Code Generation Tool

From SDSL descriptions to C++ implementation code

Microelectronics Group, University of Cantabria

Summary

QRML has been specialized to support SOA in the form of the SDSL DSL. In SOA, components provide services to other components and communicate through well-defined interfaces. SDSL adopts this paradigm and provides monitoring support as well. SDSL adds several keywords to the QRML language: the interface keyword for interface support and monitor, event, and provider keywords for monitoring support. A monitor specification declares the tracer that provides the traces and the event type to monitor.

SDSL models may be used for generating UML-MARTE and C++ templates. We implemented a generator that produces C++ code for the Runtime reconfigurable Implementation of Embedded systems (RIE) library. RIE adheres to the SOA paradigm, providing support for runtime component management and monitoring as well as for runtime system reconfiguration and edge deployment. The runtime reconfiguration allows modifying component functionality, resource allocation or deployment (e.g. selecting an edge component instead of a local implementation).
For every component in an SDSL model, the C++ code generator creates a C++ base class. Component communications are modeled with C++ virtual functions (services) that are grouped in interface classes. For every interface, the generator creates a base interface class and a remote version that is used for edge deployment.

The component class inherits from the RIE library and from interface classes that model provided services. An SDSL component model may have several alternatives with different qualities. A C++ class is generated for each alternative. These alternatives inherit from the base class for the wrapping component and share interfaces with this component (provided and required services). The RIE library provides methods for accessing components and for modifying their set points at runtime. The library supports runtime addition of new component alternatives.

The code generator also generates code for monitors, it has a method corresponding to the declared event. A component generates a monitoring event when calling that function. Once a system is operational, monitored data can be collected and visualized, e.g. FIVIS visualization framework, providing users with a holistic component-based view on the developed system and its runtime operation. It is also supported the opensource Linux LTTng tracer infrastructure.

Installation of SDSL (QRMLaddon)

Use of eclipse-based framework to generate code from SDSL

Starting from TUE QRML project it is possible to extend it, for this purpose it is created an eclipse QRMLaddon to define all the extension proposed with SDSL. This fact makes it possible to extend QRML without modifying it, so QRML will be the core to make different extensions. It is possible to have QRML descriptions '.qrml' or SDSL descriptions '.qrml2' depending on if you want to generate implementation code (SDSL approach) or if you want to use the QRML developed toolset.

Installing QRML and QRMLaddon

1. Extract the qrml_extended.zip file as provided to you into a folder of choice onto your computer.

2. Start Eclipse, close the welcome tab.

3. Choose as workspace an empty folder and click Import Projects... in the left pane and select General=>Existing Projects into Workspace and click Next.

4. Next to 'Select root directory', click Browse... and navigate again to and select the QRML_tooling\Eclipse_workspace folder in the extracted files. Back in the Import window, six projects should show up (three related to qrml and the other three related to qrmlAddon created for language extensions). Click Finish.

5. Eclipse will take some time building the workspace. It shows the progress in the lower right of the window. Wait for it to complete.

6. On the left pane, unfold the project called nl.tue.ele.es.qrml, then unfold the folder src, then the folder nl.tue.ele.es.qrml. Then right-click on the file Qrml.xtext and in the pull-down menu select Run As=>Generate Xtext Artefacts. Eclipse may say that there are errors in the project but select Proceed. Eclipse will be working for a bit. Wait until it says 'Done' in the Console at the bottom.

Repeat the same step with folder nl.tue.ele.es.qrmlAddon.

7. QRML needs to be configured so that it knows where to find and store essential files. Open the file nl.tue.ele.es.qrml\src\nl.tue.ele.es.qrml.generator\QRMLSettingsManual.xtend using the left pane and edit it. Use forward slashes.

- Folder for temporary files: Create an empty folder anywhere in your computer in which QRML can store temporary results. Add the path of this folder to the 'tempdir' entry including a slash at the end and using forward slashes between the folder names.
- External tools: Fill in the full paths for the graphviz and plantuml tools using forward slashes between the folder names. The filled in values serve as an example.
- Run-time directory: QRML needs to have access to the runtime folder named 'runtime-EclipseXtext' that corresponds to the runtime Eclipse. Add the path of this folder to the 'runtimedir' entry.

