# COdenseign and power Management in PLaatform-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>WP3</td>
<td>D3.1.2</td>
<td>SNPS</td>
</tr>
</tbody>
</table>

**Final report on system simulation and profiling**

Prepared by Bart Vanthournout (SNPS); Davide Quaglia (EDALab); Philipp A. Hartmann (OFFIS); Carlo Brandolese, Gianluca Palermo, William Fornaciari (PoliMi); Héctor Posadas, Fernando Herrera (UC); Chantal Ykman Couvreur (IMEC)

Issued by SNPS  
**Document Number/Rev.** COMPLEX/SNPS/R/D3.1.2/1.1  
**Classification** COMPLEX Public  
**Submission Date** 2012-11-26 [updated 2013-02-05]  
**Due Date** 2012-10-01

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>REASON FOR CHANGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNPS</td>
<td>1.0</td>
<td>2012-11-26</td>
<td>First Version</td>
</tr>
<tr>
<td>OFFIS</td>
<td>1.1</td>
<td>2013-11-05</td>
<td>Updated BAC++/SNPS integration</td>
</tr>
</tbody>
</table>
Contents

1 Introduction ........................................................................................................................................4
  1.1 Scope ........................................................................................................................................4
  1.2 Document structure ..................................................................................................................7
  1.3 Acronym list ...............................................................................................................................8
2 GRM simulation and profiling for overhead analysis .................................................................9
  2.1 Binary size of GRM implementation .........................................................................................9
  2.2 Performance overhead and energy gain ....................................................................................10
3 SystemC API for behavioural timing and power annotations ................................................13
  3.1 Tracing interface (model frontend) ............................................................................................14
  3.2 Trace processing and reporting (backend) .................................................................................20
4 Analysis environment for system level analysis .........................................................................23
  4.1 Analysis recording infrastructure overview ............................................................................23
  4.2 Analysis results visualization ...................................................................................................29
  4.3 Integration ..................................................................................................................................32
5 Definition of interchange format for automatic design space exploration ................................33
  5.1 Design Space Definition ............................................................................................................34
  5.2 Explorer Simulator interface .....................................................................................................38
  5.3 Additional DSE Integration Interfaces .....................................................................................40
6 Techniques for System-Level Simulation and Profiling with SCoPE+ ........................................41
  6.1 Requirements of the simulation technology for DSE ..............................................................41
  6.2 From SCoPE to SCoPE+ ...........................................................................................................42
  6.3 SCoPE+ Features and Requirements on its simulation engine .............................................44
  6.4 New features implemented in the simulation engine ...............................................................48
7 SystemC Network Simulation Library (SCNSL) .........................................................................65
  7.1 Implementation of network simulation in SystemC .................................................................66
8 References ......................................................................................................................................71
1 Introduction

Deliverable D3.1.2 is aimed at providing a description of the work executed in Task 3.1 related to the development of a proper system simulation and application profiling environment. This document is mostly based on D3.1.1, the “Preliminary report on system simulation and profiling” and gives an updated, but self-contained overview of the results in T3.1. It is the final report on the system simulation and profiling tools that have been developed and are integrated into the COMPLEX framework in T1.3 and T1.4, respectively.

1.1 Scope

The purpose of Task 3.1 is to enhance the virtual platform simulations from WP2, which are generated from Task 2.5, with analysis and profiling tools so that a design space exploration loop can be enabled. The analysis and profiling metrics that need to be implemented are derived from T1.1 as well as from T1.2. These metrics are to be used to drive the performance optimizations for the system components via the cost functions defined in T1.2.3.
The system simulation and application profiling environment corresponds to the infrastructure that is used to simulate the bus cycle accurate SystemC model with self-simulating power and timing models (n) in the COMPLEX design flow (Figure 1). The result is the simulation trace (o) of the design flow.

The goals for Task 3.1 are:

- Definition of a system-wide multi-level simulation and profiling strategy
- Simulation of communication for networked embedded systems
- Identification of metrics useful for DSE

This report discusses the following aspects of system simulation and application profiling:

- Virtual platform simulation and analysis environment generating analysis traces to drive the performance optimizations for the system components
- Implementation of the (SystemC) API for behavioural timing and power annotations
- Extension of SystemC environment with a prototype of an abstract, task based, virtual platform simulation environment supporting functional and behavioural power estimation techniques

- Definition of the interchange format of the system-level metrics to be passed from the executable system models to the design space exploration framework

- Development of the interface for the multi-level system simulation and performance analysis tool able to provide to the DSE tool the system-level metrics obtained from the simulation

- SystemC library support for the simulation of communications among networked embedded systems
1.2 Document structure

According to the scope of the work done in Task 3.1, this deliverable is organized in line with the different subtasks of T3.1. Work in T3.1 is focused to extend and enhance the existing system simulation and application profiling environments in specific areas so that they can be integrated into a single environment, so it makes sense to discuss these specific areas separately. Therefore D3.1.2 contains the following sections:

- **GRM simulation and profiling for overhead analysis**
  In this section the analysis metrics and instrumentation provided by the GRM component are discussed. It provides with specific runtime management analysis capabilities in the overall simulation environment.

- **SystemC API for behavioural timing and power annotations**
  This section discusses a generic analysis instrumentation API for power and timing analysis that can be tied to different backend analysis tools.

- **Analysis environment for system level analysis**
  In this section an analysis backend is discussed that provides with generic analysis data collection and visualization features, this extension supports functional and behavioural power estimation techniques for abstract, task based, virtual platform simulation.

- **Definition of interchange format for automatic design space exploration**
  In this section the exchange format that is used to pass data about the design metrics from the executable system models to the design space exploration framework as well as information about the system configurations that are to be evaluated by the design space exploration.

- **Techniques for System-Level Simulation and Profiling with SCoPE+**
  In this section the simulation environment is discussed that allows for a fast assessment of the best implementation solution for an embedded system at early stages of the design flow.

- **SystemC Network Simulation Library (SCNSL)**
  This section discusses extensions to the simulation environment related to the simulation of packet-based networks in SystemC.
1.3 Acronym list

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AEM</td>
<td>Application Executable Model</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CEA</td>
<td>COMPLEX Eclipse Application</td>
</tr>
<tr>
<td>CFAM</td>
<td>Concurrent Functional Application Model</td>
</tr>
<tr>
<td>CMC</td>
<td>COMPLEX Model Checker</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DoW</td>
<td>Description of Work</td>
</tr>
<tr>
<td>DRM</td>
<td>Detailed Resource Modelling</td>
</tr>
<tr>
<td>DSE</td>
<td>Design Space Exploration</td>
</tr>
<tr>
<td>GCM</td>
<td>Generic Component Model</td>
</tr>
<tr>
<td>GQAM</td>
<td>Generic Quantitative Analysis Modelling</td>
</tr>
<tr>
<td>GRM</td>
<td>Generic Resource Modelling</td>
</tr>
<tr>
<td>GRM</td>
<td>Global Resource Manager</td>
</tr>
<tr>
<td>HLAM</td>
<td>High-Level Application Modelling</td>
</tr>
<tr>
<td>M2T</td>
<td>Model to Text</td>
</tr>
<tr>
<td>MARTE</td>
<td>Modelling and Analysis for Real-time and Embedded Systems</td>
</tr>
<tr>
<td>MDA</td>
<td>Model Driven Architecture</td>
</tr>
<tr>
<td>MDD</td>
<td>Model Driven Development</td>
</tr>
<tr>
<td>MOF</td>
<td>MetaObject Facility</td>
</tr>
<tr>
<td>MOST</td>
<td>Multi-Objective System Tune</td>
</tr>
<tr>
<td>MTL</td>
<td>Model to Text Language</td>
</tr>
<tr>
<td>NFP</td>
<td>Non-Functional Properties</td>
</tr>
<tr>
<td>PAM</td>
<td>Performance Analysis Modelling</td>
</tr>
<tr>
<td>PDM</td>
<td>Platform Description Model</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RTES</td>
<td>Real-Time and Embedded Systems</td>
</tr>
<tr>
<td>SAM</td>
<td>Schedulability Analysis Modelling</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>TBC</td>
<td>To Be Completed</td>
</tr>
<tr>
<td>TBD</td>
<td>To Be Defined</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
</tbody>
</table>
2 GRM simulation and profiling for overhead analysis

IMEC worked on Use Case 2 to analyse the GRM overhead, based on GRM metrics identified in Use Case 2.

As reported and detailed in the COMPLEX deliverable D3.5.3 “Final report on run-time management” [3] (Section 10.2), the analysis mainly focussed on the binary size of the GRM implementation, the performance overhead, and the energy gain. To that end, the GRM has been integrated in a POSIX implementation of the audio-surveillance video application.

First, the GRM has then been deployed and tested on an X86-based platform running at 800 MHz. Results are summarized in the following subsections. Second, the GRM will be evaluated too on an ARM-based TI OMAP 4460 embedded platform running at 700 MHz. Obtained results will be reported in Deliverable D4.2.2, entitled “Final report on evaluation of design tools”.

2.1 Binary size of GRM implementation

The GRM is implemented in C and compiled into a library libgrm.a, which is then linked to the application. The current binary size of libgrm.a is 100.4 Kbytes, without taking the GRM databases into account. This binary size is independent from the target platform and application.

Two GRM databases are also required to store the high-level specification of the platform, the IP core types, and the application configurations:

- The current binary size of the platform database is 1506 bytes in the COMPLEX use case 2, where the platform consists of 5 ReISC cores and 1 HW accelerator, with 5 available power modes.

- The current size of the application configuration database is 88 bytes in the first development phase of the COMPLEX use case 2, where the application consists of three jobs (i.e., alarm processing, audio activity detection, and video image processing).
2.2 Performance overhead and energy gain

Figure 2 illustrates the energy-per-frame evolution of our demonstrator for two platform constraints (different energy budgets, same battery duration) with and without the GRM. Due to an optimized adaptive selection of application configurations, our GRM allows optimizing the QoS of the application while keeping the platform battery alive during its whole required duration. In contrast, this cannot be ensured without such an RRM framework. Without the GRM, only one application configuration may be activated from the start of the application. With the GRM, several ones may be successively activated during the run of the application: in this demonstrator, among the 16 available application configurations, 4 (resp. 9) are activated to satisfy the platform constraint 1 (resp. 2).
Figure 3: Performance of GRM initialization

Figure 3 illustrates the CPU processing of the GRM services executed at initialization on the X86-based platform. Both GRM_ConfigurePlatform() and GRM_ConfigureApplication() require more processing due to parsing of high-level platform specification and available application configurations. Nevertheless, these services are executed only once without any run-time overhead.

Figure 4: Performance of GRM run-time services
Figure 4 illustrates the CPU processing of the GRM services executed at run time on the X86-based platform. The performance of `GRM_SelectApplicationConfiguration()` includes the one of `GRM_EstimateElapsedEnergy()`. The performance of `GRM_ReconfigurePlatform()` includes all waiting times for IP cores being ready and for IP core reconfiguration. Nevertheless the average execution time on the X86-based platform is still < 0.5 ms.

![Figure 5: GRM CPU processing overhead](image)

Finally, a global analysis of the GRM CPU processing overhead compared to the application processing shows an overhead of only 1.16% on the X86-based platform @800 MHz (see Figure 5). Observe that this overhead is only 0.6% on a TI OMAP 4460 embedded platform running at 700 MHz. This shows that the overhead of the proposed run-time energy management mechanism is negligible and there is no significant impact on the application.

In conclusion, the experiments performed so far should indicate that the proposed combined approach of design-time exploration of application configurations with run-time optimization can improve the overall QoE of the system.
3 SystemC API for behavioural timing and power annotations

The basic infrastructure for behavioural timing and power annotations within the COMPLEX framework is based on a flexible and generic tracing interface. This tracing framework can be used to trace arbitrary user data over time, including non-functional properties like power consumption. It is used within the simulation platform to provide the outputs required by the user and the exploration and optimization framework (see D3.4.2 [12], and Section 5), following the requirements as defined earlier in the project [2].

Table 1: Requirements for simulation traces and analysis (taken from [2])

<table>
<thead>
<tr>
<th>RDV1</th>
<th>It should be able to present the simulation trace graphically (e.g. using known viewers on top of well established trace formats like VCD).</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDV2</td>
<td>It should be able to maintain the design hierarchy in the reporting (should be represented in a hierarchical naming convention, such as TopModule.SubModule1.Submodule2.Port)</td>
</tr>
<tr>
<td>RDV3</td>
<td>It should be able to trace the activity of each module of the platform in order to understand in detail the system behaviours. Core activity, power consumption for each core, memory and interconnection utilization, are possible examples.</td>
</tr>
<tr>
<td>RDV4</td>
<td>It should be able to generate compact metrics to evaluate the generated instance of the design space.</td>
</tr>
</tbody>
</table>

To enable both, a user-driven analysis based on the visual presentation of simulation results as well as the integration into an automated exploration loop as performed by MOST (see Section 5), different kinds of post-processing back-ends are needed. Especially for the generation of compact performance metrics, a dedicated reporting and accumulation mechanism is necessary. This usually implies on-the-fly pre-processing of the intermediate power values, for instance to compute the overall energy consumption of a complete simulation of a particular application scenario or configuration. Secondly, the integration with the graphical analysis framework of the Synopsys Virtual Platform solution (see Section 4) is needed. This is then usually based on more detailed traces of values/events over time. Both aspects are addressed by the tracing and post-processing mechanisms developed by OFFIS and are presented in Section 3.2.

