Moazzam Fareed Niazi | Tiberiu Seceleanu |
Hannu Tenhunen

# Towards Reuse-based Development for the On-Chip Distributed SoC Architecture

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Technical Report
No 1020, October 2011

# Towards Reuse-based Development for the On-Chip Distributed SoC Architecture

Moazzam Fareed Niazi
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, FIN-20520 Turku, Finland
moazzam.niazi@utu.fi

Tiberiu Seceleanu
ABB Corporate Research, and
Mälardalen University
Västerås, Sweden
tiberiu.seceleanu@se.abb.com

Hannu Tenhunen
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, FIN-20520 Turku, Finland
hannu.tenhunen@utu.fi

TUCS Technical Report
No 1020, October 2011

## Abstract

The development of a reusable library of components for a multi-core segmented bus platform, the *SegBus*, is presented. The library is based on a plug-in that we develop and deploy within a modeling tool which eventually used by the *SegBus* DSL while developing applications targeting the *SegBus* platform. The steps required in building the library and embed it into a plug-in are discussed with the certain use of it within the design methodology.

**Keywords:** Platform-based design, reusability, computer-aided design, CAD, on-chip interconnection networks, modeling

**TUCS Laboratory**
Communication Systems

# 1 Introduction

Recently, the functional and structural complexities of embedded systems are increasing rapidly with the major developments in the fabrication technology, which in return imposing new challenges to existing design methods. Traditional design methodologies are no longer adequate to meet the challenges of complex system design. The ability to raise the design level, and adopt design reuse techniques becomes a key competence to cope with the trends in embedded system design. The time to market is another key challenge which also favors the reusability mechanism to minimize the design efforts.

Distributed on-chip architectures or multiprocessor system-on-chip (MPSoC) paradigm gains increasing support from system designers. MPSoC is seen as one of the foremost means through which performance gains are still to be sustained even after Moore's law may become decrepit [1]. The MPSoC paradigm we consider in this study is the *SegBus* platform [4].

Moreover, the *design productivity gap* with the MPSoC paradigm remains one of a key challenge with the existing design methodologies. This challenge can be addressed by developing new *computer-aided design* (CAD) tools/frameworks, based on newer design methods together with reusable elements within it. This will not only enable us to take full advantages from MPSoC platforms, but it will further satisfy other important factors e.g. *time to market*, *quality of result*, etc.

Design decisions, particularly at higher abstraction levels, are known to bear the most impact on the quality of the eventual system implementation. Optimality of design is strictly connected to platform parameters and employed devices/components. Hence, the right selection of components with further consideration of platform parameters at high abstraction levels will support an optimum solution answering several design requirements e.g. power consumption, speed, area, etc.

The approach we deliver in this report is our continual efforts towards establishing a design methodology for MPSoC, in the context of the *SegBus* platform. In our earlier work [5][6][7], we have already introduced a *Domain Specific Language* (DSL) and an emulator program for modeling, emulating and generating execution schedule for the applications targeting the *SegBus* platform. Until now, our methodology lacks a certain way of IP component selection while modeling at higher levels of abstraction, and whose (selected IPs) behavior could be observed during emulation of a modeled system. Similarly, we were compelled to manually select and map specific IPs during implementation phase. We address here issues for evolving our design methodology further towards reusability and versatility. We introduce a library compose of a list of functional components to be used within the design methodology. The components are often referred during different stages of the system modeling - from high-level modeling to low level code generation.

To achieve our goal, we thus build a plug-in and introduce it within the modeling tool [8] to get the intended library in graphical form together with DSL. The tool runs the plug-in every time it runs for modeling applications for our platform to provide ease and choice for component selection. The realization of *SegBus* component library is necessary because it enables us to model and emulate system more accurately due to provided additional information. This certainly makes the code generation process more straight forward and accurate.

Thompson et. al. proposed a highly automated framework titled as *Daedalus* for system-level architectural exploration, system-level synthesis, programming and prototyping of heterogeneous MPSoC platforms [2][3]. Their framework allows to construct MPSoCs platforms from a library of pre-defined and pre-verified IP components, similar in a aspect to our approach of building component library, but dissimilar in another aspect where we already have a platform - the *SegBus*.