◦ The remaining entries, e.g., 'imagemagic', are there for future reference and can be ignored for now.

8. SDSL needs to be configured so that it knows where to find RIElibrary (RIEdir) and generation directory (gendir) in which generated code is going to be written. Open the file nl.tue.ele.es.qrmlAddon\xtend-gen\settings.java and specify the RIEdir and gendir variables. Use forward slashes.

Starting QRMLAddon

Go to: “nl.tue.ele.es.qrmlAddon/xtend-gen/nl.tue.ele.es.generator/QRMLAddonGenerator,java” and add the following line inside the “doGenerate” function in case it is empty:

“RIEgeneration.generate(resource);”

Right-click on the project nl.tue.ele.es.qrmlAddon.ide and select Run As=>Eclipse Application. A second instance of Eclipse will now open-up in which the QRMLaddon plugin is activated. This Eclipse is also known as the runtime Eclipse.

Again, close the welcome tab and in the left pane, select Import Projects... and select General=>Existing Projects into Workspace. Click Next and under Select root directory, click Browse... and navigate to the folder QRML_tooling\runtime-EclipseXtext of the repository and select it. One project should show up. Back in the import window, click Finish.

In the runtime Eclipse, an example model (sdsl_example.qrml2) can be opened by unfolding the project nl.tue.ele.es.qrml.runtime. You must create a model following the instructions given on “Getting-started” section.

The model is run as follows: making a minor edit to it (for instance, add a space) to trigger the tools, then press Ctrl+S to save, and press Ctrl+B to build.

The generated results can be found in the folder gen specified in the gendir variable.

Installation requirements to use code generator

Generated code has dependencies on RIE library, LTTng library and curl library. When these libraries are installed maybe makefile and subdir.mk files on Debug folder must be adapted to update the libraries installation paths. Regarding LTTng dependencies, preprocessor directive “define NO_LTTNG” has been included in each monitor “.h” file and also in each lttng “.cpp” file. These directives have been included to avoid using LTTng when compiling since Windows operative system doesn´t support LTTng installation.

Otherwise, if the using operative system is LINUX and you want to use LTTng for monitoring you must delete the preprocessor directives to include the specific LTTng code and compile the “gen/Debug/example/fitoptivis/Monitors/lttng/Makefile” with “make all” command.

Generated code usage

Code generation tool generates the templates and structure of the complete system providing a reconfigurability structure, user must fill the service interfaces functionality and the component specific behavior.

Compiling and executing generated code using RIE library

In folder Debug it is generated a Makefile to compile all project using “make all” command. Firstly, it is neccesary to compile all LTTng monitor code in case preprocessor primitives are not included. On the contrary, if preprocessor primitives are included it is neccesary to delete the folder: “gen/Debug/example/fitoptivis/Monitors/lttng” since lttng files can´t be compiled and it is also neccesary to edit “gen/Debug/objects” file to delete “*lttng-ust*” library since it would not be installed in the system.

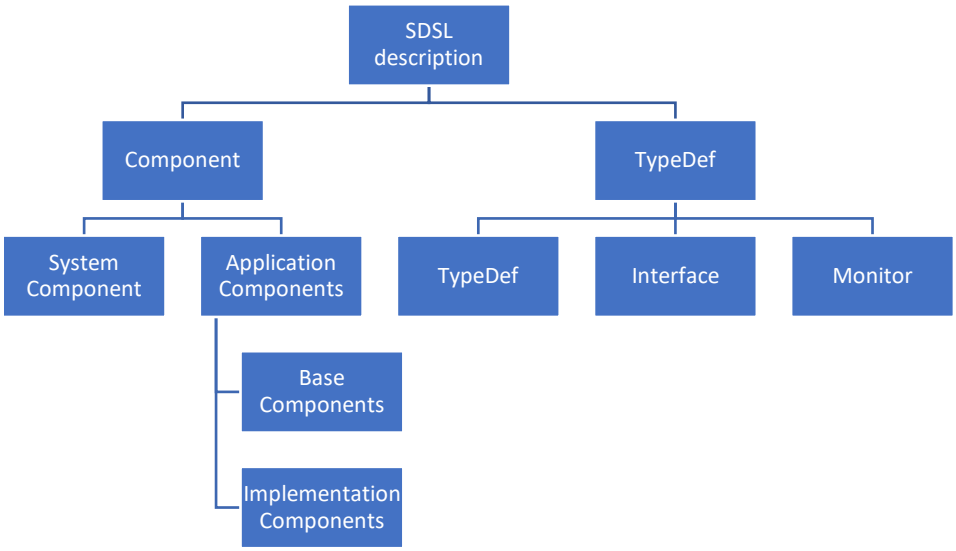
Finally, to execute the code it may be necessary to set LD_LIBRARY_PATH to include the path to FIVIS library. For this purpose export command is used, such as in the next example:

