Simulation Configuration Modeling of Distributed Communication Systems

Mihal Brumbulli and Joachim Fischer

Humboldt Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
{brumbull,fischer}@informatik.hu-berlin.de

Abstract. Simulation is the method of choice for the analysis of distributed communication systems. This is because of the complexity that often characterizes such systems. But simulation modeling is not a simple task mainly because there exists no unified approach that can provide description means for all aspects of the system. These aspects include architecture, behavior, communication, and configuration. In this paper we focus on simulation configuration as part of our unified modeling approach based on the Specification and Description Language Real Time (SDL-RT). Deployment diagrams are used to describe the simulation setup of the components and configuration values of a distributed system. We provide tool support for automatic implementation of the models for the ns-3 network simulation library.

Keywords: Simulation modeling, SDL-RT, ns-3.

1 Introduction

Simulation modeling of distributed communication systems is not a simple task. This is because of the complexity that often characterizes such systems. There are several aspects that need to be modeled, preferably using a unified and standardized approach. These include architecture, behavior, communication, and configuration. System development tools based on SDL [1] or UML [2] can model such aspects at a certain degree and independently from the target platform. Nevertheless, the lack of integration with existing simulation (and especially network simulation) libraries makes it indeed very challenging to derive the desired executable from model descriptions. This is one of the reasons that pushes the developer towards the use of general purpose languages (i.e. C/C++). It is not very difficult to obtain an executable from these models, because they are described in the same language as the simulation library. This approach is time consuming and error-prone, the models will become very soon difficult to maintain, and they cannot be used with other simulation libraries, except the one they were implemented for.

In this context, there have been several works with the aim of exploiting the advantages of both approaches. In [3] the authors show how to automatically generate an executable for the ns-2 [4] simulator from models described in

SDL. The idea of automatic code generation for network simulators is further described in [5], where SDL model descriptions are also used for deriving simulation models for ns-3 [6]. In [7] UML diagrams are used to construct simulation models, which in turn can be executed in an event-driven simulation framework like OMNET++ [8].

Although the approaches introduced so far do provide description means for some of the aspects of distributed systems, there is still work to be done regarding configuration modeling. A configuration model describes the setup and configuration values of the components of a distributed system. These aspects are still handled in different ways ranging from general purpose languages (like C++ in ns-3) to tools with limited (like NAM [9] in ns-2) or more complete (like OMNET++) modeling capabilities. What all these methods have in common is that, except of being dependent on the simulation framework, they do not integrate with existing approaches based on standardized languages like SDL or UML.

We use SDL-RT [10] as the main language of our unified approach for modeling all the above mentioned aspects including simulation configuration. In this paper we focus on configuration modeling and show how SDL-RT deployment diagrams can be used for this purpose. We consider automatic implementation to be very important, thus we provide code generation for the ns-3 simulator.

We start by giving a short overview of the ns-3 library, focusing on configuration modeling (Sect. 2). In Sect. 3 we briefly introduce our modeling approach based on SDL-RT by means of a simple example. The use of SDL-RT deployment diagrams for simulation configuration modeling is described in Sect. 4 and our code generator in Sect. 5. Finally, we present the conclusions of our work in Sect. 6.

2 The ns-3 Simulator

Ns-3 is a discrete-event network simulator for internet systems, targeted primarily for research and educational use. The simulation library is entirely written in C++ and implements all the components used in simulation configurations. These components include nodes, mobility models, applications, protocols, devices, and channels (Fig. 1). In ns-3 simulations, there are two main aspects to configuration:

- the simulation topology and how components are connected,
- the values used by the components in the topology.

Topology and Components The node is the core component in the model and acts as a container for applications, protocol stacks, and devices. Devices are interconnected through channels of the same type. Applications are usually traffic generators: they create and send packets to the lower layers using a socket-like API. A simulation configuration usually follows these steps:

