Validation of Communications Systems with SDL

The Art of SDL Simulation and Reachability Analysis

Laurent Doldi
TransMeth Sud-Ouest, France
Validation of Communications Systems with SDL
Validation of Communications Systems with SDL

The Art of SDL Simulation and Reachability Analysis

Laurent Doldi
TransMeth Sud-Ouest, France
To my parents

To Martine
To Elsa
Contents

Preface .......................................................................................... xi
Foreword ...................................................................................... xiii

1 Introduction ............................................................................... 1
  1.1 Validation of Communications Systems .............................. 1
  1.2 SDL, Language to Master Complex Systems Development .... 2
    1.2.1 Overview of SDL ............................................. 2
    1.2.2 Benefits provided by SDL ................................. 3
  1.3 Simulation Life Cycle .................................................... 4
  1.4 Contents of the Book .................................................... 6
  1.5 Tools and Platforms Used ........................................... 7

2 Quick Tutorial on SDL ............................................................ 9
  2.1 Structure of an SDL Model ............................................ 9
    2.1.1 System, block and process ................................ 9
    2.1.2 Scope of declarations........................................ 10
    2.1.3 Process ......................................................... 10
    2.1.4 Procedure ...................................................... 11
  2.2 Communication ........................................................... 11
    2.2.1 Signals .......................................................... 11
    2.2.2 Channel ........................................................ 13
    2.2.3 Signal route ................................................... 13
  2.3 Behavior ..................................................................... 13
    2.3.1 Structure of a transition ................................... 13
    2.3.2 Start .............................................................. 14
    2.3.3 States ............................................................ 15
    2.3.4 Input .............................................................. 15
    2.3.5 Save ............................................................... 16
    2.3.6 Variables ........................................................ 17
    2.3.7 Stop .............................................................. 17
    2.3.8 Task .............................................................. 17
    2.3.9 Create ............................................................ 18
    2.3.10 Output .......................................................... 18
    2.3.11 Decision ....................................................... 19
    2.3.12 Timers .......................................................... 19
  2.4 Data Types .................................................................. 20
7.3.9 Verifying an MSC with bit-state simulation ........................................ 218
7.3.10 Bit-state simulation with observer processes .................................. 220
7.4 Case Study with ObjectGeode .............................................................. 221
  7.4.1 One second to detect missing save of v76frame ................................ 221
  7.4.2 One second to detect missing input L_ReleaseReq .......................... 225
  7.4.3 One second to detect missing input L_DataReq .............................. 227
  7.4.4 Seventeen seconds to explore 87174 global states ......................... 230
  7.4.5 Add faults in block dataLink : detect output to Null .................... 235
  7.4.6 Twenty-two seconds to detect missing save of L_DataReq ............... 240
  7.4.7 Eleven minutes to detect missing input L_ReleaseReq and answer DM ................................................................. 243
  7.4.8 Eleven minutes, 2.8 million states, no error ................................ 248
  7.4.9 Exhaustive simulation with stop conditions ................................ 250
  7.4.10 Exhaustive simulation with MSC observers ................................ 251
  7.4.11 Exhaustive simulation with GOAL observers ................................ 253
7.5 Other Simulation Algorithms ............................................................. 256
  7.5.1 Tau SDL Suite ................................................................................. 256
  7.5.2 ObjectGeode: supertrace ................................................................. 256
  7.5.3 ObjectGeode: liveness ..................................................................... 257
7.6 Strategy to Master Exhaustive Simulation ............................................. 262
  7.6.1 Which simulation modes should be used ......................................... 262
  7.6.2 If simulation never terminates ....................................................... 263
7.7 Errors Detectable by Exhaustive Simulation .......................................... 264
  7.7.1 Errors detected by Tau SDL Suite ................................................. 264
  7.7.2 Errors detected by ObjectGeode ..................................................... 265
8 Other Simulator Features ........................................................................ 267
  8.1 Tau SDL Suite ..................................................................................... 267
    8.1.1 Writing in the Simulator trace ..................................................... 267
    8.1.2 Calling external C code ................................................................. 267
    8.1.3 Simulating ASN.1 data types ..................................................... 270
    8.1.4 Adding buttons to the Simulator ................................................ 270
    8.1.5 Adding buttons to the Validator ................................................ 272
    8.1.6 Setting breakpoints in the Simulator ........................................... 272
    8.1.7 Running several communicating Simulators ................................ 273
    8.1.8 Real-time simulation ................................................................... 275
    8.1.9 List of Validator options ............................................................. 275
  8.2 ObjectGeode ....................................................................................... 279
    8.2.1 Writing in the Simulator trace ..................................................... 279
    8.2.2 Calling external C code ................................................................. 279
    8.2.3 Simulating ASN.1 data types ..................................................... 281
    8.2.4 Adding buttons to the Simulator ................................................ 281
    8.2.5 Simulation scheduling like in Tau SDL Simulator and Validator .... 282
    8.2.6 List of Simulator settings ............................................................. 284
Bibliography ............................................................................................. 289
Index ......................................................................................................... 293
Preface