export LD_LIBRARY PATH:$LD_LIBRARY_PATH:/your/path

SDSL language

SDSL is an extension of QRML to support SOA (service-oriented architecture), in which components provide services to other components and communicate through interfaces. SDSL is used for describing and modelling systems.

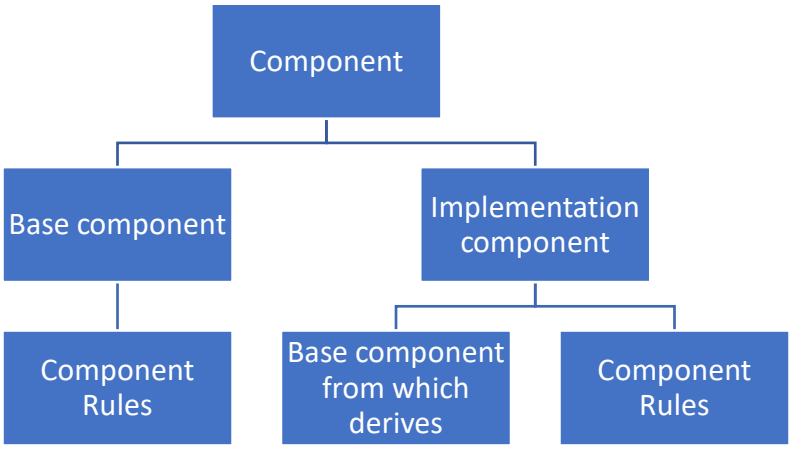
This section explains how to describe a system using SDSL to use the code generation tool. For this purpose, all the elements supported in the grammar will be detailed, next diagram shows a general view of all necessary elements for code generation.

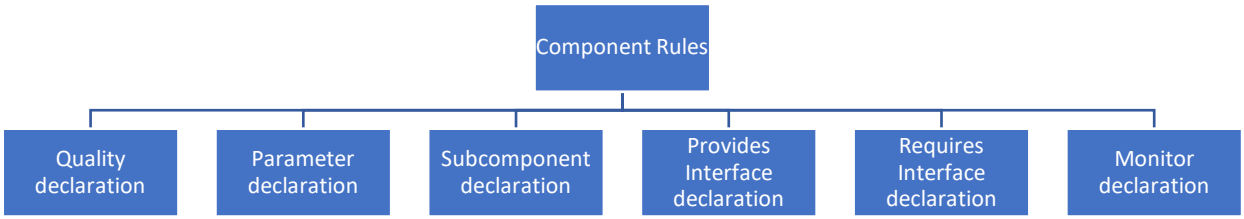


All the components, including system component, need to specify the “application” reserved word to distinguish them for implementation code generation.

- Components

  Components can be base or implementation components, in case that a component uses the “implementation of” reserved word it is an implementation component, on the contrary it is a base component. Next diagrams show all elements supported for component description:





- Base components

  A base component may have several alternatives sharing the same interfaces than the base component. A component can specify their provided/required interfaces, their parameters and qualities and the monitors which it is going to use.

- Implementation components

  An implementation component doesn't need to define again their interfaces because they are derived from base component, so it is neccesary to define its base component with reserved word "implementation of" and component can also define different qualities, parameters and monitors than the ones defined on the base component.

Additionally, a component supports hierarchy. So, a component may have several internal components connected between them. These subcomponents can´t be defined in a alternatives code fragment since hierarchical components can´t have several implementations. For this purpose, the hierarchical components must be instantiated and connected in the configuration body of the component. Root component interfaces must be connected to the correspondent subcomponent interfaces.

