Automatic synthesis of SDL models in Use Case Methodology

N. Mansurov\textsuperscript{a} and D. Zhukov\textsuperscript{b}

\textsuperscript{a,b}Department of CASE tools, Institute for System Programming (ISP), 25 B. Kommunisticheskaya, Moscow 109004, Russia; email \{nick, dmjr\}@ispras.ru

Formal description techniques (FDT’s) supported by computer-aided software engineering (CASE) tools are rapidly evolving as a response to the new challenges of the telecommunications industry, especially the need to improve “\textit{time-to-market}” of software products. In this paper we summarize our experience in using \textit{automatic synthesis} of formal models in ITU-T standard Specification and Description Language (SDL) to \textit{speed-up} the software development process. Suggested \textit{accelerated methodology} requires formalization of functional scenarios (use cases) using another ITU-T standard - Message Sequence Charts (MSC) extended with data operations. Our Moscow Synthesizer Tool (MOST-SDL) provides a \textit{bridge} from MSC models to SDL executable specifications which can be simulated using SDL tools to provide an early \textit{feedback} for the phases of requirements analysis, system analysis or system design. We present our synthesis algorithm, provide comparison with related work and discuss the results of a few case studies where the Moscow Synthesizer Tool was used together with Telelogic SDL tools for an accelerated MSC-based prototyping which involved incremental synthesis, validation and simulation of the formal model.

1. \textbf{Introduction}

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 \cite{1,2}. In particular, “\textit{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 \cite{1}. 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.

ITU-T Specification and Description Language (SDL) \cite{4} is one of the most successful telecommunications standard FDT \cite{8}. 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 \cite{6}. 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. “Success stories” of using SDL in industry mention the phases of system design, detailed design, automatic generation of implementations \cite{19} as well as testing \cite{8}.
However, the early phases of the software development process require different approaches. Use case based methodologies are becoming predominant in software development [2,1,3,11]. Use case based methodologies share the common way of capturing customer requirements as *scenarios*. Message Sequence Charts (MSC) [5] or Sequence Diagrams of the Unified Modelling Language (UML) [20] 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 modelling as compared to 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.

In this paper, we summarize our experience in using automatic synthesis of SDL models to speed-up the use case based development process. Our accelerated methodology requires formalization of use cases using MSC extended with data operations. We have developed the synthesis algorithm and the corresponding tool called the Moscow Synthesizer Tool (MOST-SDL\(^{1}\)) which provides a bridge from MSC models to executable SDL specifications. Existing SDL tools can be used to simulate the synthesized models. Additional scenarios can be generated during the simulation of the synthesized SDL model. Thus the automatic synthesis of SDL models enables *tool-aided iterative development* at the early phases of the software process. The timely *feedback* between SDL and MSC models is the prerequisite for the speed-up.

We discuss two possible ways of incorporating the MOST-SDL tool into a use case based methodology. One strategy is to use the MOST-SDL tool to speed-up the *requirements analysis* phase. The feedback provided by SDL tools allows to quickly discover inconsistencies and incompleteness of the requirements model. Another strategy is to use the MOST-SDL tool at the *system analysis* phase when the architecture of the system is being defined and independent groups of developers produce system scenarios for each architecture component. In this case the MOST-SDL tool will produce an executable architecture model of the system by synthesizing the behavior of each component and integrating it into an aggregate model. Automatically derived relationships between components can be compared to the intended ones. Exploration of the executable architecture model allows to uncover system analysis faults. We discuss the results of a few case studies where Moscow Synthesizer Tool was used in conjunction with Telelogic SDL tools [6] for an accelerated MSC-based prototyping of telecommunications-like systems which involved incremental synthesis, simulation and validation of the formal model.

The rest of the paper has the following organization. In Section 2 we discuss methodological issues of using automatic synthesis in a use case based software development process. In Section 3 we present our MSC extensions for capturing data flows over the use cases. In Section 4 we describe the details of the synthesis algorithm of the Moscow Synthesizer Tool, outline the overall structure of the synthesized SDL models and provide an illustrative example. In Section 5 we provide comparison with related work. In Section 6 we summarize our experience and make some conclusions.