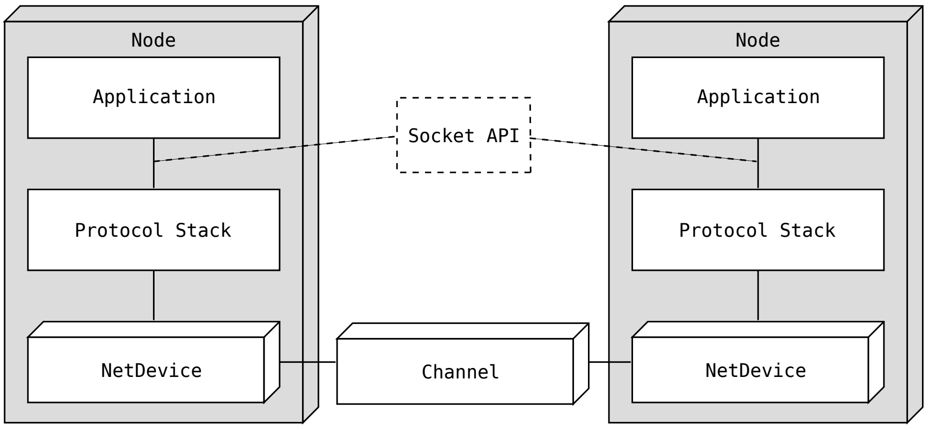


Fig. 1. Generic model for ns-3 simulation configurations

- a set of nodes is created,
- a mobility model (topology) is applied to the nodes,
- channels are created,
- devices are installed on the nodes and attached to channels,
- applications are installed on the nodes.

Configuration Values. The ns-3 library implements an attribute system that organizes the access of the configuration values of the components, thus providing a fine-grained access to internal variables in the simulation. The general C++ syntax for setting attribute values is:

```
<object-name> "->" "SetAttribute" "("
    <attribute-name> "," <attribute-value>
")" ";"
```

The `<object-name>` represents a *ns-3 Object*, which can be any of the components in Fig. 1 (i.e., Node, NetDevice, Channel, etc.). As the name suggests, the `<attribute-name>` is a string containing the attribute's name. The list of attributes for each ns-3 Object and the `<attribute-value>` they expect can be found at the ns-3 documentation.[1]

This configuration mechanism (*attribute* = *value*) looks quite straightforward, thus it can be used in tools for automatic implementation from model description. We use this mechanism in the context of SDL-RT deployment diagrams for providing a model-driven approach to simulation configuration and automatic code generation for the ns-3 library.

3 Simulation Modeling with SDL-RT

SDL-RT is based on the SDL standard [1] extended with real time concepts, which include [10]:

[1] `http://www.nsnam.org/docs/release/3.10/doxygen/index.html`

– use of C/C++ instead of SDL for data types,
– use of C/C++ as an action language, and
– semaphore support.

These extensions considerably facilitate integration and usage of legacy code and off the shelf libraries such as real-time operating systems, simulation frameworks, and protocol stacks [10]. The work presented in [11,12] shows how SDL-RT and simulation frameworks can be used in the development of complex distributed systems. In [13] the authors have successfully applied this approach in the development of the wireless mesh sensing network for earthquake early warning described in [14]. Further extensions to SDL-RT are provided by UML diagrams [2]:

– Class diagrams bring a graphical representation of the classes organization and relations.
– Deployment diagrams offer a graphical representation of the physical architecture and how the different nodes in a distributed system communicate with each other.

SDL-RT can be seen as a pragmatic combination of the standardized languages SDL, UML, and C/C++. We use SDL-RT for simulation modeling of distributed communication systems, including their architecture, behavior, communication, and configuration. Figure 2 illustrates these aspects by means of a simple example.

The client sends a request message (*mRequest*) to the server and waits for a reply (*mReply*). This sequence of actions is repeated every 1000 ms (tWait timer). The server waits for a request from the client. Upon receiving a request, it immediately sends a reply to the client.

By definition, the SDL-RT channels can only model local communication (i.e. between processes running on the same node). However, it is possible to describe distributed communication directly in the model, because SDL-RT uses C/C++ as an action language. We use the flexibility provided by C/C++ to define description means for distributed communication without changing the language. Figure 2 shows how this can be achieved via the *TCP_CONNECT* and *TCP_SEND* macros.[2] The *SENDER_ADDR* is used to access the sender's address of the last received message.

It is also possible to define patterns [15] for other types of communication. These patterns can be described in SDL-RT and implementation is handled automatically by the code generator (Sect. 5). We define the *NODE* macro to facilitate access on communication layers or other components. This allows us to reference the ns-3 node (from within behavior descriptions) and subsequently all the other components associated to it (see Fig. 1).

[2] UDP communication is also possible by using the corresponding macros.