## 1.1 Overview of the report

In the rest of the report, we proceed as follows. In section 2, we provide a short description of the *SegBus* platform, its associated *Domain Specific Language* (DSL) and emulator. Next, in section 3, we describe our design methodology which we use to design and implement applications on the platform. Furthermore, in section 4 we present, in detail, the steps required to build our proposed library and its deployment in the modeling tool. Finally, in section 5 we illustrate briefly the significance of the library with an example, followed by conclusion of the report in section 6.

# 2 Background

## 2.1 Segmented Bus Architecture

A segmented bus is a “collection” of individual buses (segments), interconnected with the use of FIFO like structures. Each segment acts as a normal bus between modules that are connected to it and operates in parallel with other segments. Neighboring segments can be dynamically connected to each other to establish a connection between modules located in different segments. Due to the segmentation of the bus lines, and their relative isolation, parallel transactions can take place, thus increasing the performance. A high level block diagram of the segmented bus system which we consider in the following sections is illustrated in Fig. 1.

The *SegBus* communication platform is built of components that provide the necessary separation of segments - *Border units* (***BU***), arbitration units - the *Central Arbiter* (***CA***) and local, *Segment Arbiters* (***SA***). The application then is realized with the support of (library available) *Functional Units* (***FU***).

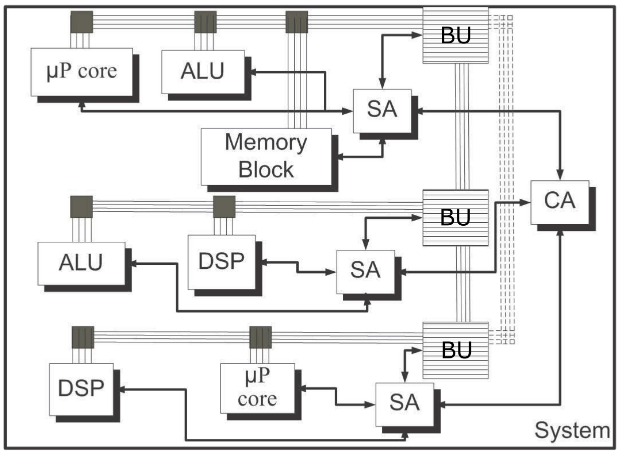


Figure 1: Segmented bus structure.

The *SegBus* platform has a single ***CA*** unit and several ***SA***s, one for each segment. The ***SA*** of each bus segment decides which device (***FU***), within the segment, will get access to the bus in the following transfer burst.
**Platform communication.** Within a segment, data transfers follow a "traditional" package based bus protocol, with ***SA***s arbitrating the access to local resources. The inter-segment communication, is also a package based, circuit switched approach, with the ***CA*** having the central role. The interface components between adjacent segments, the ***BU***s, are basically FIFO elements with some additional logic, controlled by the ***CA*** and the neighboring ***SA***s.

## 2.2 DSL for the SegBus Platform

The *Domain Specific Language* (DSL) for the *SegBus* platform is the specification language that is used to model the *SegBus* platform at higher-level of abstraction, based on stereotypes stored in the *SegBus* UML profile [5]. The DSL provides ability to model partitioned application components and platform elements in the form of high-level graphical constructs and provide methods to map application components on particular segment in a fast and correct manner.

The DSL comprises of a number of structural constraints related to the platform, written in *Object Constraint Language* (OCL) [15], to implement the correct component approach to platform design. These constraints are used to validate our models. Upon breach of any constraint requirement during the design process, the tool provides appropriate error message, so that the designer can take proper action to make the model correct according to platform requirements.

Once we model the application components as PSDF, model the platform and map the application components on to the platform correctly, we apply validation process to get the correct *Platform Specific Model* (PSM) of the application. If there exists some errors in the model, we get error message(s) and associated model element become highlighted.

Finally, the PSDF and PSM model can be transformed into XML schemes for further analysis of the desired platform configuration. We employ the generated XML schemes for emulating the performance aspects of the configured system, as described in the next section.

