Integration of SDL Models into a SystemC Project for Network Simulation

Pavel Morozkin, Irina Lavrovskaya,
Valentin Olenev, and Konstantin Nedovodeev

Institute of High-Performance Computer and Network Technologies,
Saint Petersburg State University of Aerospace Instrumentation,
Saint Petersburg, 190000, Russia
{pavel.morozkin,irina.lavrovskaya,
valentin.olenev,konstantin.nedovodeev}@guap.ru

Abstract. The paper proposes an approach for integration of a number of SDL model instances into a SystemC project. It is done by conversion of an SDL model into a C/C++ library. Implementation of the library is performed by means of post-processing of previously auto-generated C code by the CAdvanced code generator. The main benefit of the approach is reducing of the project work effort and achieving a better quality of the simulation results.

1 Introduction

Key interests of industrial telecommunication companies are increased quality of products with reduced time-to-market. Modeling is a mechanism that meets these requirements by its application at the early stages of product development. This paper focuses the communication protocol development. The ITU Specification and Description Language (SDL) [1,2] and SystemC [3–5] language can both be used separately for the modeling of the communication protocols. SystemC network models are used to explore the behavior, functional and non-functional properties of protocols. Formal SDL models focus primarily on the protocol behavior exploration at the specification development stage. In general, the development of protocol stack models in SystemC and in SDL is performed in parallel. However, resources are often limited, so this way of development carries project delay or commercial risks. This paper proposes a new approach for integrating the SDL model instances into the SystemC network model by means of a special protocol stack library. This approach minimizes the potential risks and decreases the complexity of a project. The library implemented in accordance with the proposed approach contains an SDL model of a protocol stack.

2 The Problem Statement

Nowadays, modeling plays an important role in protocol stack development. It is one of the most efficient methods of protocol mechanisms development, and is performed by implementation of high-level behavior models and testing them. SDL and SystemC are widely used languages for these purposes.

SDL is a formal description technique (FDT) [6] based on a formal semantics and is widely used for specification and investigation of event-driven communication systems. Moreover, the formal SDL specification can be taken as a part of the official protocol specification as a reference. The SystemC language is a C++ library, which provides a capability of event-driven simulation and system design using the Object Oriented Programming (OOP) paradigm and software design patterns. These features make SystemC an appropriate tool for exploration of functional and performance characteristics of protocols by simulation of network models operation. However, implementation of large SystemC projects can lead to difficulties during debugging. On the other hand, SDL has a graphical representation that helps to design a protocol stack model rapidly. Therefore, the problem that is faced is how to efficiently use the SDL and SystemC languages together.

The common use of SDL and SystemC in one model could decrease time costs for protocol development. To provide this common use, we use an SDL/SystemC co-modeling method [7,8]. In this method the SystemC tester manages the simulation of the SDL model, configures it, generates test sequences, etc. This method partly solves the efficiency problem, but has a list of drawbacks. The most important issue is that it is impossible to create different numbers of SDL model instances for the tester environments to use in network simulations. Another complexity is that the SystemC developer should understand the principles of SDL model operation.

The problem can be solved by means of a special library, which can be applied during networks simulation in SystemC. This library should implement the original SDL model and SDL simulation kernel as well as provide special services for the user (term 'user' stands for the SystemC developer, who uses the library).

3 Overview of SDL/SystemC Co-modeling Approach

3.1 Tool Choice

The approach of SDL/SystemC co-modeling described in this paper assumes that we have a SystemC project that corresponds to the whole model to be considered. The whole model contains SDL and SystemC parts. Consequently, this approach uses a C/C++ representation of the SDL system [9]. Before starting a description of the discussed approach, we need to introduce general principles of co-modeling with some requirements and important notions for modeling. Consider some abstract SDL tool. This SDL tool should meet the following requirements:

1. Provide a possibility to generate C/C++ code for the implemented model that is the equivalent of the SDL.
2. The generated C/C++ code operation should be controlled by some kind of a manager engine (*SDL_kernel*).

3. The *SDL_kernel* should provide a number of functions for initialization and simulation of the SDL model. For the further discussion it is necessary to introduce declarations for two main functions: *SDL_Init()*, which is responsible for initialization, and *SDL_Simulate()*, which is responsible for emulation of the SDL system, so that one SDL transition is executed during each call of this function. One SDL transition is a system state change from one to another.

It should be pointed out that this kind of the SDL tool already exists. For example, all these features are provided by the IBM Rational SDL Suite [10].

3.2 Modeling with SDL and SystemC

Modeling with SDL and SystemC is an approach that focuses on modeling of systems that include SDL and SystemC models. This model consists of an SDL model of a protocol layer and a SystemC model of the same layer. The SDL/SystemC co-model is represented by a SystemC project, which contains SDL and SystemC parts. The SystemC model is a master and it provides all the mechanisms for simulation. The SDL part is represented by C/C++ code, which was generated from the original SDL system. Generation of code is performed by means of the CAdvanced code generator, which is a part of the IBM Rational SDL Suite.

