

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-92¹, 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).

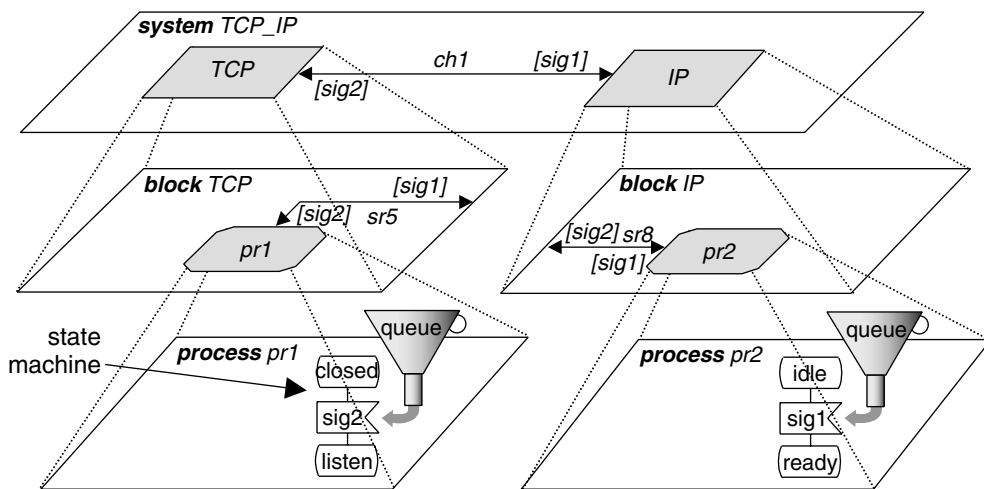


Figure 1.1 Schematic view of an SDL-92 description

Signals (*sig1* or *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.

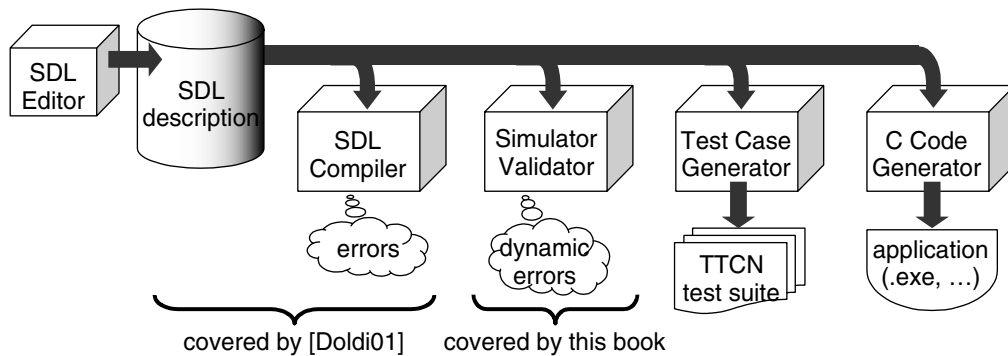


Figure 1.2 Life of an SDL description

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.

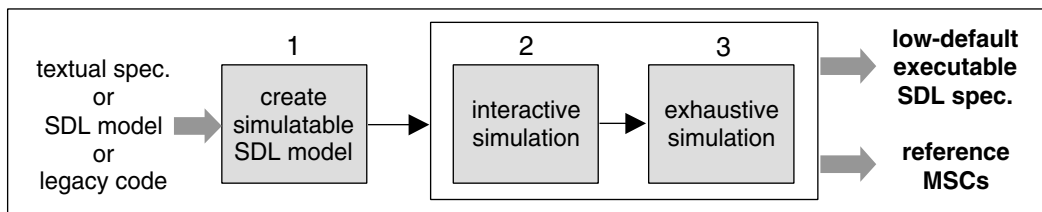


Figure 1.3 The three simulation steps

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.

