated development of telecommunications software

Nikolai N. Mansurov

Head of Department for CASE Tools
Institute for System Programming,
Moscow Russia

Abstract

In this paper we summarize our experience in building new generation formal methods-based CASE tools aimed at practical improvement of software engineering in telecommunication industry. We define an **accelerated development methodology** for the specification, design, testing and re-engineering of telecommunications software, based on extensive use of formal methods and formal languages for the description of the software **very early** in the development process and automated **re-engineering** of formal models from legacy telecommunications software. Our methodology is based on the most widely accepted telecommunication formal languages, standardized by the International Telecommunications Union (ITU): *Specification and Description Language* (SDL), scenario description language called *Message Sequence Charts* (MSC), test description language called *Tree and Tabular Combined Notation* (TTCN), data description language *Abstract Syntax Notation* (ASN.1).

This paper emphasizes the following key components of the methodology:

- Capture of requirements by use-cases using the MSC language
- High-yield requirements validation using SDL requirement models
- Synthesis of SDL requirements models from approved MSC scenarios
- Seamless refinement of SDL requirement models into design models
- Adaptable code generation from the SDL models
- Automatic recovery of SDL models from legacy software

1. Introduction and Background

This paper presents research directions of the Department for CASE tools of the Institute for System Programming. The Department was founded in October 1997 from two groups working since 1994 on various projects related to building software engineering tools for telecommunication industry [1]. The Department of CASE tools performs research in reverse engineering methodologies [2,5], development of reverse engineering tools in collaboration with industry [7], as well as development of forward engineering tools for SDL and MSC languages [3,4,6,8,9,10,11]. We are also involved in teaching a course in formal methods at Moscow State University [12,13]. The Department has research collaboration with Telecommunications Software Engineering Research Group (TSERG), School of Information Technology and Engineering, University of Ottawa and has commercial contracts for research and development with Northern Telecomm (Canada) and Telelogic AB (Sweden).

The main research goals of the Department are to define

- an **accelerated development methodology**, composed of a set of methods, for the specification, design, testing and re-engineering of telecommunications software, based on extensive use of formal methods and formal languages for the description of the software very early in the development process and automated re-engineering of formal models from legacy telecommunications software, and
- an **open architecture**, composed of languages and tools, supporting the methodology and automating it as far as possible, to make the formal methods and languages amendable for human use.

Several factors motivate our research. Modern telecommunications industry hosts highly successful software development organizations, but as new requirements and technologies arrive and more players enter the competition, there is a constant need for improvements [13]. In particular, "*time-to-market*" is becoming the dominating factor of industrial success. Other goals are more traditional and include higher quality of products, better price/performance and lower development costs [13,21]. It is generally recognized that the use of formal description techniques (FDT's) supported by computer-aided software engineering (CASE) tools is an important prerequisite for achieving these goals.

A suite of formal languages is standardized by the International Telecommunications Union (ITU): *Specification and Description Language* (SDL) [14], scenario description language called *Message Sequence Charts* (MSC) [15], test description language called *Tree and Tabular Combined Notation* (TTCN) [16], data description language *Abstract Syntax Notation* (ASN.1) [17]. ITU-T Specification and Description Language (SDL) is one of the most successful telecommunications standard FDT [13]. Industrial-strength commercial tools exist which are able to analyze SDL specifications, perform validation of SDL specifications based on state-exploration algorithms, automatically generate abstract TTCN test cases from SDL specifications and also automatically generate implementations for real-time operating systems [18]. A number of industrial case studies has been recently completed, claiming improved quality, much lower development costs and speedup in time-to-market up to 20-30% due to the use of SDL-based CASE tools [13]. "Success stories" of using SDL in industry mention the phases of system design, detailed design, automatic generation of implementations [3,4,6,9,10,11,29] as well as formal verification and testing [19,20].

However there exist certain *barriers* in adoption of formal methods in industry. We identify two major barriers – support of early development phases and existence of legacy software. Our research is aimed at lowering these barriers.

There exists a significant gap between mathematical-based formal methods and design practice at the early phases of the software development process [21]. The design of a new system usually starts in a rather tentative, exploratory, and iterative way with a Requirements Capture phase. The problem domain is surveyed, and fragments of a trial solution are sketched. Most of these sketches lead a short life, and are modified frequently. Some of them will survive and will become a permanent part of design documents, as soon as the understanding of the new system has settled sufficiently that such documents can indeed be written. In the initial phases of a design, comprehensive formal specification and verification techniques offer little help to the designer. They appear to require a level of formality and precision that is not available yet. In return, only fairly abstract properties may be established. The initial price to be paid is too high, the initial rewards are far too small [21].

Instead, the so-called use case based methodologies are becoming predominant in software development [22,30]. Use case based methodologies share the common way of capturing the customer requirements as *scenarios*. Message Sequence Charts (MSC) or Sequence Diagrams of the Unified Modeling Language (UML) [30] can be used to model use cases. The MSC language is especially attractive as an FDT for the early phases of the software development process because it is well accepted in the telecommunications industry and also because it has a well-defined formal semantics. However much less support is provided by existing CASE tools for MSC modeling as compared to mathematical-based formal methods, like SDL.

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. The key idea of the suggested accelerated methodology is to use *automatic synthesis* of executable SDL specifications from MSC models. Our accelerated development methodology supports formalization of requirements using MSCs extended with data operations. Our Moscow Synthesizer Tool (MOST-SDL) provides a bridge from MSC models to executable SDL specifications [8]. Synthesized SDL specifications can be used for requirements validation and then further refined into design models. Other steps of the methodology include fast test case generation from SDL specifications and automatic code generation from SDL designs [13].

Apart from the support for the early design phases, there is another important issue which needs to be addressed in order for formal methods-based CASE tools for communications software engineering to become common practice. Formal methodologies are only applicable to the so-called "*green-field*" projects, in which the system is developed completely from scratch. However, most projects in the industrial context involve the older, "*legacy*" base software. This software is being maintained, updated by developing new features, or reused in new projects. For the formal methods to be adopted in industry, 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 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 [5].

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, 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;

- 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 (e.g. C, C++, CHILL) as well as for different real-time operating systems (e.g. pSOS, VxWorks, etc.).

In this paper we present our methodology of *dynamic scenario-based* re-engineering of legacy telecommunications systems into a system design model expressed in SDL. Our approach consists of

- placing semantic probes [5] 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 [8].

The rest of the paper has the following organization. Section 2 provides an introduction into our accelerated development methodology. Sections 3-6 cover specification, design and testing of telecommunications software. Section 3 describes early formalization of requirements and high-yield requirements validation based on SDL requirements models. Section 4 presents one of the key aspects of our methodology – automatic synthesis of SDL specifications from scenarios. Section 5 discusses refinement of synthesized SDL requirement models into design. Section 6 describes our experience in adaptable code generation from SDL. Section 7 describes our approach to recovery of SDL specifications from legacy telecommunications software. Section 8 provides some conclusions and outlines several future research directions.

2. Accelerated development of telecommunications software

This paper presents a state-of-the-art, pragmatic development process based on the most widely accepted telecommunication standard formal languages MSC, SDL and TTCN. The purpose of our research is to demonstrate how to use modern formal descriptions techniques (FDT) and tools to dramatically reduce *time-to-market* for the industrial telecommunications software development context. Time-to-market is becoming increasingly important in determining success in the telecommunications sector.

Until very recently, computer-aided software engineering (CASE) tools were applied by relatively few organizations. Currently, however, a number of tools, which are able to handle very large systems design and development have become available and allow CASE technology to be increasingly utilized. In our paper, we will show how to adopt a CASE-based approach to achieve productivity goals in the software development cycle.

Best practice development processes today use the principle of *upfront increase* in project time and people to reduce the total cost and time to market. Figure 1 illustrates productivity improvements offered by FDT-based CASE tools [13]. We claim, that an initial investment of approximately 17% required to produce and validate a formal specification (e.g. in SDL) and generate the required test suites at the requirements capture and design phases can result in 30-50% saving at the product development and deployment [13].

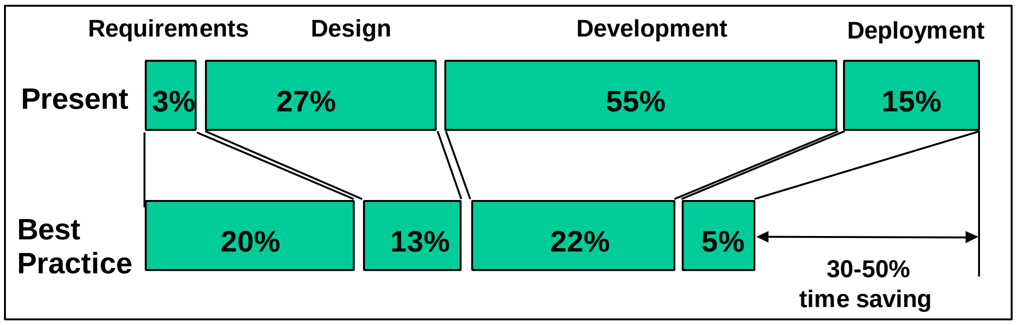

Figure 1

Incorporating formal description techniques into the development process, of course, also leads to improving the *quality* of products. Until recently, quality improvement was considered to be the main benefit of using FDTs, Therefore safety-critical applications were first to adopt them. In this paper we emphasize productivity improvements that can be achieved through the use of FDTs in a conventional (not necessarily safety-critical) industrial development process, especially with respect to delivering a new product to market. Quality improvement, as well, has a direct impact on shortening time-to-market through decreased bug fix time. However, the biggest benefits of quality improvement occur on a longer-term scale, when costly field bug fixing and maintenance are taken into consideration.

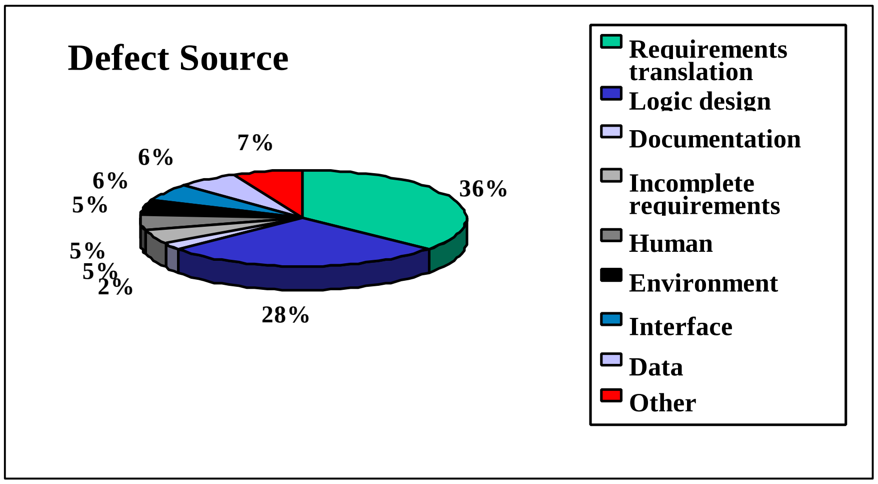

Figure 2

Significant savings can result from reducing the cost of finding and fixing defects, including field errors. According to a recent study done by the University of West Virginia and the US Air Force [13], most defects can be traced back to the early phases of software development process (see Figure 2). The cause of 36% of defects was incorrect requirements translation, 5% of defects happened because of incomplete requirements, while another 28% of defects were attributed to logical design faults. Only 31% of defects were attributed to all other causes combined.