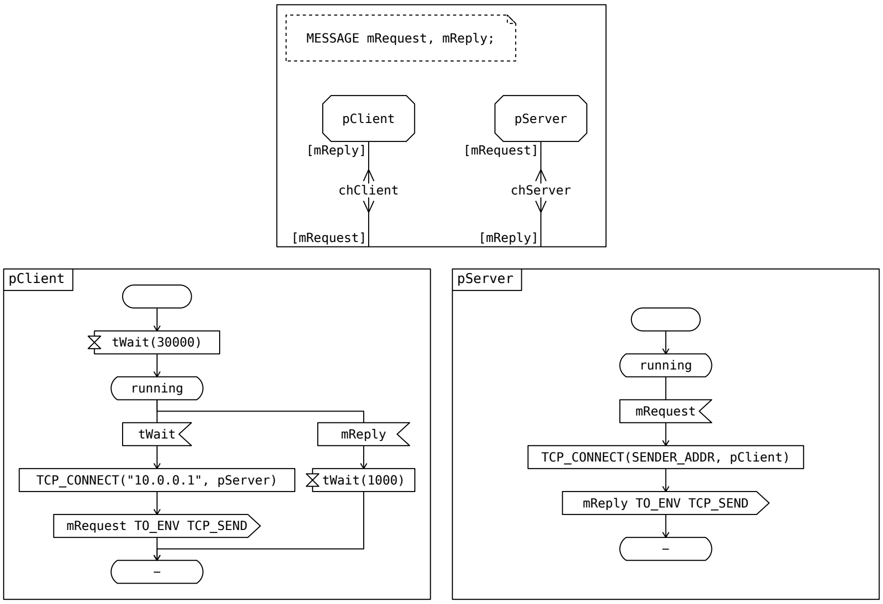


Fig. 2. SDL-RT model of a client-server application

4 Configuration Modeling

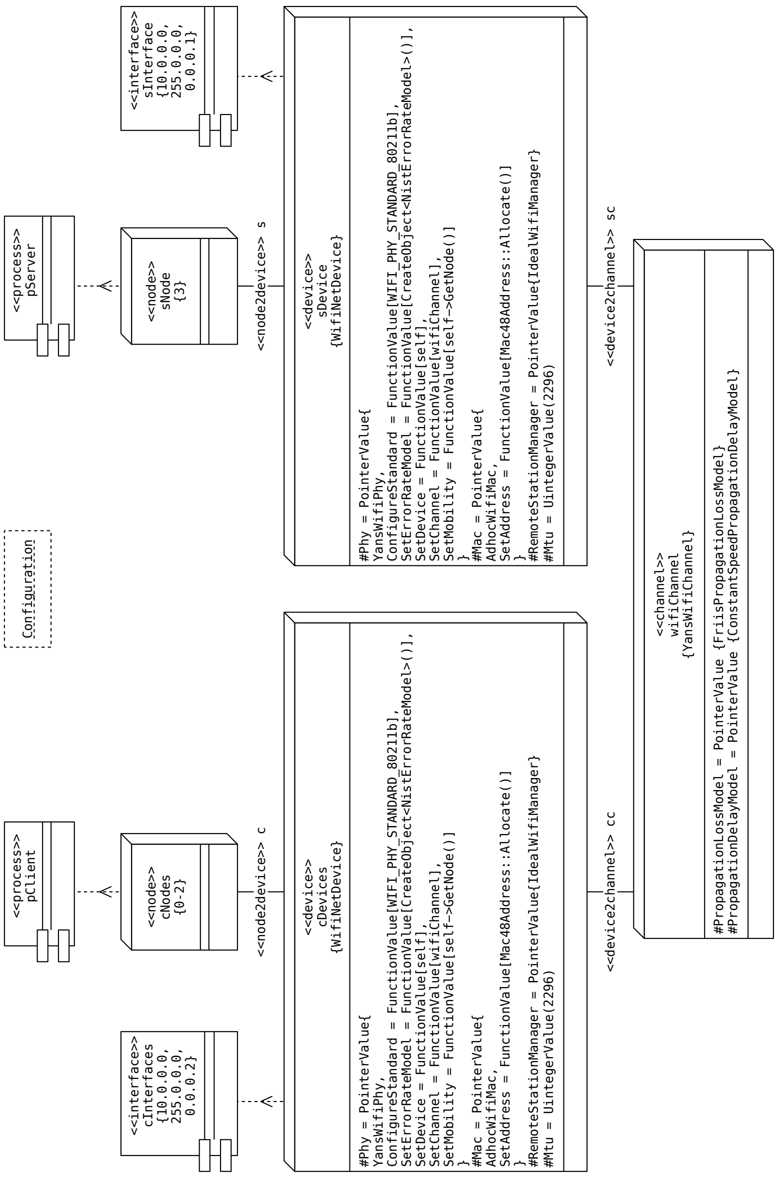
The SDL-RT deployment diagram describes the physical configuration of run-time processing elements of a distributed system and may contain [10]:

– *Nodes* are physical objects that represent processing resources.
– *Components* represent distributable pieces of implementation of a system.
– *Connections* are physical links between nodes or components.
– *Dependencies* from nodes to components mean the components are running on the nodes.