The process for connection of SDL and SystemC parts can be subdivided into the following stages:

1. Preparation of the SDL system to be the part of the whole model.
2. Generation of C/C++ code on basis of the SDL system.
3. Insertion of this C/C++ code code to the *SDL_kernel*.
4. Preparation of the SystemC part of the model.
5. Integration of the *SDL_kernel* with the generated C code into the whole model.

According to this approach, the SystemC model is a master and the SDL model is a slave. So SystemC provides the mechanisms for modeling. Co-modelling organization starts after implementation of the SDL and the SystemC parts of the model. The SystemC project should contain a special thread, which is intended for the SDL part (*SDL_thread*). This thread calls the *SDL_kernel* function *SDL_Simulate()*. Control of the *SDL_thread* can be specified in any acceptable way. The choice of this way depends on the requirements of the modeled system. Initialization of the SDL part of the model requires a call to the *SDL_Init()* function.

One of the most interesting questions in the area of SDL/SystemC co-modeling is how scheduling is organized, because of the difference in the notions of the SDL and SystemC modeling times. According to the SDL/SystemC co-modeling approach, SystemC provides all necessary mechanisms for scheduling of events. Each point of the modeling time corresponds to a number of delta-cycles, which trigger in zero time. According to the SDL/SystemC co-modeling approach one execution of the SDL transition is performed in one delta-cycle. Each transition of an SDL process from one state to another can result in scheduling of the new events. There are two ways for events scheduling – signals and timers. Using of signals means that the event should be performed at the current moment of modelling time. Such an event is processed during the next delta-cycles after a delta-delay. The timer expiration is scheduled at another moment of modelling time. So it causes a new event, which is processed when all current time events will be performed [7].

Fig. 1 shows a simple example of the SDL/SystemC co-model structure. This is an example, when two nodes interact with each other through the channel, but one node is implemented in SDL while another node and channel – in SystemC [11].

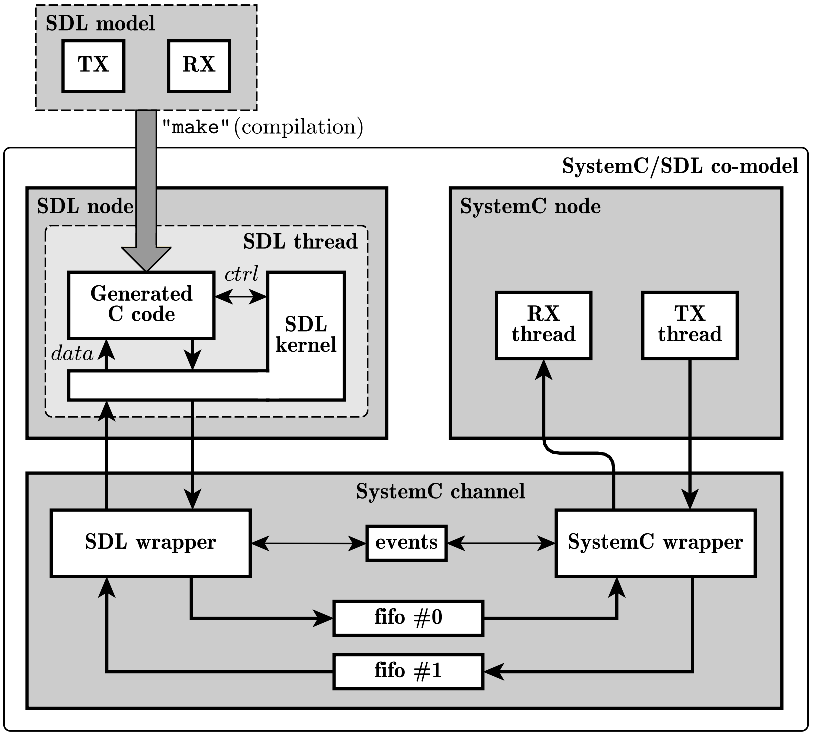


Fig. 1. SDL/SystemC co-modeling example

SDL/SystemC co-modeling approach can be successfully applied for validation of SDL models within the SystemC tester. In this case, the SDL model contains two instances of either a protocol layer or a protocol stack which work independently. Communication is performed through SystemC channels which can operate in accordance with different algorithms for error generation, signal loss, etc. SystemC test environment is a master component that fully controls the slave SDL system.

The implementation of the tester can be divided into the following general stages taking into account the general SDL/SystemC co-modeling principles:

1. Implementation of an SDL model of a protocol.
2. Implementation of special wrappers for conversion of SDL data types to the SystemC data types and vice versa.
3. Implementation of the SystemC test engine, channel for communication of the nodes and control components.
4. Writing test cases. Test cases are implemented as SystemC components, which work in accordance with different algorithms. By switching between these components, developers can change test scenarios.

The architecture of a co-model is shown in Fig. 2. It includes three main parts – SystemC test control components, an SDL part (SDL model and SDL simulation kernel) and SystemC channels. In Fig. 2 the SDL model is essentially the generated C code which implements the original graphic model.

SDL/SystemC co-modeling approach has been successfully used for validation and testing in such projects as UniPro [12] and SpaceWire-RT [13].

