Public

FP7-ICT-2009- 4 (247999)  COMPLEX

COdesign and power Management in PLatform-based design space EXploration

Project Duration 2009-12-01 – 2013-03-31 Type IP

<table>
<thead>
<tr>
<th>WP no.</th>
<th>Deliverable no.</th>
<th>Lead participant</th>
</tr>
</thead>
<tbody>
<tr>
<td>WP2</td>
<td>D2.5.3</td>
<td>MDS</td>
</tr>
</tbody>
</table>

Final report and tools on virtual system generation

Prepared by Emmanuel Vaumorin (MDS), Bart Vanthournout (SNPS), Sara Bocchio (ST-I), Davide Quaglia (EDALab), Fernando Herrera, Pablo Peñil del Campo, Eugenio Villar (UC), Kai Hylla, Tiemo Fandrey, Philipp A. Hartmann, Kim Grüttner (OFFIS)

Issued by MDS

Document Number/Rev. COMPLEX/MDS/R/D2.5.3/1.1

Classification COMPLEX Public

Submission Date 2012-12-12

Due Date 2012-09-30

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)


This document may be copied freely for use in the public domain. Sections of it may be copied provided that acknowledgement is given of this original work. No responsibility is assumed by COMPLEX or its members for any application or design, nor for any infringements of patents or rights of others which may result from the use of this document.
History of Changes

<table>
<thead>
<tr>
<th>ED.</th>
<th>REV.</th>
<th>DATE</th>
<th>PAGES</th>
<th>REASON FOR CHANGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV</td>
<td>0.1</td>
<td>02/10/2012</td>
<td></td>
<td>Initial version</td>
</tr>
<tr>
<td>EV</td>
<td>0.2</td>
<td>15/11/2012</td>
<td></td>
<td>Integrating partners new inputs (OFFIS, UC, MDS)</td>
</tr>
<tr>
<td>PHa</td>
<td>1.0</td>
<td>12/12/2012</td>
<td>67</td>
<td>Initial full version</td>
</tr>
<tr>
<td>KG</td>
<td>1.1</td>
<td>13/05/10</td>
<td>53</td>
<td>Assembled public version</td>
</tr>
</tbody>
</table>
Contents

1 Introduction ........................................................................................................... 4
2 System generation tool flow overview ................................................................. 6
3 Input blocks ........................................................................................................... 11
  3.1 SystemC executable model ........................................................................... 11
    3.1.1 SystemC generated from UML/MARTE and Matlab/Stateflow ............. 11
    3.1.2 Generation from RTL models ................................................................. 12
    3.1.3 User-defined SystemC executable model ............................................. 12
3.2 IP-XACT platform definition .......................................................................... 15
  3.2.1 Positioning IP-XACT platform in the COMPLEX flow .......................... 15
3.3 Hardware/Software Partitioning and Mapping .............................................. 21
  3.3.1 Mapping format description .................................................................... 21
  3.3.2 Integration of the environment ................................................................ 25
3.4 Virtual platform IP models ............................................................................. 27
4 Augmented TLM2 platform generation ............................................................... 31
  4.1 Task separation and augmented model generation ...................................... 31
    4.1.1 Augmented hardware model ................................................................ 33
    4.1.2 Augmented software model ................................................................ 35
  4.2 TLM2 interface synthesis ............................................................................. 36
    4.2.1 Hardware targets .................................................................................. 37
    4.2.2 Software initiators ................................................................................ 40
    4.2.3 Adaptor functions ................................................................................. 41
4.3 Overall interface synthesis and augmented model generation flow ............... 45
5 MP3 player example ............................................................................................ 47
6 References ............................................................................................................ 53
1 Introduction

This deliverable is used to reach the following milestone:

**M2.4 - Virtual system generation at M34: The different simulation technologies and performance analysis tools are integrated into a unified virtual system simulation framework**

The work of this deliverable combines the results from T2.2, T2.3, and T2.4. The goal of this deliverable is to allow the generation of an executable, multi-level, self-simulating system description, predicting the system behaviour under the use-case defined workload models with up-to cycle and/or instruction accuracy.

Based on this work, a Virtual System Generator has been developed able to read the timing and power instrumented C/C++ code for SW and user-defined HW components, which is the output of the tools developed in T2.2 and T2.4. From T2.3 it obtains virtual component implementations of memories, interconnection and pre-designed dedicated HW components.

The result is a unified simulation and performance analysis framework.

The virtual platform simulations issued from T2.5 (in combination with T2.2) are enhanced in T3.1 with analysis and profiling tools so that a design space exploration loop becomes feasible: the result of the Virtual System simulation is system timing and a power over time chart broken down into individual HW and SW tasks. Additional information, such as memory footprint and HW complexity will be provided also.

The following figure illustrates the global flow addressed by COMPLEX and how we get inputs to generate the Virtual System. The main purpose of this document is to detail the work related to the Virtual System generation using SystemC executable model, HW/SW mapping and Platform description & IP models.

![Diagram](image)

Figure 1: Top-level view on the system generation flow
It is important to notice, that the virtual platform which represents the hardware in the virtual system is subject to be used at a high or low level. In the T2.1, the virtual platform represents the hardware at a high level in order to perform power estimation. In this document, the purpose is to study how to generate the virtual platform at a low level. The D3.4.2 presents a consolidated view of these several levels.

This document is organized as follows:

- The following Section 2 gives an overview on the general steps involved in generating a virtual system.
- Section 3 presents which input data are required to enter the generation flow
- Section 4 presents the separation in tasks of the application for further mapping
- Finally, the remaining section describes an MP3 player design which is used as proof of concept for the system generation flow. This preparatory design comes closest to the scenario in use case 2. Further detailed “customized flows” for each use case are not part of this document.
2 System generation tool flow overview

The generation of a virtual system takes various steps. Several steps in the virtual system generation are interchangeable. Other steps might involve the integration of different tools or not. Some steps might be automated or not. The definition of all these aspects depends on the selection for the initial input description, on the target platform which is aimed at, and on the performance estimation tools and techniques involved. Of course, such a configuration heavily depends on purpose given to the generated virtual platform.

The COMPLEX flow enables the integration of different front-ends, which enable different ways to capture the model, and different simulation and performance estimation technologies, to play with the trade-off between simulation speed and accuracy enabled by the generated virtual system. For instance, an executable model written in SystemC could already exist, written by the user, or it might need to be generated from MARTE/UML or, for example, the virtual platform can use a pre-synthesis or a post-synthesis estimation technique.

This document explains how the COMPLEX tooling involved in the generation of a SystemC-based virtual system is integrated. The document explains the mains aspects of the integration which appear when traversing the global view of the virtual system generation, according to the general complex flow.

Figure 2: Generic view of system generation front-end, task separation and interface synthesis.

Such global view is shown in Figure 2, which is a simplified excerpt of the COMPLEX Framework, shown in Figure 5 of the DoW. Figure 2 shows that the generation of the virtual system has to take into account main aspects, such as the front-end, the task separation, the
instrumentation of the source code for SW and HW performance analysis, and the TLM2 interface synthesis.

Figure 2 shows the minimum set of pre-requisites for the generation of a virtual system in the COMPLEX framework. A first pre-requisite regards the input front-end: at least a SystemC executable specification should be available. Moreover, COMPLEX framework enables the possibility of a higher-level input from UML/MARTE and Stateflow models, which, in a latter term, can be converted into a SystemC executable. Second, the specification code that should be implemented either in HW or SW has to be separated from the executable model based on mapping information. Mapping information can be either provided by the user or automatically automated from the high-level input. Third, the separated HW and SW blocks must have been processed by the corresponding estimation tools, generating instrumented versions of the executable models. Fourth, for those parts of the system which are not obtained from the task separation, appropriate (TLM2) IP models must be available.

When the preconditions listed above are satisfied, the building blocks for the virtual system are available. In order to combine those building blocks to form a virtual system, the system generation must perform several operations. First, it must gather the input blocks for the system, i.e. the instrumented models that stem from the estimation of the separated tasks and the virtual platform IP models. Second, TLM interfaces must be synthesised to create wrappers for the instrumented components. These wrappers allow an integration of the varying components into a TLM 2.0 environment. Third, the TLM 2.0 wrappers for the HW/SW components must be interconnected with the IP models.

The result of the system generation is an executable virtual SystemC system description with all instrumented components interconnected via TLM 2.0 interfaces. This system description can be further refined and mapped to specific virtual platforms by corresponding back-ends. For instance, in the specific flow applied in use case two, two back-ends will be exerted in order to map the virtual system to two distinct refined virtual platforms. The first target platform is the SNPS virtual platform. Extended SNPS platform models will need to be instantiated to compose the system for this target. The second target platform is the ST-I platform. In order to map to this target platform, the ReISC microprocessor running FreeRTOS and the peripherals associated to this architecture must be emplaced.

How these integration aspects (front-end and back-end, task separation, integration of IP blocks, virtual system generation) are solved in different ways, they might be encrusted in a single tool or spread among different ones. Specifically, Figure 3 and Figure 4 sketch the integration solutions covered in COMPLEX, relying in use case 2 and 3 flows.

Figure 3 sketches the flow for the generation of the host-based simulation used in use case 2. In this case, the generation of an augmented virtual system is directly fed with a SystemC executable model (see Section 3.1 for details). Furthermore, mapping information for task separation and a platform definition for mapping to a target platform is provided. In case the executable model is specified by the user like in use case 2, the mapping information and the platform definition is added manually.

In this generation flow, the SMOG tool (section 4) is in charge of task separation and virtual generation tasks, including the synthesis of the TLM interfaces that are needed to interconnect the system components, and specifically the TLM2 IP components. SWAT tool is in charge of SW estimation, while HW estimation is performed through PowerOpt+. Therefore this integration enables a performance simulation based on native simulation of software and post-synthesis custom HW estimation.
Figure 3: System generation front-end, task separation and interface synthesis in the use case 2.

Figure 4 shows how the integration is solved in the use case 3. In this case, a significant part of the integration aspects are encrusted in the SCoPE+ framework.