- Global system

  The first component in the SDSL description is always going to be the system component, all the components in the system application are instantiated on this system component and connections between components are also specified in this component using the "connected to" reserved word. This component must not define any alternatives.

- Primitives Datatypes

  SDSL supports different primitive datatypes: integer, string, char, char *, short, unsigned short, long, unsigned long, float, double, boolean, Mat (OpenCV oriented datatype to deal with video oriented systems) and timespec (datatype oriented to monitoring purposes). Remote implementation of the datatypes does not support all the primitive datatypes since gRPC doesn´t support some types such as Mat or timespec.

- Datatypes

Datatype element is introduced to describe a new data structure instead of primitive datatypes supported in the SDSL. Datatype definition includes the reserved word “typedef”.
Both primitive types and new-defined ones are used for specifying the kind of data of a quality, a parameter, an interface parameter, or a monitor parameter

- Interfaces

  Service interfaces describe interfaces which can be provided or required by different components, it specifies different services with different parameters. The description of the interface includes the reserved word “interface” to define the interface name and the reserved word “service” to define a method name in the interface. The parameters type can be a primitive type or a new defined datatype. Regarding services names, services from different interfaces should not have the same name if the interfaces are used by the same component because the generator will create services with the same name and this would produce a compilation error.

- Monitors

  A monitor is described with the reserved word “monitor” to identify the tracer that provides the monitoring traces. The description should include a provider for sending data to FIVIS system using reserved word “provider” and different events description with reserved word “events”, each of the event can include several monitoring parameters.

  A monitor is used to trace the events defined on that monitor and can be instantiated on different components using the reserved word “usesmonitor”.

  - Sending data to FIVIS system

    Code generator produces specific code for sending data to FIVIS for each declared monitor. User must only pay attention to function “*send_data(...)”* with all parameters of the different events. Each time this function is called, a value is stored in a buffer to send to the FIVIS system. By default, the buffer has a size of 50 samples and each time it fills up, all the samples are sent. After that, exported data may be processed for data visualization or data analysis.

  - Sending data to LTTng

    Code tool creates the necessary files to trace some application parameters defined on each declared monitor. To use the monitors correctly, it is generated a lttng folder on Monitors folder which contains a Makefile to compile all necessary files with “make all”. To trace each event, user must call the function with the same name that the event declared and passing it the correspondent parameters. To start tracing the “make trace-init” command must be used and to stop tracing the “make trace-stop” command is used. To view and analyze the recorded events and read the LTTng traces exist several tools, but babeltrace is recommended.

Getting Started-BAC example

In this section a BAC (Biometric Access Control) application will be described using QRML and including SDSL concepts to generate implementation code automatically. Created code generator has some requirements that will be detailed along this tutorial. This example will be used to illustrate the particularities of the code generator and describe all aspects that must be included in the description to generate code correctly.

System component

System application component must have the ‘application’ reserved word to indicate its type and must also be the first component in the SDSL description. This component can´t have alternatives and it includes the instances of the different components in the application, the qualities or parameters of the system component, the monitors and the connections among components. Regarding component connections the ‘connected to’ reserved word is used to connect components making use of provided and required component declared interfaces. Connection of components is done through base components in case that a component has several implementations. On next image, the BACApplication system description is shown:

```
application component BACApplication {
    component FaceDetection faceDet;
    component FaceRecognition faceRec;
    component AccessControl accCon;
    quality Latency endToEnd;
    endToEnd==faceDet.lat.latency + faceRec.lat.latency + accCon.lat.latency;
    faceDet.outp outputs to faceRec.inp;
    faceRec.outp outputs to accCon.inp;
    faceDet.ip1 connected to faceRec.ir1;
    faceRec.ip1 connected to accCon.ir1;
    usesmonitor VideoTrace mon1;
}
```

Figure 1 System description using SDSL

Application components

Application components must include the ‘application’ reserved word to indicate its type since only this kind of components are considered to generate implementation code. Application component description may include different alternatives or implementations. So, components can be base components or implementation components. A base component description should include the different alternatives. Implementation components include ‘implementation of’ reserved word to indicate its base component. Therefore, components indicating its base component are considered as implementation components and the rest of them are considered as base components.