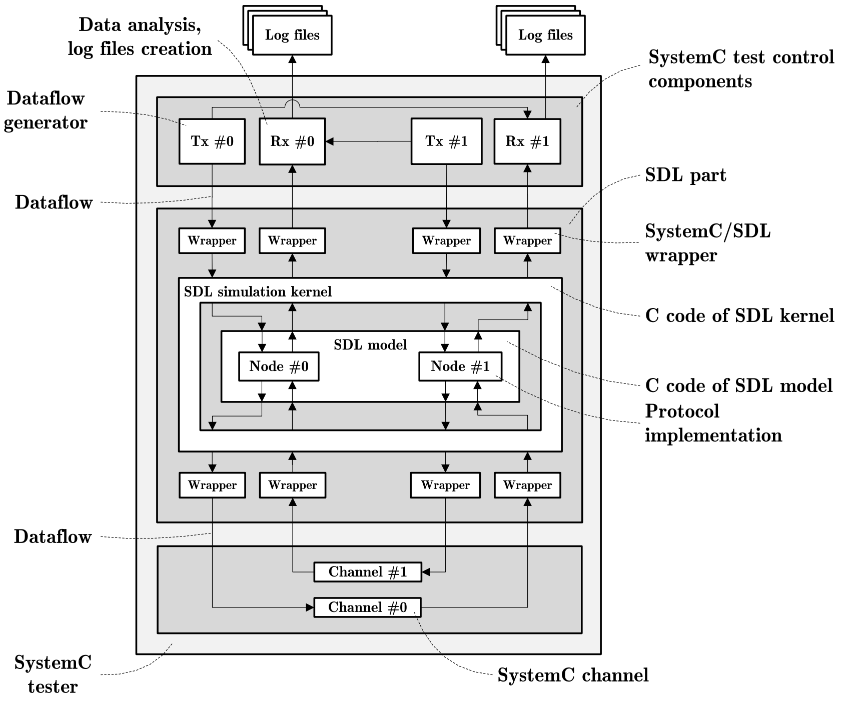


Fig. 2. Testing SDL under SystemC

4 Different Approaches to a Solution

Our goal is to develop an approach that allows creating several instances of the SDL model. Moreover, since we use the IBM Rational SDL Suite, our approach is tool specific. Especially for the SDL/SystemC co-modeling we use the CAdvanced code generator and C source code of the SDL model, which has been generated by it.

Since SystemC is based on C++ and since CAdvanced generates plain C code, there appears a task of combining C and C++ parts of the model. The use of available C++ code generators can probably solve some problems. But in the case of using our codebase, embedding a new tool requires making global changes to our projects.

We consider it is important to understand how the original SDL model is implemented in the C source code and, moreover, what are the source code equivalents for different elements of the SDL language. Another question is how we can reuse the code to have an opportunity to instantiate more than one SDL model. There are not many publications that describe the design of generation of source code from SDL and the principles of communication with an SDL simulation kernel. The paper [14] observes some details of this code generation and the principles of its functionality. Another publication [15] (a thesis) that proposes an integrated design flow for embedded systems, where the author also uses the CAdvanced code generator, describes several mechanisms that are used in SDL Simulator, and how the generated code interacts with the simulation kernel.

Based on the IBM Rational SDL Suite documentation together with [14, 15] we have conducted research in the design of model generated to find the ways for acheving our specific goal.

4.1 Integration of C Code into C++ Environment

We need to localize the SDL model instances in a memory. The most obvious approach is integration of the generated C code of the SDL model and the SDL simulation kernel into a C++ environment for its further operation in the user's code. If integration is possible, then the target library can be developed with the use of different OOP patterns. However, practical application of this approach has shown that this way entails a number of technical problems. Since the generated code of the SDL model is represented by C code that strictly conforms the ANSI C standard, the most complex problem is integration of the C code for operation in the C++ project. Consequently, the significant part of the SDL simulation kernel and generated code of the SDL model should be changed. Therefore, it can be concluded that the implementation of this approach takes a considerable time.

4.2 Code Post-processing

Another way of solving the problem is post-processing of the SDL model generated C code in order to have an opportunity of creating different numbers of

SDL model instances by the use of dynamic memory allocation. The main feature of the CAdvanced code generator is that the implementation of an SDL model represents the hierarchical structure that is called *symbol table* and organized as a tree [15]. The symbol table contains objects that represent SDL entities (system, blocks, processes, signals, etc). These objects are global variables, so static memory allocation is used.

To have an opportunity to create different numbers of SDL model instances, we need to change the memory allocation mechanism from static memory allocation to dynamic memory allocation. This can be done by the code post-processing. Ideally, we need to change the implementation of the CAdvanced code generator, but this is almost impossible as we use an existing industrial tool. On the other hand, it is a well known approach to develop an auxiliary toolchain for existing products.

5 An Approach of SDL Model Instances Integration

5.1 The Library Development Flow

The solution is aimed to develop an environment that allows creating the target library. This library provides an ability to use a different number of SDL model instances in the SystemC user's project and contains both the SDL model and the SDL simulation kernel. The library development flow and library usage in a project is shown in Fig. 3.