On the other hand, most defects were detected (and corrected) at the later phases of the development process (see Figure 3). 13,83% of defects were detected during flight tests, 51,06% of defects were discovered at the system integration phase, another 15,96% - at the software integration phase. Only 9,57% of defects were detected during requirements phase.

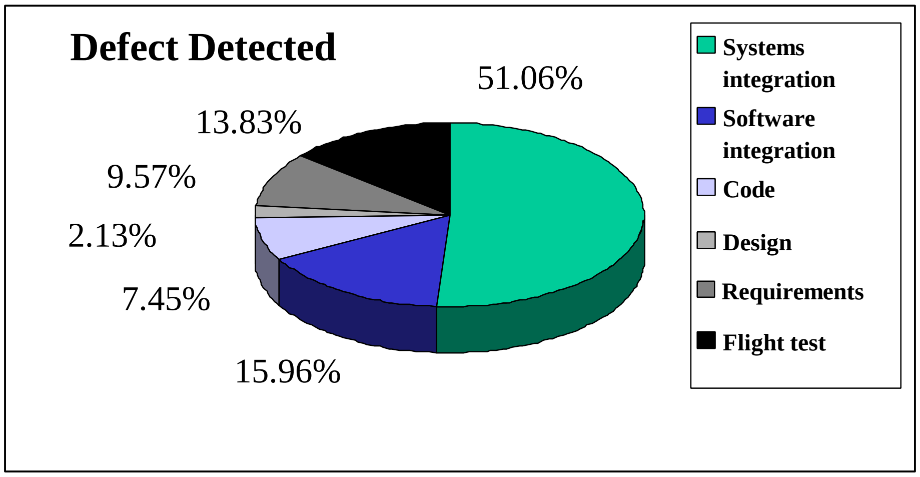

Figure 3

The main goal of our research is to lower barriers for formal methods at early phases (requirements capture) as well as at later phases (maintenance). We also use traditional benefits of formal methods-based tools and techniques for middle phases of the development process. This is illustrated at Figure 4.

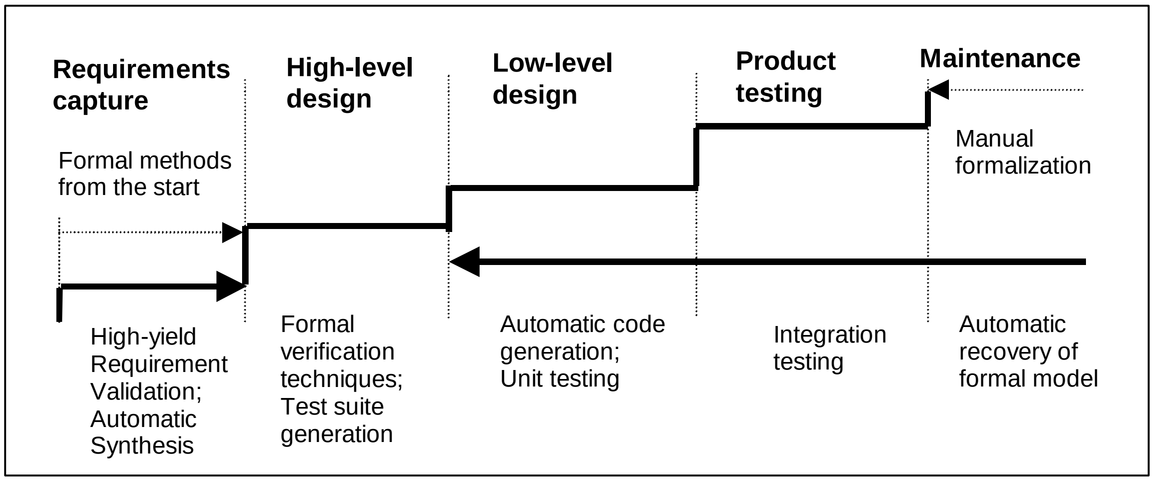
Requirements capture
High-level design
Low-level design
Product testing
Maintenance
Formal methods from the start
Manual formalization
High-yield Requirement Validation; Automatic Synthesis
Formal verification techniques; Test suite generation
Automatic code generation; Unit testing
Integration testing
Automatic recovery of formal model

Figure 4

A more detailed overview of our methodology is presented in Figure 5. This figure shows the *phases* of the methodology (right column), the most important *models* (boxes) as well as the most important information *feedbacks* provided by SDL tools (dashed lines). Relationships and feedbacks between the behavioral and the structural models at the same phase are omitted. Also omitted are the feedbacks between the models at any subsequent phases. We also emphasize two different modeling perspectives [22,30,13], namely the *behavioral path* (left sequence of boxes) and the *structural path* (right sequence of boxes). The structural path describes what entities constitute the software at the given level of abstraction and what are the relationships between them. The behavioral path describes how entities interact in order to achieve the given functionality.

An information feedback is very important because it is a prerequisite for *iterative development*. An accelerated development process can be based on such iterations. Thus the key idea of our approach is to move the use of FDTs to the earliest possible phases of the traditional development process in order to enable tool-aided iterative development.

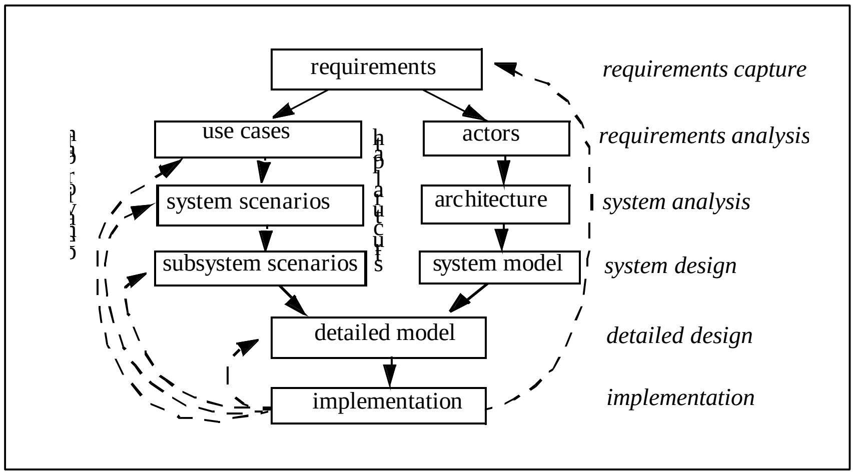

Figure 5

In a typical telecommunications oriented industrial process formalization starts at the detailed design phase. The *system model* (Figure 5) is represented as several SDL block diagrams and the *detailed model* is represented as the corresponding set of SDL process diagrams. Transitions between all phases are done manually, except for the transition from the detailed model to the implementation which is done by SDL tools. The only useful feedback provided by SDL tools is the one from the implementation back to the detailed model. Other feedbacks are less useful, because the corresponding models are not formalized and also because the transition to the detailed SDL model is usually too costly for any iterations to be feasible.

Our accelerated development methodology is an iterative use case based object-oriented methodology. It emphasizes the following key components:

- Capture of requirements by use-cases using the MSC language
- High-yield requirements validation using SDL requirement models
- Synthesis of SDL requirements models from approved MSC scenarios
- Seamless refinement of SDL requirement models into design models

- Adaptable code generation from the SDL models
- A high yield test design and automated test development process directed towards TTCN (to allow more efficient, automated testing)
- Automatic recovery of SDL models from legacy software.

The processes of accelerated development methodology are presented at Figure 6. As illustrated at Figure 6, all processes are supported by SDL, MSC and TTCN tools.

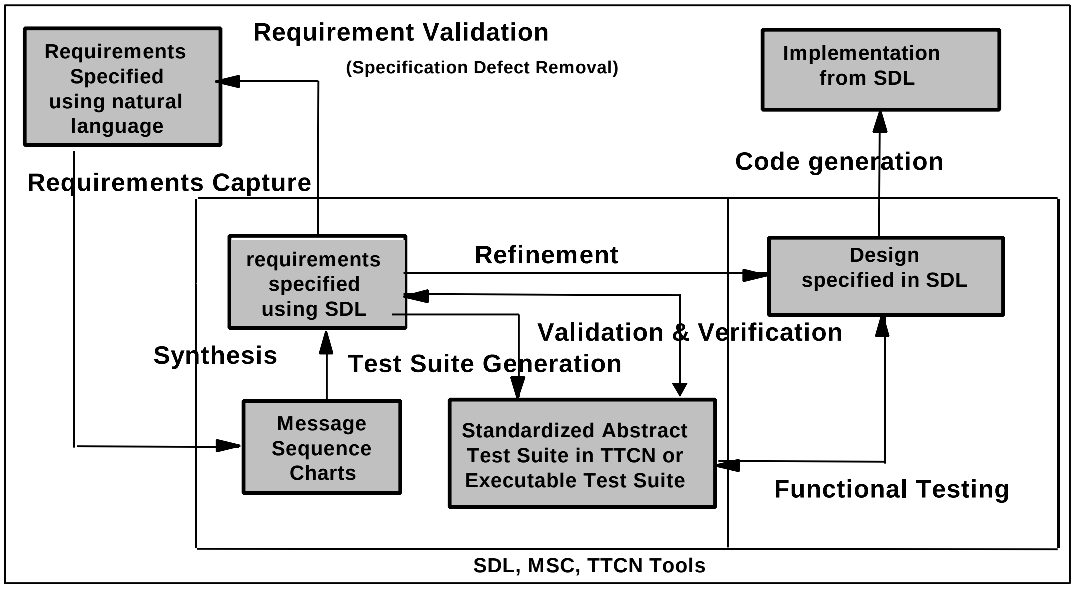

Figure 6

3. Requirements Capture and Validation

3.1. Use Case Scenarios

Use case based methodologies are becoming predominant in software development. The main concepts of a use case methodology are *actors* and *use cases* [22]. An *actor* represents an entity external to the system under development. An actor interacts with the system in order to achieve certain *goals*. A *use case* is a description of a typical (illustrative) interaction between one or many actors and the system, through which one of the actors achieves a specific goal. Such goal is called a *use case purpose*. One use case can cover several sequences of events, called *scenarios*. Several use case based methodologies were proposed [22,25,30,12].