Since the COMPLEX simulation models are mostly based on SystemC TLM-2.0, the built-in tracing capabilities of SystemC (based on sc_trace) are not sufficient, because they cannot directly cope with temporal decoupling and local time offsets. In all of the estimation techniques developed in WP2, such temporal decoupling techniques are used within the BAC++ simulation models to improve the simulation performance by reducing the number of synchronisations with the SystemC discrete-event simulation kernel. The underlying core technique of the annotation API presented to the user is based on so-called timed value streams, which are described in the following Section 3.1. These streams support local simulation time offsets to record values “in the future” based on a more flexible time handling, either via (value,starting time) or (value,duration) tuples. An additional, block-based annotation API is available as well, required to relate source code structures with abstract execution times and (potentially multiple) data streams.
The overall architecture of the COMPLEX trace generation framework is sketched in Figure 6. The power model within the observer of each augmented component in the simulation (custom hardware [9], software [7], or Power State Machine-enabled IP [8]) provides a given set of default streams (static/dynamic/overall power, current power mode), which can be selected by the user according to the hierarchical name of the component.

Additional, user-defined streams can be defined as well. In that case, the driving user processes are responsible for time synchronisation of the stream object(s), while the default streams of the power observers handle this transparently.

![Figure 6: Sketch of the BAC++ trace generator](image)

Before being processed by the final analysis backend, optional pre-processor objects can be applied to one or more input streams in order to get the desired output streams. This pre-processing is useful for data reduction, compositing, and other intermediate value/time transformations.

### 3.1 Tracing interface (model frontend)

The basic frontend API for the tracing interface is based on stream objects to collect obtained simulation data including support for a local time offset. For convenience, each stream is a first-class object in the SystemC object hierarchy and can be found according to its hierarchical name (at least as base-class object of type `timed_stream_base`). Although streams can be created during elaboration only, they can be enabled and disabled at any time during simulation. A disabled stream simply ignores all values written to it, while still respecting the local time offset.

Currently, the default C++ and SystemC data-types, as well as their variants augmented with a physical dimension taken from (a subset of) the Boost.Units framework [10] are supported for tracing directly. An extension mechanism to support user-defined value types has been developed and requires the implementation/specialisation of a `tracing_traits` class and its registration with the tracing framework (see Section 3.2). This is mostly needed for proper backend support, since especially the file generation backends require an appropriate implementation for dumping the values into the final database.
To add basic support for temporal decoupling – which is a common modelling technique in TLM-2.0 – several aspects need to be considered for handling the incoming tracing values, which are discussed in the following:

- Pushing (future) values to a stream, including support for resolving potential conflicts
- Synchronising/advancing a stream’s local time with respect to the global simulation time, without depending on SystemC synchronisation primitives
- Reading values from a decoupled stream

In contrast to the automated sc_trace interface with in-kernel support for updating the current values at any time advance during the simulation, the timed value streams presented here require explicit value and time updates. As part of the block-based annotation extension, a thin wrapper around plain variables is available, allowing to automatically push changes of the aliased variables to an associated stream.

### 3.1.1 Pushing values to a timed stream

A timed value stream is a strongly typed C++ class template with two basic member functions for pushing data into the stream (where value_type corresponds to the template parameter of the stream):

```cpp
void push( value_type const & value,
           time_type const & duration = sc_core::SC_ZERO_TIME ) = 0;

void push( time_type const & offset,
           value_type const & value,
           time_type const & duration = sc_core::SC_ZERO_TIME ) = 0;
```

The first alternative adds a given value for a given duration to the current end of the stream. Since each stream maintains a local time offset, this time offset is automatically advanced by the given duration, when this interface call is used. This interface is particularly well suited for annotating local computations without requiring external synchronisation in-between. In COMPLEX, the hardware and the software BAC++ annotations use this technique, since external synchronisation is explicitly modelled and visible within the components.

The second push alternative can be used to write (future) values to a stream, delayed relatively to the stream’s local time as given by the offset parameter. This is mostly used in case of an unknown duration (indicated by SC_ZERO_TIME). In this case, the value is assumed to be held indefinitely until overwritten again. If no duration is given, the local time of the stream is not advanced.

A synchronous timed stream, i.e. one that is not decoupled from the global simulation time, can be obtained by using one of the following three mutually equivalent push calls:

```cpp
str.push( sc_core::SC_ZERO_TIME, value, sc_core::SC_ZERO_TIME ); //(2)
str.push( value, sc_core::SC_ZERO_TIME );                      //(1)
str.push( value );                                             //(1)
```

In either of these cases, the local time offset is not advanced and a new value is pushed to the stream, being semantically held until the next value is pushed.
Secondly, the semantics for resolving multiple pushes to a single stream needs to be defined. Especially since conflicting pushes can occur in certain cases, e.g. by pushing overlapping (offset,duration) intervals to a stream. This usually happens, when there are multiple SystemC processes (i.e. initiators) triggering updates in a particular component at the same SystemC timestamp. In loosely-timed simulations, this may frequently happen in both interconnect and even target components.

It is assumed, that all pushes to a timed stream originating from a single SystemC process (local or remote) have strictly monotonous time advance. Therefore, for such simple cases, no conflict resolution is required. During the simulation, multiple pushing processes lead to an error and a conflict resolving stream needs to be used.

When using the duration-based push API, empty periods can occur in a timed stream. To define a value for these periods as well, the streams provide a so-called silence value, which can be set and updated at any time during the simulation. Within the BAC++ library, this is especially useful for the augmented custom HW components that need to provide dynamic power consumption for idle periods as well. A special parameter can be set to fill these silence periods with the last written value, effectively ignoring explicit end times of duration pushes.

The conflict resolution happens in terms of a (potentially user-defined) MergePolicy template parameter. A MergePolicy provides a resolution function taking a list of not strictly monotonously ordered (value,duration) tuples and transforms those to a single strictly ordered list of resolved tuple(s) with new values (and potentially new durations). In case of non-overlapping tuples, the ordering is simply restored. For overlapping entries, the following default merge policies are provided by the implementation already:

- **timed_accumulate_policy** (with optional `<BinaryFunction>`)  
  Split all overlapping tuples in disjoint durations by duplicating the values, then applying `std::accumulate` to each list of conflicting values to obtain the final value for each disjoint interval.

- **timed_override_policy**  
  Split all overlapping tuples in disjoint durations, taking the last value for each resulting time interval.

- **timed_discard_policy**  
  Split all overlapping tuples in disjoint durations, taking the first value for each resulting time interval.

- **timed_error_policy** (default)  
  Report an error in case of any overlapping durations.

Since this annotation framework is especially targeted at tracing extra-functional properties that are frequently tied to physical quantities, the accumulate policy is very useful to keep at least integral accuracy: When tracing power values without the correct time resolution, the correct “current” power consumption cannot be defined. By accumulating simultaneous power annotations, at least the overall energy consumption is retained.
3.1.2 Synchronising stream objects

To finalise the values written to a particular temporarily decoupled timed stream, the pushes need to be explicitly committed. This effectively freezes all tuples written to the stream until the current local time offset as it has advanced by pushes to the stream since the last synchronisation. Subsequent pushes with explicit timestamps earlier than the committed time lead to a run-time error.

In addition to the advancement of the local (start) time, attached tracing observers (pre-processors or sinks, see Section 3.2) are notified that new data is available for processing.

In case of multiple independent streams in a single component, the local time offsets need to be consistent as well. Secondly, the driving SystemC process frequently needs to consume the SystemC simulation time as well. Both operations are tightly coupled, therefore a second overload of sync functions is provided by the timed streams. Having the same semantics for their arguments, these explicit sync functions return the offset to the current absolute SystemC simulation and perform a commit on the other streams in the current component as well.

This gives the following API for timed stream synchronisation:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void commit()</td>
<td>commit all pending pushes</td>
</tr>
<tr>
<td>void commit( time_type const &amp; offset )</td>
<td>commit until offset</td>
</tr>
<tr>
<td>time_type sync()</td>
<td>commit &amp; synchronise</td>
</tr>
<tr>
<td>time_type sync( time_type const &amp; offset )</td>
<td>... until offset</td>
</tr>
</tbody>
</table>

With the sync function, a common idiom to finalise a local computation is the sync/wait call using one of the streams explicitly as shown below. Alternatively, a convenience macro to mark such a synchronisation point explicitly is available as well.

```cpp
sc_core::wait( [stream].sync() );
CPLX_SYNCHRONISATION_POINT();
```

The synchronisation in terms of an explicit stream returns the available time offset of this particular stream, and therefore the component is sync’ed until this time. Any streams with additional future pushes will have an remaining offset even after the synchronisation. Contrary to that, the CPLX_SYNCHRONISATION_POINT looks for the largest time offset and advances the local clock until the latest explicitly pushed timestamp. This may result in periods of silence in (some of) the component’s streams.

3.1.3 Block-based annotations

A frequent use case of the annotation framework is the augmentation of application source code with extra-functional properties. If only a single stream is driven from within a process, no inter-stream synchronisation is needed and the local time of the component is equal to the local time of the (only) stream.

If multiple streams are maintained, each of which has different update characteristics, keeping the local time offsets consistent quickly becomes a tedious task. Therefore, a higher level abstraction for block-based annotations of execution time durations has been developed. This convenience API separates the time annotation from the actual value updates again and handles stream syncs transparently for the user.
enum process_state { IDLE = 0x0, BUSY = 0x1 };

timed_stream<process_state> state_tr;  // silence value == IDLE
 timed_stream<unsigned> mem_load_tr; // silence value = 0

void faculty(int in0, int &out0)
{
    // some traceable variables
    timed_var<process_state> state ( state_tr );
    timed_var<unsigned> mem_load ( mem_load_tr );

    // functionality
    CPLX_TIMED_BLOCK( sc_time(500, SC_NS) )
    {
        state = BUSY;
        mem_load = 10;

        // functionality ...
    }

    // data dependent loop
    while(tmp0) CPLX_TIMED_BLOCK( sc_time(200, SC_NS) )
    {
        state = BUSY;
        mem_load = 2;

        // functionality ...
    }

    CPLX_SYNCHRONISATION_POINT();
    return;
}

Listing 1: Simple example of tracing multiple local values

The different basic blocks for annotations are wrapped within `CPLX_TIMED_BLOCK` scopes containing the duration of the block. Instead of explicit pushes to different streams, `timed_var` objects can be used and updated via plain assignments. These timed variables (or their `timed_ref` counterpart, aliasing an existing variable) are used to implicitly push the corresponding tuples upon exit of the annotated block and perform a sync (without a SystemC wait) automatically.

Marking the synchronisation point to yield to the SystemC kernel and to advance the global simulation time still needs to be done explicitly, to achieve maximum simulation performance.
3.1.4 Reading from a stream object

Complementary to the pushes to a timed stream, the post-processing parts of the annotation and tracing framework need to read values from the streams. As sketched in Figure 6, each stream can be attached to multiple readers, each of which gets notified at each commit to the corresponding stream. Read operations from different consumers are handled separately, to enable decoupled processing in different consumers as well.

In general, the notification and the time advance within the stream consumers are independent of the SystemC simulation (time) infrastructure. In the general case, this can even enable the stream processing in a different host thread to offload the main simulation CPU in today’s multi-core simulation environments.

One of the core requirements for reading tuples from a stream to process them in intermediate pre-processor stages is to support splitting a tuple with a given duration into smaller durations. To support physical quantities, this may require a specific modification of the value distribution between two tuples. For this purpose, the SplitPolicy can be defined for each stream individually. This SplitPolicy is symmetric to the MergePolicy described before, and is used to split a tuple of a given duration into two tuples with potentially different values. Internally, the SplitPolicy is also used within the implementation of the MergePolicy’s overlapping duration splits. Predefined split policies are the timed_constant_policy (i.e. keeping the same value in both tuples), and the parametrisable timed_exponential_decay_policy.

Reading pending values from a decoupled stream can be done in several ways. For most pre-processing operations, the (const)tuple_iterator based interface is sufficient. The streams provide the output interface for consuming the committed data sketched in the following listing:

```cpp
// read a value at a given time (between start and end time)
value_type const & get ( time_type const & offset = sc_core::SC_ZERO_TIME ) const;

// read (and potentially split) the first tuple
tuple_type front() const;
tuple_type front( time_type const & until ) const;

// drop first tuple
void pop_front();
tuple_type pop_until( time_type const & until ) const;

// get iterators to the pending tuples
const_tuple_iterator begin() const;
const_tuple_iterator end() const;

// consume pending tuples
void pop_all();
```

Additionally, various querying functions about the different time offsets and other utility functions are available in the stream base class.

