# Solving problems of SDL and UML for real-time software design

**by Emmanuel Gaudin,** PragmaDev

THE USE OF SDL AND UML FOR THE DESIGN OF REAL-TIME SOFTWARE IS ACCOMPANIED BY DIFFERENT PROBLEMS. SDL-RT BENEFITS FROM THE MATURITY OF C AND SDL BUT ALSO FROM THE VISIBILITY OF UML SO THAT AN SDL-RT-BASED TOOL CAN SOLVE THESE PROBLEMS.

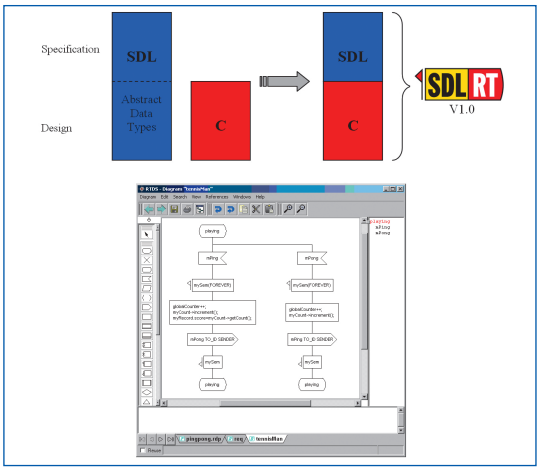


*Figure 1. SDL-RT offers graphical representation for sending and receiving messages, as well as taking and freeing semaphores*

SDL, the "Specification and Description Language", has been defined by the International Telecommunication Union (ITU) in order to describe telecommunication protocols. Experience has shown its basic principles could be extended to a lot of other fields mainly because of its graphical functional approach allowing to describe architecture and behaviour of the system. But when it comes to implementation and integration on target, the SDL data types and some of its high level concepts turn into a problem for the developers because they lose control of the final target code. Furthermore, some real-time aspects such as semaphores and pointers, were not covered in the language so most of the SDL developers started to write C inside the SDL graphical representations. Because there were no tools supporting this combination, they had to write their own C code generator out of this mix of SDL and C. All these problems prevented SDL from being widely spread. That is what SDL-RT is solving while it also includes some of the UML trend.

Real-time and embedded software are based on real-time operating systems or schedulers in which the basic structural element is the task. Several tasks execute concurrently in order to fulfil a basic function and these basic functions are gathered to realize more complex functions and so on up to the whole application. It appeared the SDL architecture gathering tasks in functional blocks that could themselves be gathered in higher level blocks was a good way to graphically represent the architecture of any real-time system. The architecture is then completed with the description of the interfaces between the different functions of the system. An interface is defined by an exchange format based on structured data, and a scenario describing the sequence of exchanges. SDL signals come with typed parameters based on abstract data types (ADT) allowing a full description of the static interface. Message sequence charts (MSC) standardized within the same ITU series provide a graphical representation of the dynamic aspect of the information exchanged within the system or with external modules such as drivers.

Real-time systems are based on independent tasks running concurrently, so it is very important not to waste CPU time whenever the task has nothing to do. That led most real-time applications to be based on finite state machines where the basic principle is to wait on an RTOS object such as a message queue as soon as the task has finished its job. Once more, the SDL finite state machines were well-suited to describe that kind of behaviour graphically. Last but not least, SDL is object-oriented at all graphical levels since its '92 version so it is possible to build up libraries of software components that can be specialized.

On paper SDL appears to be an optimized language for specification and design of real-time systems, but the technical reality is quite different. When it comes to design, SDL abstract data types (ADT) become a hurdle for several reasons. The syntax to manipulate ADTs has been defined to specify protocols, not to design them. Software designers get into trouble not having the precision they used to have with traditional programming languages such as C. There are no SDL compilers or cross-compilers on the market so C or C++ code generation is necessary to implement the SDL system on target. ADTs are based on concepts that cannot be directly translated in C or C++ so specific operators must be generated making the generated code unreadable. Integration of legacy code is difficult because a bridge needs to be setup between the SDL data types and the C or C++ data types.