Component description must include: the definition of provided and required interfaces (using ‘providesinterface’ or ‘requiresinterface’ reserved word) to communicate with other components, qualities, parameters and monitors (with ‘usesmonitor’ reserved word). In case of an implementation

component: qualities, parameters and monitors are inherited from base component. But it is also allowed to define new ones for the implementation component. Concerning service interfaces in the implementation component, they must be the same than the base component interfaces.

Next images show the SDSL description of the three different application components in the BAC example, they also include aspects of QRML which are not considered for code generation, such as 'requires' description or 'inputs'/'outputs' specification.

```
application component FaceDetection {
    outputs Image outp;
    providesinterface VideoInterface ip1;
    requiresinterface VideoInterface ir1;
    quality Latency lat;
    lat.latency==imgAna.latency;
    requires ImageCapturing imgCap;
    requires ImageAnalysis imgAna;
    usesmonitor VideoTrace mon0;
}
```

```
application component FaceRecognition {
    inputs Image inp;
    outputs ID outp;
    providesinterface IdInterface ip1;
    requiresinterface VideoInterface ir1;
    quality Latency lat;
    lat.latency==faceId.latency;
    quality Quality recQuality;
    recQuality.qual==faceId.qual;
    requires FaceIdentification faceId;
    usesmonitor VideoTrace mon0;
}
```

```
application component AccessControl {
    inputs ID inp;
    requiresinterface IdInterface ir1;
    quality Latency lat;
    lat.latency==database.latency;
    requires DatabaseAccess database;
    usesmonitor VideoTrace mon0;
}
```

Figure 2 Application components descriptions using SDSL

User-defined datatypes

Datatypes are defined on SDSL to specify a type of a quality, a parameter, an interface parameter or a monitor parameter. User-defined datatypes use the reserved word 'typedef' to describe them. They are introduced to define a new data structure different from primitive datatypes specified on SDSL. The data structure may be composed of different data which can be of a primitive type (integer, string, char *, short, long, bool, float, double, Mat and timespec) or another previously defined datatype.

On next figure it can be seen an example of datatype definition for the BAC example:

```
typedef imageData{
    int pixelFormat;
    char * pixel;
}
```

Figure 3 Datatype description using SDSL

Service Interfaces

Service interfaces describe interfaces which can be provided or required by different components, it specifies different services with different parameters. The description of the interface includes the reserved word ‘interface’ to define the interface name and the reserved word ‘service’ to define a method name in the interface. The parameters type can be a primitive type or a new defined datatype, parameter direction must also be specified, supported options are: in, out or inout. Next figure shows an interface description in the BAC example:

```
interface IdInterface{
    service ReadDataBlock{
        DataBlockType data:out;
        bool readOK:out;
    }
    service get_image{
        imageData image:out;
        bool readOK:out;
    }
}
```

Figure 4 Interface description using SDSL

Monitors

A monitor is described with the reserved word ‘monitor’ to identify the tracer that provides the monitoring traces. The description should include a provider for sending data to FIVIS system using reserved word ‘provider’ and different events description with reserved word ‘events’, each of the event can include several monitoring parameters. The monitoring parameters id and ts cannot be used since they are reserved for FIVIS implementation.

A monitor is used to trace the events defined on that monitor and can be instantiated on different components using the reserved word ‘usesmonitor’. This fact can be seen on the figure showing system description.

An example of monitor description used on BAC system is the next one:

```
monitor VideoTrace{
    provider unican;
    event Performances{
        Comp:char *;
        Latency:double;
    }
}
```

Figure 5 Monitor description using SDSL

A monitor is described with the reserved word “monitor” to identify the tracer that provides the monitoring traces. The description should include a provider for sending data to FIVIS system using reserved word “provider” and different events description with reserved word “events”, each of the event can include several monitoring parameters. Regarding these monitoring parameters, these can be of type: integer, string, char *, double, float, long, short or timespec.

The provider is necessary to identify where to store the data on FIVIS system since several providers are supported to send data to the system.