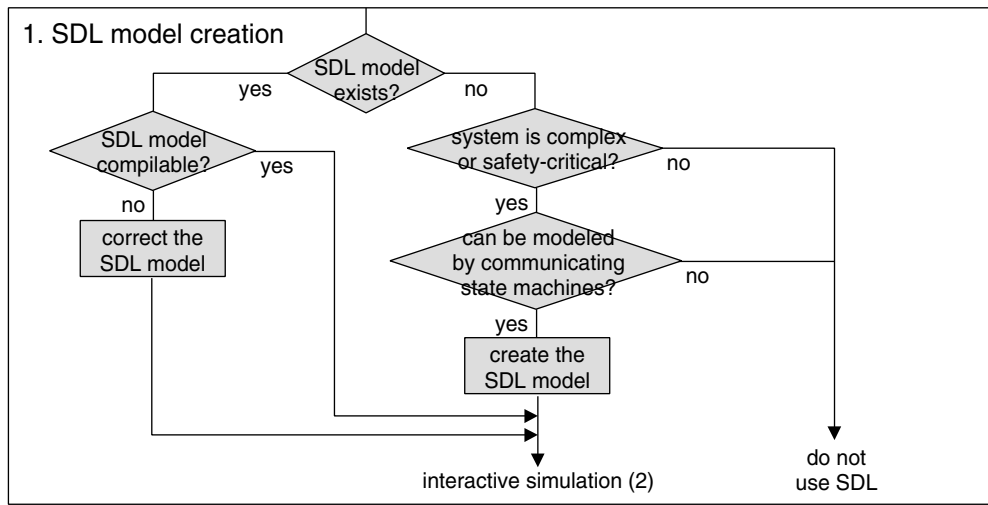


Figure 1.4 Step 1: SDL model creation

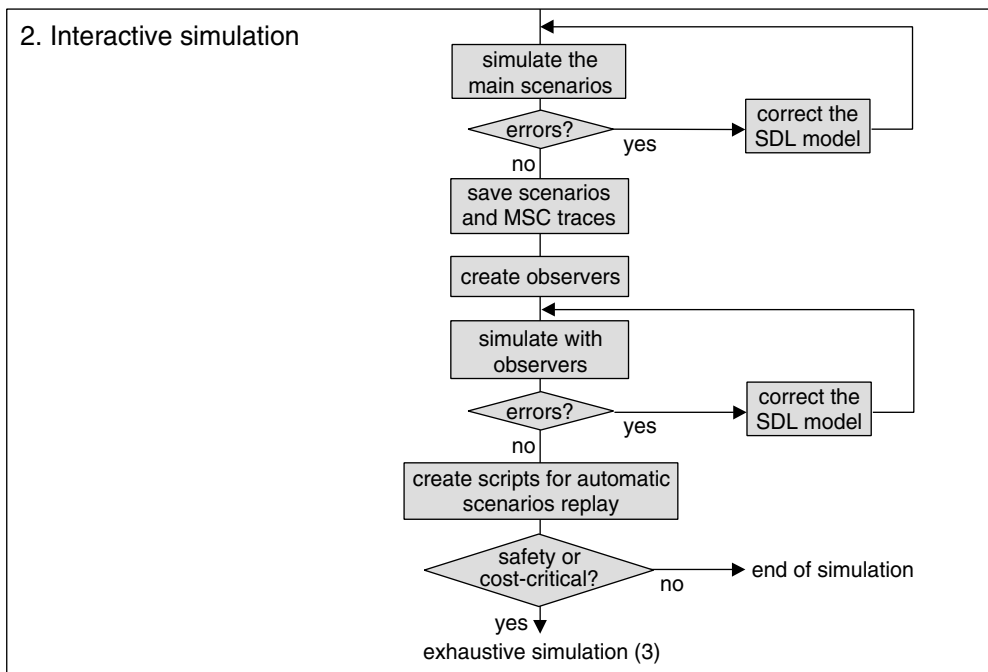


Figure 1.5 Step 2: interactive simulation

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

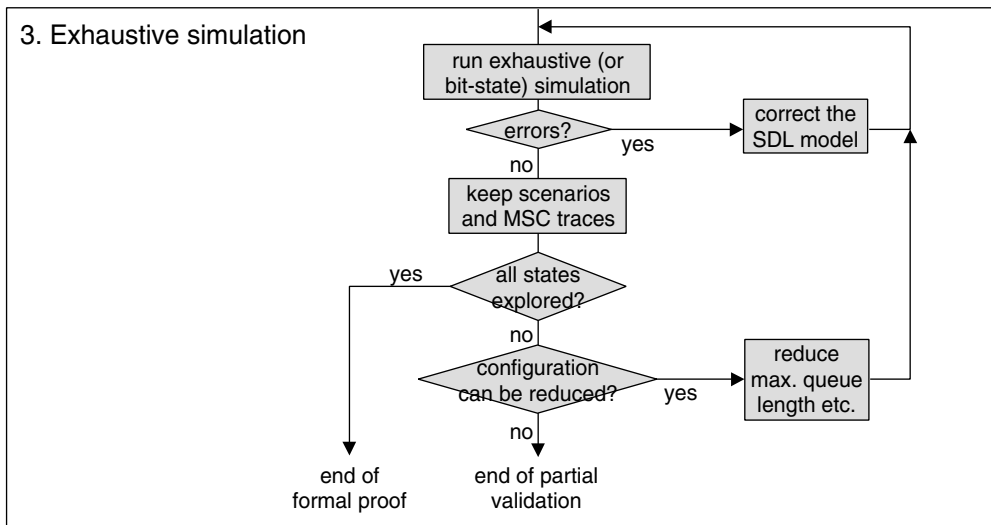


Figure 1.6 Step 3: exhaustive simulation

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 Telelogic²:

- 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.