These are the steps of the proposed library development flow:

1. **Analysis of requirements** and **implementation of an SDL model**.
2. **Obtaining a PR-model** using the GR-to-PR converter.
3. **Obtaining C code** of the SDL model with use of CAdvanced. The code consists of three parts: a symbol table, which corresponds to the SDL model architecture, a set of initialization functions and a set of PAD (Process Activity Description) functions [10] which implement the behavior of SDL processes.
4. **Code post-processing** of the obtained C code. Generation of initialization functions and patching of some parts of PAD functions.
5. **Building a target library** according to the proposed approach. Creation of the symbol table selector. Development of a user's code interface, which is a set of C++ classes.

Then all the generated source code is compiled and linked, so the user gets a target library 'component.lib'. The implementation of the SDL kernel stays unchanged during the library development flow, but the new functionality for operating with a different number of SDL model instances is added. The user's project operates with the target library and the SystemC library simultaneously. The user interface is intended for using services provided by the library.

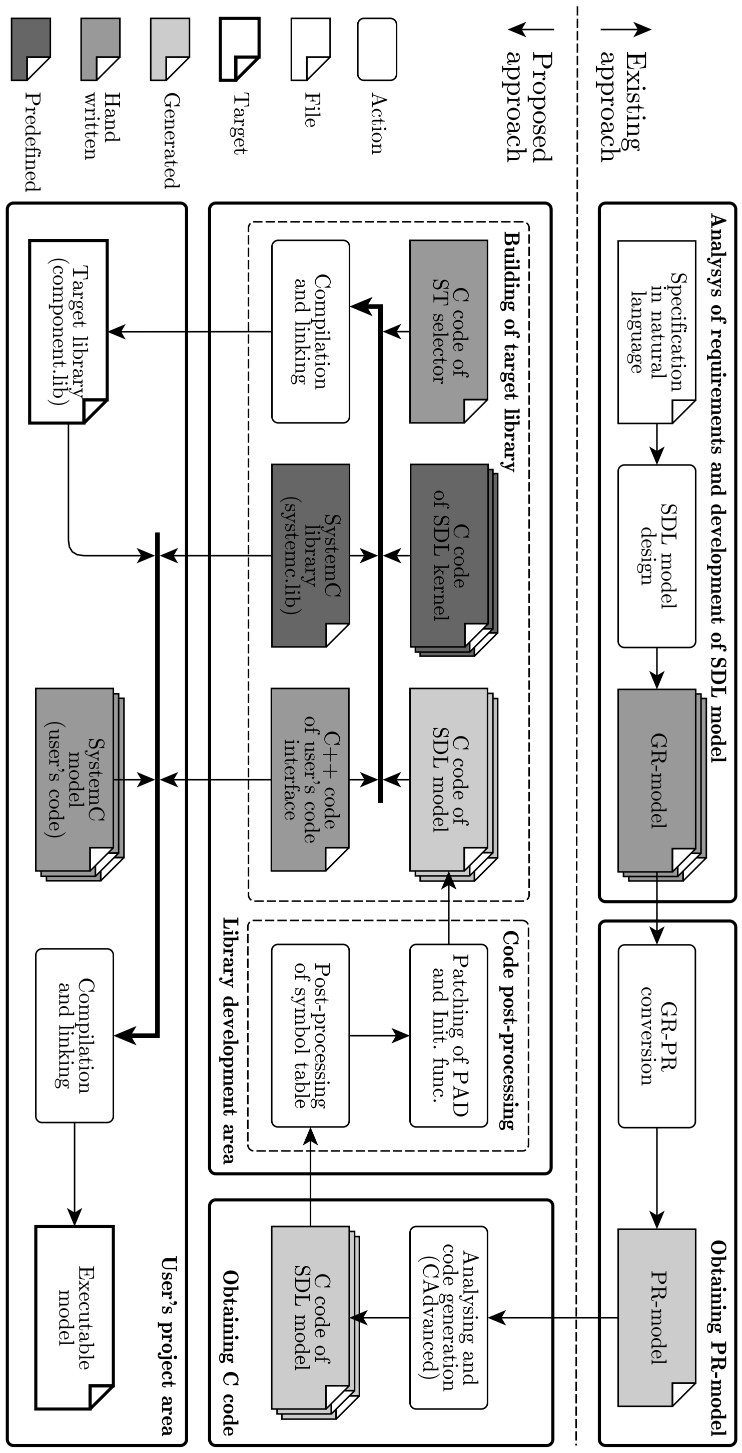


Fig. 3. Library development and usage

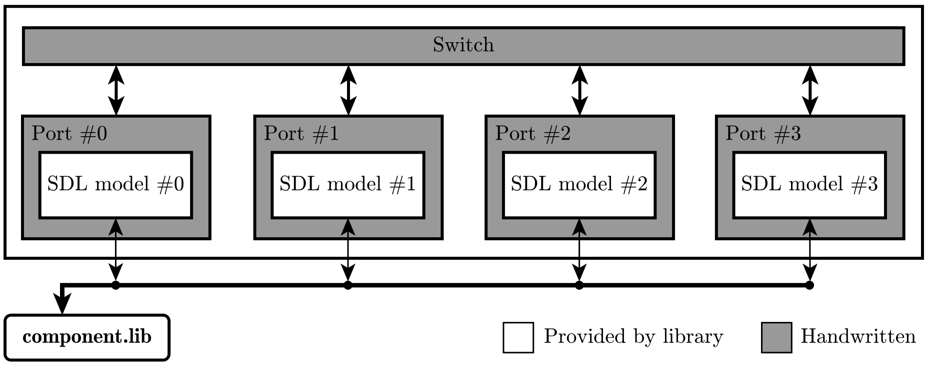


Fig. 4. SystemC model structure

5.2 Application Structure

Let us consider a simple example. Fig. 4 shows the architectural diagram of the SystemC model of SpaceWire MCK-01 switch [16].