Figure 4: System generation front-end, task separation and interface synthesis in the use case 3.
Figure 4 sketches the integration designed and developed for use case 3, in order to enable a (Eclipse-based) graphical front-end which enables the user both, the UML/MARTE capture of the model (reported in D2.1.1), and an automatic generation of the virtual system. The same graphical front-end provides a set of generators (violet arrows) (reported in D2.1.2), which automate the extraction of a set of files. These files contain the information of the UML/MARTE model required for the generation of the executable virtual system in convenient text formats, which are readable by the SCoPE+ framework (after the installation of some specific plugins for enabling the CFACM, the XML and the IP/XACT front-ends). Specifically, the CFACM front end enables to feed SCoPE+ with a platform independent model (PIM) which captures the system functionality encapsulated within components (see D2.1.1). CFAMCM code is extracted from the UML/MARTE model through the Marte2cfam generator (see D2.1.2). The CFACM code has a direct translation into SystemC, which, after integration with the SystemC stimuli, enables the generation of a SystemC Platform Independent model, shown on the left hand side of Figure 4. This SystemC PIM enables a direct link (shown as black arrow on the left hand side of Figure 4) which could feed other alternatives for the generation of the virtual system, as the one show on Figure 3. Notice that this SystemC PIM is different from the SystemC executable shown at the bottom of Figure 4, which represents the platform specific model after taking into account the platform model and the architectural mapping. This model is then a performance executable model which can provide fast estimations of time, power and other performance metrics, and thus which can be employed by the exploration tool.

For the generation of such performance model, in the virtual system generation flow of Figure 4, SCoPE+ is able to directly read CFACM code. Moreover, SCoPE+ has to read the architectural mapping information enclosed in XML files automatically generated from the UML/MARTE model. Similarly, the platform information, and moreover, information regarding the Design Space (which parameters can be configured and in which range) , is automatically produced in XML format from the UML/MARTE model. Part of the information, specifically the HW platform architecture can be handled under the IP-XACT format. A seamless connection is ensured by the Marte2IPXACT generator, and the IP-XACT plugin which enables SCoPE+ to read such format. The black arrow on the right hand side of Figure 4 shows how the IP/XACT output can serve to feed other flows for the generation of a virtual system, e.g. a virtual system based on the SNPS tool virtual platform creator. More information about the code generators is reported in D2.1.2. The XML and IP-XACT front-end plugins were available in previous versions of SCoPE and they have been refined just for COMPLEX.

Like in Figure 3, where SMOG is in charge of SW/HW separation and virtual system, generation, in the Figure 4, SCoPE+ framework solves these aspects of the integration. Moreover, SCoPE+ embeds the elements for instrumentation of the code for SW performance estimation. Moreover, SCoPE+ integrates the framework reported in D2.4.2, for high-level performance estimation of custom hardware implementation of functionality. This way, the virtual system generated under the scheme of Figure 4 enables a fast performance estimation alternative, based on native simulation and pre-synthesis techniques for the HW parts of the system. In a similar way as in Figure 3, the Figure 4 integration enables the integration of TLM2 SystemC IPs, since SCoPE+ builds internally a TLM2 platform which admits the integration of user models at the platform level. Moreover, the integration can be done from upper levels, provided that XML (or IP-XACT) wrappers are provided for the SystemC IPs.
Following sections provide an insight on how all these aspects, generically introduced in Figure 2, are solved in COMPLEX for the variety of technologies and tool involved, as shown by Figure 3 and Figure 4.
3 Input blocks

The various input blocks for the virtual system generation step in the COMPLEX flow are outlined in this section. For the different use cases that are used for the industrial evaluation, different subsets of these input blocks are used. Some inputs are automatically generated from more abstract models, while other parts are explicitly given manually by the designer.

3.1 SystemC executable model

To cover the functionality of the application, a SystemC executable model is used. This basic functional entry description is generated from a platform independent model (UML-MARTE, Matlab/Stateflow), generated from a legacy RTL model or provided manually by the system designer. For the estimation parts of the flow as well as for the HW/SW task separation steps, the corresponding restrictions (like language subsets, source organisation...) of the involved tools have to be met.

3.1.1 SystemC generated from UML/MARTE and Matlab/Stateflow

The COMPLEX flow enables the generation of SystemC code from both UML/MARTE and Matlab/Stateflow. The generation of a Platform Independent model is possible coming from both branches. Additionally, the UML/MARTE branch enables the generation of a Platform Specific Model, which is indeed a SCoPE+ executable which combines the application code with the information necessary to dump performance metrics (platform architecture, allocation of software components to hardware platform components).

Regarding the PIM SystemC model generated from the UML/MARTE, this is actually generated in two stages. First, the CFAM code structure is produced by the MARTE-to-CFAM generator (integrated in the COMPLEX plugin). Then the library based translation which defines a mapping between the CFAM macros and SystemC follows. Details on the generation of the SystemC PIM and PSM can be taken from the deliverable D2.1.2 System Generation tools from MARTE and Stateflow.

Finally, in the UML/MARTE branch, a further generator enables the extraction of SystemC describing the different verification scenarios. Each of these scenarios describes the behaviour of the external actors interfacing with the system.

The generation of SystemC code from Matlab/Stateflow descriptions is performed by using HIFSuite; the following syntax elements are currently supported:

- Charts
- Hierarchical states both parallel and sequential (AND, OR states respectively)
- Transitions
- Data (variables and constants)
- Events
- Functions implemented both graphically and as C code
- Actions on transitions and during state entering.
A Stateflow description should call the same interface which will be used by the generated SystemC code to communicate with the rest of the platform. Specifically, for the communication from the module to the platform the appropriate function calls should be present in the module whilst for the communication from the platform to the module the Stateflow description should implement the appropriate functions.

In UC1 the Stateflow description of an application for wireless sensor network is translated into SystemC and connected either to the REISC platform or directly to a network simulator implemented in SystemC.

### 3.1.2 Generation from RTL models

The generation of TLM executable models from legacy RTL models is performed by using HIFSuite. HIFSuite generates SystemC code with different TLM interfaces, compliant with the standard OSCI TLM-2.0, i.e., Loosely-timed (LT), Loosely-timed(untimed) (LT-U), Approximately-timed (AT). The TLM primitives of the OSCI TLM-2.0 library has been applied to implement such interfaces, i.e., \texttt{b_transport()} for LT, LT-U, \texttt{nb_transport_fw()}, \texttt{nb_transport_bw()} for AT.

In UC1 the RTL-to-TLM abstraction has been applied to the I2C module of the REISC platform.

### 3.1.3 User-defined SystemC executable model

Especially in Use Case 2 (Audio/Video Surveillance System), the initial input model for entering the COMPLEX flow is given in terms of a user-defined, executable SystemC model directly. For this purpose, a light-weight support library is being developed by OFFIS, to limit the input model in terms of the underlying model of computation, the set of supported communication/synchronisation mechanisms and to provide a tighter integration with the backend parts of the flow (HW/SW task separation, component estimation tools, VP back-ends and final platform software).

The public interface of the support library is aligned with the SNPS Task Modeling Library to reduce the required changes to the user-defined tasks during the refinement for the SNPS abstract platform models. The SNPS Task Modeling Library itself is not directly suitable to be used as a design entry, since a mapping of tasks to virtual processing units is always required in this case adding unwanted constraints to the input model.

From the application’s point of view, the basic modelling elements of this support library are the following:

- Generic task abstraction
- Set of pre-defined communication channels, including an abstract function call interface.

For the application-level simulation, the executable model does not add any resource constraints in terms of shared processing units or limited communication bandwidths during simulation. Nevertheless, initial performance figures can be obtained from the simulation by (manually) adding execution times and even power estimates to the abstract functional models. For this purpose, the abstract tasks already provide an interface to the COMPLEX tracing and profiling API.
The user-accessible interface for the generic task is given in terms of a base class the user-class inherits from. To improve code re-use during refinement towards a virtual platform model, different library flavours are provided:

- Pure SystemC-based implementation for unconstrained application entry-model
- SNPS Task Modeling Library flavour, directly pluggable into SNPS models
- A light-weight run-time support library for the ST-I virtual platform’s software stack, provided a C++ compiler will be available for this platform.

```cpp
namespace cplx {
    class task_base {
    : <implementation-defined base classes>
    {
    protected:
        virtual void initialize();
        virtual void task() = 0;  // main task behaviour
    };  // class task_base
    } // namespace cplx

#define CPLX_TASK( Task ) \
    struct Task : cplx::task_base
Listing 1: Overview of generic task base class
```

A significant restriction of this task-centric entry-model compared to plain SystemC is the limitation to one process per structural component. This improves the separation of communication and computation and therefore enables automatic HW/SW separation without complex side-effect analysis within a single component.

As an additional extension, two specialised task primitives are available:

- **Periodic Tasks** (cplx::periodic_task)
  
  With an additional period property, the periodic tasks are automatically re-triggered at each period. The task() function of a periodic task shall not contain an infinite loop, but is meant to run to completion upon each trigger. A basic deadline (=period) check is performed during simulation.

- **Reactive Tasks** (cplx::reactive_task)
  
  Reactive tasks are sensitive to a given event channel and their associated task() function is execution when the event is notified. Both, run-to-completion semantics and implicit state preservation via an explicit call to yield() is supported.

For the communication between the user-defined tasks, a limited set of pre-defined communication primitives is provided by the library.
• FIFO channels

For pure data-flow oriented communication, a basic FIFO channel (put/get interface) is provided. Since the packets in Use Case 2 are often quite big (e.g. image or audio frames), the interface is extended to enable separate local manipulation of the head/tail elements from the actual “transport”/synchronisation code. This reduces the number of copy operations both during the simulation and a final implementation on top of e.g. a shared circular buffer.

• Event channels

To enable a uni-directional notification of another task (without any additional data payload), an event channel has been added to the COMPLEX library. This is driven by the Use Case 2 requirements as well, where depending on the current mode a varying set of processes needs to be triggered (see D4.1.2), as well as the need for periodic activities (i.e. where the event source is triggered by a timer).

• Function Call

Especially for off-loading certain parts of the algorithm to other processor cores or dedicated hardware accelerators, a (remote) function call interface is being developed. This enables the structural separation of a particular function call, which is meant to be off-loaded after the mapping. On the caller’s side, the corresponding wrappers can then be inserted during system generation. During the HW/SW task separation (Section 0), and in particular the interface synthesis step (Section 4.1.2), the corresponding low-level interfaces are generated.

From a technical perspective, the corresponding C functions that are meant to be offloaded to dedicated HW accelerators are specified by the designer during the HW/SW separation with SMOG. In the input model, the function calls are simply simulated in the context of the calling threads, whereas after the mapping, the communication and synchronisation with the generated HW accelerator can be observed during the simulation.