### 2.3 SegBus Emulator

The *SegBus Emulator* enables us to evaluate the performance aspects of any given application running on a specific platform configuration, defined during modeling [6]. The emulator supports the analysis of various *SegBus* instances that may answer, better or worse, to specific application requirements. It helps to decide at early stages of design process which platform configuration will be most suitable for any given application before moving towards lower abstraction levels. The code generation engine, supplied by the *MagicDraw UML* [8] tool transforms PSDF and PSM of the system into XML schemes. The generated XML schemes are then employed by the emulator program to estimate the utilization of platform elements with respect to data transfers and total execution time. After the analysis of the returned results, the designer is able to make decision at this stage whether the emulated configuration will be best/optimal or not, for the target application, and can change it before moving towards lower levels of the design process. After getting the desired platform configuration for a given application, the next step is to generate the execution schedule in the form of VHDL snippets, to be later used by the arbiters.

## 3 Design Methodology

Figure 2 illustrates a general overview of the *SegBus* design process. We consider as the start an application model (AM) which is transformed into a partitioned application model (PAM) with the help of available library components (described in section 4).

Taking from the designer parameters such as the number of segments, the (independent) *PlaceTool* provides support for an initial allocation of the application functional elements onto the platform. The allocation information, stored as a text file, serves as input into the modeling environment. Here, with the *SegBus* -UML profile support, the information is processed to obtain the segmented application model (SAM). A DSL based approach provides the (structural) correctness of the SAM.

The SAM further offers data for an emulator to simulate, at high levels of abstraction, the design. The emulator provides two kinds of information. Firstly, with visual representation, the performance of the SAM can be evaluated. Secondly, control code from the *control code generator* (CCG) is obtained as VHDL

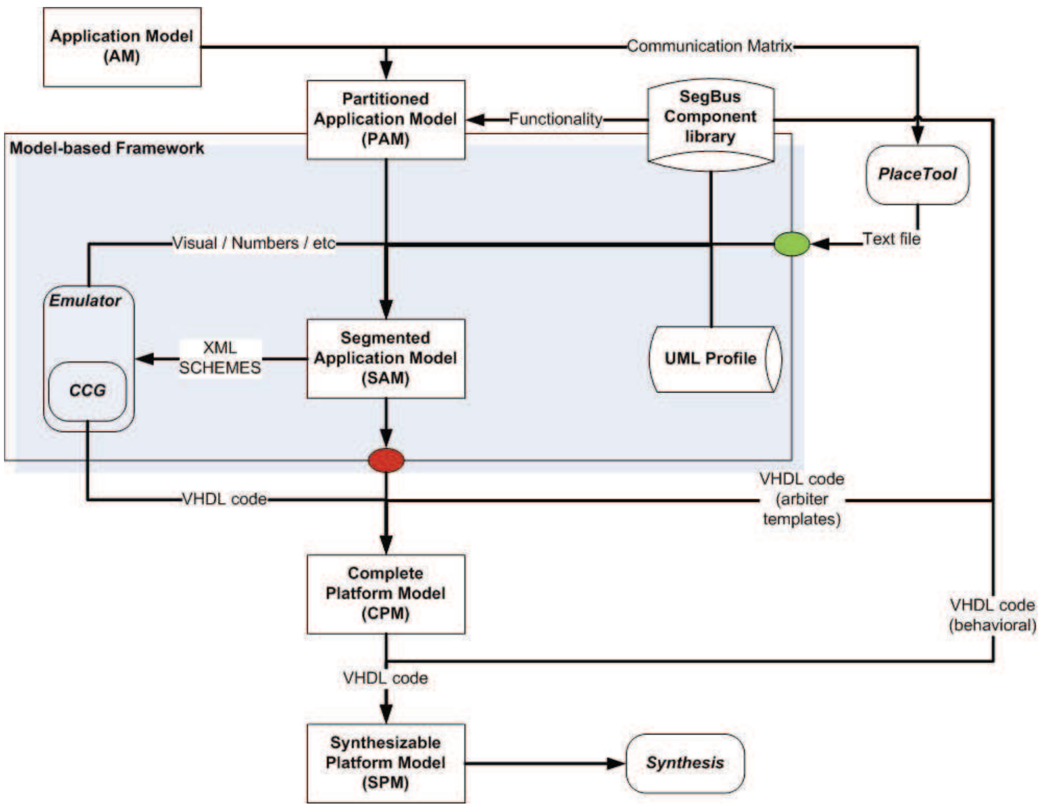


Figure 2: *SegBus* design process using the proposed framework.

snippets for the local and central arbiters. If the results from the simulation are not optimal, the designer is able to change the initial allocation of modules to segments again using the DSL. Subsequently, after a new simulation round, the new results are available.