This book is the second in a series; after a first book explaining SDL (Specification and Description Language) and about how to build an SDL model [Doldi01], this book explains how to validate the model by simulation. The two books share the same SDL case study, a simplified version of V.76, a protocol standardized by the ITU (International Telecommunications Union).

I have written this book because, after using ObjectGeode™ and Tau SDL Suite™ (both from Telelogic) for more than ten years, I felt that the numerous powerful features available in such SDL tools needed to be clearly explained to a wide audience, instead of being confined to severe technoweenies.

Readers who want to practice the exercises described in the book must contact the SDL tool vendors (see their updated list on www.sdl-forum.org), who generally provide free licenses for evaluation or cheap licenses for universities.

The first versions of ObjectGeode and Tau SDL Suite including a simulator, named Geode and SDT at that time, have been released around 1989. However, to my knowledge, 13 years after that release, this is the first book published on validation of SDL systems by simulation.

Very few other commercial tools or languages provide such a range of features for the validation and development of communications systems and software.

Some may question the need for this book, as the SDL tools have their own documentation. The answer is that the documentation of each tool, assuming one of your colleagues has not taken it away, contains thousands of pages, which is not always organized to present first the basic simulation notions and then to introduce progressively more advanced features. In the book, every notion presented is illustrated by a hands-on systematic example, which has been actually executed on the two simulation tools, with direct explanations.

Although this book describes how to validate telecommunication systems, it can be used to validate the behavior of other kinds of real-time systems that can be modeled by communicating state machines.

I hope that this book will reveal to students or managers the power of SDL simulation, and will help designers and developers in the validation of their SDL models.

Laurent Doldi
Foreword

‘Better’, ‘faster’ and ‘cheaper’ are the master words nowadays: how to build the best product, spend less and finish in time. Every project manager knows this triptych: every time she or he starts a new project, it could turn into a nightmare. CMM-I, Six Sigma, COCOMO II, MDA, MDD, XML and others are answers that have resulted from different industries involving the development of complex systems.

Modeling techniques have had significant quality and productivity impact in domains ranging from business processes to embedded real-time applications. The Unified Modeling Language (UML), Model Driven Architecture (MDA), Component Based Development (CBD), Use Case Maps, Message Sequence Charts (MSCs), and Specification and Description Language (SDL) all support modeling concepts that help reduce the impedance mismatch between models of the problem domain and designs in the solution domain.

Programming languages are no longer the necessary and sufficient condition to success. All effort is put in product development to better manage the process. Requirement management, system and software architecture, and model development are part of the artifacts of a good system or software development process.

Simulation, one of the most acclaimed requirements to UML 2.0, helps software engineers who simulate the software architecture as well as its design for a better verification and validation of what is being built. Simulation can only be based on a formal language with a clearly defined syntax and semantics. Formal abstract languages such as SDL and the upcoming UML 2.0, are the answers for modelers with concerns such as verifying architecture models and design models.

Why simulate? What to simulate? How to simulate Answers to these questions are found in this book, which is a result of the vast experience Laurent Doldi has acquired in the more than two decades during which he was involved as a consultant engineer, in complex system engineering, in tool development and in teaching classes. Whether your concern is verification or validation, you will find in this book a systematic and practical approach aimed at engineers. It will guide you through the use of tools to perform simulation, architecture or design debugging by getting coverage of your requirements expressed as test cases using MSCs, meet your quality expectations and get a faster return on investment not only for the tools but also during product development.

Jamel Marzouki
Distinguished Member of the Technical Staff, Motorola Labs, Schaumburg, IL
1

Introduction

1.1 VALIDATION OF COMMUNICATIONS SYSTEMS

Communications systems and software are more and more difficult to develop: they include complex features such as wireless and mobile access, under strong constraints such as low size, weight or power consumption, interworking, total interoperability, security, short time-to-market and low cost.

