Verification of model based design

Emmanuel Gaudin has a technical background and developed protocol stacks in SDL. He joined a modeling tool vendor in 96 as a Field Application Engineer and as a trainer to finally become technical director of the French branch. Based on that experience, he started PragmaDev in 2001 to develop an SDL-RT tool.

This paper will remind the basics of model driven engineering and how to determine if a modeling language is a good candidate. Based on that analysis, a formal verification technique will be explained and illustrated with an SDL model.

Model Driven development

Model Driven Engineering is an approach of software development based on abstract models of a system to be developed. Three types of models are used in a model driven approach:

- an abstract model of the system under development called the Platform Independent Model (PIM),
- a Platform Definition Model (PDM),
- an implementable model of the system called the Platform Specific Model (PSM).

The PIM is basically the system under development and the PDM defines the rules in order to transform the PIM into a PSM. In practice the development team works on the PIM, the PDM is defined by the application domain or the company, and the PSM is automatically generated out of the PIM and the PDM.

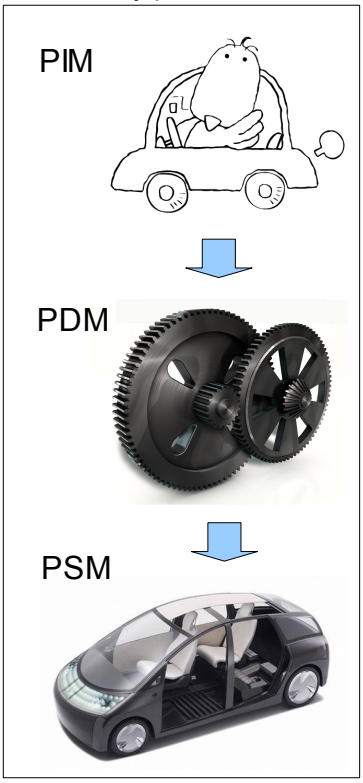
For that process to be efficient, the PIM must be abstract enough to be independent from the platform on which the system will be implemented, but at the same time it should be precise enough to be translated to a PSM. So, in order to be able to successfully translate the model, the PIM relies on a virtual machine which characteristics are:

- a number of basic services,
- strong enough semantics to be expressive.

SDL: model driven from the start

In the 80's, the International Telecommunication Union (ITU) has standardized a language to describe telecommunication protocols: the Specification and Description Language (SDL) under reference Z.100. The main goal was to describe the protocols in an unambiguous way so that all manufacturer's implementation of a standard protocol are compatible with each other. European Telecommunication Standardization Institute has extensively used SDL to describe telecommunication standards and it is obvious to state the compatibility issue has been successfully achieved. Most of the Telecommunication manufacturers have used SDL to design their software and have measured it increased quality by a ratio of 5 and reduced the overall development time by 35% in average.

Technically speaking, SDL is an abstract, event driven, object oriented, graphical language, with strong semantics of execution, and embedded Abstract Data Types. Because it embeds data types and a syntax to manipulate them, SDL models are formal (complete and non-ambiguous). An SDL model can be fully described because of that characteristic but it does not have to be. So depending on the level of precision within the model, an SDL system can be informal or very precise.



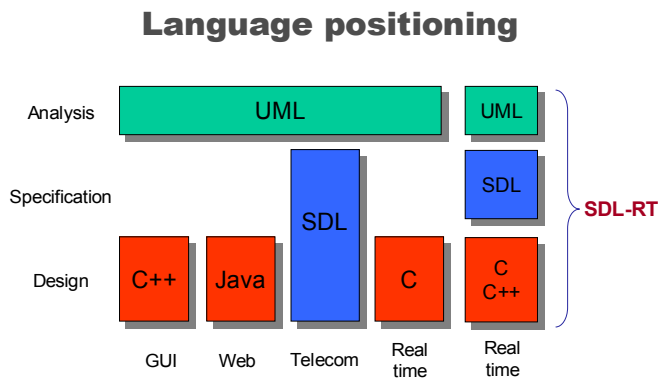
SDL built-in concepts and services such as processes, messages, timers, and procedures are supported by most of the Real Time Operating Systems making implementation on a real target straightforward. The strong semantics of SDL and its built-in services describe an SDL virtual machine on which a model is based on. That is actually the main characteristic of Platform Independent Model (PIM). The definition of possible external operators, and the implementation of the SDL services provided by the SDL virtual machine are the actual definition of the platform: the Platform Definition Model (PDM). From the SDL PIM and PDM, it is possible to fully generate the Platform Specific Model (PSM) in an executable language such as C code.