Use case driven methodologies share the common way of capturing customer requirements as functional scenarios but differ in ways of how scenarios are represented. Overview of use cases, actors and their relationships are captured in the form of informal pictures [22], scenarios are captured using tabular forms [12], plain text [22] or UML Sequence Diagrams [30]. Formalization of use cases using MSC diagrams was suggested in [25].

3.4. High Yield Requirements Validation

In this section we introduce requirements validation technique. Requirements validation is used to detect faults in the customer requirements. As was demonstrated in section 2, the feedback from the implementation to requirements is very important in the development process. According to our accelerated development methodology (see

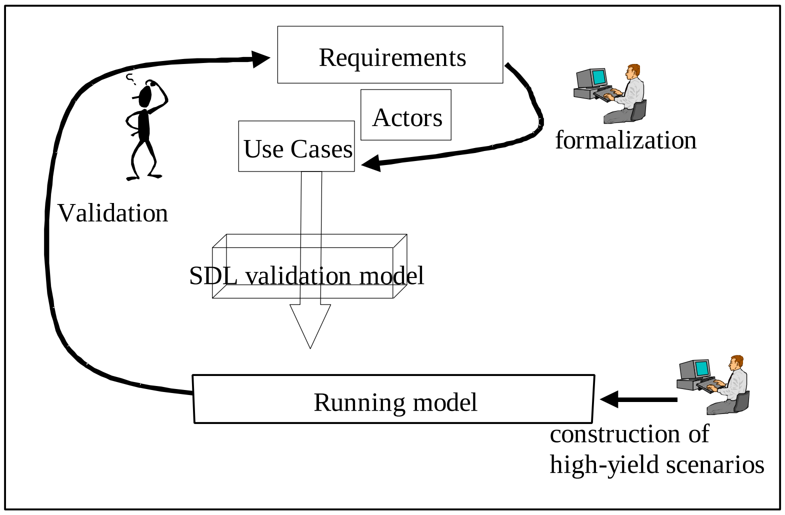

Figure 7

Figure 7), requirements validation is an iterative process consisting of the following steps:

1. Formalize requirements in the form of use case scenarios
2. Create executable model of the requirements
3. Create validation scenarios
4. Run validation scenarios through the requirements model
5. Validate the execution sequence of each validation scenario to either
   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 (from step 2).

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. 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. We recommend to use high-yield scenarios and monitor coverage to determine the thoroughness of the validation effort. High-yield validation is a risk-based process. The objective of the high-yield requirements validation is to focus the effort on the elements with highest risk.

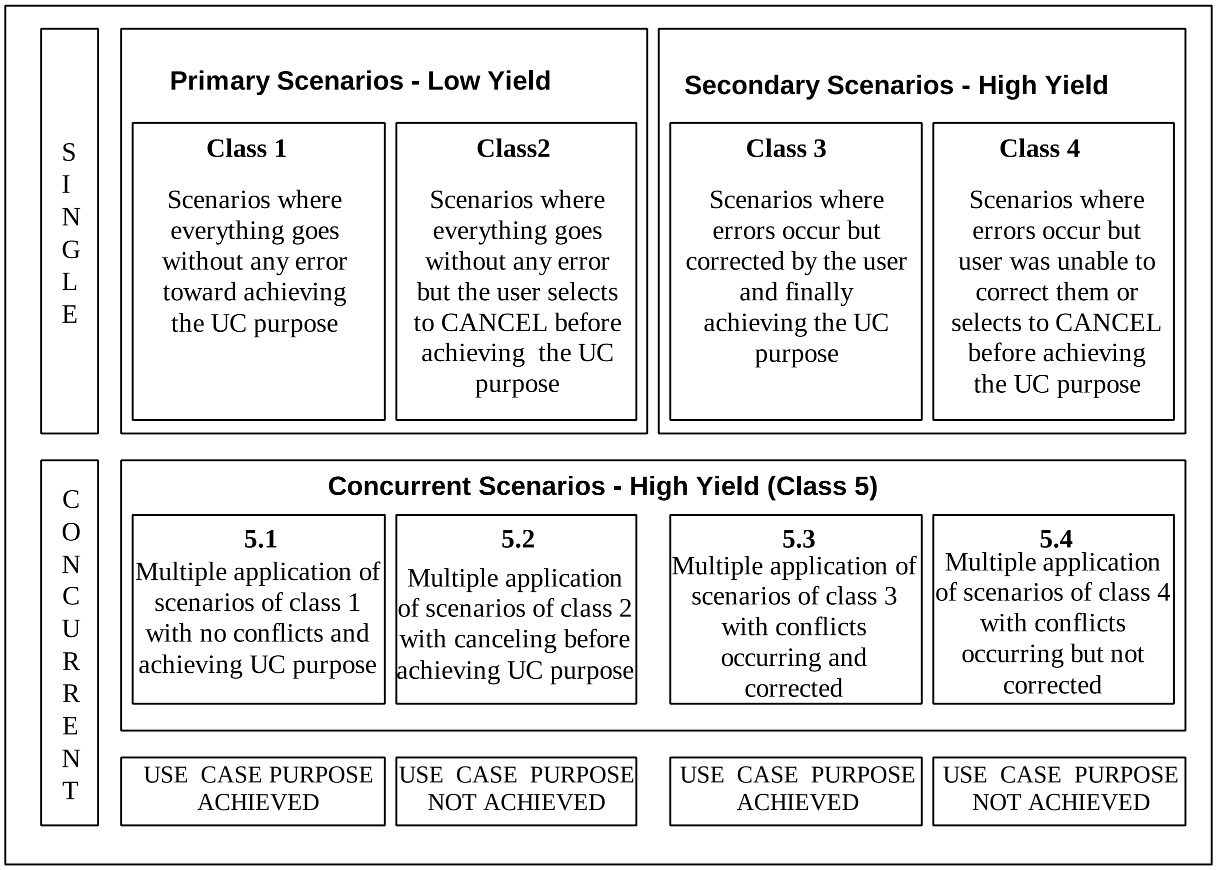
SINGLE
Primary Scenarios - Low Yield
Secondary Scenarios - High Yield
Class 1
Scenarios where everything goes without any error toward achieving the UC purpose
Class2
Scenarios where everything goes without any error but the user selects to CANCEL before achieving the UC purpose
Class 3
Scenarios where errors occur but corrected by the user and finally achieving the UC purpose
Class 4
Scenarios where errors occur but user was unable to correct them or selects to CANCEL before achieving the UC purpose
CONCURRENT
Concurrent Scenarios - High Yield (Class 5)
5.1
Multiple application of scenarios of class 1 with no conflicts and achieving UC purpose
5.2
Multiple application of scenarios of class 2 with canceling before achieving UC purpose
5.3
Multiple application of scenarios of class 3 with conflicts occurring and corrected
5.4
Multiple application of scenarios of class 4 with conflicts occurring but not corrected
USE CASE PURPOSE ACHIEVED
USE CASE PURPOSE NOT ACHIEVED
USE CASE PURPOSE ACHIEVED
USE CASE PURPOSE NOT ACHIEVED

Figure 8

Risk refers to the "cost" associated with a failure. By definition, Risk R = PF $\times$ CF (where PF is the probability of failure; CF is the cost of failure). *Yield* 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.

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 [13] we suggested the following classification of use case scenarios based on yield (see Figure 8).

In our opinion, *high-yield* validation is also *fault-directed validation*. We recommend to validate *common problems* and *high-risk areas* to increase yield on investment. In order to construct high-yield scenarios for requirements validation, we recommend to formulate fault models that express potential faults (*fault hypotheses*) and the derive scenarios directly from fault hypotheses.

If a fault model is complete and realistic then N scenarios per fault hypothesis (N is based on some heuristic) is an appropriate coverage criteria. Fault models consisting of high-risk and common errors become the basis for high-yield scenario design strategies.

Scenarios that explore *boundary conditions* have a higher likelihood of detecting serious defects than scenarios that do not. The general strategy for constructing high-yield scenarios for requirements validation involves deriving valid and invalid equivalence classes for input and output domains. Scenarios to cover valid equivalence classes broadly and invalid equivalence classes should be identified such that the elements are selected just on, or just beyond the borders of each equivalence class.

High-yield requirements validation is a form of testing applied to an early phase. 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).

3.5. Automatic synthesis of the SDL requirements model

We suggest to automatically synthesize an SDL model at the requirements analysis phase (Figure 9). The only input at this phase is the use case model which in our case should be formalized using MSC. The only structural information available at this phase consists of the set of external actors and the system (represented as distinct instances in the MSC model). Use cases and their control-flow relationships provide additional structural information which can be used by the synthesis algorithm. The behavioral information is available in the form of (incomplete) functional scenarios representing the typical interactions between the external actors and the use cases within the system. Important behavioral information can be additionally captured in the form of *data flows over the use cases* using our data extensions (see Section 4.1). The so called *synthesized requirements model* (SRM) is the projection of the information captured at the requirements analysis phase down to the detailed design (and the implementation) phase. All feedbacks shown in Figure 9 become useful because they are provided in a timely manner as the requirements model is being created, enabling the *iterative development* of the latter.

Automatic synthesis of SDL 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

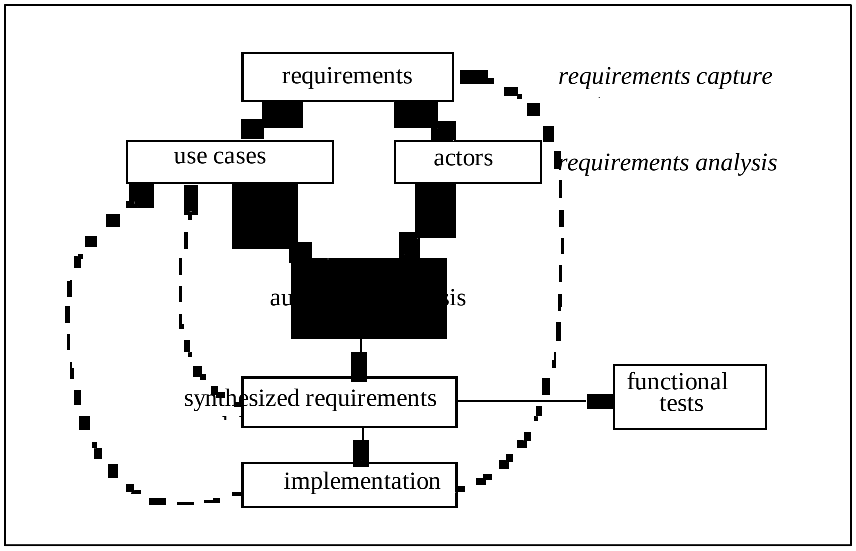

Figure 9

The synthesized requirements model can be used to *generate additional scenarios* which are *longer* than the original scenarios of the requirements model and therefore provide better understanding of the requirements [27]. 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.