\(^{1}\) The word “MOST” means “bridge” in the russian language.
2. Methodological issues

Use case based methodologies are becoming predominant in software development. The main concepts of a use case methodology are actors and use cases [2]. 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. One use case can cover several sequences of events, called scenarios. Several use case based methodologies were proposed [1,2,3,10,11].

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 [2,20], scenarios are captured using tabular forms [11], plain text [1,3] or UML Sequence Diagrams [20]. Formalization of use cases using MSC diagrams was suggested in [12].

A overview of a typical use case methodology is presented in Figure 1. 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 modelling perspectives [20,3,13], 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 1. Overview of a use case based methodology
In a typical telecommunications oriented industrial process formalization starts at the detailed design phase. The system model (Figure 1) is represented as several SDL block diagrams and the detailed model is represented as the corresponding set of SDL process diagrams [6]. 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.

2.1 Automatic synthesis at the requirements analysis phase

We suggest two ways of using the automatic synthesis to speed-up the traditional process depending on the phase to which formalization is extended. The most ambitious strategy is to synthesize an SDL model at the requirements analysis phase (Figure 2). 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 3). 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 2 become useful because they are provided in a timely manner as the requirements model is being created, enabling the iterative development of the latter.

**FIGURE 2. Automatic synthesis at the requirements analysis phase**

- requirements
- use cases
- actors
- automatic synthesis
- synthesized requirements model
- implementation
- requirements capture
- requirements analysis
- conformance tests
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 [17]. 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 test cases [8] (which at this phase of the development process correspond to conformance or acceptance tests).

2.2 Automatic synthesis at the system analysis phase

Another strategy is to use the automatic synthesis at the system analysis phase when the architecture of the system is being defined and independent groups of developers produce system scenarios for each architecture component (Figure 3). 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 3). In this case the SAM will reproduce the architectural components of the system by deriving them from the collection of system 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.

FIGURE 3. Automatic synthesis at system analysis phase
3. Formalization of Use Cases using extended MSC

We use extended Message Sequence Charts (MSC) language [5] 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).

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 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 := 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 <op> var
   var <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 4 and Figure 5. Two local data flows through scenario LocalDataFlows are shown (as two dashed lines) in Figure 4. The first flow contains the following MSC events: a: in x(p,q) from env; a: create b(p,q); b: out y to c; c: out z(r) to env; Note, that instances B and C use different (local) copies of the variable r. Thus the parameter of the message z which is sent by the instance C to the environment is not necessarily equal to p+1.
The second flow contains the following MSC events:  
\[ a: \text{in } x(p,q) \text{ from env}; \]
\[ a: \text{create } b(p,q); \]
\[ b: \text{action } 'r:=p+1'; \]
\[ b: \text{out } w(r) \text{ to env}; \]

Note that the instance \( C \) will send message \( z(r) \) to environment only when condition \( q>0 \) is true. Alternative events for the instance \( C \) can be specified using the local condition \( \text{chk}_1 \) with a different guard. Global conditions in the HMSC graph will be required when alternative events involve other instances.

**FIGURE 4. Local data flows**

![Diagram of local data flows]

Figure 5 illustrates the specification of global data flows between use cases. In Figure 5 parameters of the message \( \text{value} \) returned by the use case \( \text{UC\_POP} \) as a reaction to the message \( \text{get} \) depends on the events in the use case \( \text{UC\_PUSH} \). \( \text{push} \) and \( \text{pop} \) are assumed to be user-defined SDL procedures with in/out parameters implementing typical stack operations.

**FIGURE 5. Global data flows**

![Diagram of global data flows]

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. **Synthesis Algorithm**

4.1 **Overview of the algorithm**

In this section we describe our algorithm of synthesizing an SDL model from an (H)MSC model. The input to our algorithm is a set of HMSCs and the corresponding referenced bMSCs in the event-oriented textual form. bMSCs can contain data extensions described in the previous section. Restrictions on the input MSC language are as follows: bMSCs can not contain inline expressions, MSC reference symbols and coregions. Parallel frame in HMSC is also not supported.