---

1 The current implementation is prepared for such an optimisation, but it has not been implemented, yet.
3.2 Trace processing and reporting (backend)

The trace processing of the individual timed streams during a simulation are handled by a central trace generator backend. This central backend provides an interface to insert pre-processors to individual (sets of) streams as well as the run-time activation/deactivation of these streams.

After elaboration, the tracing streams (sources), the pre-processors and the different individual backend sinks (trace file generation, direct integration with graphical analysis environments, collecting statistical data) form a directed, acyclic dependency graph. The de/activation API respects this graph to turn off as many individual components during the simulation as possible.

Currently, only a separate VCD file generation backend has been completed. This backend already allows the analysis of the recorded traces with graphical viewers, such as GTKwave. Future integration with the other tools in the COMPLEX design flow is outlined below.

3.2.1 Pre-processing of tracing streams

In order to further increase the flexibility of the annotation and tracing framework, individual streams can be pre-processed before being written to the tracing database or file. Stream pre-processors are special classes (derived from `timed_stream_processor`), that read from a (set of) streams and provide another (set of) streams as their output.

Meaningful examples for pre-processors could be

- window averaging, for the sake of data reduction
- calculating a cumulative sum of data values (e.g., power) over a time frame
- combining individual streams of components in order to get a view of a composite

Additionally, the tracing granularity can be modified by such a pre-processing during the simulation as well, as shown in Figure 7.

![Figure 7: Flexible granularity of trace recording](image)

Each stream processor is notified of every commit on any of its incoming streams. Upon notification, a special `processing` function is called, which performs the actual read operations on the streams and pushes new values to its output streams. The following listing shows an example of a simple coarse-grained averaging stream processor, averaging all receives tuples between two explicit synchronisation requests:.
void processing() {
    time_type duration = in.end_time() - last_sync_
    value_type average = value_type();
    for( tuple_iterator it = in.begin(); it != in.end(); ++it )
        average += *it.value() * it.duration() / duration;
    in.pop_all();
    out.push( average, duration );
    out.commit();
    last_sync_ = in.end_time();
}

If a stream processor does not provide any output streams, it is called a *sink* processor. These classes are usually pre-defined and part of the trace generation backend.

### 3.2.2 Integration with Synopsys Virtual Platform solution

Usually, a stream consumer can only read values until the (absolute) time the stream has committed values to. When running independently from the SystemC simulation kernel, this can result in running behind the SystemC time progression, since there is no direct way to enforce a computation to run immediately *before* a SystemC time step, but *after* all current processes have been evaluated.

To integrate the annotation and tracing framework with the Synopsys Virtual Platform solution, the corresponding tracing backend makes use of the *Generic Data Analysis* (see Section 4.1.3), which has been added to the Synopsys instrumentation APIs recently. In GDA instrumentation, there is an option of supplying an explicit time *PullIPT* which will be asked to provide the time by the GDA analysis monitor. This example illustrates how this can be implemented. The GDA analysis will only ask this IPT for `getCurrentTime()`.

```cpp
#include "ipt/instrumentation_ifs/AbstractCoreTimeIPT.h"

struct timed_stream_time_ipt :
    public nCoWare::nIPT::AbstractCoreTimeIPT
{  
    explicit
timed_stream_time_ipt( timed_stream_base const& stream )
        : nCoWare::nIPT::AbstractCoreTimeIPT( stream.name() + ipt_name_suffix,
                                                ipt_description )
        , stream_ref_(stream)
    {}  

    virtual unsigned long long getCurrentTime() {
        return sc_time_cast( stream_ref_.local_time() ).value();
    }

private:
    timed_stream_base const& stream_ref_;  

    static const std::string ipt_name_suffix /* = ".time" */;
    static const std::string ipt_description /* = "local time of stream" */;
};
```

Listing 2 Explicit time handling when using the Generic Data Analysis APIs
This explicit time handling provides a solution to the limitations described in the previous version of this report. Fully asynchronous value recording based on temporally decoupled processes as well as user-defined pre-processor hierarchies are now supported in the Synopsys tracing backend. This enables a very detailed trace analysis based on the accompanying graphical debugger interfaces, as shown in the following screenshot. This figure shows both, a detailed value trace and a sliding window average in terms of a colored text box. Other chart types are available as well, see Section 4.2.

![Figure 8 (Dynamic) power-trace of a BAC++-augmented DFT-block](image)

The basic implementation of the Synopsys backend processors is sketched in the following sink processing function. For the supported primitive types (Integer, Double, Enum), different traits classes are provided to abstract from the (physical) values to one of these underlying data types.

```cpp
void processing () {
    // process all pending tuples
    while (in.begin() != in.end()) {
        // extract front value - local time is internally managed
        value_type value = traits_type::value_cast ( in.front() );

        m_value_ipt_apsesetValue (0, value); // record new value via SNPS GDA
        in.pop_front (); // consume tuple
    }
}
```

### 3.2.3 Collecting simulation metrics

According to the requirement **RDV4**, given in Table 1, compact metrics for the automatic design space exploration need to be generated as well. The basic report interface for the MOST exploration tool is given in Section 5.

To enable the collection of compact metrics, user-defined streaming *sink processors* producing a (set of) accumulated values from a (set of) streams can be used. These specific metric backends then report the value(s) of the metric at the end of the simulation (or explicitly upon request) in a MOST compatible XML structure. The computation of the design metrics can usually run independently of the SystemC simulation time.
4 Analysis environment for system level analysis

In the COMPLEX design flow the system simulation and profiling is required to provide with the necessary metrics and results to enable a design space exploration loop. This implies that the simulation environment is enhanced in order to provide with the relevant power and timing metrics for this exploration loop to be meaningful. The Synopsys Virtual Platforms solution serves as a simulation environment for software development, verification and validation flows as well as architecture or design space exploration.

In support of design space exploration the Synopsys Virtual Platforms solution provides a comprehensive and extendable platform-centric analysis framework. The goal of this framework is to allow the designer to efficiently debug and optimize a system architecture that depends on a strong interaction of the platform components or that depends on software running on different cores or subsystems. This means that the analysis framework is not component centric but platform centric. It allows you to inspect and correlate the state and history of software running on multiple cores, the activity in the system in general, and the interaction between the different platform components with each other as well as with the software.

Synopsys’s analysis framework provides the visibility to debug and optimize across the boundaries of processes, OSes, cores, and hardware. A plug-in mechanism allows the analysis framework to become aware of the underlying OSes running on the different cores, or the interaction between different hardware elements that communicate over a system level interconnect. This way you have full visibility into the system.

4.1 Analysis recording infrastructure overview

Several basic concepts are underlying to all tracing and analysis in the Synopsys Virtual Platform solution, as shown in the following table.

<table>
<thead>
<tr>
<th>Instrumentation.</th>
<th>Are points in the design where API calls are made to report information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitors</td>
<td>Are objects that can attach to instrumentation points to process the information that is reported and optionally record it.</td>
</tr>
<tr>
<td>Results</td>
<td>Is information recorded by a monitor.</td>
</tr>
<tr>
<td>Views</td>
<td>Are charts or tables in the user interface that present recorded results.</td>
</tr>
</tbody>
</table>

Instrumentation points may be sprinkled throughout the design. Some have been built into the SystemC kernel by Synopsys, and some may be coded into models of buses, peripherals, processors, and so on. The following figure illustrates this.
Different types of instrumentation support different types of monitors. A signal port’s instrumentation supports a monitor that traces signals. A TLM port supports monitors that trace and compute statistics. These monitors can be attached during a simulation via the Synopsys debug and analysis tools. The following figure illustrates this.

Typically, monitors record their results to a database that can be viewed in the Synopsys analysis viewer, or post-processed via Tcl scripting commands in the debug tools. The following figure illustrates this.
SystemC modules that are instrumented for software analysis contain Instrumentation Points (IPTs). An IPT provides an interface to the outside world for a specific type of instrumentation. The following IPT types are available:

- **Push IPTs**: These are used by the module to “push out” information to the outside world in the form of notifications.

- **Pull IPTs**: These are used to “pull” information from the instrumented module.

The term IPT owner is used for a module containing IPTs. Which IPTs are available on a given IPT owner varies from module to module. In addition to the IPTs explicitly provided by the model, the analysis framework adds a set of derived IPTs on top of the available IPTs. These IPTs implement extra functionality using information provided by the IPTs already provided by the model. These IPTs can be used by software analyses as well.

The central object for implementing an analysis is a monitor. A monitor uses the information provided by one or more IPTs to perform an analysis, record information to a database, and so on. You can instantiate several monitors and attach them to a module in order to perform analyses on that module.

Internally, a monitor consists of a set of probes, where each probe matches a specific type of IPT. Whenever a monitor is instantiated, the probes of the monitor are connected to the corresponding IPTs of the module. The attached probes are then used by the monitor to get information from the module’s IPT(s). For example, a specific monitor could consist of an EndInstructionExecutionProbe and a SymbolLookupProbe, which are connected to the EndInstructionExecutionIPT and SymbolLookupIPT of a core; the monitor can then use these probes to trace which symbols are executed on the core it is attached to.

Finally, monitors are instantiated through monitor factories. For every type of monitor, there is a monitor factory instance which does the instantiation of a monitor and connects it up to the IPTs.

The following figure shows an IPT owner that has two monitors connected to it. Both monitors are attached to the IPT of type B.
4.1.1 Implementing Basic Monitors

Monitors are implemented by sub-classing the Monitor class from the IPT framework. The constructor of the monitor is responsible for registering all the necessary information about a monitor. This includes creation and adding of probes (using addProbe()) and the definition of the monitor attributes. This information is used by the factory to determine which monitors can connect to which IPTs.

After a monitor has been instantiated and connected to an IPT owner, the initialize() method is called on the monitor. Every monitor is free to re-implement this method to do initialization of the database, initialization of member variables that depend on the owning module, and so on.

In order for the monitor type to be detected by the system, a monitor factory has to be present for each monitor class. Usually, it suffices to instantiate a MonitorFactory template, templated with the monitor class. If custom connection behavior is wanted for a certain type of monitor, one has to implement its own factory.

Probes

As mentioned before, there are two classes of IPTs; consequently there are two classes of probes, namely push and pull probes. Both are instantiated by the monitors in the same way, and both are added using addProbe() in the constructor of the monitor. However, there is a difference in usage by the monitor. For the pull probes, the monitor directly instantiates the actual probe, and calls the provided methods on the probe. For the push probes, it is the IPT that will call the provided methods on the interface of the probe; therefore, the monitor must do one of the following:

- Inherit from the probe interface, implement the behavior of the methods on the interface, and use a proxy probe to forward the calls on the probe to the methods implemented in the monitor
- Create its own subclass of the probe, re-implement the method’s behavior in the probe, and forward calls to the monitor

Configuring Monitors

Configuration of monitors is done through monitor attributes. At construction time, each monitor registers its attributes using addAttribute(). If a monitor wants to perform actions whenever an attribute changes, it should additionally reimplement canSetValue() (called to check whether the attributes to be set are valid) and handleAttributeValuesSet() (called whenever attribute values have been changed).

The attribute value is represented by an object of type nCoWare::Variant. Variants provide a wrapper for many basic types of values, including integers, strings, pairs, lists, and so on. Every attribute has a default attribute value, and optionally a description of the possible values. The possible values can be an enumeration, represented by a Variant of type nCoWare::Variant::LIST, where each list element is a string.
Monitor Commands

A monitor can have custom commands defined on it. The intent is that these commands can be executed on the monitor from within a tool. Custom commands are added by calling \texttt{addCommand()} in the constructor of the monitor and reimplementing \texttt{handleExecuteCommand()}. 

Custom Monitor Connections

When custom behavior is required when initializing a monitor, you have to instantiate its own subclass of \texttt{MonitorFactory}, and reimplement the following functions:

- \texttt{canCreate()} is called by the framework to check whether a monitor of this factory can connect to a given IPT owner. By default, the factory checks if the IPT owner has at least one IPT that matches one of the probes of the monitor.

- \texttt{createInstance()} is called whenever a monitor is instantiated.

4.1.2 Available Monitors

The following table lists and describes the default available monitors for each type of instrumentation. Obviously additional monitors can be created for custom analysis views and data collection.