UML, a too generic modeling approach

In 1997, the Object Management Group (OMG) standardized the Unified Modeling Language (UML), a merge of different object oriented modeling approaches coming from the database application domain. Versions 1.x of UML were too generic to support a Platform Independent Model, so version 2 of the language introduced the concept of profiles to make UML more precise within an application domain. A profile allows to introduce specialized concepts and some semantics within a UML model. At the time, the OMG did not standardized any profile, so UML 2 tools have introduced their own profile -most of the time without documenting it- making the models tied to the tools they have been designed with, and tied to the underlying profile that was used.

The ITU has taken this opportunity to standardize in July

2007 a UML profile for Telecommunication systems based on SDL under the Z.109 reference.



SDL-RT from practice to standardization

Because UML is very abstract and informal, it is mostly used in the early phases of the development process when analyzing and setting the requirements on the system. When it comes to coding, traditional textual languages are at the same level as the SDL Abstract Data Types. Because of its graphical abstractions dedicated to Telecommunication systems, SDL is positioned between the very generic UML and the very specialized coding languages.

In practice, when using SDL, telecommunication manufacturers were aiming at generating application on target so that had a very pragmatic way of using it: they wrote C code manipulating C data types instead of the SDL data and syntax. When UML became popular they started to mix the diagrams all together. SDL-RT came from these industrial practices to combine UML, SDL, and C or C++. It also introduced the semaphore concept so that each service of a Real Time Operating System has a dedicated graphical symbol.

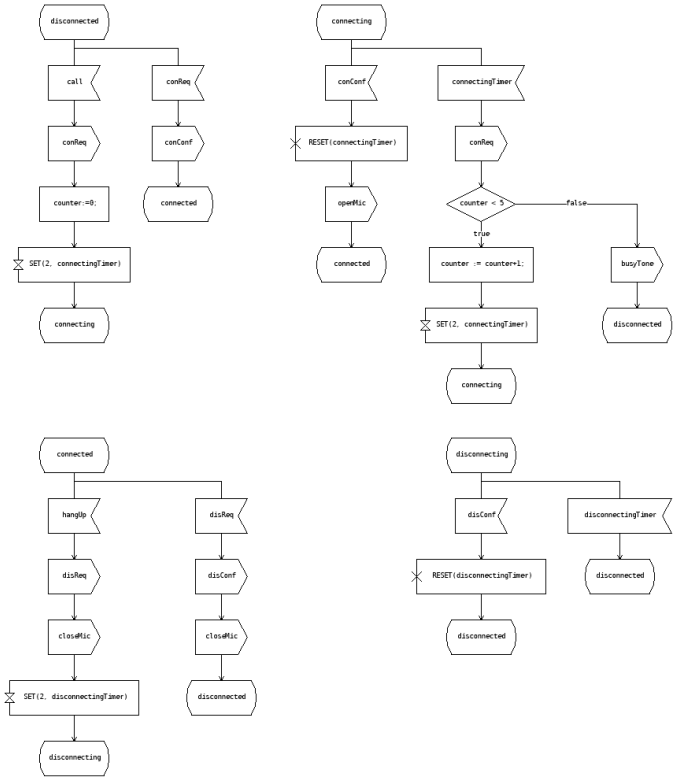
SDL-RT can be seen as a UML 2 profile dedicated to embedded and real time systems and it has all the required characteristics of a PIM.

Principles of exhaustive simulation

One of the key interest of having a model based on strong semantics is the possibility to execute it independently from a real target. Based on the interface of the system, it is possible to try all possible inputs. Once all inputs have been tried in all possible order with all possible values, that means all possible cases have been tested. This is called exhaustive simulation and we will describe the basic principles of this popular model checking technique.