The overview of our algorithm is presented in Figure 6. Individual steps of the algorithm are described below.

**FIGURE 6. Overview of the synthesis algorithm**

1. **Parsing** - this step performs syntax and semantic analysis of the MSC model. This step is applied to each MSC diagram. The output of this step is an abstract syntax tree of the MSC diagram with additional semantic information.

2. **Ordering** - this step creates an aggregate behavior graph by performing composition of all bMSCs according to HMSCs.

3. **Slicing** - this step creates an MSC slice of the aggregate behavior graph for one actor. The slice of the MSC model contains all instances related to the given actor. The slice can be considered as a non-deterministic finite automaton (NFA) which accepts all sequences of events on its instances. Switching from one sequence to another is made according to connections in the HMSC graph.

4. **Converting** - this step constructs a deterministic finite automaton (DFA) from the NFA. A well-known subset construction algorithm [7] is used. Only useful subsets are constructed, which allows to avoid the exponential growth of the number of states in the original NFA in most cases.

5. **Minimizing** - this step minimizes the DFA, produced at the previous step. Converting and minimizing steps are performed for each MSC slice independently.

6. **Generating** - this step produces the resulting SDL model from a minimized DFA. SDL graphs are generated from the DFA, while SDL structure is generated directly from the aggregate behavior graph.
4.2 MSC Events and Finite Automata

Our synthesis algorithm constructs finite automata corresponding to MSC instances. The input symbols for the finite automata constructed by our algorithm are MSC events. We distinguish between input and active events.

Input events include the following MSC events:

- message inputs $in(m,i)$ for all message names $m$ and instances $i$;
- timeouts $timeout(t)$ for all timer names $t$.

Active events include the following MSC events:

- message outputs $out(m,i)$ for all message names $m$ and instances $i$;
- MSC actions $action(a)$ for all assignments and function calls $a$;
- set timer actions $set(t,d)$ for all timer names $t$ and duration values $d$ ($d$ may be omitted);
- reset timer actions $reset(t)$ for all timer names $t$;
- stop action $stop$;
- local conditions $check(C)$ where $C$ is a boolean expression;

A non-deterministic finite automaton (NFA) is a tuple $A=(Q,E,q_0,\delta,\varepsilon)$ where

- $Q$ is the set of states;
- $E$ is the set of events;
- $q_0$ is an element of $Q$ called the start state;
- $\delta : Q \times E \rightarrow 2^Q$ is the transition function (which maps set-event pairs to sets of states). For any state $q$ and event $e$ the value of $\delta(q,e)$ is the set of all possible states to which the automaton can go from state $q$. Each transition is labeled by an event $e$. We will write $q \rightarrow (e) q'$ if $q'$ belongs to $\delta(q,e)$.
- $\varepsilon : Q \rightarrow 2^Q$ is the idle transition function. An "idle" transition without any event is allowed from state $q$ to $q'$ if $q'$ belongs to $\varepsilon(q)$. To denote this fact we will write $q \rightarrow (\lambda) q'$ where $\lambda$ is an empty sequence of events.

A deterministic finite automaton (DFA) is a special case of a NFA in which

- no state has an idle transition, i.e. $|\varepsilon|=0$
- for each state $q$ and event $e$, there is at most one transition labeled $e$ leaving $q$, i.e. $|\delta(q,e)| \leq 1$ for each state $q$ in $Q$ and for each $e$ in $E$.

Automata will be represented diagrammatically by transition graphs in which the nodes are states and the labeled edges represent the transition function (see Figure 8 and Figure 9).

We define an MSC slice (with respect to an instance $i$) as a non-deterministic finite automaton $NFA_i$ which is constructed according to the following rules:

1. The set of NFA states contains a state for each symbol in the HMSC graph. Lines connecting symbols are mapped to idle transitions between the corresponding states.
2. For each bMSC $m$ a sequence of states is created that performs the sequence of events along the instance $i$ in bMSC $m$. For each reference to bMSC $m$ the corresponding state is replaced by the new sequence of states. If the bMSC $m$ does not contain instance $i$ then the new sequence of states is empty.