For the communication primitives, dedicated ports are added as well. This enables a transparent refinement to the different backend flavours used in the Use Case 2 evaluation flow.
3.2 IP-XACT platform definition

Note: This section may be updated in a subsequent version of this report, to cover the final integration results (as part of Task 1.3) of the UML/MARTE tools developed by UC/GMV with the IP-XACT tools of Magillem and the Virtual Prototyping tools of SNPS in a consistent and self-contained manner.

3.2.1 Positioning IP-XACT platform in the COMPLEX flow

The description of the platform using IP-XACT requires data coming from several inputs. The figure below presents the global view of the process that allows going from platform specification in UML to its implementation in Platform Creator. Each line represents a “level of description” of IP or IP assembly:

- Specification: is generally a paper or specific format file describing the attributes of the platform to be uses
- UML: is used to formalize the specifications
- IP-XACT (managed in Magillem tool): is a specification of the platform structure
- PCT-script: is the input for building the virtual platform in SNPS tool

Figure 5: Dependencies between description levels of platforms to generate the PCT-script

Thus the challenge is to manage Information links between several abstraction layers:
- IP-XACT used as target repository of components specifications
- Generation (and referencing) of UML models from IP-XACT
- Referencing of specific PCT scripts from IP-XACT
- Platform specification in UML MARTE is used to generate the IP-XACT description (additional specifications are required)
- PCT description of the platform is automated

In the path going from the specification level down to the implementation level, we have identified 5 steps:

### 3.2.1.1 Step 1: IP-XACT packaging

The families of platforms which can be built are using IP components, potentially available in a library. Each IP required in the library has to be packaged in the IP-XACT format, using some attributes described at the specification level:

- Name of the IP
- Type of the IP: it may be a Processor, a Bus, a UART, a Peripheral, etc.
- The list of ports of the IP (in our case mainly TLM ports are used for communication aspects and RTL ports for clock, reset and interrupts)
- Interface of the IP are groups of ports and referencing a legal bus definition
- File Set are used to link the IP-XACT file with the files required for using the IP (for us PCT import script, and UML description file)
- Much more data may be stored in IP-XACT, but from what we intend to do in this document; this list is sufficient for now. Nevertheless, IP-XACT a good candidate for the centric representation ensuring data consistency and thus it has to be considered to be the first choice container for any additional information to be aggregated (e.g. power models)

The format of specification may be different in function of the company organization:

- Paper: the IP-XACT file can be created manually using a graphical front end (like Magillem IP-XACT Packager or another XML editor)
- Excel, CSV, or any other format: the IP-XACT file can be automatically generated (Magillem provides a CSV2IP-XACT generator)
- PCT script may pre-exist and thus be used as a specification document

### 3.2.1.2 Step 2: IP-XACT2UML

The IP-XACT files of each IP are converted into UML by an engine provided by GMV (IP-XACT schema to UML transformation). The UML file is referenced in the file set of the IP-XACT component and added in a “UML view” of the IP. IP-XACT may be used as a specification meta-model of component. The generation of UML MARTE component models is done using generation capabilities of Magillem. IP-XACT handles extensions for defining component type; this type is reported in the UML description for facilitating the assembly of platform at high abstracted level.
3.2.1.3 Step3: Platform configuration using UML

At system level, the specification of the hardware platform is done using UML, in which the following choices are done:

- Choice of a processor (name + type)
- Choice of a type of bus (type bus), that may be implicit from processor choice
- Choice of type of peripheral (HW accelerator) that allows importing external IP (e.g. HW accelerator) into the platform
- Choice of a UART
- Choice of memories (to be configured)

This UML view of the platform assembly will be the base to generate the platform assembly in IP-XACT (step4); transformation rules from UML to IP-XACT are being finalized and available in the last release of this document.

3.2.1.4 Step4: UML2IP-XACT

The UML/MARTE description of the platform is used to generate the IP-XACT description. For this, the UML/MARTE to IP/XACT generator developed by the University of Cantabria (reported in D2.1.2) is used. This generator enables the extraction and generation from the UML/MODEL, specifically, from the architectural and platform views of the system (see D2.1.1), of an IP-XACT specification of the HW platform architecture, which can include different IP blocks. A related checker ensures that the UML model can be used for the generation of a virtual system for DSE as shown in Figure 4. This means that the UML model must
include a minimum set of elements, e.g. a processor, a bus, and a memory instance. Architectural checks may also be included.

The next figure gives an example of implicit rules that must be taken into account in MARTE to IP-XACT generators

![Diagram](image)

**Figure 7:** Example of implicit rule to be used in UML to IP-XACT transformation (in step 4)

Moreover, some customizations are being performed on an ongoing for suiting to the generation of the virtual system based on the SNPS target. Specifically, the capability for configuring the IP-XACT generation (and the related checker) either for the generation of the Figure 4 virtual system, suitable for DSE, or for targeting the SNPS virtual platform has been given. Moreover, the generation will be able to read from a UML/MARTE library reflecting the components available for building a SNPS library.

Moreover, the checker is being extended for ensuring that the UML input model is suitable for enabling the generation of an IP/XACT description serving as input for the step 5, when SNPS target is enabled. Therefore, it must be also checked that the input UML model abides the platform template supported by SNPS PCT.

In order to prototype some solutions, we had to follow several tool path, as shown be next figure. Experimentations are done through 2 ways:

- Targeted solution using profile transformation (UML2XML)
- Backup solution using Magillem generation capabilities

An important work is to define the UML input format (abstraction level of platform description) and the additional information required to generate IP-XACT.
Figure 8: several experiments for UML to IP-XACT (for platform)

3.2.1.5 Automatic Generation of the IP-XACT description of the HW platform from a COMPLEX UML/MARTE Specification

Transformation rules applied on:
- Inference of the IP-XACT design
- Inference of `<componentInstances>` entry
- Inference of `<interconnections>` entry
- Inference of Context Labels
- Inference of Multiple IP-XACT Component Instances
3.2.1.6 Step 5: IP-XACT2PCT-SCRIPT

Using IP-XACT description of the platform, a generator (netlister) creates the corresponding script description for PCT. To do so, it uses a “top level” script, which is the main script document (related to the choice of the processor), with markers inside for adding:

- Component instantiation and parameterization
3. Components interconnections
4. Platform configuration
5. Bus (Specific script file)
6. Memory (Specific script file)
7. Adding a peripheral > peripheral script file

3.3 Hardware/Software Partitioning and Mapping

3.3.1 Mapping format description

At high-level, SCoPE+ enables that the application can be described in a platform independent way, finding its correspondence with the description of the application architecture in the UML/MARTE model. In the UML/MARTE model, the mapping is carried out by means of allocations of a SW component (a UML Component with either the <<rtUnit>> or <<ppUnit>> stereotype) to a component of the HW platform. The UML/MARTE to XML code generator produces the XML file that describes the corresponding mapping to the HW/SW platform.

The following XML excerpt shows how SCoPE+ enables the mapping of a single functionality, load_func1 (e.g. each one present in a different application component with the <<rtUnit>> stereotype) either to SW or to HW. This type of code is automatically generated from the UML/MARTE model.
SCoPE+ solves the integration of SW and HW estimation at several levels.

SCoPE+ enables, as shown in the excerpt, the possibility to map the same functionality of an application component (“load_func1”) either to SW or to HW, and estimate the correspondingly effects on performance. This mapping is defined within the (<Allocation> tag), where a specific instance of the executable component (<Exec_Instance …> tag) is associated to a specific “execution component” (either “func1_SW” or “func1_HW”).

“Execution components are declared within the <functionality> tag. Execution components are distinguished attending to its category, HW and SW. In the example, two “execution components” are declared for the application functionality “load_func1”: one for software estimation, “func1_SW”; and one for hardware estimation “func1_HW”. Each execution component has an associated dynamic library (.so file) which contains the instrumented code for each specific target (i.e., hardware or software).

Execution components can be seen as executable images of a component, which can be “dynamically linked” to an application component of the executable specification. That is, the executable specification can decide which instrumented code is linked to a component at elaboration time, before simulation start. This technique saves regeneration (thus
recompilation) of the virtual system along the different system configurations simulated in the design exploration process.

The UML/MARTE to XML code generator is in charge of producing the (two) execution components for each functional component in the UML/MARTE model. Therefore, this inclusion is automated and does not involve any further coding to the user.

At a ‘script’ and ‘makefile’ level, the HW/SW integration has required the integration of the high-level custom HW estimation library reported in D2.4.2 within the SCoPE+ infrastructure. The custom HW estimation library uses a parsing and annotation tool chain based on LLVM, which is different from the one used for SW estimation by SCoPE. SCoPE+ solves this mismatch by enhancing the “scope-g++” script. Specifically, “scope-g++” compilation supports now an additional estimation value for the estimation method (switch “--scope-method”). Up to now, several methods were supported for SW estimation, e.g., one based on the assignation of the cost to executed operations (“--scope-method=op-cost”), and other ones which are more precise, but which rely on the availability of the cross-compiler (“--scope-method=asm-sentences” and “--scope-method=asm-opcodes”). Now, an additional value for the switch, “--scope-method=hw-cost”, enables the instrumentation of the compiled code for custom hardware estimation. When the “--scope-method=hw-cost” is enabled, the “scope-g++” command applies the simplified compilation script reported in D2.4.2, section 2.9.5 (hwest-gcc). This way, the SCoPE+ makefile infrastructure is now able to produce a HW execution component.

A SW mapping is then a mapping of a SW execution component of an application component, onto a Hardware component such as HW processor (HW component of “processor” category). A HW mapping is a mapping of HW execution component of an application component, onto a custom HW component (HW component of “HW_resource” category). HW components are declared in the HW platform section.

Previous XML excerpt also reflects the possibility to explore the architectural mapping without needing a regeneration of the virtual system. The mapping configurability is stated in the section allocation by means of variables, whose names and values are declared in the parameters section. As can be seen in the parameter section, the variable “SWelement” can have the values “func1_SW” and “func1_HW”. Regarding to the variable “HWcomponent”, the possible values are the HW components defined in the HW platform section. This way, two mappings for the function func1 is possible.

The following XML code gives another example which does not have configuration, but with one more component. Specifically, it describes two functional components of the application, where one is mapped to SW and other to custom HW. It also reflects again that two execution components are internally used by SCoPE+ in order to implement the functionality allocation into SW component or into HW component.
Specifically, the XML code corresponds to the inference from a UML/MARTE model where the application model has two functional components, (with the MARTE <<RtUnit>> stereotype), “FComp1” and “FComp2”, which enclose functions “load_func1” and “load_func2” respectively. Each of these functional components, and thus their enclosed functionalities, have two related “execution components”. For instance, “FComp1” component has the associated “FComp1_HW” for custom HW estimation and the “FComp1_SW” for SW estimation, which like to the associated dynamic libraries, func1_HW.so and func1_SW.so.