There exist many similarities between a SDL-RT deployment diagram and the ns-3 model in Fig. 1. We use this type of diagram for describing simulation configurations for the ns-3 library. For this purpose we define a set of rules to be applied as shown in Fig. 3.

4.1 Nodes

The `<<node>>` represents a *ns-3 NodeContainer*. It has no attributes and only one property, which is the set of nodes in the container. This set is represented as a comma separated list of node identifiers. Each ns-3 node in a simulation configuration model has a unique identifier, which is assigned to it incrementally (starting at 0) by the simulation library. The total number of nodes to be created is calculated automatically by our code generator based on the highest identifier



Configuration
<<process>> pClient
<<process>> pServer
<<node>> cNodes {0-2}
<<node>> sNode {3}
<<interface>> cInterfaces {10.0.0.0, 255.0.0.0, 0.0.0.2}
<<interface>> sInterface {10.0.0.0, 255.0.0.0, 0.0.0.1}
<<node2device>> c
<<node2device>> s
<<device>> cDevices {WifiNetDevice}
<<device>> sDevice {WifiNetDevice}
<<device2channel>> cc
<<device2channel>> sc
<<channel>> wifiChannel {YansWifiChannel}

```
#Phy = PointerValue{
YansWifiPhy,
ConfigureStandard = FunctionValue[WIFI_PHY_STANDARD_80211b],
SetErrorRateModel = FunctionValue[CreateObject<NistErrorRateModel>()],
SetDevice = FunctionValue[self],
SetChannel = FunctionValue[wifiChannel],
SetMobility = FunctionValue[self->GetNode()]
}
#Mac = PointerValue{
AdhocWifiMac,
SetAddress = FunctionValue[Mac48Address::Allocate()]
}
#RemoteStationManager = PointerValue{IdealWifiManager}
#Mtu = UintegerValue(2296)
```

```
#Phy = PointerValue{
YansWifiPhy,
ConfigureStandard = FunctionValue[WIFI_PHY_STANDARD_80211b],
SetErrorRateModel = FunctionValue[CreateObject<NistErrorRateModel>()],
SetDevice = FunctionValue[self],
SetChannel = FunctionValue[wifiChannel],
SetMobility = FunctionValue[self->GetNode()]
}
#Mac = PointerValue{
AdhocWifiMac,
SetAddress = FunctionValue[Mac48Address::Allocate()]
}
#RemoteStationManager = PointerValue{IdealWifiManager}
#Mtu = UintegerValue(2296)
```

```
#PropagationLossModel = PointerValue {FriisPropagationLossModel}
#PropagationDelayModel = PointerValue {ConstantSpeedPropagationDelayModel}
```

Fig. 3. A simulation configuration for the client-server example

present in the configuration model. In Fig. 3 the highest identifier is 3, thus the total number of nodes would be 4. It is also possible to express a range of identifiers (i.e. {0–2} is the same as {0,1,2}). This feature is very useful in scenarios with a high number of nodes.

The `<<device>>` represents a *ns-3 NetDevice*. It has only one property, which is the type of the device. It can have as many attributes as necessary for configuring the device.

The `<<channel>>` represents a *ns-3 Channel*. Its type is given by its only property. It can also have as many attributes as needed.

4.2 Attributes

The general syntax of attributes for `<<device>>` and `<<channel>>` is:

```
"#" <attribute-name> "=" <attribute-value>
```

This description can be mapped to the corresponding C++ implementation using the ns-3 API (see Sect. 2). As an example consider the *Mtu* attribute in Fig. 3:

```
#Mtu = UintegerValue(2296)
```

The ns-3 implementation for this attribute will be:

```
sDevice->SetAttribute("Mtu", UintegerValue(2296));
```

Even though this mapping looks quite straightforward, it cannot handle all possible attribute values. This is because not all values can be assigned directly to the attribute. In this context, we categorize attribute values into two main groups: simple and complex. Simple values are those that can be directly assigned to the corresponding attribute (i.e. the *UintegerValue(2296)* for the *Mtu* attribute). On the other hand, complex values require the creation and configuration of a *ns-3 Object* of a certain type, for which its configuration can be described also by means of attribute values. The object can then be assigned to the attribute as a value. We extend the description means in order to provide support also for complex values. The general syntax of complex attributes is:

```
"#" <attribute-name> "=" "PointerValue" "{" <object-type>
    {"," <attribute-name> "=" <attribute-value>}
"}"
```