3. The start state of the NFA corresponds to the start symbol of the HMSC.

By construction, the language recognized by an MSC slice is equivalent to the set of all valid event traces for the corresponding MSC instance.

### 4.3 Generating SDL graphs

SDL graphs are generated from a minimized deterministic finite automaton $DFA_i = (Q, E, q_0, \delta)$. Active events are mapped onto SDL statements. DFA states are mapped to SDL states and free actions according to the following rules:

1. States which have outgoing transitions labeled only by input events are mapped to SDL states. Input events of this DFA state are mapped to the input stimuli of the SDL state. An *asterisk save* statement is added to each state to prevent deadlocks [12].

2. States which have a single outgoing transition labeled only an active event are mapped to SDL free actions.

3. States with multiple outgoing transitions labeled only by active events are mapped onto free actions starting with a non-deterministic choice between transitions using SDL decision(any) statement.

4. States which have multiple outgoing transitions labeled by both input and active events are mapped to SDL free actions starting with a non-deterministic choice between the active events. Additional alternative contains an SDL nextstate into another SDL state corresponding to input events (see rule 1).

5. Each state which has transitions labeled by events $\text{check}(C_1), ..., \text{check}(C_n)$ is mapped to a chain of SDL decision statements which select a transition with a satisfied condition. We impose the following restrictions on the usage of local conditions:
   - If a state has an outgoing transition labeled by a $\text{check}(C)$ event then all outgoing transitions from this state must have $\text{check}$ events.
   - If a state has outgoing transitions with events $\text{check}(C_1), ..., \text{check}(C_n)$ then expressions $C_1, ..., C_n$ must be mutually exclusive.

### 4.4 Generating SDL structure

We attempted to synthesize flexible typebased SDL models in order to facilitate refinement of the model and better reusability of its components (see example in Section 4.5). The synthesized model includes one SDL package and an SDL system. The synthesized package contains one process type for the system actor and one process type for each external actor (Figure 11). The package contains synthesized definitions of SDL signals for all messages used in the MSC model (Figure 11). Each process type has separate gates for incoming and outgoing signals. Start transitions are generated as virtual to allow redefinition in subtypes (Figure 10).
The package also contains a block type specifying instantiations of actors and the collaboration between external actors and the system (Figure 11). This block type contains a typebased process instance for each actor in the model. Process instances are connected by explicit channels. Process types for actors are made virtual within the block type. In order to do that additional virtual process type definitions are provided within the block type (Figure 11). These additional process types simply inherit the corresponding process types generated for actors.

We also generate an SDL system which contains a single typebased instance of the block. The generated SDL system imports definitions from the generated package (Figure 12). Redefinitions of the synthesized model can provide subtypes of the synthesized block type (Figure 12). SDL graphs for actors can be reused by directly inheriting from synthesized process types (Figure 10).

4.5 Example

We are illustrating our algorithm by considering a simple MSC model shown in Figure 7. It contains two use cases Wait and Reply where each of them has only one scenario. The instance R corresponds to the system actor and the instance S is an external actor. MSC slices for this MSC model are shown in Figure 8.

FIGURE 7. Example MSC model

![HMSC ExampleModel](image1)

MSC Wait

![MSC Wait](image2)

MSC Reply

![MSC Reply](image3)

Note that according to the definition of an MSC slice (Section 4.2) slices $N_A_R$ and $N_A_S$ in Figure 8 have the same structure as the HMCS ExampleModel in Figure 7.

FIGURE 8. MSC slices

![NAR](image4)

![NAS](image5)
Synthesized DFA for this example are shown in Figure 9. Note that although $D_A S$ is deterministic in automata terms (Section 4.2), it specifies a non-deterministic behavior in terms of MSC events: outgoing transitions from the state $q_0$ are labeled by active MSC events $out(a,r)$ and $out(b,r)$ which are not caused by any external stimuli. As can be observed from the original MSC model (Figure 7), $D_A S$ specifies the behavior of an external actor where messages $a$ and $b$ initiate use cases. We generate a non-deterministic SDL graph (Figure 10).