<table>
<thead>
<tr>
<th>Instrumentation</th>
<th>Monitor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal Ports</td>
<td>Value Trace Monitor</td>
<td>Records value changes of a signal or variable over time.</td>
</tr>
<tr>
<td>Memories and Registers</td>
<td>Register Trace Monitor</td>
<td>Records accesses and callback events on registers and bitfields</td>
</tr>
<tr>
<td>TLM ports</td>
<td>TLM port Trace Monitor</td>
<td>Records transactions and transfers on a communication port</td>
</tr>
<tr>
<td></td>
<td>Transaction Statistics Monitor</td>
<td>Records interval based transaction statistics for a communication port</td>
</tr>
<tr>
<td>Interconnect Instrumentation</td>
<td>Bus Debug Monitor</td>
<td>Records transactions and transfers on a communication port</td>
</tr>
<tr>
<td></td>
<td>Bus Analysis Monitor</td>
<td>Records interval based communication path statistics and interconnect resource utilization statistics for an interconnect</td>
</tr>
<tr>
<td>Resource Instrumentation</td>
<td>Memory Analysis Monitor</td>
<td>Records resource utilization statistics</td>
</tr>
</tbody>
</table>
### CPU Core Instrumentation

<table>
<thead>
<tr>
<th>CPU Core Instrumentation</th>
<th>Context Trace</th>
<th>Function Trace</th>
<th>Performance Profile statistics</th>
<th>Memory access statistics</th>
<th>Instruction Trace</th>
<th>Terminal Trace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A set of monitors that are used to create SW analysis views for instruction traces, function traces, cache analysis etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model Logging</th>
<th>Logging Monitor</th>
<th>Records messages emitted by models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Status trace</td>
<td>Status Trace Monitor</td>
<td>Record state changes of models</td>
</tr>
<tr>
<td>Model Member variables</td>
<td>Value Trace Monitor</td>
<td>Record value changes of a signal or a variable over time</td>
</tr>
<tr>
<td>Interfaces between port/export pairs in systems</td>
<td>IMC Trace Monitor</td>
<td>Records calls and returns made between ports and their channels/exports and optionally also the arguments passed between them</td>
</tr>
<tr>
<td>Processes and threads</td>
<td>Process Trace Monitor</td>
<td>Records SystemC process activations and event notifications</td>
</tr>
<tr>
<td>SCV instrumentation (streams and generators)</td>
<td>SCV Trace Monitor</td>
<td>Records SCV transactions</td>
</tr>
<tr>
<td></td>
<td>SCV Statistics Monitor</td>
<td>Records interval based statistics for SCV transactions</td>
</tr>
</tbody>
</table>

### 4.1.3 Generic Data Analysis

Next to the powerful framework of Instrumentation Points and Monitors there is also a simplified API that allows to use a simple instrumentation API to create custom analyses for an IP component and get access to the flexible GUI charting capabilities of Virtualizer.

The generic data analysis uses a simplified instrumentation to record data and provides access to the views with which the recorded data is presented. The basic concepts of the instrumentation are metrics and variables.

### Metrics

The metric is the fundamental measurement you are recording. You can instrument to record your own custom metrics from your models. The metrics will be traced and statistics will be recorded for them as well. The data type of a metric can be either integer, double, or an enumerated type. In addition to the distinction of these three data types, metrics can also be divided into two different kinds with respect to what their values represent. The analysis engine can be directed to treat metrics as either state values or as event values. State values are measurements that represent the state of something that persists over time; for example, the values of a device’s registers, or the rate at which power is being consumed by a mobile phone. Event values, are measurements of events that happen at independent, unrelated points in time; for example, the number of characters sent in a series of text messages, or the time taken by memory accesses. This distinction is important, since it affects the kind of statistics that are gathered; for example, an average value over time or an average value from a set of N events.
Variables
When analyzing data, it is usually necessary to look beyond the values of a metric in isolation, and to see how the values related to other things going on in the system. In generic data analysis, extra variables or conditions can be incorporated into the analysis of the recorded data. Analogous to metrics themselves, these extra variables can also be of two kinds: state variables that hold their value over time, and event variables that simply tag metrics’ values with extra attributes.

The Analysis instrumentation infrastructure provides with API’s to add these generic data analysis through the IPT framework into the visualization tools. This provides with a generic framework to trace custom data from a design and create trace and statistical views. A detailed API description has been made available to the project partners and has been made part of the product documentation.

4.2 Analysis results visualization

All data that is recorded into a database by the Monitors discussed in the previous section can be visualized via the Synopsys analysis viewer tools. The visualization is dependent on the data format as stored in the database. The database contains a set of tables, and each type of table can be visualized by one or more views or charts.

Table 3 Available visualization charts

<table>
<thead>
<tr>
<th>Chart</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Trace Chart</td>
<td>A simple trace of value changes for signals or variables</td>
</tr>
<tr>
<td>Transaction Trace Chart</td>
<td>A condensed view of transaction traffic on TLM ports or SCV streams</td>
</tr>
<tr>
<td>Transaction Statistics Chart</td>
<td>Charts of transaction statistics over time</td>
</tr>
<tr>
<td>Register Trace Chart</td>
<td>A graphical view of the register activity in your design.</td>
</tr>
<tr>
<td>Path Statistics Chart</td>
<td>A statistical view of traffic across the paths of a bus</td>
</tr>
<tr>
<td>Resource Statistics Chart</td>
<td>A statistical view of resources within a bus</td>
</tr>
<tr>
<td>Message Sequence Chart</td>
<td>A vertical view of events or transactions passing between objects over time</td>
</tr>
<tr>
<td>SCV Trace Chart</td>
<td>A view of SCV transactions on SCV generators</td>
</tr>
<tr>
<td>Diagnostic Trace Chart</td>
<td>Shows the output of a single stream object</td>
</tr>
<tr>
<td>Status Trace Chart</td>
<td>Displays status traces</td>
</tr>
</tbody>
</table>
The following paragraphs give an overview of the different visualization charts.

**Value Trace Chart**

This view is a traditional trace view with a line for each signal or variable that is traced, using a line chart or a string representation for the value in the trace. The string representation is useful for SW signals and values; the line representation is used for binary values but can also be used to show a line graph of e.g. the power consumption of a component over time, or to visualize buffer size over time as in the graph below.

**Transaction Trace Chart**

This chart gives a trace of the TLM transactions over a port, in various level of detail; providing with an overview showing only transaction start and end or a refined view with additional detail that shows the different phases of the transaction over the port.

**Transaction Statistics Chart**

This is a very much configurable view that shows various statistics about the system level communication and timing. For example the chart below shows the average communication latency statistics for a certain design.
Register Trace chart

The register trace chart is a view that shows the register activity in a design, which is most interesting for software developers. The view shows the values of the registers in a design and how their value changes, it also indicates when a value change results into a behaviour invocation in the model.

Path Statistics Chart

This chart gives a ‘heat-map’ view of a certain parameter. Statistics are collected for a certain interval size, e.g. the number of times certain behaviour has been activated. The graph represents these values in a grid view where each source of values is a line and the x-axis is divided in a set of ranges where each range is the accumulation of a set of interval measurements. (Alternatively this could also be the maximum, average or minimum of the interval measurements). The grid is collared according to a heat-map where each cell in the grid gets a graded colouring according to the value of the cell. This view allows to find ‘hot-spots’ very easily while still enabling to drill down into details in order to find, e.g. the culprit of a certain power spike. An example of such a graph below:
4.3 Integration

The analysis infrastructure of the Synopsys Virtual Platform solution is intended to be a very generic infrastructure in order to allow different use cases and problems to be solved. It is used to provide platform analysis and debugging metrics for interconnect architecture exploration as well as SW debugging and SW analysis capabilities. In the COMPLEX flow this infrastructure is to be integrated with the power and timing metrics modelling and serve as a basis for design space exploration.

In order to facilitate the integration of the analysis environment several interfaces are available:

1. The analysis infrastructure allows to record information for any model interface and internal variable. This allows creating models that use a regular SystemC port or member variable to log power and timing values.

2. The analysis infrastructure is integrated with the SystemC standard SCV tracing interfaces. These allow to create ‘generators’ and ‘streams’ of transactional information in a design and to record it into the Synopsys analysis database. An example use for this is in the SW task modelling infrastructure (D2.2.2) where the SCV tracing API’s are used to create a trace view and statistics of the task states and the scheduler. This allows visualizing statistical processing load etc.

3. The analysis visualization tools come with a scripting interface that allows to manipulate the database tables to create derived views and to export them to standard formats e.g. comma separated value lists.
5 Definition of interchange format for automatic design space exploration

When there is a discussion about “system simulation and profiling” it is not possible not to take into consideration the additional goals that the simulation and profiling tool should be designed for. Among the additional goals there is the design space exploration. In this section we present the defined interchange format that a simulation/profiling tool should respect for supporting the Automatic Design Space Exploration pointing also to some specific

The interchange format for Automatic Design Space Exploration has been defined as already declared in deliverable D1.2.1 - Definition of application, stimuli and platform specification, and definition of tool interfaces by following the MULTICUBE-XML interface (in the following called only DSE-XML interface). This interchange format has been specified previously in the MULTICUBE project and used in this project as basis for the development of the MOST framework. The idea behind this interface is to have a unique interface for enabling the integration of system models coming from different use case providers for design space exploration purposes.

![Diagram](image)

The goal of the DSE-XML interface is to addresses the interaction between the simulator and the design space exploration tools, which is essentially an automatic program-to-program interaction. In general, the interaction can be described as following:

1) The design space exploration tool generates one feasible system configuration whose system metrics should be estimated by the simulator.

2) The simulator generates a set of system metrics to be passed back to the design space exploration tool.

The specification of the formats of the input/output data to/from the simulator is defined as the explorer/simulator interface.

In order to link the use case and the simulator to the design space exploration tool, a design space definition file should be released by the use case and simulator provider together with the executable model of the use case (simulator). This file describes the set of configurable parameters of the simulator, their value range and the set of evaluation metrics that can be estimated by the simulator. This file describes also how to invoke the simulator as well as an optional set of rules with which the generated parameter values should be compliant. The rules are only used by the exploration tool in order to do not generate invalid or unfeasible solutions during the automated exploration process.
The DSE-XML specification provides an XML based grammar for writing both the design space definition file and the simulator interface files.

5.1 Design Space Definition

In this section, we describe the format of the XML design space definition file produced by the use case and simulator provider (see the following figure). For a more clear understanding, the format of the design space definition file is introduced by examples.

The overall structure of the design space definition data file is composed of a preamble, which defines the namespace and a version.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<design_space xmlns="http://www.multicube.eu/" version="1.4">
  <simulator> ... </simulator>  
  <parameters> ... </parameters>
  <system_metrics> ... </system_metrics>
  <rules> ... </rules>
</design_space>
```

The remaining part of the file describes the simulator invocation method (<simulator> ... </simulator>), the set of parameters of the simulator which can be configured (<parameters> ... </parameters>), the system metrics which can be estimated by the simulator (<system_metrics> ... </system_metrics>) and the rules which have to be taken into account by the exploration tool in order to generate feasible configurations.

Simulator invocation

The simulator invocation is performed by shell command execution. The <simulator_executable> marker is used for specifying the complete path name of the executable:

```xml
  <simulator_executable path="/path/my_simulator_executable" />
</simulator>
```

The path is specified by using Unix conventions. The simulator executable is invoked with three arguments:
my_simulator_executable \  
   --xml_system_configuration=sc_path_name \  
   --xml_system_metrics=sm_path_name \  
   --reference_xsd=xsd_file_name

where sc_path_name is the path name of XML file describing the system configuration (whose format is specified in the explorer/simulator interface XML specification). The sm_path_name is the path name of the output XML file which should be used by the simulator for producing the system metrics output. The latter XML file is, again, compliant with the explorer/simulator interface XML specification. The argument --reference_xsd=xsd_file_name is used for specifying the position of the reference explorer/simulator interface XSD file in the file system. This argument can be used by the simulator for validating the input and output files exchanged with the explorer tool.

Parameters specifications

The <parameters> ... </parameters> is used by the use case and simulator provider to specify the names, the types and the ranges of the parameters that can be explored by the DSE tool. The section contains a list of <parameter> markers:

```xml
<parameters>
  <parameter name="seed"  
    description="RNG seed"  
    type="integer" min="0" max="10" />
  <parameter name="fetch_queue_size"  
    description="instruction fetch queue size"  
    type="integer" min="1" max="8" step="2" />
...
</parameters>
```

For each parameter a unique name must be provided. This name will be used for generating configurations at the input of the simulator. The parameters types can be divided into two categories: Scalar types, Variable vector types (not used within the use cases of the project and so not detailed in this document).

The scalar parameter types are defined in the following table which shows the also the corresponding min, max, and step attributes. The attributes are used for specifying the progression associated to the type.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Min attribute</th>
<th>Max attribute</th>
<th>Step attribute</th>
<th>Items list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>Mandatory</td>
<td>Mandatory</td>
<td>Optional (default=1)</td>
<td>NA</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>exp2</td>
<td>Power of two</td>
<td>Mandatory</td>
<td>Mandatory</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>string</td>
<td>String</td>
<td>NA</td>
<td>NA</td>
<td>Mandatory</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Integer and boolean progression.** The integer type specifies a simple sequential integer progression associated that specific parameter. The min and max attributes (which are mandatory) specify the boundaries of the progression. The step attribute can be used to produce non-unitary progressions. The Boolean type is an integer progression with min=0 and max=1.
Exponential progression. The values associated with an "exp2" parameter type should be computed by the design space exploration tool by using a power of two progression. For example:

```xml
<parameter name="il1_cache_block_size_bytes"
    description="..." type="exp2" min="8" max="64" />
```

should be interpreted by the exploration tool as a parameter with range values { "8", "16", "32", "64" }.

String progression. In the case of string parameters, a list of possible string values should be used instead of the min/max attributes:

```xml
<parameter name="bpred" description="branch predictor type" type="string">
    <item value="nottaken"/>
    <item value="taken"/>
    <item value="perfect"/>
    <item value="bimod"/>
    <item value="2lev"/>
    <item value="comb"/>
</parameter>
```

Multiple instances of scalar parameters. In the case the design space is composed of a subset of scalar parameters each one with the same type and range, the design space definition can be represented in a more compact way by adding the attribute instances, instead of declaring one line for each scalar parameter. If the attribute instances is not declared it is considered equal to 1. As an example, in the case of a set of 6 integer parameters with a 0 to 10 range, we can write:

```xml
<parameter name="seed_vector" description="RNG seed vector"
    type="integer" min="0" max="10" instances="6" />
```

System metrics specification

The `<system_metrics>` section is used by the use case and simulator provider to specify the names, the types and the units of the system metrics that can be estimated by the simulator:

```xml
<system_metrics>
    <system_metric name="cycles" type="integer" unit="cycles"/>
    <system_metric name="instructions" type="integer"
        unit="insts" description="..." />
    <system_metric name="power_consumption" type="float"
        unit="W" description="..." />
    <system_metric name="area" type="float" unit="mm2"/>
</system_metrics>
```

The optional “description” attribute is a generic string describing the nature of the system metric. It is important to note that a strict requirement for the DSE phase is that the metrics involved in the exploration should be represented by scalar values. As an example, traces are not supported. To support time variable values such traces in the DSE phase, aggregate values should be used, such as average, maximum, minimum or standard deviation.

Multiple set of metrics. Especially when the exploration should be done regarding a set of applications, or input stimuli, there is a need of describing in a more compact way the replication of the system metrics for each instance. To do this, has been introduced within the
<system_metrics> section the attribute “repeat” that describes how many times the content of the <system_metrics> section should be unrolled to take into consideration the different evaluations for each application/stimuli.