Respecting such an array of constraints requires a high quality of the specifications or standards used for their development. This is why the specifications of many communications systems are based on SDL (Specification and Description Language) or at least contain SDL parts describing complex behaviors. Examples of such systems are the GSM second-generation mobile telephony system, the UMTS third-generation mobile telephony system, the ETSI HiperLAN 2 Broadband Radio Access Network or the IEEE 802.11 wireless Ethernet local area network.

Validation of such systems by simulation of SDL models is useful, for example, at the following stages:

• when standards are created by the organizations, to check that the behavior of the system is correct, to generate Message Sequence Charts (MSCs) (sequence diagrams) illustrating typical use cases, or to generate TTCN (Tree and Tabular Combined Notation) test cases to test the conformance of future implementations;

• before the implementation of a standard by a company, because standards rarely contain a finished SDL model ready to be translated into the application code;

• to provide nonambiguous low-error specifications to a contractor, enabling a quicker and less expensive implementation;

• after changes in the specifications, to check that the system has not regressed.

During all these stages, the simulation allows the detection of specification or design-level anomalies, preventing them to be embedded in the implementation. Once the code is loaded into a target device embedded into a complex test environment, each error detected is more difficult and expensive to analyze than during an SDL model simulation: the error can come not only from the specification but also from the coding, from the testing environment, from the hardware and so on.

Also, SDL simulation enables the execution of the specification before the target hardware and software platform is available: board, board support package, compiler and so on.
The SDL Simulators, especially in exhaustive mode, quickly find error scenarios far beyond human imagination: bugs that could appear after millions of devices have been sold, revealed by a modification of their environment, can be detected and fixed during the specification or design phase.

Concerning safety-critical systems such as fault-tolerant aircraft systems architectures, medical or car devices, the validation by simulation can prove formally that their specified behavior is correct, according to a set of criteria.

1.2 SDL, LANGUAGE TO MASTER COMPLEX SYSTEMS DEVELOPMENT

1.2.1 Overview of SDL

SDL stands for Specification and Description Language. It is standardized by the ITU (International Telecommunication Union) in the Z.100 Recommendation [SDL00, SDL99].

In SDL-92\(^1\), the architecture is modeled as a system containing blocks, as depicted in Figure 1.1. Each block may contain either blocks or processes. Each process contains an extended finite state machine. State machines communicate by exchanging signals through channels (or signal routes).

![Figure 1.1 Schematic view of an SDL-92 description](image)

Signals (\textit{sig1} or \textit{sig2} in Figure 1.1) arriving on a state machine are queued. By consuming a signal from its queue, a state machine executes a transition from one state to another state. During the execution of a transition, a wide range of actions can be performed by a state machine: signal transmission to another state machine, assignment, procedure or operator call, loop, process instance creation and so on. The execution semantics of SDL is accurately described and includes the semantics of actions in state machine transitions.

Data types are described using predefined types or constructs such as Integer, Boolean, Character, struct, Array, String, Charstring, Powerset. ASN.1 can be used in an SDL model.

\(^1\) SDL-92 means the 1992 version of SDL plus the corrections introduced in Addendum 1 to Recommendation Z.100 of 1996, sometimes called SDL-96.
to describe more complex data types, using constructs such as choice (similar to union in C),
optional fields, Bitstring or Octetstring and providing standardized encoding rules.

SDL is object-oriented: it provides the notions of classes, inheritance, polymorphism (in
SDL-2000) and so on found in object-oriented programming languages.

SDL is frequently used with MSCs, similar to UML (Unified Modeling Language) Sequence
Diagrams.

1.2.2 Benefits provided by SDL

SDL being a graphical language enables you to visually design models, instead of using only a
textual notation. SDL provides graphical structuring features (blocks, etc.), state machines and
communication through signals that are not available in programming languages such as C++
or Java.

During the modeling process and after its completion, and because SDL has a complete
semantics, the SDL description can be rapidly checked and debugged using the powerful
tools available today (see Figure 1.2), namely the compilers and simulators, enabling very
fast model correction.

![Figure 1.2 Life of an SDL description](image)

Bugs are found and corrected before the implementation begins. SDL simulators provide high
caliber debugging features, from symbol by symbol stepping to automatic simulations using
various strategies (random, exhaustive, bit-state, supertrace etc.) coupled to automatic error
detection by observers. Simulation scripts allow automatic nonregression testing of the SDL
description in a few seconds by automatic replay of scenarios, with observers on-line checking
the SDL behavior. Simulators generate MSCs representing a visual trace of the simulation.