The synthesized requirements model can be used to automatically generate functional test cases [19,20].

4. Synthesis of SDL specifications from use cases

In this section we present technical details of the automatic synthesis of SDL models from scenarios formalized in MSC. We describe formalization of scenarios, our data extensions to MSC, provide some details of the synthesis algorithm, and provide an illustrative example.

Our approach consists of:

- Formalizing use cases using High-Level MSCs (HMSC)
- Mapping scenarios to finite state machines (FSM)
- Specifying flows of data through scenarios using some data extensions to MSC
- Automatically synthesizing SDL validation models, such that they are:
  - complete (both structure and behavior) & ready to run

- non-deterministic
- typebased

4.1. Formalization of Use Case Scenarios Using MSCs

We used the following guidelines when designing our formalization methodology.

- Each scenario is formalized using *a Basic Message Sequence Chart* (bMSC) [15]
- Control-flow relationships between scenarios are formalized using *High-Level MSCs* (HMSC) [15]. Control-flow relationships between scenarios include alternative (sub-) scenarios, iterations of (sub-) scenarios, "uses" and "extends" relations between scenarios [22]
- Data-flow relationships between scenarios are formalized using our *data extensions* [8] to MSC language
- Certain *composition rule* is used [8]:
  - Sequential rule (single use case is executed "to completion")
  - Parallel rule (different use cases can execute simultaneously)
  - Multiple instance rule (multiple instances of the same use case can execute simultaneously)

We use extended Message Sequence Charts (MSC) language [15] to formalize *scenarios*. Each *use case* is formalized by a high-level MSC (HMSC) which represents control-flow relationships between scenarios of the use case. Our extensions to the MSC language describe the *flows of data* through individual scenarios (local data flows) as well as data flow dependencies between scenarios (global data flows). Let's consider our data extensions [8] in more detail.

1. **Variable definitions**. We allow to define variables of different types. SDL semantics is assumed. Variable definition is placed into a text symbol in any MSC diagram. A local copy of each variable is propagated to 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. SDL semantics is assumed. 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. SDL semantics is assumed for parameter passing.
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. SDL semantics is assumed.
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.

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. Note that timers with parameters are not supported.

The concept of data flows over scenarios is illustrated in Figure 10 and Figure 11. Two local data flows through scenario *abc* are shown (as two dashed lines) in Figure 10. The first flow contains the following MSC events: a: **in** x(p,q) **from env**; a: **create** b(p,q); b: **out** w(r) **to env**; Note, that instances *B* and *C* use different (local) copies of the variable r. Thus the parameter of the message w which is sent by the instance *B* to the environment is not necessarily equal to p+1.

The second flow contains the following MSC events: a: **in** x(p,q) **from env**; a: **create** b(p,q); b: **out** y(p) **to** c; c: **in** y(p) **from** b; c: **action** 'r:=p+1'; c: **out** z(r) **to env**;

Note that the instance *C* will send message z(r) to environment only when condition r>0 is true. Alternative events for the instance *C* can be specified using the local condition local with a different guard. Global conditions in the HMSC graph will be required when alternative events involve other instances.

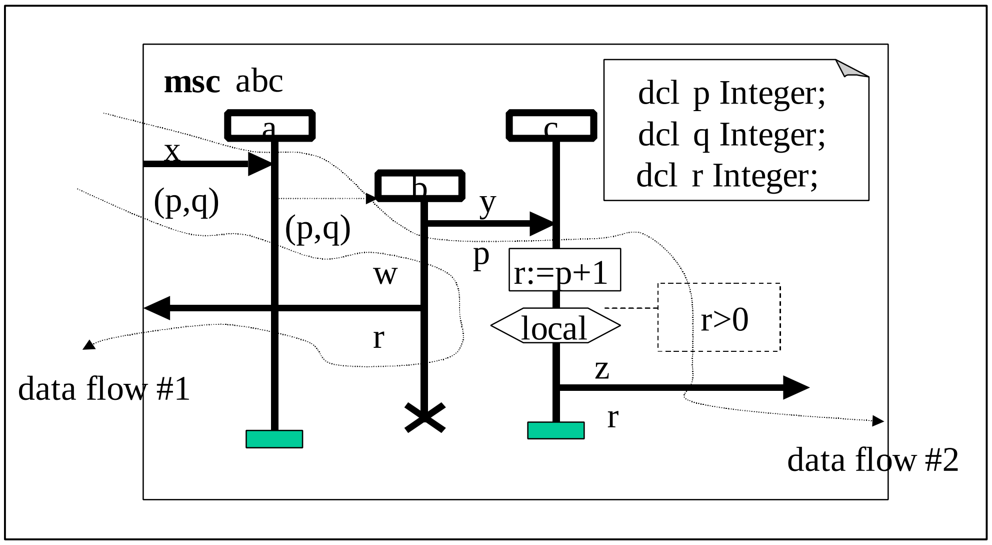

Figure 10

Figure 11 illustrates the specification of global data flows between use cases. In Figure 11 parameters of the message *value* returned by the use case *pop* as a reaction to the message *get* depends on the events in the use case *push*. push and pop are assumed to be user-defined SDL procedures with in/out parameters implementing typical stack operations.

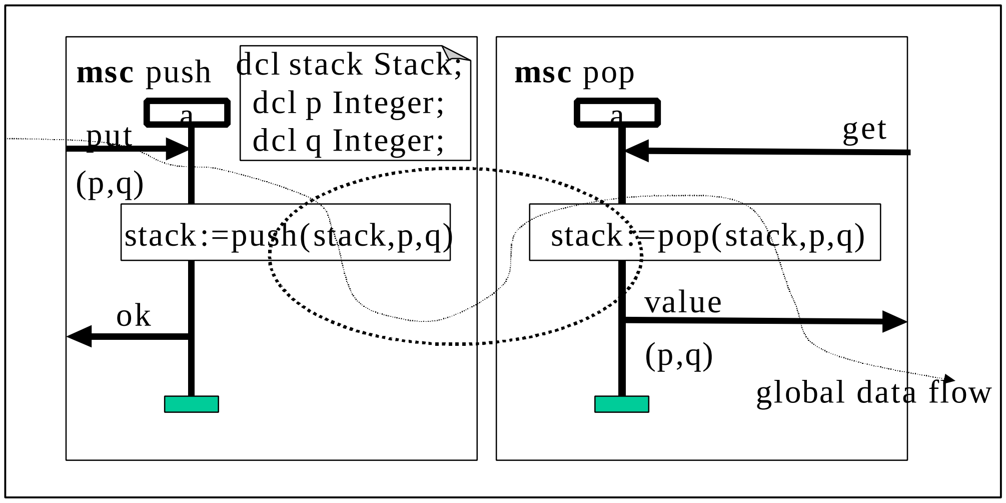

Figure 11

Our main motivation in adding data extensions to MSC is to allow more accurate specifications of functional requirements. However the same data sub-language turns extended MSC into a powerful FDT for design phases.

4.2. Synthesis Algorithm

Our synthesis algorithm consists of the following steps [8]:

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

To describe out synthesis algorithm, we use the concept of *event automata*. An **event automaton** is a finite automaton corresponding to *a single MSC instance*, such that the input symbols for the automaton are MSC events, involving the given MSC instance. We distinguish between three categories of MSC events: input, active and idle events. An idle event is a trivial (empty) event, which was added to simplify algorithm description.

- input events (require synchronization with other instances, decision about event is taken by another instance)
    - message input *in(i,m)*
    - timeout *timeout(t)*

- active events (do not require synchronization with other instances, decision is local to the current instance)
  - message output *out(i,m)*
  - action *action(a)*
  - set timer *set(t,d)*
  - reset timer *reset(t)*
  - stop action *stop*
  - local condition *check(c)*

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, producing all valid event traces for the instance *i*. We use the following algorithm to construct MSC slices:

1. initial states of the event automaton correspond to symbols at HMSC graph with idle events
2. for each bMSC 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

This algorithm is further illustrated at Figure 13.

4.3. Example

We are illustrating our algorithm by considering a simple MSC model shown in Figure 12. 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.

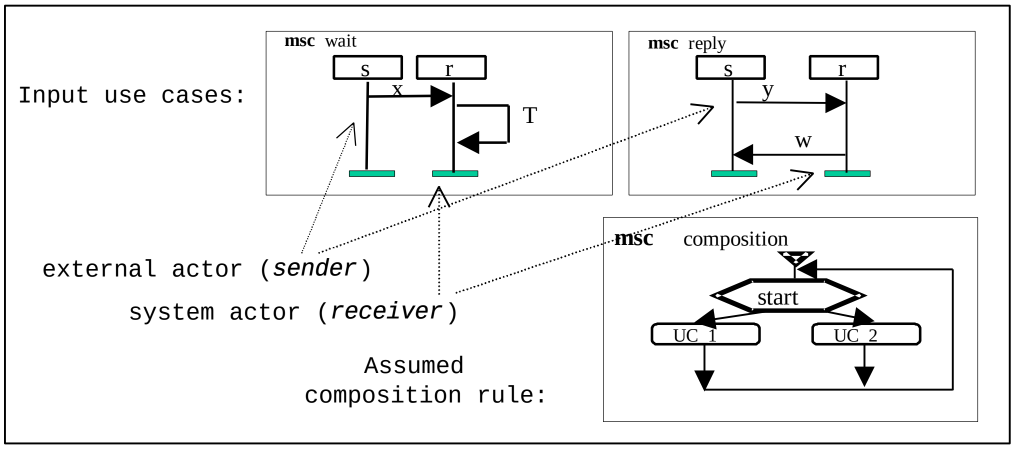

Figure 12

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.