```xml
<system_metrics repeat="4">
  <system_metric name="cycles" type="integer" unit="cycles"/>
  <system_metric name="power" type="float" unit="W"/>
</system_metrics>
```

**Feasibility rules**

The <rules> section is used by the exploration tool in order to not generate invalid or not feasible solutions during the automated exploration process. The behaviour of the simulator when these rules are not met is undefined. Each rule is a Boolean expression which should evaluate to true.

Each Boolean expression can be an operator acting on either a <parameter> or <constant> leafs or other Boolean expressions. This allows creating complex expression trees of rules. Rules are ‘AND’ed by default by the exploration tool. Each rule is identified by a <rule> marker and it has an optional ‘name’ attribute.

```xml
<rules>
  <rule name="l2bs_ge_l1bs">
    <greater-equal>
      <parameter name="l2_cache_block_size"/>
      <parameter name="l1_dcache_block_size"/>
    </greater-equal>
  </rule>
</rules>
```

The following operators/markers can be used: <greater>, <greater-equal>, <less>, <less-equal>, <equal>, <not-equal>, <expr>. The <expr> marker can be used for introducing generic expressions such as: { + - * / }. Moreover, for combining complex expressions the following markers/operators can be used: <and>, <or>, <not>. 
5.2 Explorer Simulator interface

The specification of the formats of the input file for the simulator and the output file from the simulator (see the following figure) is defined as the explorer/simulator interface. Also in this case, the format of the explorer/simulator interface is introduced by examples.

Simulator input file

The simulator input file should contain a preamble and a sequence of <parameter> sections where, for each parameter, the name and the value is specified:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<simulator_input_interface xmlns="http://www.multicube.eu/" version="1.3">
  <parameter name="seed" value="1" />
  <parameter name="fetch_queue_size" value="1" />
  ...
</simulator_input_interface>
```

The number of <parameter> sections and the name of the parameters is the same as defined in the XML Design Space description file, if no multiple instances of scalar have been defined. The multiple instances of scalar parameters defined in the XML Design Space description file will be expanded into a set of scalar parameter equal to the value of the instances attribute. To discriminate among the different instance of the scalar parameter a suffix '_#' containing the instance number will be added at the parameter name. For example, the values of the multiple instances of the scalar ‘seed_vector’ previously defined, is the following:

```xml
<parameter name="seed_vector_1" value="1" />
<parameter name="seed_vector_2" value="5" />
<parameter name="seed_vector_3" value="6" />
<parameter name="seed_vector_4" value="3" />
<parameter name="seed_vector_5" value="9" />
<parameter name="seed_vector_6" value="5" />
```
Simulator output file

The simulator output file contains a preamble and a sequence of <system_metric> sections where, for each metric, the name and the value is specified:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<simulator_output_interface xmlns="http://www.multicube.eu/" version="1.3">
  <system_metric name="cycles" value="3000" />
  <system_metric name="instructions" value="1500" />
  <system_metric name="power_consumption" value="2.5" />
  <system_metric name="area" value="25" />
</simulator_output_interface>
```

The number of <system_metric> sections and the name of the system metrics should be the same as defined in the XML Design Space description file.

In presence of the <repeat> attribute within the XML Design Space description file the name of the system metrics should be augmented with a suffix ‘_#’, containing the instance number. E.g:

```xml
<system_metric name="cycles_0" value="3000" />
<system_metric name="power_0" value="2.5" />
<system_metric name="cycles_1" value="1547" />
<system_metric name="power_1" value="4.7" />
```

In the case of errors during the simulator execution, the simulator output file should contain a single <error> marker indicating the error reason:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<simulator_output_interface xmlns="http://www.multicube.eu/" version="1.3">
  <error reason="memory-full" kind="fatal"/>
</simulator_output_interface>
```

The attribute reason is a generic string that can contain a report about the error cause. The kind can be "fatal"/"non-fatal". Fatal errors should block the overall exploration process while non-fatal errors should force the explorer to skip to the next configuration. Overall, the error strings of the simulator are meant to be related to memory-full or disk-full problems, file system permissions problems, license problems, and internal exceptions and other. If an <error> marker is present in the output file, <system_metric> markers are ignored by the exploration tool.

Dependencies with the design space definition file. The parameter names used in the simulator input file should correspond to the parameters declared in the design space definition file. The system metrics used in the simulator output file should correspond to the system metrics defined in the design space definition file.
5.3 Additional DSE Integration Interfaces

The XML interface definition for the design space exploration phase has been agreed to be used as basis for all the exploration done within the project. However to facilitate the integration of the large set of simulators/use cases/design tools some addition/simplifications has been done on top of it.

Access to SCOPE+ internal metrics

To improve the planned integration with the SCOPE+ tool, we gave the possibility to select within the <system_metrics> section of the XML Design Space Definition files also some SCOPE+ internal metrics. In fact, previously SCOPE+ had the possibility to export as system metrics only a sub-set of metrics that were globally evaluated, e.g. System power and application latency, and not all the metrics related to specific HW or SW components, e.g. bus usage or execution time of specific functions. To give the possibility to export as system metric to the design space tool also some “internal metrics”, we extended the design space files letting the possibility to add a <scope_metric_path> section within a <system_metric> section, specifying components details.

An example, the access to the value of the maximum execution time of the function “audioControl_RX” within the SW component “audioController” and its export within the system metric “Exec_time_Maximum_audioController_audioControl_RX” can be described as follows:

```xml
<system_metric name="Exec_time_Maximum_audioController_audioControl_RX" type="float" unit="ms">
  <scope_metric_path component="audioController/audioControl_RX" internal_metric="exec_time/@maximum"/>
</system_metric>
```

MOST generic wrapper

To improve the integration of the large set of simulators/use cases/design tools with MOST design space tool, we built what we called “MOST generic wrapper” (MGW). The MGW is a python script that is able to hide to the designer that has in charge the design space integration all the details related to the “Explorer Simulator Interface” reducing its integration effort only to the definition of the design space and to the generation of a simple script (e.g. bash) able to launch the execution of the simulator/use case model/design tool.

The details related to the MOST generic wrapper can be found in the Deliverable D3.4.2 – Intermediate Report on Design Space Exploration.
6 Techniques for System-Level Simulation and Profiling with SCoPE+

As stated in the Description of Work (DoW) one of the fundamental steps in the optimization of the DSE, is the “identification of proper system simulation and application profiling in order to identify bottlenecks, and to provide statistics and metrics to be used throughout the system-level exploration activities”. This is specifically the goal of task T3.1, where the University of Cantabria develops SCoPE+, a framework enabling early simulation and estimation of performance for a wide set of HW/SW implementation solutions.

This way, SCoPE+ enables a fast assessment of the best implementation solution for an embedded system at early stages of the design flow. For it, SCoPE+ incorporates novel techniques for enabling:

- A significant save on recoding effort, since, the performance estimation can take into account different HW and SW allocations for the functional (application) components without requiring any refinement of the functional (platform independent) code.

- Fast and accurate simulations, since SCoPE+ relies on native simulation, incorporating the SCoPE system-level performance estimation techniques, for automatic annotation of the code, consideration of cache impact, modelling of the RTOS, etc.

- System-level simulation techniques, for taking into account the impact of communication penalties, which depend on the architectural mapping and on the communication resources available.

- Interfaces between SCoPE+ (simulation framework) and MOST (exploration tool) which enable the automation of the exploration.

6.1 Requirements of the simulation technology for DSE

The time spent in design exploration can be roughly approximated through the following expression:

$$T_{exp} \approx N_{simulations} \left( t_{simulation} + t_{preparation} + t_{sim-expl.\ interface} \right)$$

A good exploration tool will be able to reduce the number of simulations ($N_{simulations}$) by exploiting the knowledge acquired during the exploration about the impact of the design parameters on the goal functions and the output metrics. This enables a more efficient search. This is subject of the T3.4 related reports, where MOST exploration tool is reported.

Moreover, there are still three main factors which can heavily impact on the exploration time and which can be strongly optimized from the simulation engine side: the simulation time required for each solution explored, the time required for the preparation of the new solution; and the time consumed by the communication of the simulation framework and the exploration tool.

Current design exploration frameworks find limitations in all these aspects. In order to get sufficient accuracy, common simulation and estimation solutions are based on the execution of cross-compiled binary code. For its execution on the host system, techniques such as
Instruction Set Simulators (ISS) and binary translators are used. These solutions require codes completely refined for the target platform. The application SW must be recoded using the target operating system API. Explicit execution flows (threads and processes) are required in all the resources to execute all the mapped components. Additionally, SW/SW, HW/SW and HW/HW communications must be explicitly implemented in the refined code. Specifically, communications between components mapped to HW and SW require SW drivers to perform HW/SW communications.

Therefore, relying on these simulation techniques requires a huge effort and time cost for enabling a sufficiently exhaustive exploration process. Moreover, in an ESL design flow, much of that cost can be reduced if simulation engines provided a sufficiently generic front-end to postpone code refinement effort after architectural mapping, when refinement cost is more productive. In turn, architectural mapping is decided after DSE.

Therefore, what is required is a simulation technology able to provide fast simulation and sufficiently accurate performance figures from and input description of the system where the functionality is sourced as platform independent code, thus ready and suitable for a later implementation process; cleanly separated from platform description and architectural mapping information; and which does not require any refinement of functional description for each architectural mapping explored.

Summarizing, several main goals have to be taken into consideration for the reduction of the exploration cost, thus to enable an optimum simulation technology for DSE:

- The reduction (ideally, elimination) of the recoding effort required when exploring several architectural mappings
- Faster (ideally, infinitely fast) simulation technologies
- well-defined (ideally standard) interfaces between the simulation framework and the exploration tool
- Sufficiently accurate (ideally exact) as a fundamant for the design decisions

### 6.2 From SCoPE to SCoPE+

In a previous work, the University of Cantabria made a step forward towards an optimum simulation framework for DSE by developing SCoPE [13]. SCoPE combines easy creation of platform models with fast techniques for SW execution modeling. SCoPE provides a simulation technique alternative to instruction set simulators (ISS) obtaining speed ups in factors about x100, while still achieving performance estimation errors of about 10%. Moreover, simulation speed-ups are also got with regard to virtualization techniques [14]. To do so, SCoPE applies the latest techniques on annotated native co-simulation with high-level TLM HW platform models.

Despite SCoPE works on source code, but it still works on partially refined SW codes. In effect, C/C++ functionality provided to SCoPE relied on a RTOS API, therefore, some effort for adapting the original functional code to an specific operating system API is required. Additionally, HW/SW communication still depends on drivers.
In COMPLEX, SCoPE is being extended in order to overcome this limitation. The resulting framework, named in short SCoPE+ has been provided with a new front end, named Concurrent and Functional Application Component Model (CFACM) front-end.

The CFACM front end provides an API which enables a text-based description of a platform independent model (PIM) of the system, which contains system functionality partitioned into components. Despite it is possible, the main purpose of the CFACM front-end is not to aim a direct usage by the user. The CFAM front end has been designed to facilitate the capture of the system as a UML/MARTE model complemented with C/C++ pure code. Specifically, the PIM information is contained by C/C++ functionality, plus the component-based application architecture, which can be graphically depicted in the standard UML/MARTE notation.

An innovative aspect of SCoPE+ is about system-simulation, that is, how each architectural solution, that is, each platform specific model (PSM) is simulated.