After testing the SDL description, code generators can autocode it: it is not necessary to write
a single line of code to get the application running, except when communicating with non-SDL
parts or optimizing performance if severe constraints exist. Just by pressing a button, a code
generator produces, without manual coding errors, one or several binaries running on one or
several computers or boards, without executive or under Unix, win32, Posix™, VxWorks™,
VRTX™, Chorus™, PSOS™ and so on. Execution on the target system produces a visual trace
in the form of MSC sequence diagrams.

Also, test cases in TTCN or in another test language can be generated by very sophisticated
tools, ranging from transformation of an interactive simulation scenario into TTCN to automatic
generation of TTCN test cases covering all SDL symbols or automatic generation of TTCN
test cases corresponding to user-defined test purposes.
1.3 SIMULATION LIFE CYCLE

The life cycle of simulation can be split into three steps, as illustrated in Figure 1.3:

1. Production of an SDL model ready for simulation, starting either from a textual specification, as in the V.76 case study presented in the book, or from an SDL model, for example coming from a protocol standard, or from legacy code that must be reverse-engineered because no useful specification documents exist;

2. Interactive simulation, offering a good level of validation and automatic nonregression testing;

3. Exhaustive simulation, the top level in validation, reserved for safety-critical or cost-critical systems. Cost-critical means that leaving errors in the system would be very expensive, because its code will be embedded into millions of devices.

Simulation produces a low-default executable specification, plus reference MSCs (sequence diagrams) that provide an excellent documentation and which can be used as test cases to test the system implementation: as they have been generated by simulation, they are consistent with the validated SDL model.

![Figure 1.3 The three simulation steps](image)

Step 1 is shown in Figure 1.4: if an SDL model exists for the system, it must be compiled. If there are errors, the SDL model must be corrected. Some SDL models may need to be completed, for example to add missing data type declarations. If no SDL model exists, a decision must be taken to use or not to use SDL: if the system is not complex or not safety-critical, such an investment is not necessary. Beware of systems that seem to be simple but are not, in terms of behavior. Then, if the system (or a part of it) cannot be modeled using extended finite state machines communicating with signals through queues (a kind of mailbox), use another language, such as a synchronous language. Otherwise, continue with Step 2.

Step 2 is shown in Figure 1.5: first, the main scenarios (the use cases) are simulated step by step. Each scenario is stored (files .scn, .com or .cui) to be replayed automatically later. Each MSC trace is also stored. When an error is found, the SDL model (or an specification if it is wrong) is corrected.

To detect automatically when the SDL model is wrong or when it is correct, observers can be created. Then the SDL model is simulated together with its observers, replaying the scenarios stored previously, to check that the observers work correctly. Again, when an error is found, the SDL model is corrected.

To replay scenarios automatically, scripts can be written: after each model correction or evolution, all the test scenarios are automatically replayed in a few seconds and the simulator reports any error or any violation detected by the observers.
Simulation detects SDL symbols never executed: more scenarios must be created to try to cover them.

If the system is not safety- or cost-critical, this level of simulation is sufficient. Otherwise, continue with Step 3.

Step 3 is shown in Figure 1.6: exhaustive (or bit-state) simulation is run. When an error or an observer violation is found, the SDL model (or an observer) is corrected. The scenarios
generated by the simulation must be kept, together with the MSC traces of the scenarios that satisfy the observers.

If all the reachable states of the SDL model have been explored, the simulation is finished: the SDL model correctness has been formally proved (according to the configuration used).

Otherwise, because the large number of global states prevents exploring them, the model configuration must be reduced, for example, by allowing one or two signals only per process input queue, limiting the exploration depth and so on. If the configuration cannot be reduced, the simulation is finished, and the validation is partial because some global states remain unexplored.

The simulator also detects SDL symbols never simulated: they indicate SDL transitions or branches that could be removed, or reveal missing test signals or test values to be transmitted to the SDL model by the simulator.

All these steps are detailed in the book, illustrated by numerous hands-on exercises.

1.4 CONTENTS OF THE BOOK

This book is divided into eight chapters. Chapter 1 is the present introduction. Chapter 2 is a quick tutorial on SDL-92, the language used for the exercises in the rest of the book. Chapter 3 contains the simplified version of the V.76 protocol specification, some analysis MSCs and the corresponding SDL model used during the simulation exercises. Chapter 4 explains how to validate the V.76 SDL model using interactive simulation. Chapter 5 introduces observers, what they can detect, how to build them and how to use them during interactive simulation. Chapter 6 describes random simulation. Chapter 7 presents exhaustive simulation, how to use it with observers and other simulation algorithms such as bit-state or liveness. Chapter 8 illustrates other simulator features such as calling external C code or adding buttons to the simulators.
Each chapter contains hands-on exercises with solutions for the two main SDL tools commercially available: ObjectGeode and Tau SDL Suite, both from Telelogic.