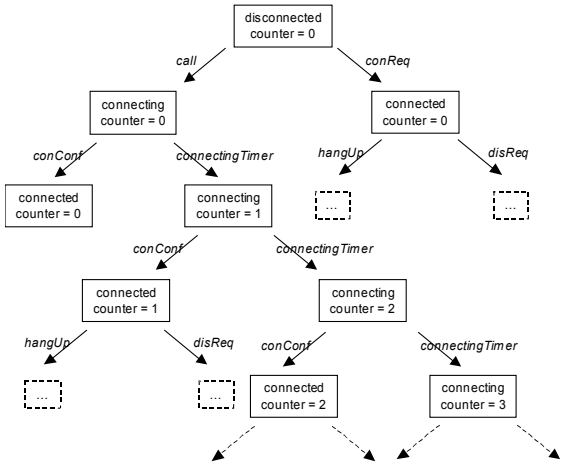
Global system state

It is important to first introduce the concept of global system state: that is a complete picture of the overall system. It combines the states of all finite state machines, the values of their local variables, the values of the object attributes, and the values of all global variables. For a given global system state, a given input will always produce the same result.



In the above simple example the global system state is the state of the finite state machine and the value of the only local variable: counter. The combination of these two values fully describe the system state.

When an input is applied to the system, its global system state will change to a new one. Executing all possible inputs on the system builds what is called a reachability graph or a behavioral tree. One path in the tree is a standard scenario.



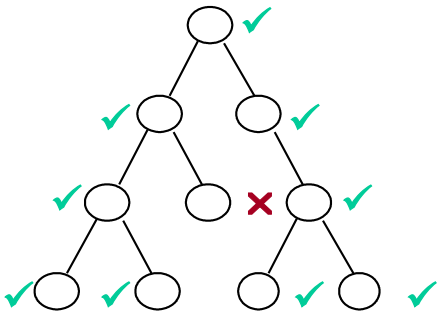
Each incoming event leads the system to a new global system state.

Considering embedded applications are usually multi-threaded, a path from one global system state to the next might be defined by a full transition (from one state

to the next of a finite state machine) or by an atomic instruction (because blocks of instructions can be pre-empted by the RTOS). Of course, if a branch of the reachability graph is an atomic instruction, that makes it much larger than if based on full transition.
It is also important to notice that even a very simple state machine might lead to a very large graph. For example a one state automaton with a simple *int* on 16 bits will generate 2^{16} global system states with full transition branches. That means it will not be possible to explore all possible states on a real system because of the combinational explosion.
In order not to execute several times the same branches, all visited states must be remembered during an exploration. To do so, mathematical techniques are used such as hash tables that compresses the system information to set a bit in a table. Whenever the same bit is set there is a probability related to the size of the table that it is the same state. Another way to reduce the reachability graph is to cut branches in the tree. A simple way to do so for example is to reduce the possible values for some variables (for example instead of trying all possible values for an integer, try 1, 5, 10, and 256). That technique requires to master the system to be checked in order to set the right values.

Observers

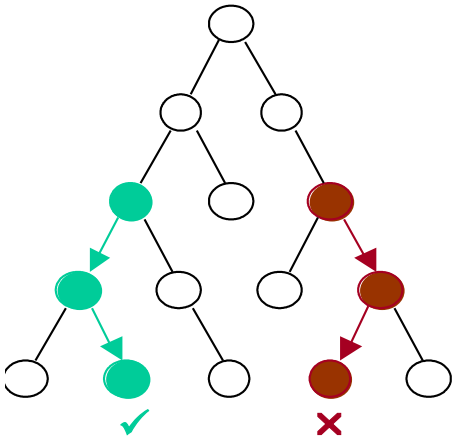
Each time a node in the graph is reached, static and dynamic rules can be verified. A static rule is based on the system state itself, for example the value of a state combined with the value of a variable. Where a dynamic rule is based on the evolution of the system states, so it can be a piece of a scenario: for example it is not possible for one of the state machine in the system to go directly from the disconnected state to the connected state.



Static rules verification

The static rules, the dynamic rules, and the rules to restrict the graph are described in an observer. It is basically an automaton that is evaluated every time a new state is reached. It has access to all internal information (states, variables and others) and decides either one of the rules have been violated, if the branch should be cut, or if the exploration should go on.

The same technique can be used to generate test suites. In that case, the observer defines the test objectives and the exhaustive simulation will find all possible scenarios to get to the objectives.



Dynamic rules verification

Conclusion

Model checking requires the model to be based on a language with strong semantics in order to be able to explore the reachability graph and verify as many rules as possible. In the embedded domain, SDL is a good candidate because it is a formal language with concepts similar to the ones available in real time operating systems.
Exhaustive simulation technologies have been around for quite some time and have been proven to be efficient on real industrial cases. In order to be more widely used, it requires an easy to use tool, especially when it comes to observers, and more over the time and expertise to properly restrict the graph and define the rules.