In the execution of each different system implementation, the component structure of the functional application is preserved. However, the platform architecture and the architectural mapping might change (e.g., by considering a different number of cores in the MPSoC a cluster, or by moving the implementation of some functionality from software to hardware). This will mean that the performance loads associated to the functionality computations and communications present in the PIM, will change in general, from solution to solution, and thus the execution and communication flows can vary from solution to solution.

SCoPE+ provides a solution for performance system-level modelling since it enables an automatically estimation and annotation of the performance impact of each functionality and communication of the initial and unmodified PIM, which will vary in general for each architectural mapping. These load variations can modify somehow the execution flow, so SCoPE+ let take into account a possible impact of performance on behaviour.

Figure 9: CFAMC front-end for SCoPE+.

Then, the generators developed in task T2.1 enable a direct and automatic translation to the text-based representation which can be directly read by SCoPE+.
obtains the information from the CFAMC model and the XML-based information, in turn extracted from the UML/MARTE model extracted through the generators reported in D2.1.2.

The architectural mapping information, present in the XML files, serves for knowing where each functionality is mapped and thus for estimating the load of computation, through the techniques reported in D2.2.2 and D2.4.2 for SW and custom hardware performance estimation. The same architectural mapping information is used to assign a cost to the system-level communications. Analysis and profiling metrics under consideration (stated in D1.1.1 and D1.2) are also extracted from the UML/MARTE model, and drive the performance optimizations for the system components in terms of the cost function defined in D1.2.3. The analysis information is provided in the form of final metrics that are obtained at the end of the simulation. The abstraction level for the metrics is fixed at task granularity, modelling the HW platform with approximate times.

An interchange format has been defined to pass analysis information from the virtual platform system to the exploration framework. The metrics obtained are reported in XML files, since it is one of the most effective languages for communicating different tools.

The development of the complete SCoPE+ tool has been performed separately in several tasks in the COMPLEX project. In D2.2.2, the solutions for high-level SW performance modelling of the functional components allocated as application SW is reported. In D2.4.2, a novel technique for evaluating custom HW implementations of functional components has been reported. Finally, in T2.5 the interfaces required to receive all the information from the UML/MARTE models have been implemented and reported in D2.5.2, except for the XML interface between SCoPE+ and the exploration tool, which is being reported in D3.4.2.

None of the aforementioned tasks for building SCoPE+ would be useful without a new simulation engine. In order to perform complete system simulations and evaluations, a simulation engine capable of integrating all these solutions together with other facilities capable of modelling the rest of the elements of the system and all the resulting interactions and side effects is required. Then, the estimations obtained can be adequately reported to be used by the rest of the tools involved in the corresponding COMPLEX design flow.

### 6.3 SCoPE+ Features and Requirements on its simulation engine

In task 3.1, the SCoPE+ simulation engine is being developed. It is being done by improving the previous SCoPE infrastructure and, moreover, by integrating and coordinating the new features developed in COMPLEX. It requires an integration effort in terms of front-ends and interfaces with the UML/MARTE modelling infrastructure (reported in D2.5.2), and with the exploration front-end (reported in D3.4.2) which can be even directly handled and/or monitored by the user or by a front-end (such as the COMPLEX Eclipse Application reported in D2.1.2). Moreover, the simulation engine of SCoPE+ has also to perform a coordination of the simulation libraries (i.e., software and hardware) developed in T2.2. and T2.4, in order to makes feasible system simulation and the output performance figures consistent.

The result is a simulation engine able to model complete systems, described in UML/MARTE; supporting a component-based description of the application, which integrates pure C/C++ functionality; where application components can be allocated either to a SW implementation on top of a MPSoC platform, or to custom HW; and which seamlessly communicates with an exploration tool, such as MOST. Furthermore, the SCoPE+ simulation engine models the side effects provoked by the combination of the different elements, such as
delays and power consumption of communications between components mapped to different resources.

The new features of the SCoPE+ simulation engine can be then enumerated in terms of the new SCoPE+ features reported in D2.2.2:

- Simulation of Component-based Platform Independent functionality
- Modeling of SW/SW communications
- System-level simulation of multi-OS execution
- System-level simulation of HW/SW communication
- Component Traversing Flows

Most of these features are in charge of supporting most of the new features of SCoPE for SW simulation and performance integration, reported in D2.2.2, for the integration of the custom HW estimation technology, reported in D2.4.2, and of some of the new APIs reported in D2.5.2. Following subsections are devoted to provide technical insights about the new features of the simulation engine.

### 6.3.1 Simulation of the Component-based Platform-Independent Functionality

In order to enable the simulation of the CFACM code an adaptation layer has been developed. This layer translates the CFAM function calls into the calls of one of the modelling APIs already supported by SCoPE, the SCoPE POSIX API. The SCoPE POSIX API is a POSIX subset which, as well as one of the RTOS API supported by SCoPE, serves also as a reference API to access the OS model integrated in SCoPE.

The CFAMC-to-POSIX translation infrastructure has been encapsulated in a single plugin, the CFACM plugin. This has the advantage of providing a clean and separable extension to the SCoPE kernel. Indeed, during the development in COMPLEX, several versions of the CFACM plugin are being developed, which enabled a clean and decoupled development. Extensions and fixes proper of the CFAMCM front-end did not require necessarily new versions of the SCoPE kernel.

The SCoPE POSIX API was sufficient for developing the adaptation layer, so no additional POSIX functions have been developed. However, the implementation of the layer requires in certain cases more information from the simulation engine than the one that can be obtained by only using the POSIX functions. As a consequence, some additional hook functions have been developed in the simulation engine to enable the access of the layer to private information contained in the classes of the simulation engine.

### 6.3.2 Modeling of SW/SW communications

As reported in D2.2.2, section 3.2.2, In order to correctly simulate all the effects resulting from the SW mapping of the functional components, it is required to simulate both the resource allocations of the components and the communication performed through the component interfaces. The simulation of these features have been solved through the support of the CFACM API (Concurrent Functional Application Component Model), reported in D2.5.2.
As was mentioned in D2.2.2, information is read from the CFACM model, in turn extracted from the UML/MARTE model.

The implementation of this support has imposed several requirements to the simulation engine. For instance, specific hook functions have been provided in order to dynamically stating from the CFAM layer which processor is executing a specific execution flow. Moreover, functions for modelling the intra-component communication effects resulting from the use of the interfaces have been also provided.

### 6.3.3 System-Level simulation of multi-OS execution

As was reported in Section 3.2.6 of D2.2.2, SCoPE+ enables system-level modelling of multi-OS execution. That is, it enables that the Platform Independent Model access the OS services in an implementation agnostic way, that is independently from which RTOS instance is providing the service to a functional component, but providing performance estimations which take into account the mapping of functional components to different RTOS instances.

The SCoPE+ solution is solved at two levels. Global synchronization is solved at the simulation engine level. The SCoPE+ simulation engine has been enhanced to enable a new capability for the SW simulation, namely the simulation of global operating systems. Using this feature, all communication and synchronization resources can be shared among all the different RTOS instances modelled (in other words, it can be understood as if the they worked at a hypervisor level).

On top of that, the implementation of the layer in charge of translating CFAM calls to POSIX calls also considers the current operating system where each SW component is being simulated. Then, the CFAM layer handles the side effects produced in performance and scheduling by the use of common OS services from different operating system instances. These effects specifically account for communication delays and power consumption of communication devices resulting from the use of CFAM services.

### 6.3.4 System-Level simulation of HW/SW communication

A similar problem than the one presented when modelling multiple operating systems appears when components using CFAM services are mapped to HW. In that case, it is required to communicate and synchronize HW components with the SW OSs.

The solution is similar than the one applied for the simulation of multi-OS execution. A two-level solution is provided. The infrastructure developed to model a global operating system in the SCoPE+ tool has also been extended to control the execution of HW resources such as ASICs and FPGAs. Following the SW-centric approach of the Complex project, a new kind of OS modeling solution has been implemented to control the operation of functional components mapped to these HW resources. This solution controls the execution, synchronization and communication of HW components using CFAM services.

### 6.3.5 Multi-level simulation

In D2.2.2, multi-level simulation was reported as the ability for modeling and simulating platform independent code together with SW code. The support of this feature, reported in D2.2.2, is intimately linked with the previous points. In effect, the support of this feature requires the possibility to simulate a CFAM component (which can potentially be allocated
either to SW or to HW) with a SW code (e.g. POSIX code). Therefore, multi-level simulation requires enabling the complete integration of all the new methods presented in the previous lines with the corresponding ones of the original SCoPE tool.

The connection of PIM components and SW code is possible thanks to the architecture selected for SCoPE+, where the CFACM layer relies through the SCoPE POSIX API on top of a global operating system (explained later).

Since not only features related to SW simulation have to be provided for full support of multi-level simulation, this feature is being still developed in T3.1.

6.3.6 Component Traversing Flows

As mentioned in D2.2.2, in the COMPLEX specification methodology, an execution flow of the system functionality can traverse several application component instances, which in turn can be mapped to different resources (or not).

When one functional component (client) accesses some functionality in another component (server) calling a function from its interface, in a platform-independent code no additional execution flows are explicitly created (by default). As in any other function call, the execution flow performing the call is the one that executes the function. The flow stops executing the calling code, enters the called function, executes the function, returns and continues the execution of the calling functionality.

If client and server are mapped to different resources, the client execution flow must automatically change from one resource to another to execute the server functionality. Thus, execution flows must change the associated resource dynamically. Considering that both resources can be in a single symmetric multiprocessor node, in different nodes of a network, or one in HW and the other in SW, a complex simulation engine is required to simulate all the possible resource changes.

Information from the UML/MARTE models has been used to intercept the function calls inserting new wrappers, as described in D2.5.2. These wrappers are in charge of performing all the operations required to automatically modify the execution of the platform independent codes to model the resource allocations. This process does not modify the platform-independent code, which is executed as it is. As a result, by using SCoPE+, it is possible to perform a functional debug and verification of the system functionality, at the same time that impact assessment is done.
6.4 New features implemented in the simulation engine

Considering all the requirements listed above, the work done in task 3.1 is based on the evolution of the simulation kernel of the original SCoPE tool for system-level simulation. The main extensions performed in the simulation infrastructure are focused in the following features:

- Global Operating System model
- OS controller for ASICs and FPGA resources
- Support for dynamic assignment of resource allocation to tasks
- Modeling of performance effects resulting from communications with void data
- Support for interfaces with the TLM models and the DSE tools

6.4.1 Global Operating System modeling

The Operating System model included in the original SCoPE tool is a POSIX-compliant SMP system. The OS model controls the operation of processes and threads, handling priorities, policies and interrupts. Time annotations and preemptions are also modelled.

However, the model presents a limitation for its use under the COMPLEX specifications. As said above, the code that can be modelled in the original SCoPE tool must be composed of POSIX-based, application-SW components. In other words, the code to be simulated must be refined SW, not platform-independent code.

The reason for these limitations relies on the architecture of the modelling infrastructure, which reflects the architecture of common real SW infrastructures. The SMP OS model execution relies on three main layers, implemented as C++ classes: one containing the elements of the SMP OS model that are shared by all the processes and processors, one layer containing the local OS elements related with each processor, and the last one, which includes the SystemC infrastructure in charge of modelling the processor execution. In the simulation, an instance of the OS class is required for each operating system included in the platform to be modelled. Additionally, as many instances of the classes that model the second and third layer are needed as processors are in platform controlled by the OS.

With this structure each task is assigned to an OS in the original SCoPE tool. As a consequence, a task can execute on any of the processors controlled by the OS, but not in other resources. Thus, it is not possible to generate execution flows traversing different resources, depending on the allocation of the functional components. This previous implementation does not enable modelling the huge interaction resulting from the simulation of system-level, platform-independent components. Direct function calls between components mapped in HW and SW or calls between components of different OSs are not possible. Drivers and explicit communication APIs must be used instead.

The new SCoPE+ tool enables these direct communications by implemented a Global Operating System infrastructure. The Global Operating System adds a new level to the three levels described above. In this new implementation there is a higher level composed by the global operating system, a second layer integrating the specific elements required to describe each instance of real OS in the platform, a third modelling layer with the OS elements related to each processor, and a fourth layer with the SystemC resources used to model the processor.
The new global layer is instantiated once per simulation. It integrates the elements required to model each OS in the original SCoPE tool in a shared way. The global OS controls all the executable resources of the HW platform where a platform-independent component can be allocated, including ASICs and FPGAs. At the same time, all tasks are managed by the same global operating system. Thus, the infrastructure enables an execution flow to traverse different executable resources.

The global operating system, also integrates all the OS resources, such as semaphores, mutexes, streams or message queues. All tasks can access all the resources, independently on the resource were they are mapped. Thus, it enables easy implementation of the CFAM API layer developed in T2.2.

To correctly model the selected resource allocations, the global operating system must limit the resources where a task can execute. Each task can execute in one or a set of resources depending of the component that is executing on each moment. Thus, the CFACM macros developed in T2.5 inform the simulation engine about the points where a flow enters and exits new component, in order to change the HW resources where it can execute. Resource masks are used to indicate the processors and HW resources where the task can be executed on each moment.