The HW components “processador_1” and “virtex” are defined in the HW components sections. The specific mapping is described in the allocation section by means of the allocations “func1_map” and “func2_map”. In this case, the “func1_map” mapping states the specific processor in charge of executing the SW implementation of the functional component “FComp1”, while the “func2_map” mapping states the specific custom HW resource in charge of executing the custom HW implementation of the functional component “FComp2”.

```
<HW_Platform>
  <HW_Components>
    <HW_Component category="processor" name="processador_1"/>
    <HW_Component category="HW_resource" name="virtex"/>
  </HW_Components>
</HW_Platform>

<Functionality>
  <Exec_Component name="FComp1_SW" category="SW" file="EXAMPLE_PATH/libs/func1_SW.so" function="load_func1"/>
  <Exec_Component name="FComp1_HW" category="HW" file="EXAMPLE_PATH/libs/func1_HW.so" function="load_func1"/>
  <Exec_Component name="FComp2_SW" category="SW" file="EXAMPLE_PATH/libs/func2_SW.so" function="load_func2"/>
  <Exec_Component name="FComp2_HW" category="HW" file="EXAMPLE_PATH/libs/func2_HW.so" function="load_func2"/>
</Functionality>

<Allocation>
  <Exec_Instance name="func1_map" component="FComp1_SW" hw_resource="processador1"/>
  <Exec_Instance name="func2_map" component="FComp2_HW" hw_resource="virtex"/>
</Allocation>
```
3.3.2 Integration of the environment

In the Deliverable 2.5.2, there were included the way the HW/SW partitioning and the corresponding mapping are described in the XML files in order to be read by SCoPE+.

Although the work done until M24 has been quite complete, the application of the infrastructure to Use Case 3 and an increase in the exploration possibilities of the methodology have resulted in several integration issues. These new issues mainly rely on the consideration of the environment in the simulation infrastructure.

The integration of environments has resulted in three main development activities. First, the environment itself has to be automatically integrated in the simulation from the UML information. Second, new modelling components are needed by the system simulator to emulate the effect of IO communications between the system and the environment. And finally, the inclusion of the environment opens a new area of exploration.

3.3.2.1 Adding stimuli environments to the SCoPE+ simulation

As described in D2.1.2, the system environment can be described in the UML methodology, generating codes that can emulate the execution of this environment in a SystemC simulation. As a result, SystemC threads, ports and signals are used to model the environment behaviour.

However, although SCoPE+ has been implemented on top of SystemC, it is oriented to simulate SW codes. Thus, SCoPE operates with an internal model of computation completely different from the one used by the SystemC kernel. As a result, the application of these SystemC environments to the SCoPE+ models requires additional integration components.

Together with the SystemC code itself, the connection requires from different wrappers. These wrappers have been generated in two ways, in order to support both services provided by the system and required by the environment, and services provided by the environment and required by the system. Services provided by the environment are accessed as function calls from the system. As a result, the operation of the wrapper is just to convert the function calls to SystemC channel calls. Thus, a single wrapper is automatically generated for accessing all these services.

However, services provided by the system imply active requests from the environment. That is, each request from the environment starts a chain of execution flows that perform all the operations the system must perform to provide the service required. Then, these wrappers are, in fact, active components that act as service initiators. There is a wrapper automatically generated for each part of the environment, which starts its execution at the beginning of the simulation and handles all the incoming requests from the environment. To support that, a main function is also automatically generated initialising the system and all these wrappers, and making the connection between the SystemC channels and the wrappers. To execute it, a library is compiled and a static executable component is added to the XML system description file. An “Execution component” is declared within the <functionality> tag.

```xml
<Functionality>

  <Exec_Component name="System" category="SW"
  file="EXAMPLE_PATH/libs/[scenario.name]/SC_[scenario.name/].so"
  function="init_scenario_main" />

</Functionality>
```
Additionally, the XML file has to include the statement that points to the simulation library that contains the SystemC code of the stimuli scenario. Since the SystemC code cannot be directly supported by the SCoPE+ infrastructure, it cannot be loaded as an executable component. After several alternatives, the final solution selected to integrate that code, is to add this code as a SystemC plugin. Then the code can be loaded together with the underlying SystemC infrastructure of SCoPE+ simulations.

```
<Plugin name="environment" entry="init_[scenario.name/].main"
    library="EXAMPLE_PATH/libs/[scenario.name/]/SC_[scenario.name/].so"
    args="EXAMPLE_PATH/libs/[scenario.name/]/SC_[scenario.name/].so"
/>
```

### 3.3.2.2 Adding IO devices to connect the environment

To enable the communication between the system and the environment, an additional HW resource has to be considered in the model of the HW platform. Specifically, an UML component specified by the MARTE stereotype <<HwI_O>> models these type HW resources. In order to enable the integration of the Stimuli scenarios in complete DSE process, some modification have to consider in the XML file system description. The first of these modifications is the creation of a new component category is defined: “hwIO”.

```
<HW_Component category="hwIO" name="nameComponentName"/>
```

The application of these components represents a challenge for the generator, since two different kinds of IO devices appear. Now, there are IO components that interconnect elements within the system and IO components that connect the system with its environment. For example, in a distributed system, such as Use Case 3, the system contains different computing nodes interconnected by networks (a Ethernet and a serial communication). These networks are considered internal to the system model and then the network interfaces have to connect parts of the system, not the system with the environment. But, at the same time, external serial connections provide information from the environment. Thus, these interfaces represent the same type of physical component, but require different implementation in the model.

In order to support that, the UML modeling infrastructure and the generation tool have been updated.

### 3.3.2.3 Supporting different environments

Finally, the most complex integration issue comes from the support of different environments. The COMPLEX UML/MARTE methodology includes the modelling of the system environment in order to specify some stimuli scenarios. The stimuli methodology defines a set of scenarios which describe concrete behaviours of the external actors interfacing with the system. These stimuli scenarios specify different uses cases to explore the different design alternatives captured in the model.

In the case that several scenarios are presented in the model the XML statement has to include the corresponding exploration variables defined as “__nameVariable”.

---

Page 26
As a consequence of that, in the Design space XML file these exploration variables should be specified and defined all the scenarios that are going to be considered in the simulation.

All wrappers and generated communications have been developed in order to support automatically exploring the effect that different environments have in the system.

Additionally, modifications in the structure of files generated by the project and modification in the Makefiles have been required to support the automatic exploration of system environments.

### 3.4 Virtual platform IP models

In any system there are a large number of components that are reused as existing HW IP. When generating a virtual system a model for these IP blocks needs to be inserted. For the COMPLEX flow these components need to be available as SystemC models. To cover the use cases considered in the project the following models will be available for the Synopsys platform:

- **Processor IP**: these are represented in the platform by an instruction set simulator integrated with the SystemC simulation kernel. Following processor types are available: ARM926EJS, ARM11MPCORE, other processor models can be integrated with the COMPLEX virtual system generation flow but their availability in the project can be limited due to 3rd party licensing requirements.

- **Memory IP**: generic memory models can be used to represent the different memories in the system.

- **Interconnect IP**: a generic TLM2.0 LT/AT interconnect model and also AMBA-AHB and or simple AXI models are available. Again, more complex interconnect models (e.g. NIC301) can be integrated with the COMPLEX virtual system generation flow but their availability is limited due to 3rd party licensing requirements.
Peripheral IP: in order to support the OS that is running on the processors in the system a number of peripheral and IO components need to be available when generating the virtual system:

- UART model: a generic serial interface component is available, but also a standard 16550 UART model.
- Ethernet controller: in case network connectivity is required in a platform there is a SMC91C111 Ethernet controller, the model complies with the programmers interface of the device and also implements connectivity to the physical Ethernet on the host
- USB: Synopsys Designware USB model
- LCD Panel: via a PL111 primecell IP model
- Keyboard and Mouse interface: a generic keyboard and mouse interface or a primecell PL050
- GPIO PL061 model
- Programmable interrupt generator:
  - Timer: a programmable timer & interrupt generator

Next to the Synopsys virtual platform that provides the IP models listed above, use cases 1 and 2 in the COMPLEX project also utilize the ST virtual platforms. Those platforms combine conventional cycle-accurate software simulation (e.g., using an Instruction Set Simulator (ISS)) with SystemC-based abstract TLM.

The typical topology of these two platforms includes:

- Initiator modules: A CPU model is set for RTOS-based software simulation. The CPU model includes the ISS of the CPU and supports a blocking transport interfaces. An additional hardware IP module is included for customized computation. This HW IP module is connected to the interrupt controller of the ISS by a channel in order to trigger the software processing element in case of an interrupt event.

- Target modules: These could be memory components or peripheral devices. LT style is applied. In case there is more than one memory module, this topology can represent an embedded system with application data partitioning.

- Combined initiator/target module: The Direct Memory Access (DMA) controller allows directly moving data between memory locations and devices without intensive handling from the CPU Model. It plays roles as both an initiator (for reading and writing data) and a target (being programmed by a DMA requester).

- Interconnection: An OSCI TLM-2.0 LT-style STBus model constitutes the interconnection architecture. It is extensible by adding more initiators and targets.

All the HW IPs for Use case 1 and Use Case 2 are memory mapped peripheral devices and they derive from one generic class, peripheral, that constitutes the basic functionality for the HW IPs which header is the following one:
struct peripheral : public sc_core::sc_module {
  public:
    tlm_utils::simple_target_socket<peripheral> socket;
    unsigned int base;
    unsigned int top;

  //Member Methods
  virtual void b_transport( tlm::tlm_generic_payload& trans,
                           sc_core::sc_time& delay ) = 0;

  void set_address(unsigned int base_address, unsigned int top_address) {
    base = base_address;
    top = top_address;
  }
};

All HW IPs are target devices; therefore they should provide an implementation of the following mechanism:

virtual void b_transport( tlm::tlm_generic_payload& trans,
                           sc_core::sc_time& delay ) = 0;

That constitutes the TLM 2.0 communication mechanism between the peripheral and the core/cores.