1.5 TOOLS AND PLATFORMS USED

The exercises of the book have been developed using the following commercial off-the-shelf tools, both developed by Telelogic:\(^2\):

- ObjectGeode Version 4.2 for Windows
- Tau SDL Suite Version 4.0 for Windows

The Unix versions of these tools are very similar to their Windows version. The main difference is the way they are launched: it is generally performed by double clicking on an SDL file in Windows and by typing a command in Unix.

The V.76 SDL model used in the book and its associated files can be downloaded in ObjectGeode and Tau SDL Suite formats on ftp://ftp.wiley.co.uk/pub/books/ldoldi/.

---

\(^2\) ObjectGeode has been developed by Verilog, with France Telecom R&D (former CNET) know-how. Then, in 1999, Telelogic has acquired Verilog.
Quick Tutorial on SDL

For readers not familiar with SDL, this chapter presents the most frequently used constructs of SDL-92. A detailed step-by-step tutorial on SDL-92 [SDL92] and a presentation of SDL-2000 are provided in [Doldi01].

2.1 STRUCTURE OF AN SDL MODEL

2.1.1 System, block and process

SDL provides the following entities to structure a description:

- system: top level, outermost construct;
- block: must be contained in the system or in a block;
- process: must be contained in a block;
- service (optional): must be contained in a process;
- procedure (optional): can be placed anywhere.

Figure 2.1 shows an example of a system containing three blocks.

Figure 2.2 shows that block1 contains two blocks.

Figure 2.3 represents the contents of block1 and block3. Block1 contains one process, and block3 contains two processes. The state machines contained in processes (or services) instances communicate together or with the environment by transmitting and receiving signals (or remote variables or procedures) through channels and signal routes.
2.1.2 Scope of declarations

A declaration is used to define signals, data types, variables and so on. In SDL, a declaration is visible in the current entity and its children. For example, a signal declared in the system is visible in the whole system. Variables cannot be declared in a block or in a system; therefore global variables do not exist in SDL.

2.1.3 Process

A process, if it does not contain any service, contains a state machine. Figure 2.4 shows a process in which the variable V76par has been declared. A variable is local to a process and thus is not visible, for example, from another process. Our process contains only one state, ready.

Each process has one or more instances, running in parallel, and independent. Two numbers are used to specify the number of instances for a process:

- the first number indicates the initial number of instances (when the system is started),
- the second number indicates the maximum number of instances running at a certain moment.

Figure 2.5 depicts the two main possibilities for the number of instances. Each process instance contains four implicit variables:

- self: contains the Pid of the current instance,
quick tutorial on SDL

1.1 Process

2.1.4 Procedure

2.2 Communication

2.2.1 Signals
Once declared, a signal can be inserted into the “wires” (channels and routes). The signal name can be directly inserted into the [ ] symbols of the channel or signal route, or the signal name can be inserted into a signal list (like in the channel DLCaSU on the left part of Figure 2.7).

The “wires” that you see in SDL descriptions are either channels or signal routes: if a “wire” is connected to a process, it is a signal route, otherwise it is a channel.
2.2.2 Channel

A channel can carry signals either in one or in both directions. Figure 2.8 shows channels ch1 and ch2 carrying signals to block_b_1, while channel ch3 carries signals from block_b_1 to outside. Channel ch4 carries signals both in and out of block_b_1.

By default, a channel contains a FIFO (First In First Out) queue used to delay the signals.¹

![Figure 2.8](image1)

**Figure 2.8** Channels in block DLCb

2.2.3 Signal route

The difference between channels and signal routes is that signal routes do not delay signals. Figure 2.9 shows an example of four signal routes.

![Figure 2.9](image2)

**Figure 2.9** Signal routes in block_b_1

2.3 BEHAVIOR

2.3.1 Structure of a transition

As depicted in Figure 2.10, a transition in a process begins with a state, immediately followed by an input or a spontaneous transition, or a priority input, or a continuous signal or a save. No other symbol is allowed after a state.

¹ When using an SDL editor, an option allows you to specify if you want the channel to delay signals or not. In ObjectGeode and Tau SDL Suite, by default, channels do not delay signals.
2.3.2 Start

Every process must contain exactly one start symbol. When a process instance is created, the first transition ready to be executed is the transition beginning from the start symbol.