Finally, it is important to note that all the elements have been modified to enable considering HW implementations (ASICs and FPGAs) as resources controlled by the global operating system, since the original SCoPE tool only supports processors. Details on the integration of HW implementations are described in the next subsection.

**Status**: Integration done.

### 6.4.2 HW integration in the Global Operating System

In order to enable direct calls to functions provided by components mapped to HW or exploring HW and SW allocations of functional components, the simulation infrastructure hides the allocation details to the platform-independent code. Thus, for modelling the execution of platform-independent components, the infrastructure provided by SCoPE+ is symmetric when modelling HW and SW allocations.

Solutions for estimating the cycles and power consumption required to execute sections of platform-independent code in SW and HW implementations have been developed in T2.2 and T2.4. However, these solutions estimate the performance of the components, separately and independently of the rest of the system. In T3.1, the goal is to develop an infrastructure where this information is combined with the functional execution of the components in order to model the entire system. Thus, combined and side effects resulting from the execution of entire systems can be modelled.

In order to model entire systems, a solution for modelling HW allocation in a compatible way with the modelling of SW allocations has been developed. Considering that the original SCoPE tool and the COMPLEX project are both oriented to SW centric approaches, the selected implementation for SCoPE+ is also SW centric. Thus, SCoPE+ shows a SW-like infrastructure for both HW and SW resource allocations of functional components. All executable resources are modelled by a global operating system in the same way, independently on their real nature.
To model ASIC and FPGA devices valid for HW allocations, alternative implementations for the third and fourth layers of the modelling infrastructure described in the previous subsection have been developed. The third layer is in charge of modelling the scheduling and interrupts management. The fourth layer models the time advance and the communication with the rest of the system through the bus.

Considering both layers, the main difference between the modelling of processors and HW components used for HW allocations are the following:

- Processors can execute only one task at a time, while ASICs and FPGAs can perform multiple tasks at a time.
- ASICs and FPGAs can execute one or a limited number of calls of the same task at a time.
- Processor models can perform only one external call or I/O operation at a time, while ASIC and FPGA models can perform one per active task.

As a consequence, new layers based on the use of dynamic SystemC threads have been implemented, enabling the modelling of HW allocations in the global design.

**Status:** prototype done.

### 6.4.2.1 Integration of HW components for multiple service calls

The integration of the components modeled with the HW modeling technique developed in WP2 has required additional effort for use case integration. It has required modifying different parts of the modeling infrastructure. The main problem is that the HW modeling solution uses global variables and functions that makes the modeling to be non reentrant. As a result it is not possible to run different functions mapped in HW concurrently.

In order to solve that limitation, the entire code of a HW mapped component (containing functional code and communication wrapper) is integrated in a protected dynamic library. That way, all its elements are saved and separated when multiple calls are performed. However, this solution only protects calling functions from different components, but not calling in parallel functions of the same component.

To ensure correct operation of the HW components, multiple loadings of the protected libraries are performed. When the simulation is built, one copy of the component dynamic library is loaded, connecting the communication wrappers with the rest of the components of the system. During simulation, every time a service of the component is called, this first library is initially accessed. Then, this library creates a clone of itself, redirecting the call to the new copy of the library. When the call finishes, the cloned library returns the results to the original library and the cloned library is destroyed.

All the elements of the simulation infrastructure, specially the macros used to automatically generate the wrappers, have been modified in order to support this entire infrastructure.
6.4.3 Support for dynamic assignment of resource allocation to tasks

The next feature integrated is the ability to simulate infrastructures where the resource associated to each execution flow (thread or process) can change dynamically. This required feature does not rely in an attempt of simulating systems where the real flows change the resource at run-time. The reason for this dynamic solution, as explained above, is based on the fact that initial platform-independent codes and final implementations can be different.

Following the SW paradigm applied in the project, functional platform-independent components are usually built as a sequence of function calls potentially integrated in different components. Thus, the execution of a thread starts calling a function of one component, which can call a function of other component, which, again, can call another component, and so on. Explicit calls to synchronization and communication channels are not required. As a consequence, when one functional component (client) accesses some functionality in another component (server) using a system call the simulation does not create any additional execution flow. In function calls, the execution flow performing the call is the one that executes the function. The flow stops executing the calling function, enters the called function, executes the function, returns and continues the execution of the calling function.

The result is that the number of threads and processes of the implementation can be different from the number of execution flows used in the platform-independent code. Thus, several execution flows of the real implementation can be modeled with a single flow. If these real flows are allocated to different resources, the simulation must move the resource where the modeling thread is executing, in order to obtain a correct simulation.

The basic idea of SCoPE+ is execute directly the platform-independent code to model the resulting system, without performing code modifications. Thus, the modeling thread which executes the platform-independent codes can enter different components, and potentially different resources, depending on the component allocation decided.

Following the example considered above, if client and server are mapped to different resources, the client execution flow in the simulation must automatically change from one resource to another to execute the server functionality. Thus, execution flows must change the associated resource dynamically. Considering that both resources can be in a single symmetric multiprocessor node, in different nodes of a network, or one in HW and the other in SW, a complex simulation engine is required to simulate all the possible resource changes.

To support such capabilities, the basic idea applied is to use the global operating system infrastructure described above, that covers the entire system. Different functions have been implemented in the simulation engine for modeling this change on the HW resource that executes the code.

Information from the UML/MARTE models has been used to intercept the function calls inserting new wrappers. These wrappers are in charge of performing all the operations required to automatically modify the execution of the platform independent codes to model the resource allocations, calling the new functions provided by the simulation engine. This process does not modify the platform-independent code, which is executed as it is. As a result, it is possible to verify and debug the functional components using SCoPE+.

As stated before, the functional model used in this work considers that services are accessed by clients through function calls. Thus, the execution flow of each task changes from one component to another, and usually from one HW resource to another, depending on the
allocation of each component. The solution proposed to model this change of HW resource is to consider a global operating system and use the processors' mask to fix where the tasks is executing at each moment. However, the last problem we found is how to transform the original function calls in the functional model to do so, in an automatic manner where no additional recoding effort from the designer is required.

The basic idea to do that is to generate a Container/Component infrastructure for all system components. Here, the Component contains the part of the functional model related with the system component. The Container is a wrapper in charge of detecting inputs and outputs from the component.

Component and Container both provide the same interfaces, with the same function prototypes. Each time a Component requests a service from another Component, the call is redirected to the Container of the second component instead of the Component itself.

Thus, the problem is limited to how to create automatically the Container for each component. For doing so, information in high level models such as UML/MARTE is required. In these models, information about the components in the system, the interfaces, the functions on each interface and from where they are called is present. From this information, the Container and the modification in the function calls can be created through specific generators already developed in T2.1.

To generate the Container, the generator creates a code that acts the wrapper, enabling the monitoring of the functions provided by its interface. The code generated is based on the CFACM macros described in D2.5.2.

For example, in the following code, a container implementing an interface with three functions is shown:

```c
CREATE_CONTAINER(ProcessingComponent)
  : INHERIT_INTERFACES(public ImageProcessor_PI)
{
    COMPONENT(ProcessingComponent);

    SPORADIC_FUNCTION(startOperation, m_ImageProcessor);
    SPORADIC_FUNCTION(stopOperation, m_ImageProcessor);
    SPORADIC_FUNCTION(getState, m_ImageProcessor, state);
}
```

Internally, each function is implemented by a Macro as:

```c
#define SPORADIC_FUNCTION(function, arg_type) \ 
    void function(arg_type arg1) {\ 
    int node = getCurrentProc();\ 
    if(node!=m_node)\ 
      changeCurrentProc(m_node);\ 
      modelCommunication(node, m_node, size);\ 
      component->function(arg1);\ 
      modelCommunication(m_node, node, size);\ 
      changeCurrentProc(node);\ 
    } else{\ 
      container->function(arg1);\ 
    } 
```

In that example, the functions `getCurrentProc()` and `changeCurrentProc()` are used to change the resource where the code have to execute. Additionally, the function `modelCommunication(...)` is used to model communication performance, as explained in the next section.

For more details of the integration of SW and HW components in the extended simulation engine, and the wrappers generated, please refer to D2.5.2: “Intermediate report and tools on virtual system generation”.

**Status:** First prototype done. Deep verification needed.

### 6.4.4 Modeling of performance effects resulting from intra-component communications

Function calls performed between different components require sending information from client to service at function calls to be computed, and returning the results when the call ends. These data transfers result in delays, and provoke power consumption in the communication components. Additionally, the use of the communication resource is increased, and delays in other communications due to collisions can be increased.

To model that, the wrappers used to integrate the functional components in the SCoPE+ simulations (T2.5) also require services from the simulation engine to model communications. Information about the size of the data to be transferred on each communication is extracted combining information obtained from the code and the interfaces described in UML/MARTE.

Additionally, in order to model the transfers, it is required to know the path the transferred data have to follow. Two options for path extraction have been implemented. The user can describe the paths in UML/MARTE, information which is later used in SCoPE+ to model transfer impacts. If no paths are manually described, SCoPE+ has been extended to extract automatically the paths from the UML/MARTE platform description.

The infrastructure developed has been integrated in the COMPLEX design flow, including UML/MARTE and Eclipse. The modeling capabilities of these elements have been used to improve the capabilities the designer has to drive the evaluation process, as is shown in following sections. Nevertheless, their use is not mandatory for the application of the simulation engine itself, and they can be avoided if the proposed solution is integrated in another infrastructure.

#### 6.4.4.1 Obtaining communication paths between functional components

To perform accurate modeling of the system communications, the first step is to know the paths the information follows on each transfer. Usually, SW systems are built creating several layers. One of the lower SW layers is the communication layer. Each communication in the functional model is not implemented in a completely different way. Communications usually take advantage of the services of this layer, to reduce the design effort.

As a consequence, similar communications make use of the same service in the communication layer, resulting in a similar communication type. For example, it is common to find that in a system, transfers sent from all processes in processor “a” to processes in processor “b” use the same lower layer service, being performed in the same way.
Taking advantage of that, it can be considered that it is not required to extract the communication path for any communication made in the system, but only extract the paths required to support communication among all the HW resources of the system: processor and application-specific HW. This simplification substantially reduces the computational effort required to model communications during system simulation. As a result, simulation overhead implied by the communication model is minimal, something which is critical if we want to explore large design spaces, with thousands of possible points.

To minimize the simulation time, the proposed approach is to obtain the resource-to-resource paths for each simulation just before it starts. Then, the transfer can be modeled only indicating the initiator and target.

To obtain these paths, two solutions have been implemented in T3.1. The first one extracts the paths automatically from the platform description done in UML/MARTE. Additionally, a solution has been proposed to enable the designer to fix some of these paths manually, instead of letting the tool to select one. To do that, an additional UML/MARTE diagram and an additional XML file format have been defined to describe the communication paths.

The algorithm used to extract the communication paths between different processing resources is the following:

1. The user defined paths described in the corresponding XML file is read, if any.
2. If there is a path defined to connect a resource “A” with a resource “B”, but not in the opposite direction, it is analyzed if the same path can be used in both directions.
3. If no paths are defined to communicate two resources, the automatic path mechanism is used.
4. If no path is found to connect two elements, an error is reported when this path is requested during simulation.

**Automatic path extraction process**

To extract the communication path between two processing resources, a branch and bound like algorithm has been implemented. Information about the HW platform is obtained from the XML file obtained from the UML/MARTE model, as described in D2.1.2. In this file all HW instances are listed, including their interconnections. Interconnection information also includes master/slave information. Summarizing, the path extractor knows who is connected to each HW component and in which direction data can be sent.

The algorithm looks for the faster path capable of communicating initiator and target. Additionally it is defined an intermediate component where information can be exchanged (a memory or a network).

To extract the path, two search trees are created containing all the elements with an already known connection path with the initiator or the target. Each node of the tree represents a HW component and it is linked in the tree to the components connected to it, as described in the XML file.

The procedure starts identifying initiator and target as leaf in their respective trees. Then it is checked if there are common leafs in both trees. If not, the HW components connected to the leaf components in the XML file are added as new leafs, and the checking is performed again. When a hit is detected, a complete path has been found. The cost of the path is set as maximum cost and only faster solution are searched. To do that, all the branches which
current cost is larger than this maximum are pruned. When all branches have been bounded, the minimal path is selected.

The radial structure of the algorithm helps finding a fast solution earlier, since usually these solutions has a smaller amount of components in the path.

**User defined paths**

In certain cases, the user can prefer to provide a path instead of letting the tool to automatically take the decision. When multiple possible paths can be found, it can be interesting to the user to have a way to provide it, in order to ensure that the simulation will follow designer decisions.

To do that, it has been enabled the possibility of using a UML/MARTE sequence, diagram to describe the communication path in Eclipse. Then, a plug-in generates a XML file indicating the paths selected. The resulting XML file has the following format:

```xml
<interconnections>
  <connection origin="processor1" target="processor10" link="memory1">
    <component name="comp1"/>
    <component name="comp2"/>
    ...
    <component name="memory1"/>
    ...
    <component name="last_comp"/>
    <connection/>
    ...
  </connection>
<interconnections>
```

**6.4.4.2 Modeling Intra-component Communications**

In order to create a model of the system capable of evaluating the impact of communications, the elements provided by SCoPE to create the platform models have been minimally modified. The original SCoPE tool provides models of buses, memories, DMA, bridges, networks and network interfaces, which are developed on top of a base class that handles the TLM2 protocol proposed by the OSCI.