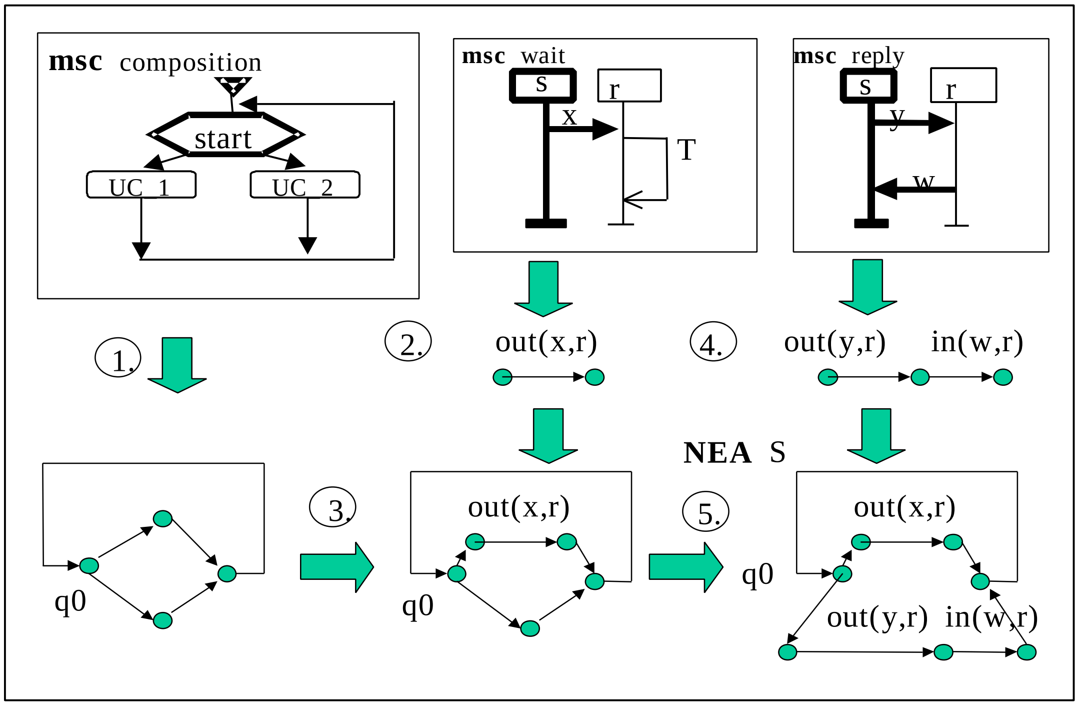

Figure 13

Figure 13 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 4,5) process instance S from MSC reply. The resulting MSC slice is presented in the bottom right corner of Figure . It is labeled non-deterministic event automaton (NEA).

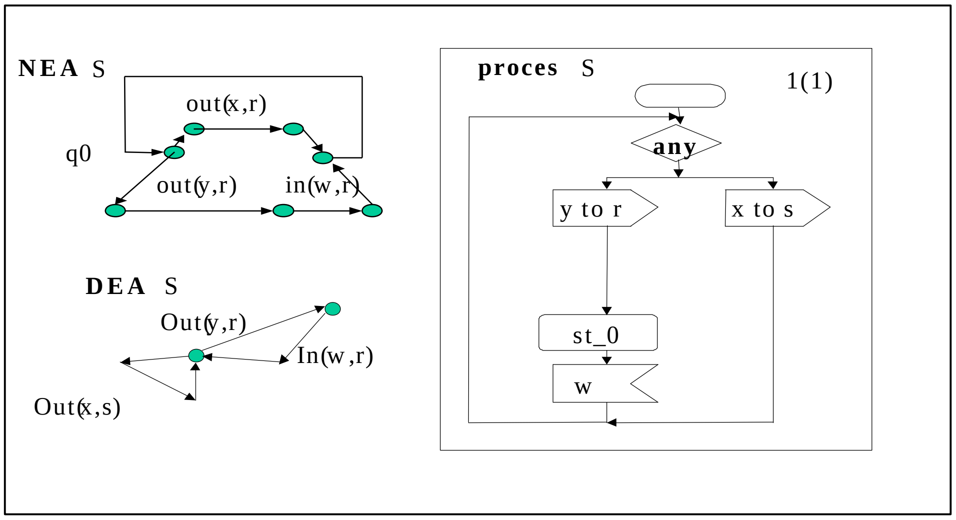

Figure 14

Figure 14 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 15 illustrates the generated SDL structure for our example.

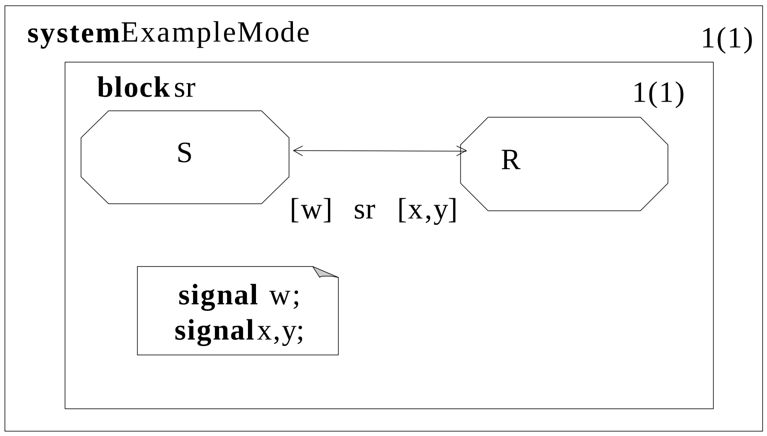

Figure 15

4.4. 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 MSCs to other languages [23,24,28]. Synthesizing SDL specifications from MSC is addressed in [23]. Survey of work on a more general subject of protocol synthesis is available in [26].

Methodological issues of generating a formal executable specification from a set of use cases are addressed in [24]. This paper summarizes experience in manually developing a LOTOS specification of a telecommunications standard on the basis of use cases provided by industry. 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 [28] 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 [27] 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 bMSCs. According to [27], 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 [27]. The Waterloo synthesizer derives static process structure based on the instances in bMSC. Similar approach is taken in the Moscow synthesizer. Additionally, the Moscow synthesizer derives *dynamic process structure* by considering bMSC 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 [23] 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 [23]. 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 [23]. Thus the Concordia synthesizer produces only behavioral components. Composition of bMSCs using HMSC was not addressed in [23] 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 [27]) which allows us to synthesize executable requirements models (similar to [24]) as well as executable architecture models [27]. 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. Refinement of Requirement Models into Design Models

In this section we present our support for design activities. Our accelerated development methodology involves seamless refinement of requirement models into design models.

5.1. Architecture Validation

We suggest to use automatic synthesis at the system analysis phase. During this phase the *architecture* of the system is being defined and independent groups of developers produce system scenarios for each architecture component (Figure 16). The input at this phase is a set of system scenarios. Normally a system scenario will be a projection of the corresponding use case from the requirements analysis phase. The structural information available in the system scenarios consists of the set of external actors and the architecture components (represented as distinct instances in the MSC model). The 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). The feedbacks provided by SDL tools through the SAM go both to the system scenarios as well as to the architecture model (Figure 16).

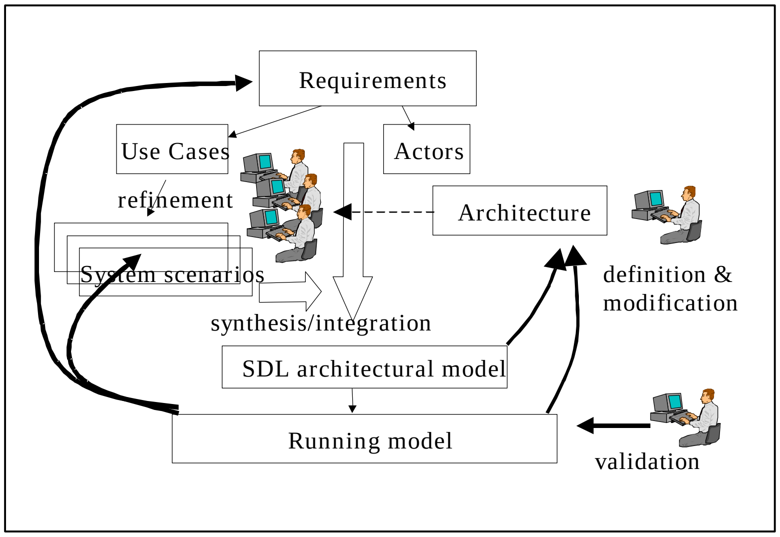

Figure 16

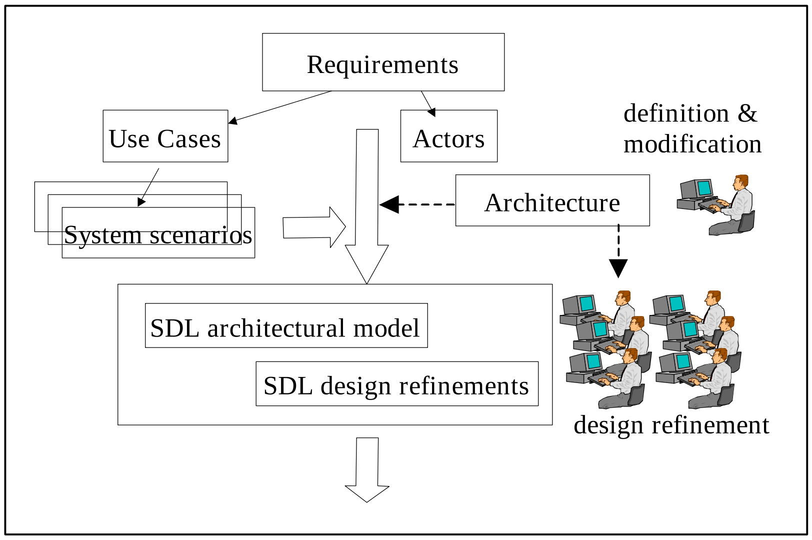

Figure 17

In this case the SAM will reproduce the deriving them from the collection of system architectural components of the system by scenarios, synthesize the behavior of each component and integrate the model. Automatically derived relationships between components can be compared to the intended ones (described in the architecture model). In our experience the synthesized architecture model is helpful in uncovering system analysis faults. This step is a direct continuation of requirements validation activity described earlier.

5.1. Vertical Decomposition of MSCs

SDL architecture model can be manually refined into design model (Figure 17). Vertical decomposition of MSC models can be used to introduce design details into requirement models. Figure 18 demonstrates this technique. In use case reply from example at Figure 12, 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 presented at Figure 19.