The *PropagationLossModel* is an example of a complex attribute value:

```
#PropagationLossModel = PointerValue {
    FriisPropagationLossModel
}
```

In this case an object of type *FriisPropagationLossModel* needs to be created before it can be assigned as an attribute value:

```
Ptr<FriisPropagationLossModel> obj =
    CreateObject<FriisPropagationLossModel>();
wifiChannel->SetAttribute(
    "PropagationLossModel", PointerValue(obj)
);
```

The descriptions introduced so far can cover all possible ns-3 attribute values. Nevertheless, they are not sufficient for ensuring the minimal required configuration for normal operation of devices and/or channels. This is because the ns-3 attribute system itself is incomplete. In order to address this issue, we define a new group of attribute values named *function values*. As the name suggests, these allow us to express function calls in an attribute-value fashion, where the name of the attribute is actually the name of the function and the value represents the list of parameter values. The syntax of function attributes is:

```
"#" <attribute-name> "=" "FunctionValue" "["
    [<parameter-value> {"," <parameter-value>}]
"]"
```

As an example consider the *Mac* attribute:

```
#Mac = PointerValue {AdhocWifiMac,
    SetAddress = FunctionValue[Mac48Address::Allocate()]
}
```

This is a mix of complex and function values and is mapped to C++ as:

```
Ptr<AdhocWifiMac> obj = CreateObject<AdhocWifiMac>();
obj->SetAddress(Mac48Address::Allocate());
sDevice->SetAttribute("Mac", PointerValue(obj));
```

We also define *self*,[3] which can be used to reference the channel or device inside their configuration values (see Fig. 3).

4.3 Components

The `<<process>>` represents an instance of a SDL-RT process defined in the architecture (see Fig. 2). It must be linked with a `<<node>>` using a dependency relation. The dependency means that there is a running instance of the process for each of the ns-3 nodes in the container defined by the `<<node>>`.

The `<<interface>>` represents a range of ip addresses to be assigned to the devices. It has neither parameters nor attributes and must be linked with a `<<device>>` using a dependency relation.

4.4 Connections

As the name suggests, the `<<node2device>>` links nodes to devices. In terms of simulation configuration this means that, for each node in the container defined by `<<node>>`, a `<<device>>` of the specified type is added to it.

The `<<device2channel>>` attaches the devices to the specified channel.

[3] Not to be confused with SDL-RT's *SELF* that is used in behavior descriptions.

4.5 Topologies

The only aspect that cannot be modeled with SDL-RT deployment diagrams is the topology of the nodes. By topology here we mean the actual position of the nodes in the coordinate system used by the network simulator. This is very important especially for configuration models involving wireless and sensor networks.

In fact the topology can be modeled using attributes for the `<<node>>` or C++ code inside SDL-RT comments.[4] A major drawback of these solutions is that they hide the position of the nodes relative to each other. This can be addressed by using topology generators with graphical user interfaces like NAM or NPART [16], which can generate topologies for the ns-2 simulator. Our code generator (Sect. 5) can transform these into ns-3 topologies and apply them to the nodes as specified in the configuration model.

Fig. 4 shows a simple grid topology, which can be applied to the model in Fig. 3.

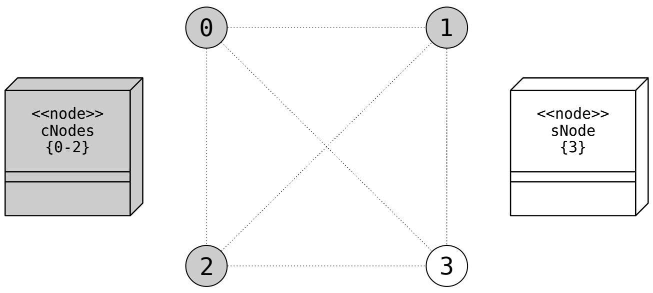


Fig. 4. Sample topology for the client-server example

Nodes with identifiers 0, 1, and 2 are client nodes (nodes represented by *cNodes*); the node with identifier 3 is a server node (*sNode*).

It is important to note that the node identifiers used in the configuration model must exist also in the topology description. If this is not the case, an error containing information on missing nodes is reported and the code generation will fail. Nevertheless, the existence of redundant nodes[5] in the topology description is treated as a warning. In this case the code generation will succeed and the redundant nodes will be ignored by default.