Following method defines the memory address space of the device.

void set_address(unsigned int base_address, unsigned int top_address)

These two methods constitute the minimal set for having a target module (i.e. a memory can be modelled with just these two methods). Of course, designers can add methods/functionality to implement specific behaviour. For example the structure and methods used by the SPI are illustrated in Figure 11. Two main methods in the model implement a typical SPI mechanism. The b_transport() method, which is inherited from the standard peripheral class, listens to the target socket and waits for configuration information from requestors. Upon receipt, the configuration information (i.e., transmitting/receive mode, data, and size of transfer) is saved in programmable registers of the SPI controller. Then the clock_commit() function, activated at every clock cycle, begins an SPI operation according to registers configuration setting. When the entire SPI transfer is finished, the clock_commit() function will interrupt the CPU by setting the proper interrupt output port.

Figure 11: SPI structure
The clock signal will be the global clock of the SoC, used also for the ISS of the CPU; this signal provides a common synchronization for the platform.

![Diagram of SPI connection in Use Case 1](image)

**Figure 12**: SPI connection in Use Case 1

The SPI constructor has also two additional parameters: `bool i2s_mode` and `bool i2s_clock` to configure the peripheral with the optional i2s protocol. Parameter descriptions are in Table 1. Once the platform will be constituted, proper values will be set.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Value (Default)</th>
<th>Description</th>
</tr>
</thead>
</table>
| I2S_mode       | 0, 1            | It indicates if I2S implementation is required.  
0: No I2S functionality required. Only SPI functionality will be generated  
1 : I2S functionality is implemented in addition to the SPI features. |
| I2S_clock      | 0,1             | It selects the Master clock generation availability in I2S mode.  
0 : Master clock not available  
1 : Master clock is available as an output for external component  
Note: This parameter has a meaning only if I2S_mode is set |
4 Augmented TLM2 platform generation

From the input blocks described in the previous sections, a self-simulating virtual platform shall be assembled that allows collecting data on the extra functional characteristics of the system. This data allows analysis and comparison of system configurations during the design space exploration. With respect to the COMPLEX flow related to use case two, the input is a user defined executable SystemC model and the augmented models are obtained from individual software and hardware estimations. The augmented models that result from the estimation and characterisation in tasks T2.2, T2.3, and T2.4 must be re-combined to form an instrumented system description. The generated system shall comply with the IEEE 1666™-2005 Standard for SystemC [1] and the external interfaces of components shall be based on OSCI TLM 2.0 [2]. Therefore, all input blocks are connected via TLM 2.0 interfaces which need to be synthesised. Appropriate wrapper functions and modules will need to be generated in order to accomplish the interconnection and communication between the system components. The following sections describe in more detail, how the augmented models are obtained and how these models are reintegrated into an augmented TLM2 platform.

Some of the main contributions in this effort during the last period of the project are the extension of the interface synthesis to consider accesses via the LRM API and the support for multiple clients in TLM wrappers for hardware slaves (see 4.2.1).

4.1 Task separation and augmented model generation

When the input model is available as executable C or SystemC code and estimation shall be performed by some external estimation tools instead of SCOPE+ in order to obtain augmented TLM models, as it is the case in use case 2, the corresponding parts of the input model that should be estimated need to be separated from the input design and prepared for estimation. This step is supported by the SMOG tool. This tool reads an input design’s source and writes a self-contained portion of the code into an individual target directory. The generated source is accompanied by a simple test bench and Makefile, as well as configuration files for the supported estimation tools. When targeting software, a configuration file for the software estimation tool SWAT (see software estimation described in [12]) is generated. For hardware, a configuration for a simulation script is generated that can be used with the high level synthesis tool PowerOpt (see hardware estimation described in [10]). Both tools represent the exemplarily deployed estimation tools within the COMPLEX flow. The SMOG tool is also involved in the generation of the virtual system and is therefore described in this document.

Even though the executable input design may consist of SystemC/TLM models, a block of code that should be separated must be encapsulated in a single C-style function. The function specifies the boundary of the code implementing a task’s behaviour that is bound for detachment. Its signature defines its interface. The function may call additional sub-routines. Every sub-routine is considered as part of the separated code and included in the separated source code in order to yield a self-contained implementation. The restriction to have a block available as a C-function also originates from the necessity to satisfy the input language requirements of the applied software estimation tool. For the transformation of a member function into a C-function, data members that are accessed by the original member function must be explicitly passed to the C-function variant via extra parameters. For a general approach, it is suggested to add just one additional parameter to the C-function. This parameter should be a pointer to a struct that holds all former data members. It therefore
resembles an explicit ‘this’ parameter. An exemplary transformation of a member function into a free C-style function is shown in Figure 16. Concerning further restrictions regarding the coding style which might be mandatory for tasks that shall be realised as hardware, please refer to PowerOpt’s Language Modelling Guide [8].

After identifying the designated C-function that marks the top level of the block, SMOG gathers all function and type definitions that are used from the top level function and writes them to separate source code files that represent the input to the appropriate estimation tool. The output of SMOG depends on the actual separation target, as SWAT and PowerOpt have different input constraints. When targeting software and estimation with SWAT, SMOG generates C source files to respect the input language subset of SWAT. For estimation with PowerOpt, C++ code is generated. The generated Makefile for separated hardware is prepared in such a way that it can be used to directly load the separated code into PowerOpt, i.e. it applies PowerOpt’s dale-cc front-end instead of gcc as C/C++ compiler. Extra code that contains a main function forming a test bench is only created when the separated code itself does not already include the main function, which is normally only the case when separating code for a software target. The added main function reads input data from a binary stimuli file and calls the separated function with this data. The resulting test bench code can be used by the estimation tool to run a standalone simulation of the separated block. Note that this test bench is only able to reproduce the input data sequence for a selected block, as it is given by the stimuli file, but not the timing of the original environment. However, this simple form of test bench is sufficient for estimation purposes in most cases, as the timing behaviour is rather important when simulating the platform design as a whole, than when characterising a single component. When targeting hardware and the use with PowerOpt, the test bench code is enhanced with pragmas for hiding test-bench code from synthesis and estimation, as well as macros provided by PowerOpt for tracing stimuli data and array initializations that are needed for estimation. SMOG also provides the functionality to generate an instrumented version of

Figure 13: Artificial example on moving a member routine's functionality into a C-function

```c
struct Calc::public sc_module
{
    void proc();
    private:
        float coeffA, coeffB;
};

void Calc::proc()
{
    float varA, varB;
    return coeffA * varA + coeffB * varB;
}

void CCalc( Calc_data *this_)
{
    float varA, varB;
    return (*this_).coeffA * varA + (*this_).coeffB * varB;
}
```

```c
struct Calc_data {
    float coeffA, coeffB;
};

struct Calc::public sc_module
{
    void proc();
    private:
        Calc_data data_; 
};

void Calc::proc()
{
    return CCalc( &data_ );
}

void CCalc( Calc_data *this_)
{
    float varA, varB;
    return (*this_).coeffA * varA + (*this_).coeffB * varB;
}
```
the original input block, in case no stimuli data is available. The instrumentation dumps the input data of the instrumented function to a binary file during a separate simulation step. The content of the binary file corresponds to the format expected by the generated test bench code. A more detailed insight into the SMOG tool is given in Figure 14.

Relevant parts of the executable SystemC model are read into SMOG by first creating a binary intermediate representation using the EDG C/C++ compiler front-end from the FOSSY synthesiser (1) and then mapping the binary representation to an AST and writing it to an ASCII format based on s-expressions (2). The user has full control over the set of source files that shall be processed by SMOG by specifying them using command line options. Instead of reading the source of a whole design, it is advisable to focus on those source files of the design containing code which should be separated according to the HW/SW mapping information. This speeds-up the import of source code, decreases the size of memory needed for the AST, and reduces the possibility to encounter un-supported constructs in the input code, e.g., within test bench code. If the set of loaded source files is insufficient, SMOG prints warnings that list the required functions that are not defined inside the specified source files. The separation part (3) searches the function selected for separation in the AST and generates the estimations’ input source files that contain this function and all called sub-routines. It also always creates a Makefile and triggers creation of a test-bench with a main function if function main is not part of the separated code.

4.1.1 Augmented hardware model

With the COMPLEX enhancements added to the high level synthesis tool PowerOpt, the estimation of separated functionality targeting hardware in the BAC++ estimation flow results in an executable BAC++ hardware model. This section shortly describes this BAC++ hardware model. More details can be found in [10]. As mentioned in D2.4.2, the created BAC++ model represents a white box IP which is tripartite. As shown in Figure 15, it consists of a functional and a non-functional model, as well as of an observer.
Communication with the environment utilizes a designated register interface. The register interface itself is represented by a structure which is located in a TLM2 wrapper module. Details on this register interface are given in section 4.2.1 which describes the TLM2 interface wrapper. The BAC++ model accesses this structure via references passed during instantiation.

The non-functional model represents the local resource manager (LRM) of the hardware component and accesses the LRM section of the register interface, which is generic and has a fixed layout. The LRM manages the switching of power modes. The encapsulating container class provides an interface method called \texttt{LRM\_notify()}, which can be used by the TLM2 interface to notify the non-functional model of write accesses to the LRM register interface, i.e. power mode change requests.

The functional model provides a function-call-like interface for executing the behaviour of the C/C++ function, from which the BAC++ code was generated. Each created HW BAC++ module implements a member method \texttt{call(call\_param\_reg\_map\_type \&params)}, which simply forwards the function-call and its parameters to the functional model. The call parameters are specified by the call-related section of the register interface, whose layout depends on the signature of the C/C++ function. It is represented by a struct type that contains a data member for each input and output parameter of the function. More details on the register interface and the associated struct type can be found in section 4.2.1. The functional model implements the behaviour of the module based on the controller and data path resulting from estimation, reading the input values from the register interface struct and writing back the output values.