When it comes to integrating the code generated out of an SDL system onto an RTOS, some SDL semantic aspects are not directly supported. Let's take two simple examples. The SDL priority concept applies on messages while RTOS priority concept applies on tasks. SDL semantic considers that an automaton transition can-not be interrupted while an RTOS could actually interrupt execution at any time, especially if system calls occur during the transition involving tasks with different priority levels. In

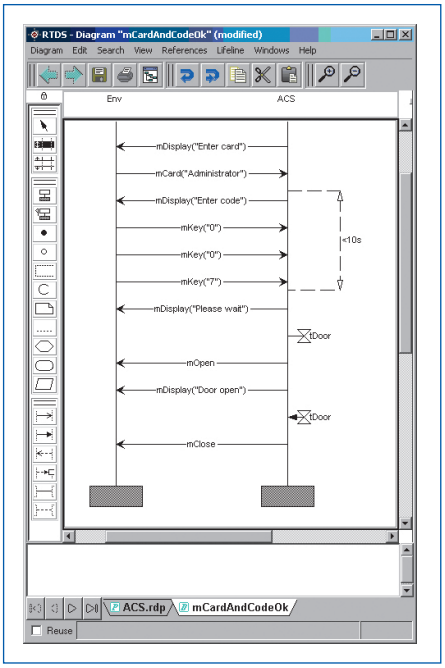


*Figure 2. MSC specification example*

order to guarantee that the generated code behaves as specified, an SDL virtual machine must be generated, making integration on target very tricky and access to some RTOS services almost impossible. The concept of semaphore does not exist in SDL while it is one of the most classical synchronization mechanisms in real-time applications. This is due to the fact that SDL has been defined to specify protocols to communicate with remote entities so the concept of semaphores could not be used.

For all the above reasons and in order to implement systems described in SDL, tools became extremely complex to manipulate, very expensive, and required long training sessions as well as highly skilled consultants to get the system to work on the target. In the end all the benefits of using a graphical language were lost during integration on target. The main reason for that situation was that tool vendors wanted to abide by the standard whatever the technical problems implied.

In the field, SDL users started to write C code inside their SDL diagrams and broke up the SDL semantic to fit technical reality. But since this mix of C and SDL was not supported by the tools on the market (because not conforming to the official standard) these very same users developed their own tool chains. The result is that every major telecommunication manufacturer has its own internal C code generator out of this mix of SDL and C, namely Alcatel, Nokia, Nortel, EADS and Sagem. The first SDL-RT release aimed at formalizing the industrial habits of mixing C code within SDL graphical representations. In order to extend the usage of such a language to all real-time applications, SDL-RT also introduced graphical symbols to handle semaphores.

The Unified Modelling Language (UML) – as its name states – is a merger of several representations. Its main objective was to document application and to ease communication among all the people involved in a software development. Since its first release in 1997, with the help of strong marketing, UML has spread quite quickly in a large number of areas. This language is characterized by its full object-oriented approach where any element of any system is basically described as a class. This generic approach provides a very high level of abstraction so that it can be applied to any concept. On the other hand UML is not dedicated to any application domain and is not precise enough to describe applications in detail. As a matter of fact, it is mostly used during the early analysis or specification phase. The weak link between the UML diagrams and the design raises two recurring problems: synchronization between the UML documentation and the final code, and a lack of a structuring framework to drive the development process. These problems could actually explain the success of UML by the fact that development teams could actually still work as they used to, and document independently with this language. The benefits of such a usage are very limited and that is why some large companies have decided to stop making UML mandatory.

In the real-time domain only the class diagram, the sequence diagram, and the state chart are used among the nine diagrams available in the version 1.x of the standard. Since UML tools are basically drawing tools with very limited code generation possibilities, they are much more affordable than the ones based on SDL. That also helped the technology to spread.

Several experiences of intensive usage of the language within large development teams eventually ended in to disaster. From the fact that the tools were generating an oversized code with low performance and no debug on target facilities, tools and methods managers raised a major question: is a full object-oriented approach applicable in the real-time and embedded domains where the traditional way is functional? Difficult to give a straight answer to such a question, but it appears that successful object-oriented developments have initially started with a functional approach and then factored in components organized in libraries in order to be re-used later. The UML marketing strength was so strong that SDL had to reposition itself as a graphical modelling language. The last SDL version, SDL'2000, integrated the UML class diagram and paved the way to merge with the next major version of UML under discussion at the time: UML 2.0.

The main objective of this new version of UML is to support the MDA (model driven architecture) approach in order to define specific profiles for each application domain. It is already possible to evaluate the status of these languages.

On the SDL side, despite the initial commitment of the different tool vendors, the latest version of the language has not been implemented in any tool. Generally speaking, the move towards UML – meaning less precision – did not seduce SDL users because of its complexity. The SDL Forum has actually initiated a task force to define a sub-set of SDL to simplify the language, and is – on the other hand – working at defining a UML profile based on SDL. In the field, SDL users are still working with the '96 version of the language.