5 Code Generation

SDL-RT descriptions are used as a basis for the generation of an executable for the ns-3 simulator. This implies C++ code generation from SDL-RT

[4] C++ code can be included in the model by using SDL-RT comments (the *Configuration* rectangle in Fig. 3).

[5] These are node identifiers that appear in the topology description but not in the configuration model (SDL-RT deployment diagram).

architecture, behavior, and deployment models. Our tool for code generation is integrated with PragmaDev's RTDS.[6]

5.1 Architecture and Behavior

We have already covered SDL code generation for network simulators in [5]. In this paper we introduce further improvements to our approach by providing a generic model as shown in Fig. 5.

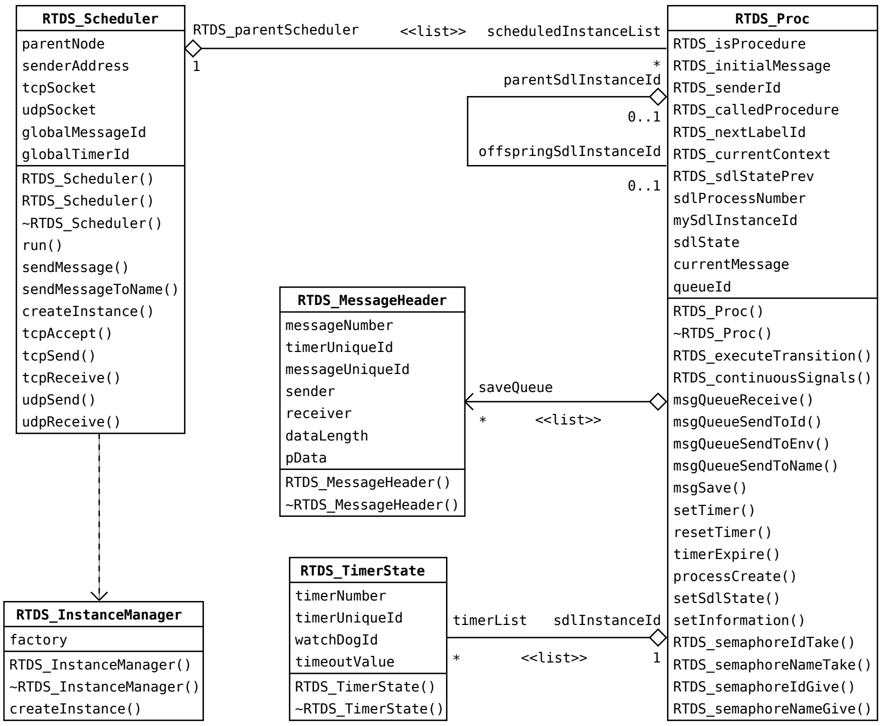


Fig. 5. Generic model for code generation

RTDS_Scheduler keeps track of all process instances running on a node and handles communication between these instances. The creation of process instances is managed by *RTDS_InstanceManager*. Communication can be local or distributed. Local communication is implemented via shared memory [10]. In this case the sender and receiver process instances are running on the same node, which means that they can be accessed by the same *RTDS_Scheduler*.

[6] `http://www.pragmadev.com/product/index.html`

On the other hand, distributed communication is handled via ns-3 sockets (TCP or UDP) and is implemented by *tcpAccept*, *tcpSend*, *tcpReceive*, *udpSend*, and *udpReceive*. There exists a one-to-one relationship between the *RTDS_Scheduler* and the ns-3 node (the `<<node>>` in Fig. 3). This concept is implicitly included in the `<<node>>` definition.

RTDS_Proc provides basic functionality for the SDL-RT processes. All SDL-RT processes (i.e. pClient and pServer in Fig. 2) extend this class by implementing the *RTDS_executeTransition* member function. Each process instance is associated with only one *RTDS_Scheduler*.

RTDS_MessageHeader encapsulates SDL-RT messages. It includes also some additional information required for handling local communication between process instances.

RTDS_TimerState implements the SDL-RT timer. The core functionality is given by the *watchDogId* attribute, which is a *ns-3 Timer*.