Abstract base classes providing the basic functionality are available for each part of the created BAC++ module. For BAC++ HW modules the most important base class is \texttt{cplx::white\_box\_ip::module}. It contains the basic functionality of HW modules. During the characterisation process, a class representing the particular HW module is derived from this class. The base class also contains an instance of the extra-functional model as well as an instance of an observer. The latter is instantiated with the appropriate values, characterising the particular design. That is internal apparent resistance, for example.
4.1.2 Augmented software model

The software estimation in the BAC++ estimation flow is performed using the SWAT tool according to the flow described in [12]. The software estimation tool SWAT requires executable C code as input and produces object code with back-annotated timing and energy estimates. The resulting object files need to be linked to the rest of the design in order to replace the definition for some software functionality by a corresponding annotated version. As a bridge to the C++ tracing mechanism, the object code uses a call-back function to propagate the gathered extra functional data. Object files generated by the basic swat-core-ba tool use a call-back function that provides the id of an executed basic block. When using this classic version, the call-back function or downstream code needs to map the basic block id to the associated extra functional properties. The files that provide this basic block related information are also provided by SWAT and can be easily imported in order to build a proper look-up table. Extensions to the SWAT tool implemented during the COMPLEX project provide object files that resemble BAC in object form. These object files utilize another more convenient call-back function that directly provides the extra functional characteristics of an executed basic block. To illustrate the call-back functions’ signatures, Listing 2 shows call-back stubs that can be used in a C++ environment. The body of the stubs need to execute functionality that interacts with the used C++ tracing back-end. The API used for tracing extra-functional data in the call-back functions is the same as for hardware in order to allow a uniform handling of system blocks during the design space exploration following the BAC++ estimation flow.

```c
// basic call-back used by SWAT object code:
extern "C" __swat__bbtrace(int id)
{
  // obtain extra functional properties related to current basic block
  // tie up and pass extra functional data to tracing interface
}

// extended call-back used by SWAT object code in COMPLEX:
extern "C" __swat__bacpp(int id, float t, float e)
{
  // tie up and pass extra functional data to tracing interface
}
```

Listing 2: Call-back functions used by the annotated software model for tracing purposes

For use of the back annotated software model within the COMPLEX estimation flow the applied call-back function (or downstream code) must implement the proposed COMPLEX tracing mechanism described in [13].

As SWAT requires executable code, the estimation input must also include definitions for those functions that are not supposed to stay software but shall be implemented as hardware. In order to obtain resulting object code where these parts can be easily replaced, the corresponding functions are best located in separate files. SMOG supports this approach by placing every function that is used from the separated software block into a separate file, while SWAT allows ignoring certain input files during estimation and back-annotation. When back-integrating the annotated software model into a virtual platform, the object code containing definitions of functions that are outside the software block need to be replaced by code that translates a call into TLM transactions. These adaptor functions are also generated by SMOG as part of the interface synthesis.
4.2 TLM2 interface synthesis

When re-integrating annotated models into a virtual platform based on TLM2 models, as required in use case two, the corresponding interfaces to allow TLM2 based communication with and in between these annotated models need to be added. For instance, if an annotated software function calls other functions that have been separated to hardware, the corresponding calls need to be translated into TLM transactions, in order to initiate a communication with the hardware component, resembling a remote method invocation (RMI). The required interfaces differ for software and hardware components. The most prominent difference is caused by the fact that a software block usually initiates the communication with other blocks, taking the position of a master, while a hardware block generally serves as an accelerator for certain functionality and takes the position of a slave. The augmented code for software will therefore contain e.g. function calls which need to be delegated to the corresponding hardware components. As a consequence, the TLM2 interface that is needed to integrate a software block into a system must provide an initiator socket, while a hardware block requires a wrapper that provides a target socket. The different wrapper modules for interfacing annotated models with a TLM2 environment are described in the following sections. A common constraint for the automation of the interface synthesis is that the interfaces shall be based on the TLM 2.0 target-initiator interface. Communication shall be based on the TLM 2.0 base protocols and the kind of data packets that are transferred over these sockets shall be limited to the TLM 2.0 generic payload.

The internal operation of SMOG when generating wrapper modules and adaptor functions for use in a TLM2.0 platform is based on the same input processing steps as required for task separation. Figure 16 shows the involved SMOG modules. As with task separation, the input source files need to be loaded into SMOG by transforming the source code into an internal AST representation (1) + (2). The TLM adaptor generation (3) searches the specified function in the AST and generates the appropriate adaptor module, depending on the target that was used when separating the same function for estimating (HW or SW). For software tasks, only a wrapper module is generated that can serve as TLM initiator. For hardware tasks, the needed TLM target wrapper module is accompanied with a separate header file specifying the register interface struct type, as well as with the adaptor functions that translate a call to the separated function into TLM transactions.

Figure 16: SMOG TLM adaptor generation tool flow
4.2.1 Hardware targets

The annotated model resulting from estimating a separated piece of code with PowerOpt™ is a BAC++ module. This module is a C++ class which encapsulates a functional model, providing the original functionality enhanced by extra functional property annotations at basic block granularity, and a power model, implementing a local resource manager (LRM) that allows managing and switching of power modes. See also section 4.1.1 on the augmented hardware model. This module serves as a slave component, providing the encapsulated functionality. A way to incorporate such a component into a TLM2 virtual platform model without modifying the module, is to add a wrapper class that turns TLM communication that either represents an invocation of the hardware block’s functionality, or accesses to the LRM interface, into execution of the functional or power mode model inside the BAC++ module. Such a wrapper requires at least one target socket, in order to receive TLM2 transactions. This socket could be used to handle both functional and power mode related requests. Optionally, a separate second target socket could be used in order to distinguish between the functional and the LRM interface. As the TLM2 based communication is memory mapped, the wrapper also needs a register interface, or register map, that allows mapping the addresses of TLM2 transactions to either function parameters, return values, protocol signals for synchronizing functional requests, or registers of the LRM interface.

The SMOG tool is capable of generating the source code for such a TLM2 wrapper class. By default, the generated wrapper provides a single target socket that is utilized by both, the functional and the power mode interface. The protocol for blocking communication based on the TLM2 base protocol is implemented in two separate methods, a method `b_transport` to handle transactions targeting the functional model and a method `lrm_transport` to handle transactions that target the power mode model. Having two distinct transport methods for the two interfaces allows supporting a second socket for the LRM interface if desired without extra effort. By using sockets of type `simple_target_socket`, the corresponding transport methods only need to be registered at the right socket. If there is just one socket, the `b_transport` implementation detects this case and forwards transactions that are related to the LRM to the `lrm_transport` method. The distinction between functional and power mode related TLM transactions is based on the addresses used in the transactions’ payloads, based on a register map that specifies the layout of the register interface. This register interface is implemented as a struct member, whose structure is used to derive the addresses of nested register members. It also serves as a buffer, implementing the storage locations for the register interface. The register interface consists of two main sections, the registers of the LRM interface and the set of registers needed to implement a remote invocation of the functional model (i.e. the call interface). The general layout of such a register interface is shown in Figure 17.

The LRM related section of the register interface follows the specification in [14]. It was chosen to precede the call interface, because unlike the call interface’s register number and size its size is fixed. The call interface covers registers for buffering inputs and outputs of the functional model, as well as control registers for synchronizing possibly conflicting functional requests. This control section includes one message register that contains information on the state of the functional model and one client register for every component that might request execution of the functional model. The slave component communicates its current state and the currently served client via the message register to its clients. The clients notify requests and their actual state via individual client registers to the slave. Concurrent requests (but no input data) are buffered in the corresponding client registers. The control registers’ constellation and layout is based on the synchronisation mechanism described in [3].
In case a hardware block has only one single client, the generated interface code allows to replace the set of control registers by a single register that represents a combined request-acknowledge handshake signal. Doing so also simplifies the handshaking mechanism in that requesting and waiting for a grant from the target module is not required. This optimisation is enabled by setting the variable `OPTIMIZE_FOR_SINGLE_CLIENT` when building the platform model.

The struct type implementing the register interface in the generated TLM wrapper is organized into logical sets of registers by further nested struct members. Figure 18 illustrates how the implementation of the register interface described above looks like and how the parameter section of the call-related registers is derived from the signature of an input function that is separated to hardware.

![Figure 17: General layout of the register interface](image)

![Figure 18: Mapping a function signature to a register interface struct allowing one client](image)
The register slots for the parameters of the function get the same name as the corresponding parameters. They are sorted with respect to their direction, i.e. in, out, or inout. SMOG tries to detect the direction of a parameter by its type. For instance, pointer or reference parameters are considered as inout, all other parameters as inputs, and a return value as output. This default classification can be overridden by the user via command line options. The ordering can optionally be manifested by encapsulating the sets of parameters into additional struct objects called in, inout, and out. By default, this extra nesting hierarchy is disabled when generating the source code for the register interface struct type. For the sake of clarity, the conditional extra nesting is not shown in Figure 18. If the input function has a return value, a member called “result” is added to the set of output register members in the param struct type.

The order in which requests from multiple clients are processed depends on the scheduling performed by the target wrapper. The actual implementation simply processes the requests in the order in which they were registered (FIFO). Exchanging or customizing the scheduler in the target wrapper is currently not implemented, but adding this feature is possible at manageable effort. Figure 19 illustrates the protocol when two clients are accessing the functional interface of the target wrapper. It also shows the internal states of the target wrapper. The state transitions are triggered by accesses to the Ctrl register interface or by internal events, for instance when the wrapped functionality has finished execution. Like the layout of the register interface for handling RMI, the applied protocol is also based on [3].

Figure 19: Sequence diagram showing two clients accessing the same target object

Remember that the actual implementation is based on blocking communication. Therefore, a client that either wants to send input data or desires to read output data is blocked until the target is ready. This has the advantage that just one single buffer for input/output data is required. A possible extension of the communication mechanism could extend the register
interface by individual parameter registers for every client, or corresponding FIFO buffers. This would for instance allow any client to send its input data while the wrapped functionality is processing a call from another client at the cost of increased memory requirements.

In order to reduce the amount of automatically generated code, fixed and reusable elements of such a target wrapper have been moved to corresponding base classes that are part of the COMPLEX library. Figure 20 gives a schematic view of a target wrapper module for a hardware component as it is generated by SMOG.