The VHDL code for arbiters, the information from the chosen library components and the final allocation contained by the SAM are converged into the complete platform model (CPM). At this stage, we get the principle benefit of the proposed library which provides pointers to employ specific components which are listed in the library and stored at a particular location. From here, with the addition of the VHDL code for platform elements: functional units, specific platform units (border FIFOs, templates for arbiters, synchronizers, etc) we obtain the synthesizable platform model (SPM).

In the following sections we briefly pinpoint the focus of the developments of the *SegBus* component library.

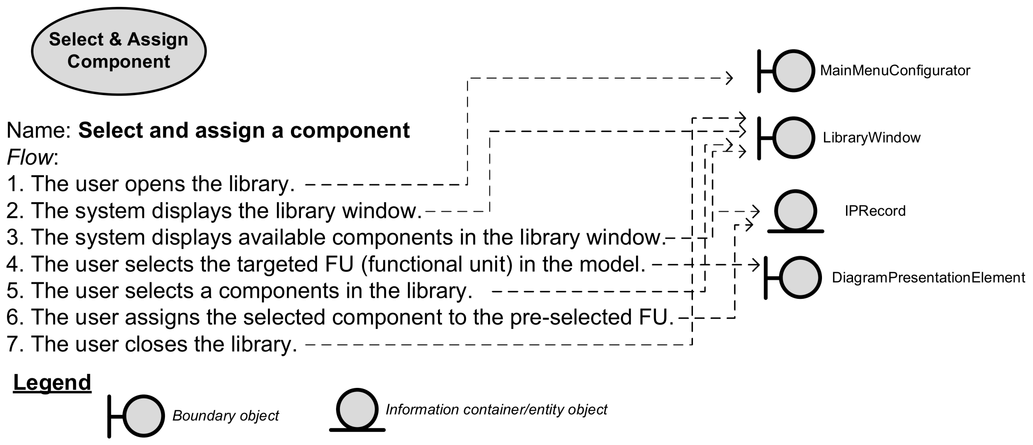

Select & Assign Component
Name: Select and assign a component
Flow:
1. The user opens the library.
2. The system displays the library window.
3. The system displays available components in the library window.
4. The user selects the targeted FU (functional unit) in the model.
5. The user selects a components in the library.
6. The user assigns the selected component to the pre-selected FU.
7. The user closes the library.
MainMenuConfigurator
LibraryWindow
IPRecord
DiagramPresentationElement
Legend
Boundary object
Information container/entity object

Figure 3: An analysis model of a “Select and assign component” use-case.

## 4 The *SegBus* Component Library

Reusability refers to a phenomenon where an object can be used more than once with or without minor modification. It reduces the development and implementation time and additionally enables us to handle the time-to-market challenge. Here, we build a library of often used components to make them reusable for further developments on the platform.

The development of the library starts by the *use case analysis*, which is the most common technique used to identify the system requirements that will ultimately helps us to design classes to satisfy the use cases. The requirements are the foundation on which the system is built. The ambiguous and incomplete requirements lead us to an incompetent system and the design efforts are suffered. We identify following important functional requirements that must be included in the library for a better use of it.

- The library can be opened in the tool.
- A new component can be added in the library.
- An existing component can be removed from the library.
- A component in the library can be selected and assigned to a (pre-selected) model element.
- The library can be closed.