5.2 Configuration

The SDL-RT deployment model (see Fig. 3) and a ns-2 topology file serve as inputs to the code generator for configuration implementation. First, the model is checked against the rules defined in Sect. 4. If it doesn't satisfy any of the rules, corresponding errors are reported and no code is generated. On the other hand, in case of success (no errors were detected), the model is transformed into C++ code for the ns-3 library as follows:

1. All the ns-3 nodes are created as part of a global container.
2. A position is assigned to each node according to the topology description.
3. The nodes are grouped into containers as described in the configuration model.
4. The channel of the specified type is created.
5. The network devices (as part of a container) are created, added to the nodes, and attached to the channel.
6. The protocol stack is installed on the nodes. This step is handled automatically by the generator, therefore it doesn't need to be specified in the model.
7. The interfaces are created and attached to the devices.
8. An instance of *RTDS_Scheduler* is created for each node. Process instances are created and associated to the nodes via the *RTDS_Scheduler*.

All these steps are illustrated in Fig. 6, which shows the code generated from the model in Fig. 3 (only the server part).

```
int main(int argc, char **argv)
{
    // 1. Create nodes.
    NodeContainer globalContainer;
    globalContainer.Create(4);

    // 2. Assign positions to nodes.
    Ptr<ConstantPositionMobilityModel> pos_3 = CreateObject<ConstantPositionMobilityModel>();
    pos_3->SetPosition(Vector(0.0, 0.0, 0));
    NodeList::GetNode(3)->AggregateObject(pos_3);

    // 3. Group nodes into containers.
    NodeContainer sNode;
    sNode.Add(NodeList::GetNode(3));

    // 4. Create channels.
    Ptr<YansWifiChannel> wifiChannel = CreateObject<YansWifiChannel>();
    Ptr<FriisPropagationLossModel> p_0_0 = CreateObject<FriisPropagationLossModel>();
    wifiChannel->SetAttribute("PropagationLossModel", PointerValue(p_0_0));
    Ptr<ConstantSpeedPropagationDelayModel> p_1_0 =
        CreateObject<ConstantSpeedPropagationDelayModel>();
    wifiChannel->SetAttribute("PropagationDelayModel", PointerValue(p_1_0));

    // 5. Create devices, add them to the nodes, and attach them to the channel.
    NetDeviceContainer sDevice;
    for(uint32_t i=0; i<sNode.GetN(); i++)    {
        Ptr<WifiNetDevice> device = CreateObject<WifiNetDevice>();
        sNode.Get(i)->AddDevice(device);
        device->Attach(wifiChannel);
        sDevice.Add(device);
        Ptr<YansWifiPhy> p_0_0 = CreateObject<YansWifiPhy>();
        p_0_0->ConfigureStandard(WIFI_PHY_STANDARD_80211b);
        p_0_0->SetErrorRateModel(CreateObject<NistErrorRateModel>());
        p_0_0->SetDevice(device);
        p_0_0->SetChannel(wifiChannel);
        p_0_0->SetMobility(device->GetNode());
        device->SetAttribute("Phy", PointerValue(p_0_0));
        Ptr<AdhocWifiMac> p_1_0 = CreateObject<AdhocWifiMac>();
        p_1_0->SetAddress(Mac48Address::Allocate());
        device->SetAttribute("Mac", PointerValue(p_1_0));
        Ptr<IdealWifiManager> p_2_0 = CreateObject<IdealWifiManager>();
        device->SetAttribute("RemoteStationManager", PointerValue(p_2_0));
    }

    // 6. Install the protocol stack on all nodes.
    InternetStackHelper protocol_stack;
    protocol_stack.InstallAll();

    // 7. Attach interfaces to the devices.
    Ipv4AddressHelper sInterface_address("10.0.0.0", "255.0.0.0", "0.0.0.1");
    Ipv4InterfaceContainer sInterface = sInterface_address.Assign(sDevice);

    // 8. Create a RTDS_Scheduler and process instances for each node.
    RTDS_Scheduler *sNode_scheduler[1];
    for(uint32_t i=0; i<1; i++)    {
        sNode_scheduler[i] = new RTDS_Scheduler(sNode.Get(i));
        RTDS_Env(sNode_scheduler[i]);
    }
    for(uint32_t i=0; i<1; i++)    {
        pServer(sNode_scheduler[i]);
    }

    // Run simulation.
    Simulator::Run();
    Simulator::Destroy();
    return 0;
}
```

Fig. 6. Code generated from the configuration model in Fig. 3

6 Conclusions

Simulation modeling is not a simple task. This is especially true for distributed communication systems due to their complexity. The lack of a unified approach covering all the aspects of such systems makes modeling even more challenging. The existing methodologies and tools, which are based on standardized languages like SDL or UML, do provide modeling means but they are not complete.