The model contains a *Switch* module and four ports connected to four independent SDL model instances. The *Switch* and *Port* modules are implemented in SystemC. According to the proposed approach it is possible to design a switch model, where each port includes the implementation of full protocol stack in SDL. In this case the network layer is implemented in SystemC while the bottom ones – in SDL. The structure of the application implemented in accordance with the proposed approach is shown in Fig. 5.

It consists of the following parts:

1. **The SystemC library**, which includes SystemC kernel.
2. **The SystemC model** implemented by a user.
3. **The target library**, which provides an ability to create a number of different SDL model instances. The library is divided into three parts: the user's model interface, which describes the services for communication between the users SystemC model and the SDL kernel; the SDL kernel, which performs scheduling of the generated SDL model and the SDL model itself. For communication with C++ classes a basic `xInEnv/xOutEnv` [10] mechanism is used. Implementation of the SDL model has four parts:
   (a) A set of PAD functions. These functions implement behavior of SDL model processes.
   (b) A selector of a symbol table (ST).
   (c) A set of SDL model instances. Each of them has its own symbol table, but all instances have a set of common PAD functions.
   (d) A set of initialization functions. These functions are used for instantiation of each new symbol table with the use of the dynamic memory allocation.

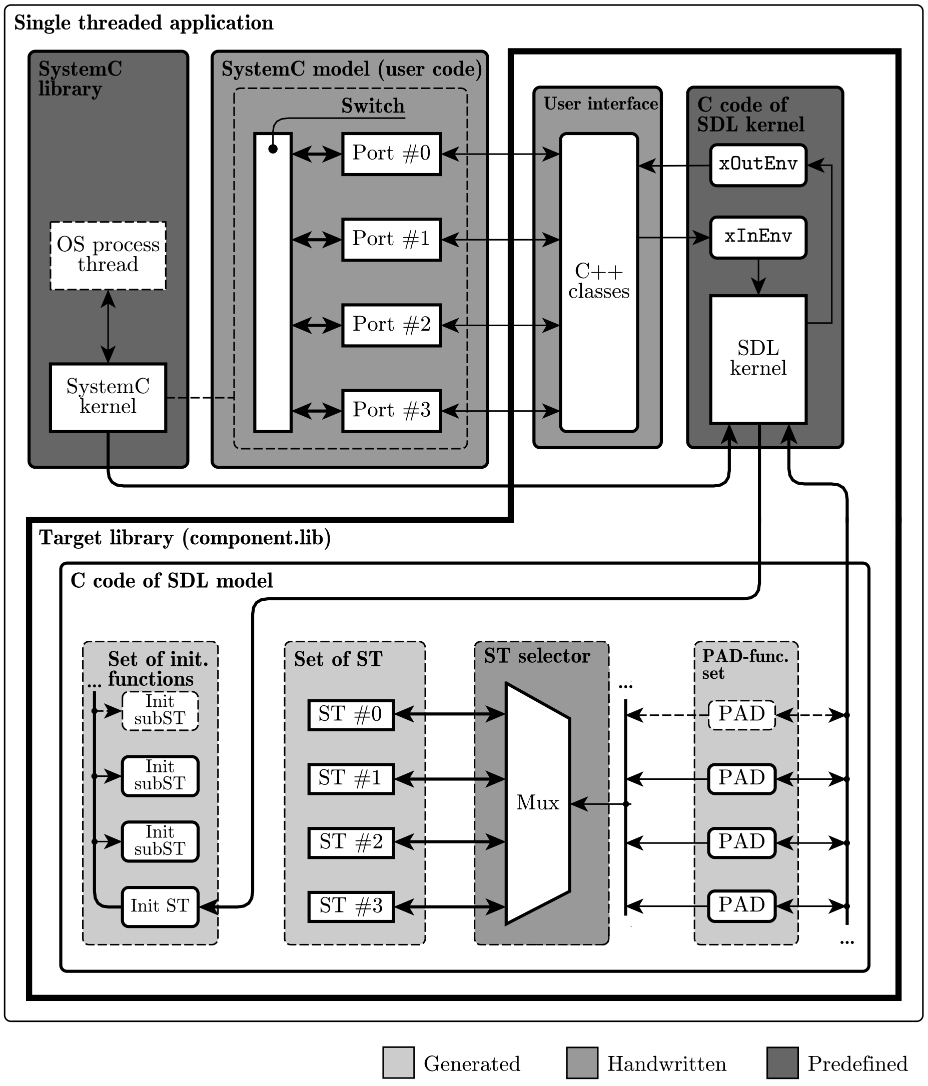


Fig. 5. Application structure

User's SystemC model is a single threaded application, which is controlled by the SystemC kernel. The Switch module with its ports communicates with the SDL kernel via user's interface. The SDL kernel is responsible for scheduling of SDL model processes. The kernel calls different PAD functions and each PAD function chooses an SDL model instance by means of ST selector. ST selector uses a symbol table identifier, which is generated by a user's C++ object which represents the SDL model (for example *Port #0*). The ST selector clearly indicates the required SDL model instance.