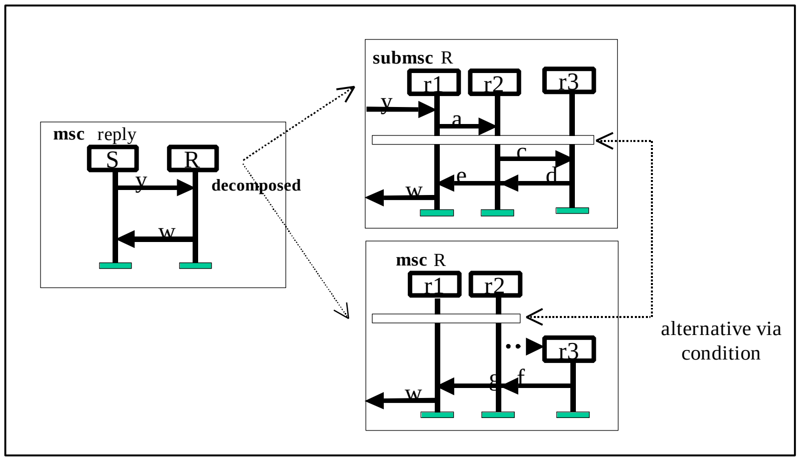

Figure 18

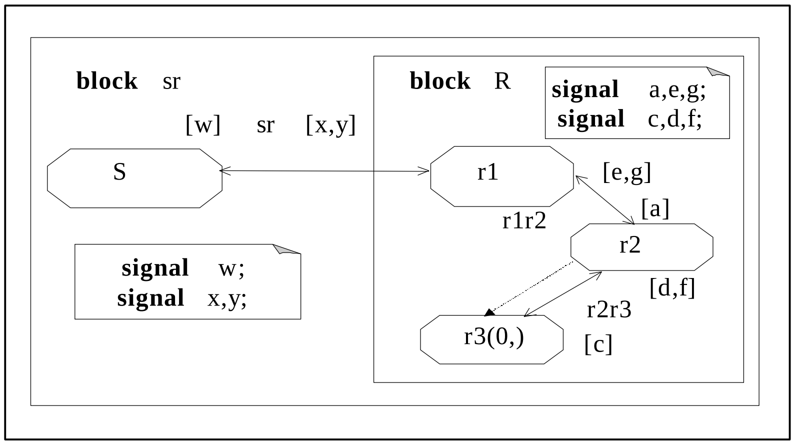

Figure 19

Our synthesizer tool further supports refinement process by generating typebased SDL models [8]. The following features are provided in order to support seamless refinement of the synthesized requirement models into design [8,13].

- SDL structure is generated as a package containing one process type for each actor
- Block type diagram is provided which defines communication between typebased process instances
- Process types are virtual and can be redefined
- Start transitions in process types for external actors are virtual and can be redefined
- Specialization of actors is relatively stable with respect to the changes in the MSC model

6. Adaptable code generation from SDL specifications

In this section we present our experience in developing industrial-strength automatic code generators for SDL [1,3,4,6,9,10,11,31]. First, we discuss the need to adapt the generated code in industrial context. We compare several approaches to adaptable code generation and present some of our thoughts, why it is difficult to adapt the generated code. Then we present the concept of declarative mappings [11,31] as a promising technique for generating highly adaptable code.

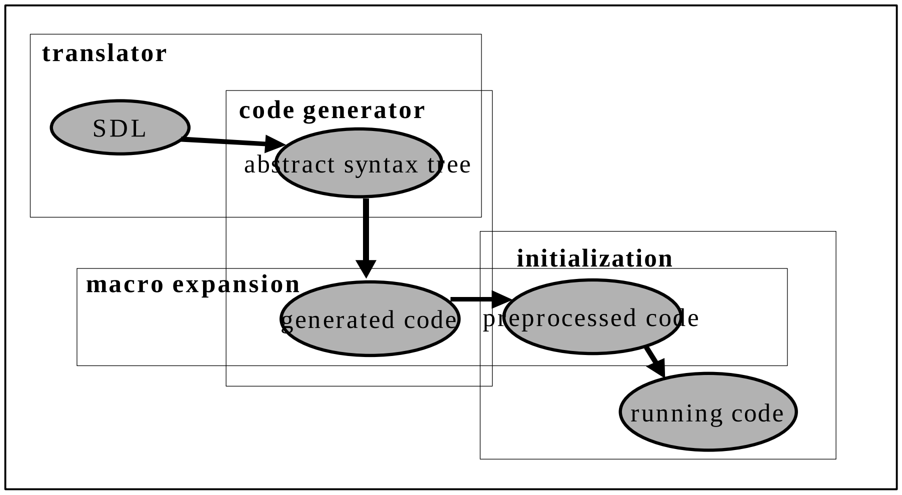

Figure 20

Automatic code generation from SDL has a much broader scope than can be expected. Generated code is used internally in many SDL tools for simulation, validation, and test case generation. Some SDL tools use generated code for performance evaluation. But one of the biggest promises for accelerated software development is associated with the use of the generated code for production [3,29]. However surprisingly few industrial projects are known, where automatically generated code is used in final products [3]. Figure 20 shows main transformations involved in generating target code.

There exist several issues in production use of the generated code [3,29]:

- target language (CHILL, C, C++, Java, etc.)
- quality of the generated code [6,10,11]
  - size

- execution speed
- readability
- interfaces to the generated code [3]
- integration with legacy code
- direct use of the implementation language with the generated code [3]
- different generation schemes (optimizations)
- deployment of the generated code

Each project has *unique* requirements to the generated code which means code generation should be *adaptable* (i.e. allow to change decisions about the structure and the behavior of the generated code). As we put in [3], the ideal situation is when a code generator is another “team member”, which completely fits into the development environment of a particular project.

Several research groups are working in the direction of adaptable code generation. Approaches to adaptable code generation include:

- Run-time support system
- Macros in generated code
- Object-oriented frameworks
- Configurable code generator
- Target code in comments
- Target Language Annotations
- Deployment Annotations
- Declarative mappings

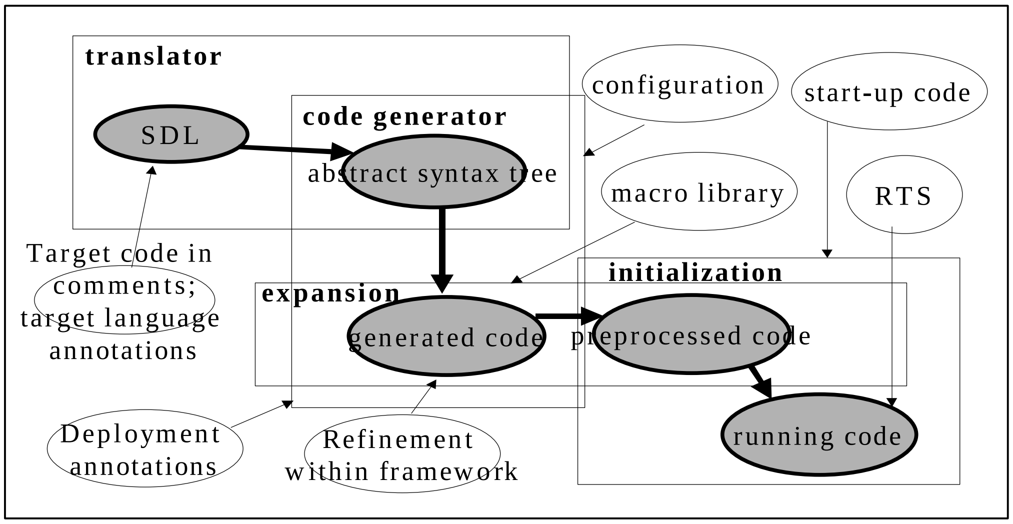

Figure 21

Different stages of the translation process can be used for adaptation (Figure 21).

The major challenge of code generation is the design of a mapping [6,11]. The **mapping**: an interrelated set of decisions on how to represent executable and structural source constructs by target constructs, so that the resulting generated program is both syntactically and semantically correct.

The following mapping-related issues limit design choices for an adaptable code generator:

- Often executable constructs of the source language have semantically close equivalents in the target language and can be *represented directly* by these equivalents (e.g. loops, conditionals, assignments, etc.)
- Semantically distant executable constructs can be *modeled* by groups of executable (and perhaps a few minor structural) constructs of the target language. Often, such groups can be encapsulated in the run-time support library.
- Few structural constructs can be represented by semantically close equivalents (e.g. objects, procedures, modules, packages, etc.)
- Some structural constructs require transformation during translation (but this prevents adaptable code generation)
- Most structural constructs can be interpreted at run-time (but this is inefficient)
- Some structural constructs can be represented by groups of structural (and perhaps also executable) constructs (but this can lead to fragmentation [6] and prevents adaptable code generation)

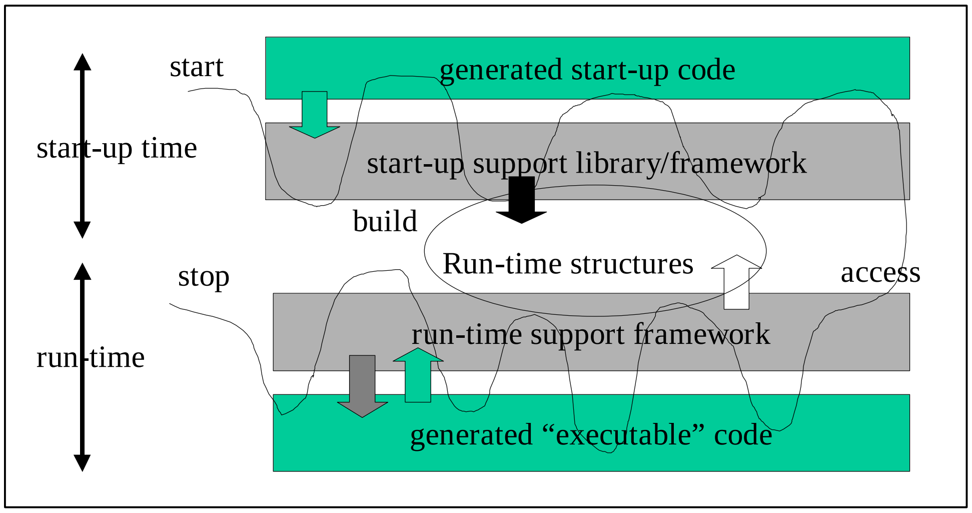
start
start-up time
generated start-up code
start-up support library/framework
build
Run-time structures
stop
access
run-time
run-time support framework
generated "executable" code

Figure 22

Adaptation is quite hard to achieve because of the following factors:

- Mapping decisions are not localized
- Mapping decisions are highly interrelated
- There are not many mappings available because of the semantic and syntactic constraints of the target language (although there can be more individual mapping alternatives)
- Mapping decisions are ***irrevocable***

In [6,11] we have introduced the concept of declarative mappings as a technique for generating adaptable code from SDL (see Figure 22). Mapping is called *declarative* when:

- Decisions about the representation of the problematic structural constructs are *postponed* until run-time
- Generated code contains "*declarations*" of what constructs of the source language constitute the system
- "Declarations" are executed at the *start-up time* of the generated program
- Start-up support is a *library* which determines, what run-time structures are being built
- Run-time support is a *framework* for the generated "executable" constructs and the dynamically created run-time structures

Automatic code generation is quite an obvious way to dramatic improvements in time to market. However, as we demonstrated above, considerable effort needs to put into integrating code generation into realistic industrial-strength software development. We believe that research on adaptable code generation from SDL is very important for wider adoption of SDL in an industrial context. In our experience, there are no simple «one-size-fits-all» solutions when is comes to adaptable code generation. A combination of approaches, outlined above, including our technique of declarative mappings can be used.