In this paper we introduced simulation configuration modeling as part of our unified approach based on SDL-RT for modeling all aspects of distributed communication systems. We showed how SDL-RT deployment diagrams serve this purpose quite well, despite some limitations regarding the attribute values and topology description. To overcome these limitations we extended the description capabilities for attributes in order to provide support for more complex configuration values. Also, we defined a mapping between the configuration model in SDL-RT and topology generator tools.

We have already implemented our approach, thus providing code generation for the ns-3 network simulator. We believe that our tool can be adapted (without much effort) to support also other simulation libraries. The only limitation is the programming language. It has to be C/C++ because this is the language used by SDL-RT.

References

1. International Telecommunication Union (ITU): Z.100 series, Specification and Description Language, `http://www.itu.int/rec/T-REC-Z.100/en`
2. OMG: OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.4.1. Tech. rep., Object Management Group (2011)
3. Kuhn, T., Geraldy, A., Gotzhein, R., Rothländer, F.: *ns*+SDL – The Network Simulator for SDL Systems. In: Prinz, A., Reed, R., Reed, J. (eds.) SDL 2005. LNCS, vol. 3530, pp. 103–116. Springer, Heidelberg (2005)
4. Breslau, L., Estrin, D., Fall, K.R., Floyd, S., Heidemann, J.S., Helmy, A., Huang, P., McCanne, S., Varadhan, K., Xu, Y., Yu, H.: Advances in Network Simulation. IEEE Computer 33(5), 59–67 (2000)
5. Brumbulli, M., Fischer, J.: SDL Code Generation for Network Simulators. In: Kraemer, F.A., Herrmann, P. (eds.) SAM 2010. LNCS, vol. 6598, pp. 144–155. Springer, Heidelberg (2011)
6. Henderson, T.R., Roy, S., Floyd, S., Riley, G.F.: ns-3 Project Goals. In: Proceeding from the 2006 Workshop on ns-2 – the IP Network Simulator (WNS2 2006), article 13. ACM Press (2006)
7. Dietrich, I., Dressler, F., Schmitt, V., German, R.: SYNTONY: Network Protocol Simulation Based on Standard-Conform UML 2 Models. In: 2nd International Conference on Performance Evaluation Methodologies and Tools (ValueTools 2007), ICST, article 21 (2007)
8. Varga, A., Hornig, R.: An Overview of the OMNeT++ Simulation Environment. In: Proceedings of the 1st International Conference on Simulation Tools and Techniques (Simutools 2008), ICST, article 60 (2008)

9. Estrin, D., Handley, M., Heidemann, J.S., McCanne, S., Xu, Y., Yu, H.: Network Visualization with Nam, the VINT Network Animator. IEEE Computer 33(11), 63–68 (2000)
10. SDL-RT Consortium: Specification and Description Language - Real Time. Version 2.2, `http://www.sdl-rt.org/standard/V2.2/html/SDL-RT.htm`
11. Ahrens, K., Eveslage, I., Fischer, J., Kühnlenz, F., Weber, D.: The Challenges of Using SDL for the Development of Wireless Sensor Networks. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) SDL 2009. LNCS, vol. 5719, pp. 200–221. Springer, Heidelberg (2009)
12. Blunk, A., Brumbulli, M., Eveslage, I., Fischer, J.: Modeling Real-time Applications for Wireless Sensor Networks using Standardized Techniques. In: SIMULTECH 2011 - Proceedings of 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications, pp. 161–167. SciTePress (2011)
13. Fischer, J., Redlich, J.P., Zschau, J., Milkereit, C., Picozzi, M., Fleming, K., Brumbulli, M., Lichtblau, B., Eveslage, I.: A Wireless Mesh Sensing Network for Early Warning. Journal of Network and Computer Applications 35(2), 538–547 (2012)
14. Fleming, K., Picozzi, M., Milkereit, C., Kühnlenz, F., Lichtblau, B., Fischer, J., Zulfikar, C., Ozel, O., et al.: The Self-organizing Seismic Early Warning Information Network (SOSEWIN). Seismological Research Letters 80(5), 755–771 (2009)
15. Schaible, P., Gotzhein, R.: Development of Distributed Systems with SDL by Means of Formalized APIs. In: Reed, R., Reed, J. (eds.) SDL 2003. LNCS, vol. 2708, pp. 158–158. Springer, Heidelberg (2003)
16. Milic, B., Malek, M.: NPART - Node Placement Algorithm for Realistic Topologies in Wireless Multihop Network Simulation. In: Proceedings of the 2nd International Conference on Simulation Tools and Techniques (SimuTools 2009), ICST, arricle 9 (2009)