The requirements accumulate to different cases which subsequently gear us towards a *use-case driven development* approach. A *scenario* is a sequence of steps describing an interaction between a user (system designer) and a system (the *SegBus* DSL in our case) [9]. Each requirement, then, evaluates to a number

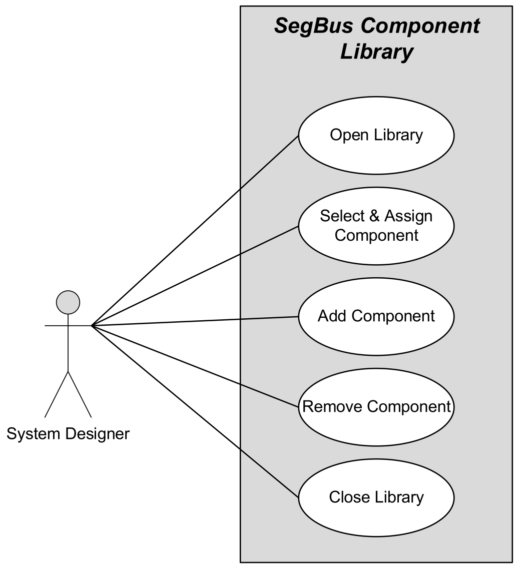


Figure 4: Use case model of the *SegBus* component library.

of scenarios. Further, a *use case* is a set of scenarios tied together by a common user goal. By examining the requirements, we come up with the a use-case model, as illustrated in figure 4.

Next, we build an *analysis model* based on the developed use-case model, which figures out the possible stereotypical classes (analysis objects) responsible for certain roles in the system. These roles are: the boundaries (interface with the outside world i.e. screens/forms), the entities (information container) and the control objects (coordinators of the use case execution) [10]. With analysis model, we are able to analyze a use case's flow with respect to analysis objects, which further makes it possible towards a robust system. Figure 3 illustrates a use case flow with analysis objects. The analysis models of other use cases build in similar fashion.

## 4.1 Implementation approach

A *class diagram* describes and represents the structure of the system. The description contains the types of objects a system is composed of, and the static relationships exist among them. Here, we develop a class diagram of the library on the basis of the developed use case analysis model, as discussed previously. Later, we implement the library based on the static structure as described in the class diagram. Figure 5 shows the class diagram of the *SegBus* component library.

Each class, in the shown figure, is responsible for a particular role within library's functionality. Below, we discuss individual classes and their role briefly:
• **IPRecord.** This class is a *java bean* - a class which allows access to its properties using dedicated *setter* and *getter* methods. An object of this class is used to

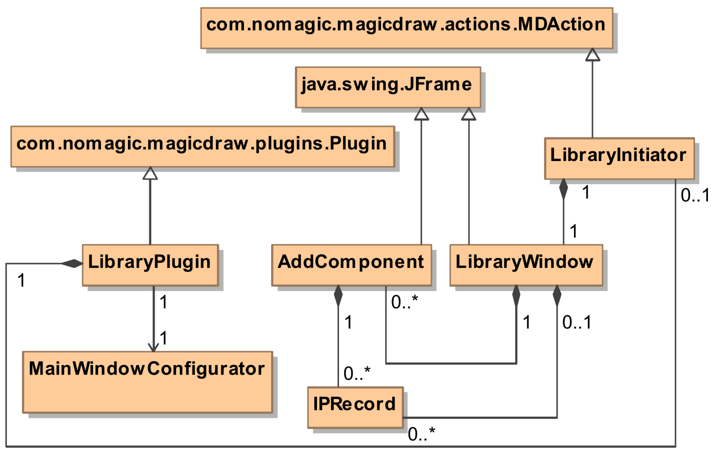


Figure 5: The class diagram of the *SegBus* component library.

hold important properties of a library component: name, feature size (technology), power consumption, storage location, etc.

• **JFrame.** The class is part of the "Swing" package of the Java language [11] and it is used to produce a top-level window with a title and a border.

• **LibraryWindow.** This class is directly extended from *JFrame* class of the Java standard API [11]. The class is a central *window* which holds important graphical components of the library. The window is the starting point where the available library components can be seen in a graphical form. Additionally, the class provides capability to add, remove and assign the library components to model's element(s).

• **AddComponent.** This class is also directly extended from *JFrame* class of the Java standard API like the previous one. The class provide a graphical window which holds other graphical components (text fields, etc.) used to add new components in the *SegBus* component library.

• **Plugin.** The *open API* of the MagicDraw tool [8] is a collection of classes which allows us to write our own plug-ins, create actions in the menus and toolbars, change UML model elements, etc. The *Plugin* class also from the open API and is the base abstract class for any plug-in of the tool. The plug-in under development must be extended from this class. Every plug-in has it own descriptor (discussed in section 4.2).