6 Main Principles of Model Memory Organization and Management

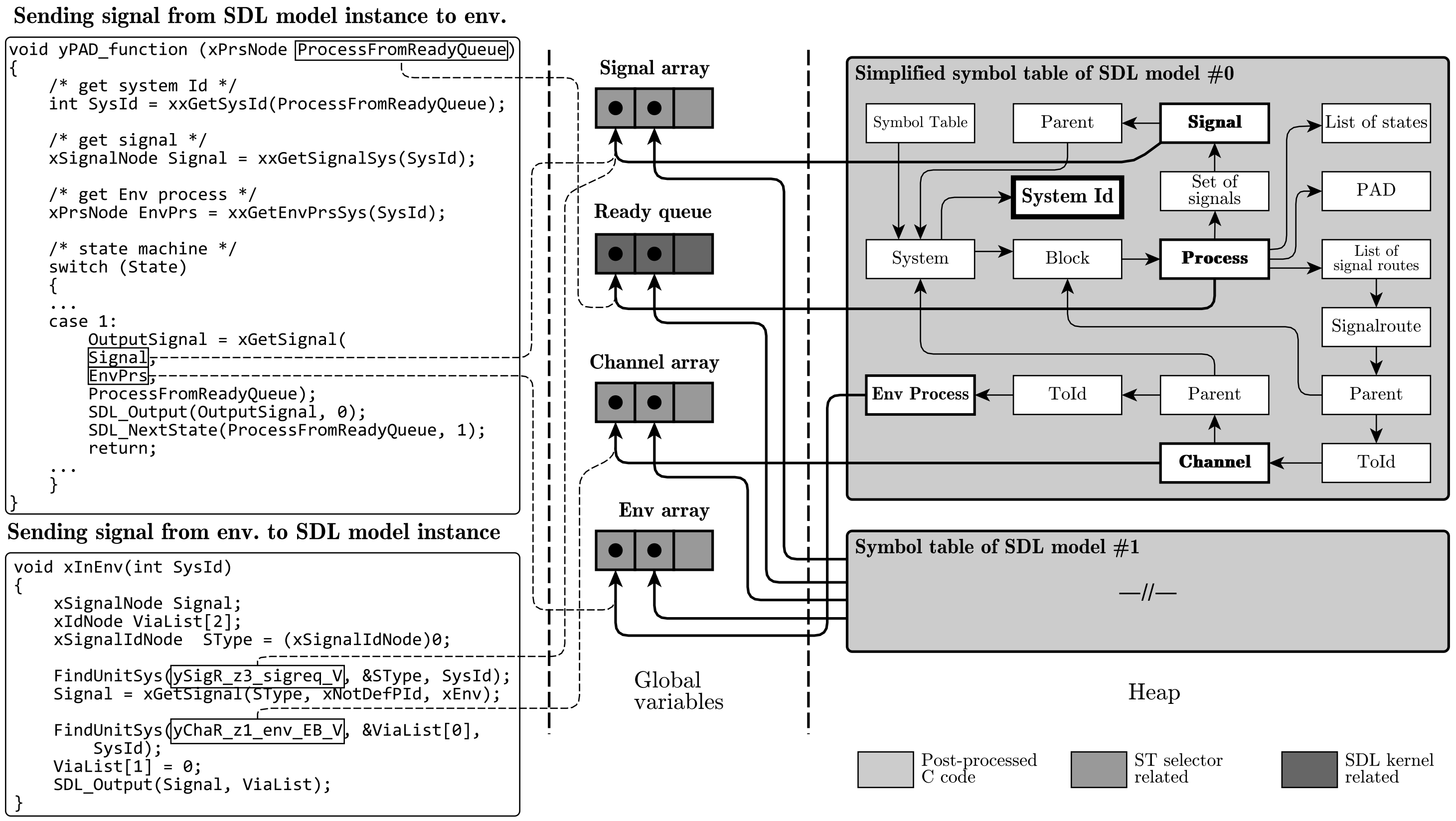
As an example we take the simple SDL model to clearly explain the proposed approach. The SDL model, which is taken as the basis for the example, consists of one block communicating with the environment by means of signals 'sig.req' and 'sig.rsp'. This block contains only one process, which operates with the same signals.

First we should generate the C code from the SDL model. It is done by means of CAdvanced code generator. This C code contains a set of interacting data structures and each of them could be a huge hierarchical tree. These structures represent the SDL model symbol table. Thereafter, it is possible to convert the generated C code to the XML. The XML representation contains 226 nodes (a node comprises a C data structure and its fields) with 251 connections between them for such a simple example.