7. Re-engineering of SDL specifications from legacy

CASE-based approaches (often using SDL tools) offer significant improvements in quality, productivity, and time to market. However, in order for CASE-based communications 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 often very poorly documented. Up to now, this fact has constituted a "*legacy barrier*" to the cost effective use of new development technologies [5,13].

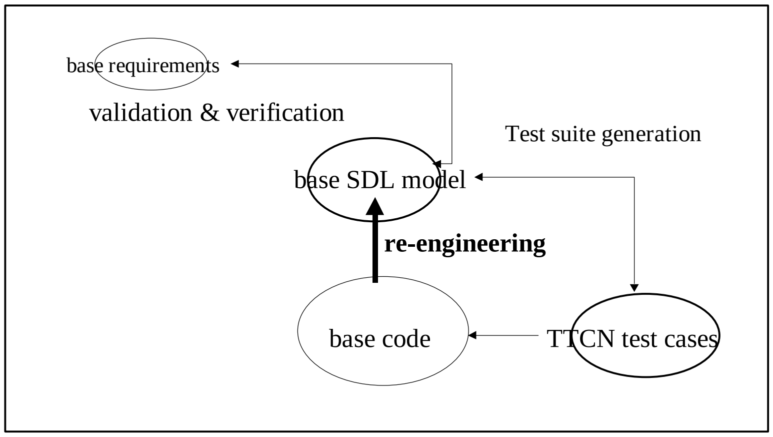

Figure 23

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, which often produces more intuitive results and does not involve the costly use of the target hardware (Figure 23);
- automated generation of regression test cases for the base software platform (Figure 23);
- analysis and validation of the formal specifications of the new features built on top of the SDL model of the base software platform (Figure 24);
- feature interaction analysis including existing and new features (Figure 24);
- automated generation of test cases for new features (Figure 24);
- automatic generation of implementations of the new features (Figure 24). Such implementations are retargetable for different implementation languages (e.g. C, C++, CHILL) as well as for different real-time operating systems (e.g. pSOS, VxWorks, etc.). Adaptable code generation should be used for better integration into existing development processes (Section 6).

In this paper we present our methodology of *dynamic scenario-based* re-engineering of legacy telecommunications systems into a system design model expressed in SDL [5].

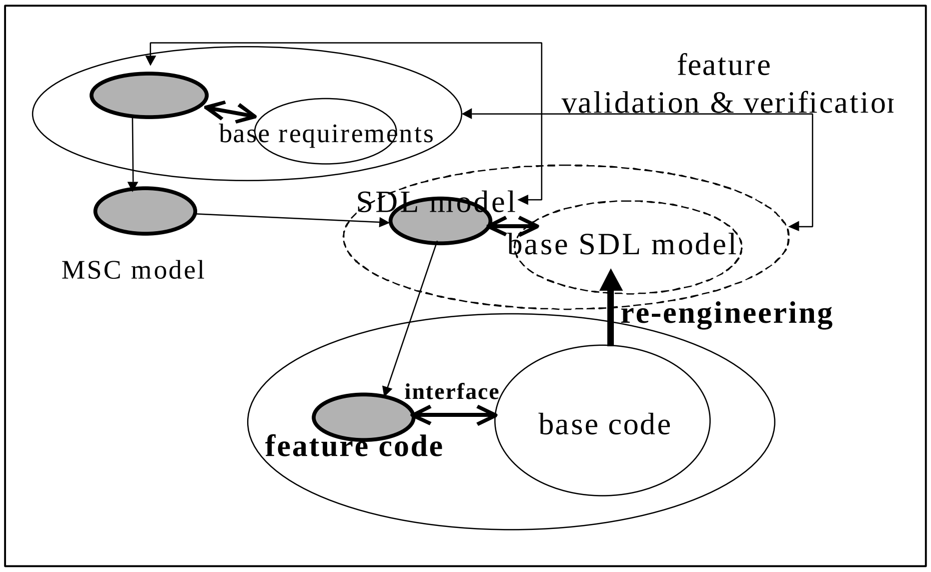

Figure 24

Our approach consists of

- placing semantic probes [5] 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-92 model from this set of Message Sequence Charts (MSCs) using the Moscow Synthesizer Tool [8].

This process is repeated until the SDL design model satisfies certain validity constraints [5]. 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 [5].

7.1. Re-engineering Methodology Overview

Dynamic scenario-based re-engineering of legacy software into SDL models is a process, where an SDL model is synthesized from *probe traces* [5], collected from *dynamically* executing the *instrumented* legacy system (see Figure 25). More specifically, in the process of scenario-based re-engineering, the SDL model is synthesized from a higher-level representation - *extended MSC-92 model (*later referred to simply as MSC model) which is abstracted from probe traces. The execution is driven by a *test suite* [5].

The enabling technology for our dynamic scenario-based re-engineering process is automatic synthesis of SDL models from a set of MSCs [8]. So far automatic synthesis of SDL models from MSC was considered only as a forward engineering technology (see Section 4). 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.

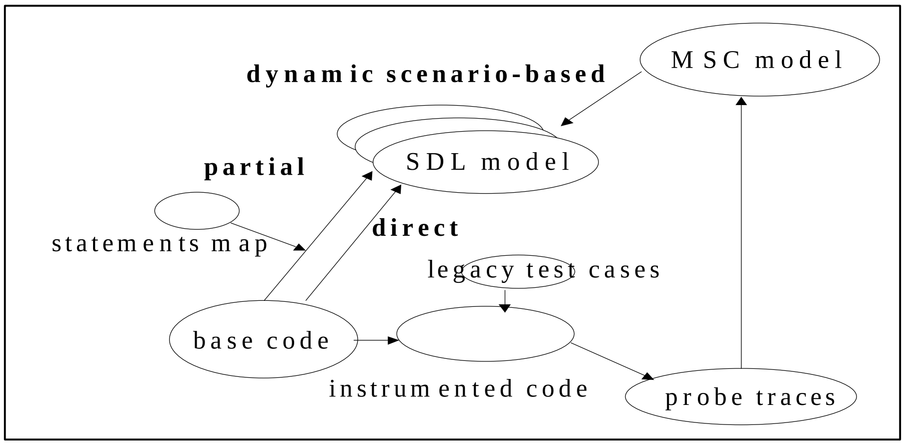

Figure 25

An alternative approach to re-engineering of legacy software into SDL models, the so-called *direct* automatic re-engineering [2,5] is also shown in Figure 25. In contrast, the direct re-engineering approach derives an SDL model statically from the source code by performing semantic-preserving translation [2]. An alternative to direct re-engineering is described in [21]. We call this approach *partial* re-engineering. During the partial re-engineering only a selective translation is performed, under control of the so-called *statements map* (see Figure 25). The statements map is a table which contains all statements from the source program (in some canonical form). The idea of the statements map is to (manually) provide the target model statement for any *relevant* source statement. The framework of the target model is extracted automatically. Detailed comparison of re-engineering approaches is contained in Section 7.3.5.

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 26. In Figure 26 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 26 represent flows of data, which determine the sequence of methodology steps. A detailed description of methodology steps is contained in the next section.

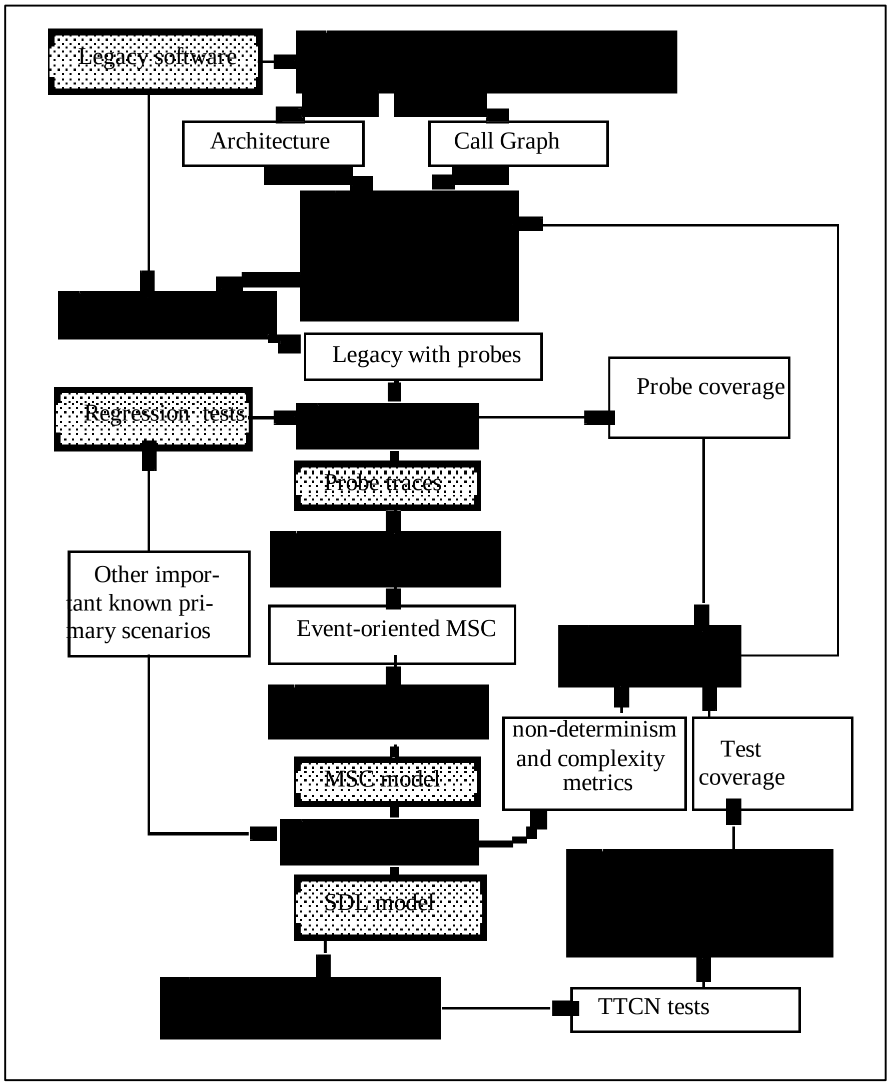

Figure 26

7.3. Detailed steps in the methodology

7.3.1. 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 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 [7]. 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. It also affects the level of details in traces and thus (among other things) the amount of adaptation work for automatically generated a test suite [19] (see also step 11).

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 [5] 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 [5]. 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, the resulting SDL model is used to produce TTCN test cases (using SDL tools, according to known techniques [19,20]). During this process we are more interested in secondary higher-yield scenarios. Discussion of the yield of scenarios with respect to requirements validation was presented in Section 3.

7.3.2. 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.