• **LibraryPlugin.** The class is directly extended from "com.nomagic.magicdraw.-plugins.Plugin" class of the open API. The class contains overridden methods: *init()*, *isSupported()* and *close()*. The initialization method (init) initializes the plug-in and registers the action it intends to perform. Similarly, the "isSupported" method is used to provide information about compliance of the plug-in with other versions of the tool. Lastly, the close method is used to mention specific actions we intend to perform while closing the plug-in.

• **MainWindowConfigurator.** The *action manager* of the tool is responsible for managing the actions and categories. It has a list of categories where it registers different actions by shortcuts and ids. The *MainWindowConfigurator* class implements *AMConfigurator* interface from the open API, which in turn, is used to configure the action manager. The class overrides a method named as *configure()* which provides information to action manager about our plug-in. We create, with the help of this class, a separate menu for the library in the tool's main menu bar.
• **MDAction.** The class belongs to the "com.nomagic.magicdraw.actions" package of the open API. It provides methods which can be overridden to offer functionality whenever a related action is occurred within the tool.
• **LibraryInitiator.** This class extends from the "MDAction" class. The class overrides a "actionPerformed" method to instantiate the "LibraryWindow" object whenever a designer chooses the library to run in the menu bar.

## 4.2 Plug-in setup

Plug-ins are the only way to add or change functionality to the MagicDraw tool [12]. We shape up the *SegBus* component library in a plug-in form such that it can be effectively utilized as a pool of reusable IP components by the tool.

A plug-in must contain: a directory, compiled Java files packages into *jar* files, a plug-in descriptor file and optional files to be used by the plug-in. We use Apache Ant [13] to compile the source code of the library together with the provided class library of the tool, and further package it into a *jar* file (described below). Following, we show the script, which we use to compile and package the source code into a *jar* file.

```
<project name="SBLibrary" basedir="." default="main">
  <property name="lib.dir"     value="../../lib"/>
  <path id="classpath">
    <fileset dir="${lib.dir}" includes="**/*.jar"/>
  </path>
  <target name="clean">
      <delete dir="build"/>
  </target>
  <target name="compile">
      <mkdir dir="build/classes"/>
      <javac srcdir="src" destdir="build/classes"
             classpathref="classpath"/>
  </target>
  <target name="jar">
      <mkdir dir="build/jar"/>
      <jar destfile="build/jar/SBLibrary.jar"
           basedir="build/classes">
          <manifest>
              <attribute name="Main-Class"
                         value="SBLibrary.LibraryPlugin"/>
          </manifest>
      </jar>
  </target>
</project>
```

The tool, on every startup, scans plug-ins directory and look for further sub-directories. If a sub-directory contains a plug-in descriptor file (named as "*plugin.xml*"), then the *plug-in manager* of the tool reads it. Next, if requirements specified in the descriptor file are satisfied, then the plug-in manager execute the specified class by calling its *init()* method. Therefore, the specified class (*LibraryPlugin* in our case) must be derived from "com.nomagic.magicdraw.plugins.Plugin" class in order to successfully get triggered by the plug-in manager.

We thus perform all the necessary steps to make up the appearance of the library as an executable plug-in. To achieve this, we create a directory (*$(MagicDraw)\plugins\SBLibrary*), compile and build a packaged *jar* file (*$(SBLibrary)\build\jar\SBLibrary.jar*) based on its general principles and write a descriptor file. The content of the descriptor file are shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
 <plugin id="SegBus.Library.Plugin"
        name="SegBus IP Library Plugin"
        version="1.0" provider-name="Moazzam"
        class="SBLibrary.LibraryPlugin">