According to the proposed approach the C code should be divided into a number of post-processing steps. Initialization functions were generated and PAD functions were patched. Initialization functions are called each time a new SDL model instance is initialized. In the case when the original SDL model was designed with use of packages, the CAdvanced generates an implementation of each package and places them into separated source file. Each package has its own initialization function. Therefore, we need to post- process all source files to have opportunity of build the symbol table in memory using dynamic memory allocation. After the initialization each SDL model instance is separately stored in the heap. Since initialization is performed with the use of the same function, which does not returns any value, it is not possible to get access to all symbol tables (as each new instance is initialised by a consecutive calling of initialization functions). To solve this problem some special nodes of the SDL model symbol table are added to special arrays. These arrays are used to determine the necessary nodes of symbol table of SDL model instance while sending signals from environment [1] to SDL model or sending signals from SDL model to an environment. Communication mechanisms are shown in Fig. 6.

The heap stores two symbol tables of the SDL model. A set of arrays, which are global variables, is stored in a data segment and contains pointers to signals, channels and environment processes, since every SDL model instance has its own environment. The SDL kernel extracts the first process from the *ready queue* [10] and calls associated PAD function.

A PAD function must obtain information about SDL model instance before it can send a signal to any process. It is done by using of a multiplexer, which is able to choose an instance depending on the information from arrays. The signal array, the channel array and the environment processes array are used for a required SDL model instance choice. Multiplexer does it by using traverse of hierarchical part of symbol table. The system identifier is stored on a system level of the hierarchy of SDL entities [1]. Such an identifier is associated with each new SDL model during the initialization stage.



Sending signal from SDL model instance to env.

```
void yPAD_function (xPrsNode ProcessFromReadyQueue)
{
    /* get system Id */
    int SysId = xxGetSysId(ProcessFromReadyQueue);

    /* get signal */
    xSignalNode Signal = xxGetSignalSys(SysId);

    /* get Env process */
    xPrsNode EnvPrs = xxGetEnvPrsSys(SysId);

    /* state machine */
    switch (State)
    {
    ...
    case 1:
        OutputSignal = xGetSignal(
        Signal,
        EnvPrs,
        ProcessFromReadyQueue);
        SDL_Output(OutputSignal, 0);
        SDL_NextState(ProcessFromReadyQueue, 1);
        return;
    ...
    }
}
```

Sending signal from env. to SDL model instance

```
void xInEnv(int SysId)
{
    xSignalNode Signal;
    xIdNode ViaList[2];
    xSignalIdNode  SType = (xSignalIdNode)0;

    FindUnitSys(ySigR_z3_sigreq_V, &SType, SysId);
    Signal = xGetSignal(SType, xNotDefPId, xEnv);

    FindUnitSys(yChaR_z1_env_EB_V, &ViaList[0],
        SysId);
    ViaList[1] = 0;
    SDL_Output(Signal, ViaList);
}
```

Fig. 6. Communication mechanisms

So during the execution of a PAD function and sending a signal to environment by `xOutEnv` function or to other process by `SDL_Output` function, it uses the multiplexer to determine a symbol table of the SDL model instance. When signal is sent from environment (from user's SystemC model) to SDL model instance by `xInEnv` function, it uses the multiplexer which performs search for a channel and a signal in arrays for identification of the SDL model instance.

7 An Example of the Approach Application

This example gives more details of the proposed approach and shows how the SystemC developer can use it in his project. Let us assume that we need to create a network model in SystemC and also we need to use it for an exploration of non-functional properties of a protocol while the SDL model of a protocol has already been implemented. To simplify the SDL model we use the same SDL model as we used in section 6. The example is shown in Fig. 7.

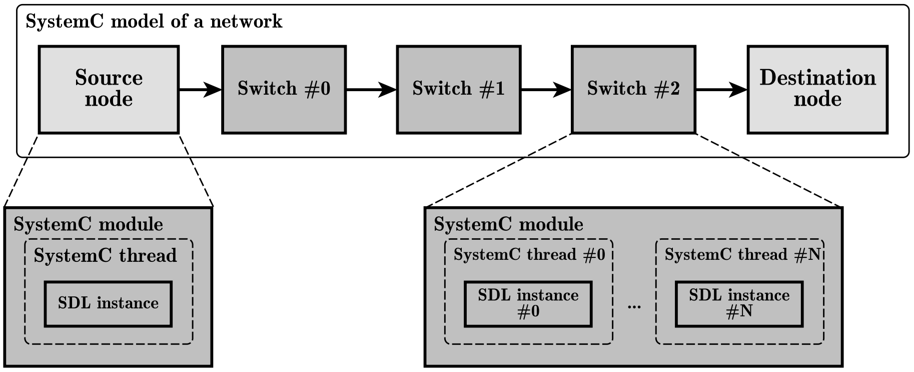


Fig. 7. Example of the approach application

The SystemC model includes the *Source* node, three switches and the *Destination* node. The *Source* node is responsible for data generation while the *Destination* node is responsible for its reception. Each switch contains the SystemC module, which includes a number of SystemC threads, each of which corresponds to an independent instance of the SDL model. All these instances are created by the user in C++. The library controls all of them.