Version 2.0 of SDL-RT has integrated the class diagram and the deployment diagram from UML that complement the SDL representations while keeping consistency between the different views. The (still under validation) version 2 of UML has integrated existing features from SDL such as the block diagram into the structural diagram and the MSC into the sequence diagram. It is also possible to define domain-specific profiles but unfortunately there is no standard real-time profile defined. Major industrial companies actually doubt such a profile will ever be standardized because of the conflict of interests among the different editors who are voting at the OMG.



Nevertheless the first UML 2 tools are already available on the market but the initial objective of the new standard to ease portability and readability is missed since each editor has defined its own proprietary profile. These profiles are not public and tools rarely offer a profile ed-

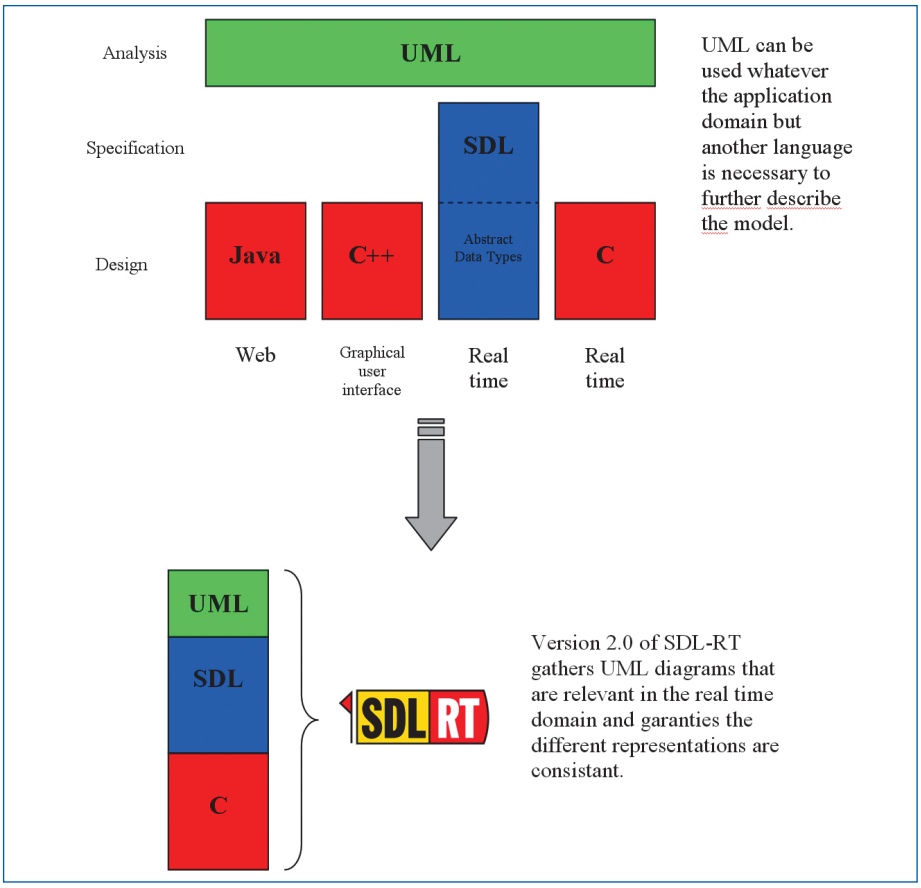


*Figure 3. Version 2.0 of SDL-RT gathers UML diagrams and ensures consistence*

itor, so in the end there are as many versions of UML 2 as there are tool vendors on the market. Users will definitely lose from that situation making readability and portability more difficult than before.

Whether the tools are based on SDL or UML, editors had a prophetic attitude regarding their language and spent a lot of energy to demonstrate through seminars and advertisements they were right. It worked out so well that it created technical fanatics in the industry, so dazzled by the prophecy that they would not hear about any other technology. Technical reality brought them back to reason, but at what cost? More thoughtful, such as SDL-RT, are proposing pragmatic solutions inspired by the usage in the industry and wishes from the final users without ignoring the current trends. That is why they gather some UML diagrams, most of the SDL diagrams, and the classical programming languages such as C and C++ within a consistent framework. The objective is to offer all the benefits of the different languages for the real-time and embedded domain without their drawbacks. The diagrams are stored in a public XML format in order to be able to re-use the information without any specialised tool. Apart from the language it is, of course, the quality of the tool that will convince the end users since the objective of a development team is not to conform to a standard but to efficiently produce software with the best level of quality and performance. ■