Synthesized DFA for this example are shown in Figure 9. Note that although $D_A S$ is deterministic in automata terms (Section 4.2), it specifies a non-deterministic behavior in terms of MSC events: outgoing transitions from the state $q_0$ are labeled by active MSC events $out(a,r)$ and $out(b,r)$ which are not caused by any external stimuli. As can be observed from the original MSC model (Figure 7), $D_A S$ specifies the behavior of an external actor where messages $a$ and $b$ initiate use cases. We generate a non-deterministic SDL graph (Figure 10).
5. 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 [13,17,18]. Synthesizing SDL specifications from MSC is addressed in [12]. Survey of work on a more general subject of protocol synthesis is available in [15].

Methodological issues of generating a formal executable specification from a set of use cases are addressed in [13]. 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 [18] 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 [17] 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 [17], the MSC traces are useful for visualizing execution sequences that are longer than 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 (Section 2.2), however we also use automatic synthesis to create executable requirements models (Section 2.1). 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 [17]. 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 [12] 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 [12]. 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 [12]. Thus the Concordia synthesizer produces only behavioral components. Composition of bMSCs using HMSC was not addressed in [12] 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 [17]) which allows us to synthesize executable requirements models (similar to [13]) as well as executable architecture models [17]. 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.

6. Conclusions

We have shown how automatic synthesis of SDL models can be used to speed-up the use case based software development process. Our main motivation is to extend the use of formal description techniques to the early phases of the development process. Our accelerated methodology uses MSC with data extensions to formalize use cases. The speed-up is achieved through enabling tool-aided iterative development and rapid transition from requirements analysis to design and validation. Automatically synthesized requirements models and architecture models are executable and can be simulated using existing SDL tools. Data extensions are essential to provide accurate capture of the functional requirements.

MSC language with data extensions can be used as an accelerated prototyping language. Using MSCs for forward engineering combines high descriptive power of a specification language with fast code generation.
The Moscow Synthesizer Tool was applied to several small-sized case studies of telecommunications-like software systems, in particular the ToolExchange project, described in [9].

The Moscow Synthesizer Tool was also used at the Moscow State University in the undergraduate course “Formal specification methodologies: MSC and SDL” [10]. The course introduces a use case based software development methodology. MSC are used to formalize use cases. SDL is used to capture the results of architectural analysis [2] as well as to perform system and detailed design. The course involves practical assignments, where each student is presented with a realistic case study. Each assignment involves going through all phases of the development process and constructing a series of formal models, driven by the original use cases, up to the point when a complete executable SDL model of the case study is developed. In parallel, we experimented with the MOST-SDL tool. The MOST-SDL tool was applied to the formalized use case model as soon as it became available. Later, the automatically synthesized SDL models were compared with the models developed manually by students. Results of this comparison show, that the automatic synthesis of SDL models allows more than 40% reduction of the development time, and a dramatic improvement of the quality of SDL models.

In our experience, the MOST-SDL tool has a good potential as a training tool, because students or new developers can inspect (rather non-trivial) SDL models produced by the tool, while specifying functionality in a “low learning curve” MSC formalism.

Simulation of the synthesized models provided very useful feedback to the analysts. In particular, we discovered that the non-deterministic SDL graphs of the external actors resulted in excellent test suites. Additional scenarios, generated during the exploration of the synthesized model, yielded a high number of problems in the original requirements and architecture models. Tool-aided iterations between formal models in MSC and executable SDL specifications resulted in considerable speed-up. Automatic derivation of test cases from the synthesized SDL models is a promising future research direction.

The Moscow Synthesizer Tool described in this paper was used for re-engineering of legacy telecommunications software [9]. Probe traces were dynamically collected from a suitably instrumented legacy system. The traces were then converted into MSCs. An SDL model of the system was then obtained by applying the MOST-SDL tool. Detailed description of our dynamic scenario-based approach to re-engineering of legacy is presented in [9].

Our experience with Moscow Synthesizer Tool shows that automatic synthesis of SDL models from scenarios has a good potential in allowing significant reductions of the time-to-market for a “real-world” telecommunications industry. It also offers a very cost-efficient way of introducing MSC and SDL-based formal methodology into an industrial environment.

7. References