7.3.3. 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 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.** 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 [15] into places where loops or branching are possible. Note, that we are using an extended event-oriented MSC-92 notation as the input to the MOST-SDL tool [5]. 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). Synthesizer technology is briefly described in the next section. A more detailed description is contained in [5] and also in Section 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.

7.3.4. Investigation 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 10. **Generate TTCN test cases.** Using an SDL model for automatic generation of test cases is one of our ultimate goals. Techniques for automatic generation of test cases form SDL models are described elsewhere [19,20].

Step 11. **Execute tests on legacy and assess coverage.** Execution of the automatically generated test case may require some *adaptation* (conversion of abstracted interfaces into original interfaces) [19]. Adaptation of test interfaces to system interfaces is shown in Figure 27. Probe traces and the corresponding automatically generated test cases use *abstracted interfaces* of the system (according to selected *probe placement strategy*).

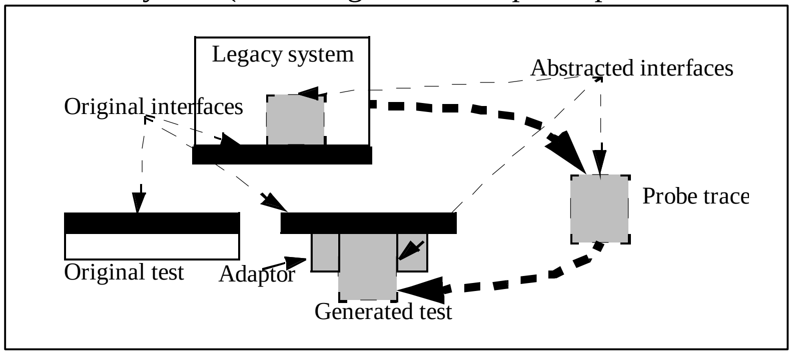

Figure 27

An abstracted interface consists of internal functions (grey box) within the system, which are not necessarily accessible from the environment of the legacy system. In Figure 27 this is illustrated by a grey cavity inside of the rectangle representing the legacy code.

Original tests use original interfaces of the system. The original interface consists of externally accessible functions (black box). In Figure 27 this is illustrated as the outer bottom layer of the legacy code. The functionality of the *adaptor* (illustrated as hatched area) is to convert an abstracted interface back into the original one.

Our experience demonstrates that some compromise should be found between the "depth" of the abstracted interface and the simplicity of the adaptor. An abstracted interface at the logical level greatly simplifies probe traces and leads to more meaningful generated SDL models. In some cases a special-purpose direct access to the internal functionality should be provided for the ease of implementing an adaptor.

Step 12. **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.

7.3.5. Comparison with related approaches

The alternative approach to re-engineering of legacy software into SDL models is the so-called direct automatic re-engineering (Figure 25). Direct re-engineering approach derives SDL model statically from the source code by performing semantic-preserving translation [2]. 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 [2]. In contrast, SDL models which are synthesized according to our dynamic scenario-based approach always contains *less* information than the implementation.

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 [2]. The level of detail of directly generated SDL models can 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*.

The so-called partial re-engineering [21] is an interesting alternative to direct re-engineering. According to this approach, only the framework of the model is extracted automatically (in [21] 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: 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. 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. The biggest advantages of 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.

In our experience, the use of dynamic scenario-based re-engineering methodology combined with subsequent 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.

As compared to a direct static re-engineering approach, 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 re-engineering methodology provides an efficient means of improving understanding of both legacy code and regression test suites. Automatic re-engineering of legacy telecommunication software into SDL models is the 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.

8. Conclusions and Future Research Directions

We summarize our experience in building new generation formal methods-based CASE tools aimed at practical improvement of software engineering in telecommunication industry. We presented our **accelerated development methodology** for the specification, design, testing and re-engineering of telecommunications software, based on extensive use of formal methods and formal languages for the description of the software very early in the development process and automated re-engineering of formal models from legacy telecommunications software. Our methodology is based on the most widely accepted telecommunication formal languages, standardized by the International Telecommunications Union (ITU): *Specification and Description Language* (SDL), scenario description language called *Message Sequence Charts* (MSC), test description language called *Tree and Tabular Combined Notation* (TTCN), data description language *Abstract Syntax Notation* (ASN.1).

We emphasized the following key components of the methodology:

- Capture of requirements by use-cases using the MSC language
- High-yield requirements validation using SDL requirement models
- Synthesis of SDL requirements models from approved MSC scenarios
- Seamless refinement of SDL requirement models into design models
- Adaptable code generation from the SDL models
- Automatic recovery of SDL models from legacy software

One of the major goals of our research is to lower barriers for adoption of formal methods at the early phases of the development process (requirements capture) as well as at the later phases (maintenance). We use well-known benefits of formal methods-based tools and techniques for the middle phases of the development process.

Our accelerated development methodology creates an upfront increase in project time and effort to reduce the total cost and time to market. We claim that an initial investment of approximately 17% required to produce and validate a formal SDL specification and generate test suites at the early phases can result in up to 30-50% saving of the total project time. We develop the new generation formal methods-based tools to support our methodology which will further increase the time to market.

The ideas and results presented in this paper represent our on-going research. Our intention is to continue the definition of the accelerated development methodology and the corresponding open architecture, composed of languages and tools, supporting the methodology and automating it as far as possible. In particular, we are investigating fast test case construction techniques using SDL tools, implementation of the partial re-engineering approach for SDL, and architecture analysis of legacy telecommunications software. Other future research directions of the Department for CASE tools include extensive case studies aimed at assessing the practical application of various approaches to re-engineering, as well as the combined use of the forward and reverse engineering techniques for improving time to market for typical telecommunication feature development.

9. References

[1] N. Mansurov, A. Kalinov, A. Ragozin, A. Chernov (1995), Design Issues of RASTA SDL-92 Translator, in Proc. of the 7-th SDL Forum, Oslo, Norway, 26-29 September, 1995, Elsevier Science Publishers B.V. (North-Holland), pp. 165-174

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

[3] N. Mansurov; A. Ragozin; A. Chernov; I. Mansurov: “Industrial strength code generation from SDL”, in A. Cavalli, A. Sarma (Eds.) SDL’97: TIME FOR TESTING –

SDL, MSC and Trends, Proc. Of the 8-th SDL Forum, Evry, France, 23-26 September, 1997, , Elsevier Science Publishers B. V. (North-Holland), pp. 415—430.
[4] N. Mansurov, A. Ragozin, A. Chernov, I.Mansurov, Tool suppport for algebraic Specifications of Data in SDL-92, in Proc. Formal Description Techniques IX, Kaiserslautern, Germany, 8-11 October 1996, Chapman & Hall, p 61-76.
[5] N. Mansurov, R. Probert, 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] N. Mansurov, A. Ragozin, Generating readable programs from SDL, in Proc. SAM'98 workshop, 29 June-1 July, 1998, Humboldt University, Berlin, 1998.
[7] N. Rajala, D. Campara, N. Mansurov, inSight Reverse Engineering CASE Tool, in Proc. of the ICSE'99, Los Angeles, USA, 1998.
[8] N. Mansurov, D. Zhukov, 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).
[9] N.Mansurov, A. Chernov, Generation of executable code for Algebraic Specifications, in Voprosy kibernetiki: System Programming Applications, N. 2, Moscow, 1996 (in Russian)
[10] N.Mansurov, A. Ragozin, Generation of code with simple and readable structure from SDL-92, in Voprosy kibernetiki: System Programming Applications, N. 3, Moscow, 1997 (in Russian)
[11] N. Mansurov, A.Ragozin, Using declarative mappings for automatic code generation from SDL and ASN.1, in Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999, Elsevier Science Publishers B.V. (North-Holland).
[12] N. Mansurov, O. Majlingova, Formal specification methodologies: MSC and SDL languages (lecture notes and tutorial), Moscow State University, Department of Computational Mathematics and Cybernetics, 1998 (in Russian).
[13] R. Probert, N. Mansurov, Improving time-to-market using SDL tools and techniques (tutorial), Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999.
[14] ITU-T (1993), CCITT Specification and Description Language (SDL), ITU-T, June 1994
[15] Z.120 (1996) CCITT Message Sequence Charts (MSC), ITU-T, June 1992
[16] R.L. Probert, O.Monkewich, TTCN: the international notation for specifying tests for communications systems, Computer Networks, Vol. 23, No. 5, 1992, pp. 417-738
[17] ISO/IEC 8824: Specification of Abstract Syntax Notation One (ASN.1) (1989).
[18] Telelogic (1998), Telelogic ORCA and SDT 3.3, Telelogic AB, Box 4128, S-203 12 Malmoe, Sweden, 1998
[19] Anders Ek, Jens Grabowski, Dieter Hogrefe, Richard Jerome, Beat Koch, Michael Schmitt, Towards the Industrial Use of Validation Techniques and Automatic Test Generation Methods for SDL Specifications, in Proc. of the 8-th SDL Forum, Evry, France, 23-26 September, 1997, Elsevier Science Publishers B.V. (North-Holland), pp. 245-261
[20] R. Probert, H. Ural, A. Williams, J. Li, R. Plackowski, Experience with rapid generation of functional tests using MSCs, SDL, and TTCN, submitted to Special Issue of Formal Descriptions Techniques in Practice of Computer Communications, 1999

[21] G. Holzmann, 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.
[22] I. Jacobson, M. Christerson., P. Jonsson, G. Overgaard, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Reading, MA, 1992.
[23] G. Robert, F. Khendek, P. Grogono, 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
[24] R.Tuok, L. Logrippo, Formal specification and use case generation for a mobile telephony system, Computer Networks and ISDN Systems, 30 (1998), pp. 1045-1063.
[25] M. Andersson, J. Bergstrand, Formalization of Use Cases with Message Sequence Charts, MSc Thesis, Lund Institute of Technology, May 1995
[26] R. L. Probert, K. Saleh, Synthesis of communication protocols: survey and assessment, IEEE Transactions on Computers, 40(4), pp. 468-475, April 1991
[27] S. Leue, L. Mehrmann, M. Rezai, Synthesizing ROOM Models from Message Sequence Chart Specifications, University of Waterloo, Technical Report 98-06, 1998
[28] S. Some, R. Dssouli, and J. Vaucher, From scenarios to timed automata: Building specifications from user requirements, In Proc. 2nd Asia Pacific Software Engineering Conference, IEEE, December 1995.
[29] R. Singh, J. Serviss, Code Generation using GEODE: A CASE Study, in Proc. of the 8-th SDL Forum, Evry, France, 23-26 September, 1997, Elsevier Science Publishers B.V. (North-Holland), pp. 539-550
[30] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modelling Language Reference Manual, Addison-Wesley, 1999
[31] A. Ragozin, Automatic generation and execution of programs from SDL specifications, PhD thesis, Moscow State University, 1999