 <requires>
   <api version="1.0"/>
 </requires>

 <runtime>
   <library name="build/jar/SBLibrary.jar"/>
 </runtime>
</plugin>
```

Interested readers can find out the detailed semantics about various elements and related attributes used in the above descriptor file in "*Open API* user guide" [12].

## 5 Example use of the component library

In this section, we demonstrate the effective use of the presented *SegBus* component library at a specific phase of the design process. As described in section 3, we initially partition a given target application on the basis of available components in the proposed library, and later after building a SAM model of the application employing the *SegBus* DSL and before moving towards building the CPM model, we again use the *SegBus* component library and assign particular library elements to respective model elements of the SAM in a graphical manner. Figure 7 shows the main library window after it has been invoked from a dedicated menu in the tool.

New components can also be added into the library with ease. The library plug-in provides a separate window where we get facility to add new components. The window can be invoked from the main library window by pressing "Add" button. Figure 8 depicts this window where we provide information about the new library component to be added.

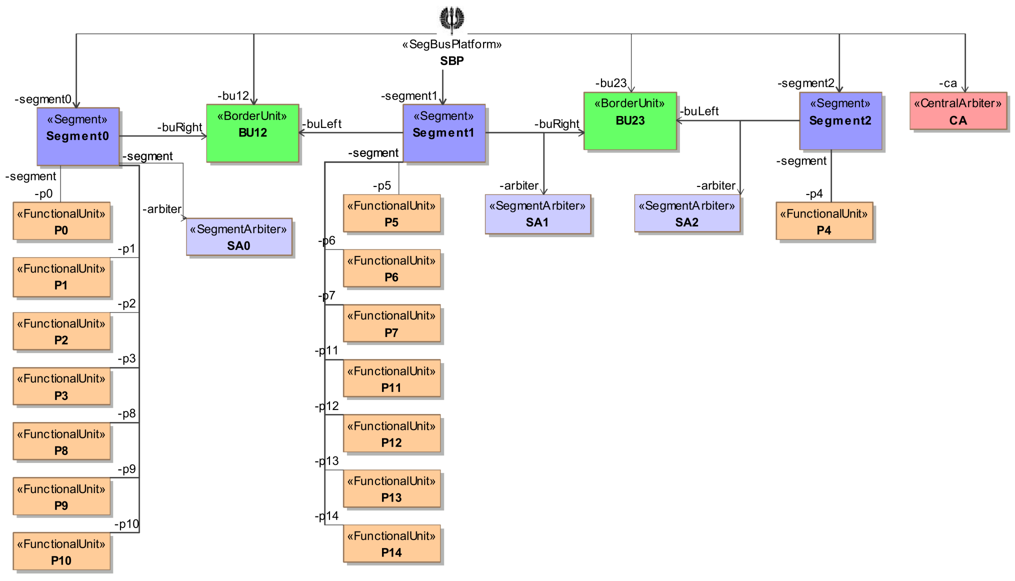
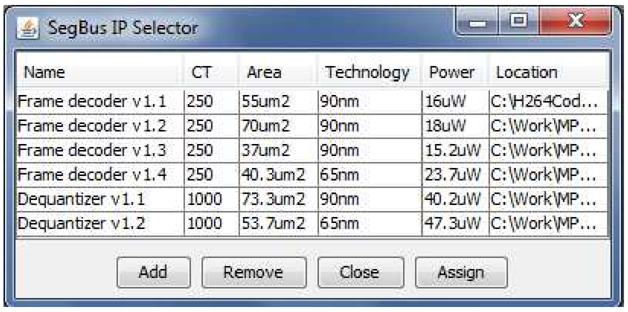


Figure 6: The SAM model of the example application in 3 segments - linear topology configuration.



| Name | CT | Area | Technology | Power | Location |
|---|---|---|---|---|---|
| Frame decoder v1.1 | 250 | 55um2 | 90nm | 16uW | C:\H264Cod... |
| Frame decoder v1.2 | 250 | 70um2 | 90nm | 18uW | C:\Work\MP... |
| Frame decoder v1.3 | 250 | 37um2 | 90nm | 15.2uW | C:\Work\MP... |
| Frame decoder v1.4 | 250 | 40.3um2 | 65nm | 23.7uW | C:\Work\MP... |
| Dequantizer v1.1 | 1000 | 73.3um2 | 90nm | 40.2uW | C:\Work\MP... |
| Dequantizer v1.2 | 1000 | 53.7um2 | 65nm | 47.3uW | C:\Work\MP... |

Figure 7: The main window of the *SegBus* component library.

Next, we show the main role of the library during modeling an application which targets the *SegBus* platform. Figure 6 shows the SAM model of the (simplified) MP3 decoder (layer III) [14] as per the described methods in [5]. We select a model element in the SAM model, for instance process P9. We then open the library of components, select a particular component from the list of library components and assign it to the pre-selected model element by pressing the "Assign" button in the library's main window. This will transfer the information about the library component to the model element (process P9 in this case). Figure 9 shows the process P9 after post-selection process from the library.

The component selection from the library and their assignment to model elements shifts important design information to CPM which ultimately allows us to build the intended right system according to supplied design parameters and constraints.