A fragment of source code of the SystemC module of a *Switch* is shown in Listing 1.1. This fragment shows the part of source code of the `switch_module` class and `sdl_model` class. Constructor of the `switch_module` class is responsible for initialization of a module and creation of a corresponding new instance of the SDL model. The `sdl_model` class contains functions which form the user's interface and which are used in the `switch_module` class. The `sdl_model_thread` firstly waits for a request event. After the event has been generated the thread

```
/**** part of user's interface ****/
class sdl_model {
public:
    // initialization of the new instance
    sdl_model ();

    // function for sending sig.req to the instance
    void send_sig_reg ();
    ...
};

/**** part of user's code ****/
class switch_module : ... {
public:
    ...
    void sdl_model_thread ();
private:
    sc_event sig_req_event;
};

// switch module ctor
switch_module::switch_module(sc_module_name name) :
    sc_module(name){
    // thread creation and event setting
    SC_THREAD (sdl_model_thread);
    sensitive << sig_req_event;

    // creation of a new instance of the SDL model
    sdl_model_instance = new sdl_model;
    ...

}

// switch module thread
void switch_module::sdl_model_thread (){
    while(1){
        wait(sig_req_event); // waiting for input event

        // sending sig.req signal to the instance
        sdl_model_instance->send_sig_reg();
    }
}
```

Listing 1.1. Part of SystemC module source code

handles it and the signal is sent to SDL model instance using function call. The function is provided by the library and it is one of a set of functions of the user's interface. Therefore, the instance can be used in such a manner as if it is a SystemC component. Thus, SystemC developer can work with any instance of the SDL model of a protocol not knowing anything about its implementation.

The proposed approach gives an opportunity to focus on implementation of the SystemC model of a network comprising hundreds of nodes rather than on implementation of the SDL model of a protocol.

8 Conclusion

This paper gives an overview of the problem of integration of different numbers of SDL model instances into the SystemC project. The paper proposes and explains the elaborated approach. The SDL model is encapsulated inside a self-contained C++ class and could be easily instantiated. During instantiation of every new copy of the SDL model, it is placed on heap. For this opportunity, the generated source code of the SDL model has to be post-processed and the memory allocation mechanism should be changed from the original static memory allocation to a dynamic memory allocation. In addition, we provide an example of the successful application of the approach. The proposed approach is expected to reduce the project work effort and help in achieving a better quality of the simulation results. However, there is still a number of open questions and tasks for future work: definition of rules for code post-processing, memory management, proving of a model implementation and behavior correctness.

The future work would be mostly focused on the creation of a special tool. This tool is planned to be applied during the SpaceWire-RT standard model implementation and validation.

References

1. International Telecommunication Union: Recommendation Z.100 (12/11) Specification and Description Language - Overview of SDL-2010, `http://www.itu.int/rec/T-REC-Z.100/en`
2. Mitschele-Thiel, A.: Systems Engineering with SDL: Developing Performance-Critical Communication Systems. John Wiley & Sons (2001)
3. Institute of Electrical and Electronics Engineers: IEEE Standard for Standard SystemC Language Reference Manual, `http://standards.ieee.org/findstds/standard/1666-2011.html`
4. Black, D.C., et al.: SystemC: From the Ground Up. Springer (2010)
5. Grötker, T., Liao, S., Martin, G., Swan, S.: System Design with SystemC. Kluwer Academic (2002)
6. Turner, K.J.: Using Formal Description Techniques – An Introduction to Estelle, LOTOS and SDL. John Wiley & Sons I (1993)
7. Balandin, S., et al.: Co-Modeling of Embedded Networks Using SystemC and SDL. International Journal of Embedded and Real-Time Communication Systems 2(1), 24–49 (2011), `http://www.igi-global.com/article/modeling-embedded-networks-using-systemc/51648`

8. Gillet, M.: Hardware/software co-simulation for conformance testing of embedded networks. In: Finnish-Russian University Cooperation Program in Telecommunications seminar
9. Olenev, V., et al.: SystemC and SDL Co-Modelling Methods. In: Proceedings of 6th Seminar of Finnish-Russian University Cooperation in Telecommunications Program, pp. 136–140. State University of Aerospace Instrumentation (2009)
10. IBM Rational. IBM Rational SDL Suite User's Manual. IBM Rational (2009)
11. Stepanov, A., et al.: SystemC and SDL Co-Modelling Implementation. In: Proceedings of 7th Conference of Finnish-Russian University Cooperation in Telecommunications Program, pp. 130–137. State University of Aerospace Instrumentatio (2010), `http://www.fruct.org/publications/fruct7/files/Ste.pdf`
12. UniPro protocol stack by Mobile Industry Processor Interface Alliance, `http://mipi.org/specifications/unipro-specifications`
13. SpaceWire-RT project, `http://spacewire-rt.org`
14. Haroud, M., Biere, A.: SDL Versus C Equivalence Checking. In: Prinz, A., Reed, R., Reed, J. (eds.) SDL 2005. LNCS, vol. 3530, pp. 323–338. Springer, Heidelberg (2005)
15. Dietterle, D.: Efficient Protocol Design Flow for Embedded Systems. Brandenburg University of Technology (2009), `http://systems.ihp-microelectronics.com/uploads/downloads/diss_dietterle.pdf`
16. SpaceWire switch MCK-01, `http://multicore.ru/index.php?id=850`