![Figure 20: Generated wrapper module for hardware (TLM target)](image)

### 4.2.2 Software initiators

For software components, the annotated models gained from SWAT consist of object code for power and timing annotated C functions. Functions that are called from the software, but are not part of it (i.e. moved to hardware), are ignored during estimation and yield non-annotated object files that are simply not considered when building the final platform model. In order to utilize the annotated object code in a TLM2 virtual platform, corresponding TLM2 modules must be created, that provide an interface which allows initiating RMI like communication with slave hardware components and a means to execute the software functions whose implementation is taken from the annotated object code. The SMOG tool can also generate this kind of TLM wrapper module. The module provides one initiator socket for processing transactions with slave (hardware) components and a thread process that executes the encapsulated (annotated) software function. A schematic view of a generated initiator wrapper module is shown in Figure 21.
As the initiator wrapper module is expected to replace an ISS, the encapsulated function is assumed to be function `main`. As the original main function cannot keep its name in the TLM2 environment due to a conflict with the main function in the SystemC simulation environment, the `main` function is assumed to be renamed to `sw_main`. This requires touching the object code for `main` and renaming `main` into `sw_main` before linking the different objects into a platform executable. This constraint seems to be acceptable, because the step of renaming the function in the object code can be easily automated, for instance by applying a corresponding shell script. As with the hardware target wrapper module, fixed and reusable elements of an initiator wrapper module for software have been moved to base classes that are part of the COMPLEX library. These also provide methods for building a look-up table from SWAT generated bbmodel files and for performing a look-up from a call-back function as described in 4.1.2. The annotated software model and the TLM wrapper alone would be insufficient. At least as soon as the software component needs to communicate with other components, TLM adaptor functions are needed to translate the sub-routine calls from the software into TLM transactions. These adaptor functions are described in more detail in the next section. As the adaptor functions need to access the initiator socket of the software wrapper module in order to send TLM transactions, the wrapper provides helper methods, e.g. `tlm_write`, for assembling a TLM generic payload and sending it to a connected slave via the wrapper’s initiator socket. These helper functions are implemented in the base classes inside the COMPLEX library.

### 4.2.3 Adaptor functions

In order to make a call from a software block to a function that was separated to hardware, the call is translated into the corresponding TLM2 transactions that form a remote method invocation (RMI). This is achieved by replacing the original software functions by adaptor...
functions that provide the same signature, but execute the protocol depicted in Figure 19 based on TLM2 transactions. As the RMI is based on blocking communication, a call to the adaptor function is blocking too. In order to allow exploitation of parallelism in the initiator code, the main adaptor function is using sub-routines that encapsulate sub-steps in the protocol (request grant, write inputs, wait until execution finished, read outputs). Instead of calling the blocking main adaptor function, an initiator could also call these sub-routines directly, e.g. in order to be able to execute additional code in between. However, this usually requires additional modifications in the original source code, i.e. replacing one function call by several calls to resemble the applied RMI protocol. Listing 3 lists the main adaptor function and sub-routines for a dct function separated to a hardware block.

cplx::sw_wrapper_base* obtain_client_initiator_module()
{
    sc_core::sc_process_handle hndl = sc_get_current_process_handle();
    cplx::tlm_initiator_wrapper_base *initiator
        = dynamic_cast<cplx::tlm_initiator_wrapper_base*>(
            hndl.get_parent_object());
    assert(initiator);
    return initiator;
}

void dct32_request()
{
    const unsigned int base_addr = 0;
    volatile dct32_register_if_type* reg_p =
        (volatile dct32_register_if_type*)(base_addr);
    unsigned int busCycles = 0;
    cplx::sw_wrapper_base *initiator = obtain_client_initiator();
    unsigned int client_id = initiator->get_client_id(base_addr);

    /* synchronize with initiator
     * (let cpu time that elapsed before this transaction pass): */
    initiator->synchronize();

    /* notify request via client's register: */
    typedef cplx::client_register_type client_reg_t;
    cplx::client_register_type client_reg;
    client_reg.state = cplx::client_register_status_request;
    initiator->simple_tlm_write<client_reg_t>(
        (unsigned long)&(reg_p->call_if.ctrl.initiator_reg[client_id]),
        &client_reg, busCycles);

    /* notify initiator on consumed bus cycles for this communication: */
    initiator->adjust_global_cycle_count(busCycles);
}

void dct32_write(const int in[32] )
{
    dct32_request();
    const unsigned int base_addr = 0;
    volatile dct32_register_if_type* reg_p =
        (volatile dct32_register_if_type*)(base_addr);
    unsigned int busCycles = 0;
    cplx::sw_wrapper_base *initiator = obtain_client_initiator();
    unsigned int client_id = initiator->get_client_id(base_addr);

    /* synchronize with initiator
     * (let cpu time that elapsed before this transaction pass): */
    initiator->synchronize();

    /* notify request via client's register: */
    typedef cplx::client_register_type client_reg_t;
    cplx::client_register_type client_reg;
    client_reg.state = cplx::client_register_status_request;
    initiator->simple_tlm_write<client_reg_t>(
        (unsigned long)&(reg_p->call_if.ctrl.initiator_reg[client_id]),
        &client_reg, busCycles);

    /* notify initiator on consumed bus cycles for this communication: */
    initiator->adjust_global_cycle_count(busCycles);
}
(let cpu time that elapsed before this transaction pass):*/
initiator->synchronize();

/* wait for grant: */
typedef cplx::message_register_type message_reg_t;
message_reg_t msg_reg;
while(true)
{
  initiator->simple_tlm_read<message_reg_t>(
    (unsigned long)&(reg_p->call_if.ctrl.target_reg),
    &msg_reg, busCycles);
  if(msg_reg.valid==1 && msg_reg.client_id==client_id &&
    msg_reg.event==
    (unsigned int)cplx::message_register_event_method_ready
  ) break;
}
/* write arguments: */
int in_w_1;
for(in_w_1=0; in_w_1<32; ++in_w_1)
{
  initiator->simple_tlm_write<int>(
    (unsigned long)&(REG(reg_p,in,in[in_w_1])),
    const_cast<int*>(&in[in_w_1]),
    busCycles);
}
/* notify args ready: */
typedef cplx::client_register_type client_reg_t;
cplx::client_register_type client_reg;
client_reg.state = cplx::client_register_status_arg_ready;
initiator->simple_tlm_write<client_reg_t>(
  (unsigned long)&(reg_p->call_if.ctrl.initiator_reg[client_id]),
  &client_reg, busCycles);
/* notify initiator on consumed bus cycles for this communication: */
initiator->adjust_global_cycle_count(busCycles);
}

unsigned int dct32_finished()
{
  const unsigned int base_addr = 0;
  volatile dct32_register_if_type* reg_p =
    (volatile dct32_register_if_type*)(base_addr);
  unsigned int busCycles = 0;
  cplx::sw_wrapper_base *initiator = obtain_client_initiator();

  /* synchronize with initiator
  (let cpu time that elapsed before this transaction pass):*/
  initiator->synchronize();

  /* read message register: */
typedef cplx::message_register_type message_reg_t;
message_reg_t msg_reg;
initiator->simple_tlm_read<message_reg_t>(
  (unsigned long)&(reg_p->call_if.ctrl.target_reg),
  &msg_reg, busCycles);
/* notify initiator on consumed bus cycles for this communication: */
initiator->adjust_global_cycle_count(busCycles);
unsigned int client_id = initiator->get_client_id(base_addr);
return msg_reg.valid==1 && msg_reg.client_id==client_id &&
   msg_reg.event==
   (unsigned int)cplx::message_register_event_method_finished;
}

void dct32_read(int out[32])
{
    const unsigned int base_addr = 0;
    volatile dct32_register_if_type* reg_p =
        (volatile dct32_register_if_type*)(base_addr);
    unsigned int busCycles = 0;
    cplx::sw_wrapper_base* initiator = obtain_client_initiator();
    unsigned int client_id = initiator->get_client_id(base_addr);
    /* synchronize with initiator
     * (let cpu time that elapsed before this transaction pass):*/
    initiator->synchronize();
    /* read outputs: */
    int out_r_1;
    for(out_r_1=0; out_r_1<32; ++out_r_1)
    {
        initiator->simple_tlm_read<int>(
            (unsigned long)&(REG(reg_p,out,out[out_r_1])),
            &out[out_r_1], busCycles);
    }
    /* notify return ready: */
    typedef cplx::client_register_type client_reg_t;
    cplx::client_register_type client_reg;
    client_reg.state = cplx::client_register_status_return_ready;
    initiator->simple_tlm_write<client_reg_t>(
        (unsigned long)&(reg_p->call_if.ctrl.initiator_reg[client_id]),
        &client_reg, busCycles);
    /* notify initiator on consumed bus cycles for this communication: */
    initiator->adjust_global_cycle_count(busCycles);
}

void dct32_adaptor(const int in[32], int out[32])
{
    dct32_write(in); // request grant and write input data
    unsigned int busy = 1;
    while(busy != 0)
    {
        busy = !dct32_finished(); // wait for results
    }
    dct32_read(out); // read output data
}

Listing 3: Adaptor functions turning a function call into TLM transactions

These adaptor functions are also generated by the SMOG tool automatically. Though they are called from the software code encapsulated in an initiator wrapper module, these adaptor functions are generated together with the target wrapper for a function separated to hardware. This is because the adaptor functions are specific to a certain target wrapper and its register interface, but independent from the initiator wrapper module encapsulating the calling code.
This independence from the initiator module is achieved by utilizing polymorphism and accessing the initiator and its socket using a pointer to the common base class.

The creation and dispatching of a generic payload data package for every single TLM transaction is encapsulated in the `simple_tlm_write` and `simple_tlm_read` template methods of the initiator module, whose implementation is part of the COMPLEX library. Note that the value will have to be converted to the endianness of the target before using it as payload data in case the endianness of the communicating blocks differs. The actual implementation assumes that the endianness is identical. The address used in a transaction must correlate to a defined address space that is known to both initiator and target and must be specified by the user as part of the platform description and passed to SMOG when generating the adaptor functions. The address layout of the register interface is based on a struct type that contains data members for all input and output parameters of the function, as well as data members for data that needs to be exchanged to handle the protocol (handshake), as described in 4.1.1. The offsets of the members inside the struct are used for determining the addresses used in the TLM transactions. By mapping a pointer of that struct type to the base address from the platform definition, the offsets of the struct members can be directly used as addresses in the TLM transactions. An advantage of this approach is that it allows the creation of rather compact code that does not have to handle and calculate absolute addresses. Additionally, the identification of parameter accesses by using the original parameter names as member identifiers in the struct type increase the readability of the synthesised code. However, this approach comes with one restriction. Refining the virtual system by mapping it to a virtual platform will switch from a host based simulation of the SW to one that is based on an instruction set simulator (ISS). In case the cross-compiler which is used to compile the code for the ISS uses another alignment and packs e.g. array elements or struct members differently than the compiler that is used to compile the synthesised interface code, then the TLM initiator and target wrappers might map parameters to different addresses and communication via TLM transactions could fail. An alternative could be to use an explicit mapping of input and output data to the available addresses in the shared address range. But this approach was considered to result in less readable and less compact code.