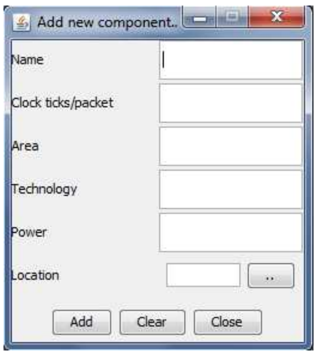
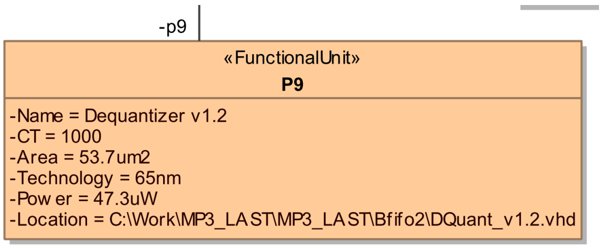


Figure 8: A window for adding components into the library.



-Name = Dequantizer v1.2
-CT = 1000
-Area = 53.7um2
-Technology = 65nm
-Pow er = 47.3uW
-Location = C:\Work\MP3_LAST\MP3_LAST\Bfifo2\DQuant_v1.2.vhd

Figure 9: A functional unit after being assigned a component from the library.

## 6 Conclusions

We have presented a technique for designing applications targeting the *SegBus* platform with the help of a collection of reusable hardware/software components stored in centralized library. The technique enables us to tackle evolving challenges with the current trends in embedded system development by implementing the phenomenon of reusability within the design process.

## References

[1] *International Technology Roadmap for Semiconductors.* 2007 Edition.

[2] M. Thompson, T. Stefanov, H. Nikolov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere. *A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs.* In proceedings of 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2007, pp. 9-14.

[3] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, E. Deprettere. *Daedalus: Toward composable multimedia MP-SoC design.* In proceedings of 45th ACM/IEEE Design Automation Conference (DAC), 2008, pp. 574-579.

[4] T. Seceleanu. *The SegBus Platform - Architecture and Communication Mechanisms.* Journal of Systems Architecture, Vol. 53, Issue 4, April 2007, pp. 151-169.

[5] M. F. Niazi, K. Latif, T. Seceleanu, H. Tenhunen. *A DSL for the SegBus Platform.* The 22nd IEEE International System-on-Chip Conference (SOCC), 2009, pp. 393-398.

[6] M. F. Niazi, T. Seceleanu, H. Tenhunen. *A Performance Estimation Technique for the SegBus Distributed Architecture.* The 39th International Conference on Parallel Processing Workshops (ICPPW), 2010, pp. 89-98.

[7] M. F. Niazi, T. Seceleanu, H. Tenhunen. *An Automated Control Code Generation Approach for the SegBus Platform.* The 23rd IEEE International System-on-Chip Conference (SOCC), 2010, pp. 199-204.

[8] MagicDraw UML. http://www.magicdraw.com

[9] M. Fowler and K. Scott. *UML Distilled. Second Edition* Addison-Wesley, ISBN: 020165783X, 2002.

[10] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process* Addison-Wesley Professional, 1999.

[11] Java Platform. http://www.oracle.com/technetwork/java/index.html

[12] MagicDraw Open API user guide, version 17.0. http://www.magicdraw.com

[13] Apache Ant$^{\text{TM}}$. http://ant.apache.org/

[14] C. Park, J. Jang and S. Ha. *Extended Synchronous Dataflow for Efficient DSP System Prototyping.* Journal Design Automation for Embedded Systems, Springer Netherlands, vol. 6, no. 3, 2002, pp. 295-322.

[15] OMG. *Object Constraint Language (OCL) 2.0 Revised Submission, version 1.6.* Jan. 2003.

ISBN 978-952-12-2644-1
ISSN 1239-1891