Note that after transmitting any transaction a wait-statement must be executed in order to pass back the control to the simulation engine. Otherwise, the target would not get a chance to respond to a transaction. Corresponding wait statements are located in the TLM read and write methods, which suspend the initiator by the delay annotated to the TLM payload. These wait statements are activated by default but can be disabled via a constructor parameter of the wrapper module, if the number of wait statements shall be reduced to a minimum for performance reasons.

4.3 Overall interface synthesis and augmented model generation flow

Figure 22 shows how all the aspects described in the previous sections are combined into a flow for assembling an augmented TLM platform.

In step (1), the functionality that is targeted for hardware is separated from the input source using SMOG. The resulting source code is meant to be estimated with the high level synthesis tool PowerOpt. The code is complemented with a simple test bench and a Makefile which are generated by SMOG in the same step. For power estimation, input stimuli are required. To obtain input stimuli, instrumented source is generated in step (2) using SMOG from the same
input as used for the task separation in step (1). Building the input design using the instrumented source in step (2b) creates an executable that creates a binary stimuli file which can be read from the test bench that was generated in step (1) as input to PowerOpt. The estimation in step (3) then creates an augmented BACpp model of the hardware functionality. Analogous to the hardware part, the task separation using SMOG is performed on the software part of the input design in step (4). The resulting code is then estimated and annotated in step (5) using the SWAT software estimation tool. The result from software estimation is BACpp like annotated object code for the software parts. As the software estimation requires executable code, it might also require code that shall not be implemented as software. These parts must be removed from the annotated object code afterwards in step (5b). Normally, this simply involves selecting the proper object files resulting from software estimation. After creating the augmented models for hardware and software in the previous steps, corresponding TLM wrapper modules and adaptor functions are generated in steps (6) and (7), again using SMOG.

Furthermore, existing IP models for parts of the system are encapsulated in wrappers that add protocol and power state machines for non-intrusive extraction of extra functional properties. These wrapper components result from IP power and timing annotation in step (8). This step is not related to a single tool but consists for instance of characterizing given IP components and specifying the corresponding protocol and power state machines. It is described in detail in [11].
5 MP3 player example

As a proof of concept, the applicability of the system tool flow for generating an augmented TLM platform is demonstrated using an MP3 player design. This design consists of an instruction set simulator to run MP3 decoder software, a memory that holds the MP3 encoded data used as input for the decoder, a hardware accelerator for a discrete cosine transformation (DCT) which is separated from the MP3 decoder software, an AC97-like output component, as well as a router which connects the ISS with the memory, the hardware accelerator, and the output component. Figure 23 depicts the components of the MP3 player design.

In this example design the MAD MPEG Audio Decoder (libmad) [4] serves as MP3 decoder. The libmad library is patched to allow execution on the ISS and to turn the DCT routine into a call back, supporting switching of implementations. The design emplaces a patched version of the OpenRISC 1000 Architectural Simulator (Or1ksim) [5] as instruction set simulator, which allows multiple instances. The integration of the Or1ksim ISS into the TLM environment is accomplished following the Application Notes at [6]. The router and memory blocks are TLM IP models taken from the SCML 2.0 modelling library [7]. The output of the decoder algorithm is passed to an AC97-like custom IP consumer component. This component periodically reads samples from its input buffer at a frequency of 41 kHz. As the FIFO buffers for the left and right channel samples may not run empty, the processing period of this component imposes a timing constraint on the MP3 decoder design.

The described virtual platform for the MP3 player design serves as a bottom-up testing ground for parts of the COMPLEX flow and several added automation steps. The platform containing the ISS was chosen to allow execution of the design while the instrumented code of the SW block needed for a host based simulation is not available. Likewise, the
instrumented HW block inside the HWDCT TLM wrapper can be replaced by the original function for starters, until a working BACpp hardware model is available.

In a first step the MP3 player design is transformed into a plain C model. This step primarily consists of rather minor modifications of ISS related code in the software to allow execution on the host machine. The resulting executable C-model is then processed by the task separation functionality of SMOG, to obtain the separate hardware and software source code packages that can be passed on to estimation. For the hardware part, the function dct32 of the MP3 player design is used as top level function and all source files that comprise the executable C model are passed to SMOG. The tool analyses the design and generates source code, containing the dct32 function and all other needed functionality from the libmad library, as well as a test bench and a configuration file for the simulation and estimation in PowerOpt.

Figure 29 illustrates the estimation flow as it is applied to the function dct32 from the MP3 player example. The source code for the function and all additional source files that are utilised by the DCT functionality is passed to SMOG. The tool reads and analyses the source files and generates source code that contains only those parts from the original code, which are minimally necessary to implement the functionality. The result of the task separation is then passed to PowerOpt in order to obtain the BACpp for the HW DCT task. A TLM wrapper for the BACpp resulting from a PowerOpt estimation is generated by SMOG as part of the interface synthesis. Task separation and interface synthesis can be performed in a single tool session by providing multiple command options to the SMOG tool.

For the software part, SMOG is used to gather all source code that is needed to perform the decoding of MP3 data, i.e. the functionality that is provided by the software running on the ISS in the virtual platform model of the MP3 player, in order to retrieve the input source for the software estimation. As the software estimation using SWAT requires an executable model, the dct32 function is included in this code, but is located in a separate file in order to simplify a selective estimation and later replacement. Furthermore, the MP3 data is contained in form of an array, which must be replaced later on. Like the estimation of the hardware task using PowerOpt, the obtained software code is run through SWAT. The files that contain code
that is only included in order to make the input source code linkable but which shall not be considered as part of the software are ignored during estimation by proper settings in the configuration file. An annotated executable model for the software is obtained in form of object code. A manual re-integration of the annotated code produced by SWAT shows that the semantics of the code is not altered by the annotation. In addition, power and timing figures can be obtained from the annotated software model.

Executing the unmodified, annotated binary provided by SWAT does give power and timing numbers for the decoding process when run as software, excluding the dct32 function, which is part of the processed source code but ignored for estimation. The resulting binary contains an un-instrumented software version of the dct32. This implementation must be replaced in the final platform model. For the integration of the BACpp like object code generated from SWAT, the relevant annotated object files must be used when linking the final executable model, while the non-relevant object files must be replaced by appropriate object files that contain corresponding definitions for the otherwise missing symbols. Not relevant are those object files, that contain symbols for functions or arrays that are not supposed to be part of the software in the final platform. Their replacements are usually implementations of adaptor functions that transfer a function call into TLM transactions and are generated by SMOG. As depicted in Figure 25, the object file that is obtained from SWAT and which contains an annotated version of the main function must be touched in order to rename the main function, e.g. into sw_main. This is necessary, because the object file is relevant for the final platform model where we want to call the main function from a TLM wrapper that represents the software task. As the final platform is supposed to utilise the OSCI simulation kernel, not renaming the main function from the software task would cause a conflict with the main

![Figure 25: Annotation flow for the MP3-Player Software](image-url)
function inside the simulation kernel, which we do not intend to replace. Next to the TLM wrappers that are required to incorporate the annotated software task in the platform model, a definition of the call-back function _swat_bacpp or __swat_bbtrace is needed, depending on the applied SWAT annotation flow. The call-back function is called from the annotated software task and represents its interface to the trace generation. As the signature of these functions does not depend on the design, they have been made part of the complex library.

After separating the hardware and software parts of the MP3 player design, the annotated models obtained from estimation need to be incorporated into the virtual platform in order to form the augmented HW/SW system model. The integration is achieved by applying the TLM wrappers, generated by SMOG. The generated TLM initiator calls the annotated software function (sw_main) and the linked dct32 adaptor function transforms calls to the dct32 function into corresponding TLM commands. The generated TLM target wrapper for hardware collects call arguments that are transferred via TLM commands and issues a call to the contained BACpp HW model’s dct32 method. While the TLM target for the hardware DCT just replaces the handwritten version from the original platform design shown in Figure 23, the TLM initiator replaces the Or1kSim ISS and its TLM wrapper.

Figure 26 shows the resulting platform after integrating the annotated hardware and software models and extending the IP models with protocol and power state machines (PrSM and PSM). The PSM approach for non-invasive power estimation of TLM IP is described in detail in [11].

Figure 26: Augmented TLM platform of the MP3 decoder

Executing the augmented TLM platform does not only allow testing the functional correctness of the design, but also provides information on the components’ activity and their extra functional properties. This allows performing an exploration with low power consumption
and high performance (timing) as goals and for instance clock frequency and voltage as parameters. We exemplarily performed a manual exploration of the augmented MP3 decoder platform by running simulations based on two different platforms. These differed regarding the application of the TLM adaptor functions generated by SMOG. One version of the platform applied the single TLM wrapper function *dct32_adaptor*, causing the software code and hardware DCT to operate sequentially, as the software is blocked during a DCT call. Another version did not use the top-level adaptor function, but applied the adaptor subroutines to achieve a pipelining of hardware DCT and software PCM synthesis. This of course required a modification of the input source. Both platforms were simulated with several voltage and frequency settings, assuming that there are two logical voltage islands, one holding the custom hardware and another holding the rest of the system. The platform that implemented a pipelining of PCM synthesis and DCT allowed to operate the DCT at a much lower frequency (half of that of the CPU frequency) and achieved the lowest power consumption at the cost of a minor overhead in communication. Figure 27 shows power traces that were obtained by executing the pipelined augmented TLM MP3 decoder platform at a voltage of 1.8 V and frequencies of 150 MHz (CPU and IP) and 100 MHz (custom hardware DCT). At the top, the power trace for the SW is given. Please note that a preliminary CPU model was used for the software annotation, leading to assumedly rather inaccurate power values. Below follow traces of the Router IP model, the BACpp model for the custom hardware DCT, and the output AC97 IP component.

![Power traces](image)

Figure 27: Power traces for some components of the augmented TLM MP3 decoder platform
The rather unspectacular trace for the memory component is not shown, as it displays continuously high power consumption, independent from the actual operation mode (reading or idle).

Note that the LRM interface was not utilized in the manual exploration. Amongst others, due to the absence of a GRM in the MP3 decoder design. However, the augmented TLM MP3 decoder platform was used for testing the LRM interface by modifying the libmad MP3 decoder software code to perform direct calls to the LRM C API functions. It was tested to read and write the LRM register interface in the custom hardware wrapper and to request power mode changes. For instance, the hardware DCT was set to power off mode while reading from memory and powered back on afterwards. Furthermore, invalid power modes and transitions were requested to check the LRM response returned from the API functions.
6 References

[10] “Final report and tools on custom hardware estimation and model generation”. COM-
PLEX deliverable D2.4.2, January 2012