First, TLM2 protocol is based on the use of structs for transferring the communication information. These structs include information about the type of communication (read/write), address, package size, data buffer, etc. Furthermore, the struct includes a void pointer for user-defined additional information.

To model only communications creating an infrastructure capable of working in parallel with the functional model, a new package type has been defined. This package contains all the members of the struct, but the data buffer, with is null.

The inherited class has two operation modes, for communication channels (buses/networks) and for HW components. In the modification performed, the part inherited by the HW components has been extended to detect the new package type. When the package is detected, the HW component wait the reception time, indicated by the component parameters, and automatically returns, indicating that transfer has been completed. No functionality in the HW component is executed.
Second, system communication paths are supposed as a combination of two simple communication structures:

- HW component / Channel / HW Component, and
- HW component / HW component.

Thus, the internal modelling of a communication transfer extract all the simple communication structures in the path, and inject a package of the new type in all the initiators of the simple communications. Then, the simulation infrastructure gets the corresponding delay in all the structures and applies it to the functional code modelling. As a consequence, total communication delay is estimated. Additionally, as channels are used as in normal communications, collisions and other side effects are considered.

### 6.4.4.3 Link Between Functional and Communication Models

As stated before, the Container is a wrapper in charge of detecting inputs and outputs from the component, and thus, in charge of modeling resource allocation and intra-component communications (Figure 10).

Component and Container both provide the same interfaces, with the same function prototypes. Each time a Component requests a service from another Component, the call is redirected to the Container of the second component instead of the Component itself. Then, the Container calls the communication model indicating the initiator, target and size of information to transfer. Information size depends on the arguments of the function. `malloc` and `calloc` functions have been wrapped to save the size of each buffer created, to enable getting correct transfer sizes.

After that, the same function in its Component is called to execute its functionality. When the function returns, the Container calls again the communication channel, to model how the service returns the obtained results, and continues the execution in the first Component. The modification of the function call can be done by changing the pointer to the interface called or by using `#define` clauses, avoiding manual recoding.
**Status:** Integration done. Real evaluations needed to optimize the evaluation of the impact of communications in real implementations.

### 6.4.5 Support for interfaces with the TLM models and the DSE tools

Three different files are obtained from the simulation: a file with system metrics, a file with task metrics and a file with metrics of HW components.

The exploration of completely configurable HW/SW systems, including the possibility of deciding HW and SW allocation of functional components, requires more configurable parameters and output metrics to be integrated in the simulator.

Additional configuration parameters have been integrated in order to support the parameters supported by the UML/MARTE specification. Additionally, the output metrics have been extended to provide more information to the designer. The original SCoPE tool only reported system metrics, considering the system as a black box. The new tool also reports information about system internals. Information of the performance of the internal HW components is reported. Additionally, information about the performance of the SW components in the different allocations is reported. Latencies, throughputs, delays and iterations are metrics reported to help the designer to optimize the system. Furthermore, these metrics can be used to check the system constraints during the exploration process.

**Status:** Integration done.

### 6.4.6 Support for different communication semantics

As previously described in section 8.4.3 “Support for dynamic assignment of resource allocation to tasks”, the SPORADIC_FUNCTION macro was initially developed to solve the management of different allocations of client and server components. However, this macro only implements an initial, non configurable kind of communication between the components.

The definition of the UML/MARTE specification methodology done in T2.1 proposes a wider range of communication types. Depending on a list of parameters that are associated to each communication interface, different semantics can be described. As a result, the extension of the communication mechanisms is needed. In order to support that extension, the number and characteristics of the communication solutions implemented as macros in the simulation engine has been extended.

**Semantics taxonomy**

The implementation of these communication semantics has first required of an analysis of the different possible combination of attributes in order to simplify their support. This analysis has revealed several dependencies in the possible attributes that was not detecting during the definition of the UML modeling methodology.

The attributes that can be applied to the communication interfaces, and, thus, to the provided services, can be divided in three groups: communication semantics, deadlines and priorities.

While semantic information identify how the client and server execution flows operates, the deadline information enables the user to set deadlines to the services that are checked at runtime and the priority allows to fix the policy and priority of the requested services. These three groups are orthogonal among them, so no dependencies have been found there.
However, several dependencies have risen within the semantic properties. To solve that, the semantics indicated by each possible value have been clarified considering implementation possibilities, instead of starting from an abstract point of view, as done in T2.1.

These properties are the following:

- **occKind** - identify the occurrence of service. The possible values are:
  - **Periodic** invocations involve that no explicit object invoking the function/method is required.
  - **Sporadic** invocation means that whatever the object invoking functionality, it can do it at any moment, without a constraint in the delay among consecutive invocations.

- **exeKind** - indicates how the service must be executed with respect to the request. It results from the combination of two characteristics:
  - **Immediate** invocation means that the service cannot start in blocked mode, that is cannot have entry conditions which provoke the blocking of the process.
  - **Deferred** invocation means that the service can have entry conditions.
  - **Local** invocation means that the service is executed locally
  - **Remote** invocation means that the service is executed remotely

- sinchKind – indicates the synchronization type between client and server. It can be:
  - **Synchronous** invocations mean that the caller object waits for the called object to compute the service and return (usually implemented as a function call).
  - **Asynchronous** invocations mean that the caller object immediately returns, and so that the service can be computed by the called object at some given moment after the invocation (usually implemented through a dynamic thread creation at the called object to attend the service). In this sense, it is told that the call is deferred, so used as a synonymous.

- **concPolicy** – indicate the relationship between client and server:
  - **Guarded** invocation means that the service execution must be protected from multiple simultaneous executions. The simulator must ensure the function is guarded.
  - **Sequential** invocation means that several calls to the service client execution can never be executed in parallel and are executed sequentially.
  - **Concurrent** invocation means that the service execution must be protected from multiple simultaneous executions. The user must ensure the code support that

Once analyzed all the possibilities, it has been identified that Immediate/Deferred semantics are not orthogonal to the previous semantics. There are incompatibilities and dependencies with protected/unprotected semantics.

At the same time, both sequential and concurrent **occKind** attributes let a concurrent access, which, in contrast to guarded semantics, let services to be called concurrently. The difference consists in that the sequential service declares that its concurrent execution might cause side effects once there is no protection mechanism. However, a concurrent service declares that the function is re-entrant and that its concurrent call is side effect free. Since this distinction depends on the functionality (C/C++) code, linked to the CFAMCM code produced from the model to the effects of CFAMCM code generation, this attributes are equivalent.
In COMPLEX, a synchronous/asynchronous semantics depends on the type of component. \(<\text{PpUnit}>\) components will always provide synchronous services. In contrast, \(<\text{RtUnit}>\) can provide either synchronous or asynchronous services.

In MARTE, it is possible to make explicit this semantics by using \textit{synchkind} attribute of the \(<\text{RtService}>\) stereotype. Related to the COMPLEX methodology, an alternative would be to stereotype the operations in the functional view. This information could be also used at the verification view, to check the coherence with the messages of the sequence diagrams used for documenting environment-system interaction. However, in COMPLEX, the use of the \(<\text{RtService}>\) stereotype within the functional view has been avoided (to have only functional information in such a view).

In COMPLEX, the \(<\text{RtService}>\) stereotype can be used at the C&C view related to the ports enclosing exported services. However, the use of the \textit{synchkind} attribute is still optional. This means that it is required to define a default semantics or rule for inferring if a service provided by the RtUnit is either synchronous or asynchronous. The proposal of this document (v1.2) is to take the synchronous semantics as the default one. This enables back compatibility with previous versions of SCoPE+ or simple simulators no supporting asynchronous semantics, which will support all COMPLEX models (except those which explicitly use the “synchKind” attribute). It is also “safer” in the sense of covering all possible combinations of parameter types. Notice that if a service has an “out” or an “inout” parameter, and the model defines the service as synchronous, there is a semantic incompatibility.

As a consequence of that, several incompatibilities appear:

- \textit{PpUnits} cannot provide asynchronous services (no new threads or execution flows are created).
- Cyclic services are not called by any component, they are autonomously triggered. Therefore, Synchronous/Asynchronous semantics are not applicable.
- Asynchronous services should not have “inout” “out” parameters (since no client is going to be waiting for them). “delayedSynchronous” semantics stated in MARTE specification makes no sense from our point of view (it defines an asynchronous synchronization, which is a contradiction by itself) (A checking of this is recommended).
- Since in the methodology, the “synchKind” attribute is optional, a default semantics has to be stated when this attribute is not present (in this document, the synchronous semantics as default semantics is proposed as default semantics).
- “exeKind” semantics is used for RtUnits to the same effects of “concPolicy” in PpUnits.
- Regarding “exekind” semantics, remoteImmediate and localImmediate semantics, that is, immediate semantics, they cannot be protected (once blocked, the service cannot longer be immediately executed). Deferred semantics has not restrictions.
- remoteImmediate semantics is incompatible with asynchronous semantics (since the service is executed by the caller).
Summarizing, the resulting possible combinations are the following:

<table>
<thead>
<tr>
<th>UML/MARTE (COMPLEX Methodology)</th>
<th>Executive semantics of the service/function/method</th>
<th>When is executed: (SynchKind)</th>
<th>Protection for concurrent accesses: (concPolicy for PpUnit / exeKind for RtUnit)</th>
<th>Regularity of the invocation (occKind)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML Operation (Service):</td>
<td>Provokes the Generation of a Concurrent Thread</td>
<td>Synchronous/ asynchronous</td>
<td>Protected/ unprotected</td>
<td>Periodic/ Sporadic</td>
</tr>
</tbody>
</table>

- occKind=periodic (<<RtSpecification>>) 
  - exeKind= deferred in <<RtUnit>> component
  - Yes, at owner comp. 
  - N/A
  - Protected
  - Periodic

- occKind=periodic (<<RtSpecification>>) 
  - exeKind= immediate in <<RtUnit>> component
  - Yes, at owner comp. 
  - N/A
  - Unprotected
  - Periodic

- concPolicy = guarded in <<PpUnit>> component
  - No
  - Synchronous
  - Protected
  - Sporadic

- concPolicy = sequential|concurrent in <<PpUnit>> component
  - No
  - Synchronous
  - Unprotected
  - Sporadic

- occKind =sporadic 
  - exeKind= deferred 
  - synchKind=synchronous (default) (<<RtSpecification>>) 
  - in <<RtUnit>> component
  - No
  - Synchronous
  - Protected
  - Sporadic

- occKind =sporadic 
  - exeKind= immediateRemote 
  - synchKind=synchronous (default) (<<RtSpecification>>) 
  - in <<RtUnit>> component
  - No
  - Synchronous
  - Unprotected
  - Sporadic

- occKind =sporadic 
  - exeKind= deferred 
  - synchKind=asynchronous (<<RtSpecification>>) 
  - in <<RtUnit>> component
  - Yes, potentially blocked (only once call can be executed at the same time)
  - Asynchronous
  - Protected
  - Sporadic

- occKind =sporadic 
  - exeKind= immediateLocal 
  - synchKind=asynchronous (<<RtSpecification>>) 
  - in <<RtUnit>> component
  - Yes, at called Component (server)
  - Asynchronous
  - Unprotected
  - Sporadic

**Implementation of communication semantics**

Once identified the types of communications that have to be implemented to support the different possibilities supported by the UML/MARTE methodology, different macros have been developed to automatically implement their characteristics.

First, the name of all macros supports the possibility to add a prefix and a suffix. The prefix indicates if a deadline is provided, and the suffix if a priority has been defined for the function.
Additionally, different macros model the execution of the different communication semantics, following the table above:

<table>
<thead>
<tr>
<th>UML/MARTE (COMPLEX Methodology)</th>
<th>CFAMCM Macros Optional Prefix/Suffix</th>
<th>Additional semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>absDi (&lt;&lt;RtSpecification&gt;&gt;)</td>
<td>[PREF ]=INSTUMENTED_</td>
<td>Deadline</td>
</tr>
<tr>
<td>Priority (&lt;&lt;RtSpecification&gt;&gt;)</td>
<td>[ _SUFX]=PRIORITY</td>
<td>Priority</td>
</tr>
</tbody>
</table>

Additionally, a set of functions have been implemented in the simulation engine to provide the facilities required to perform the communication characteristics desired for each macro. As a result, the simulation environment can support all the communication properties described in the UML/MARTE modeling methodology.
6.4.7 Additional support for HW platform modeling

The component models that enable creating the virtual platform for the simulation have been extended to support the UML modeling methodology. These modifications focus on three components: the generation of new IO and communication components and the modification of networks’ integration.

New IO devices have been developed to enable adequate modeling of the communication between the system and the SystemC environments created with the new COMPLEX UML methodology. Since this kind of environments run directly on SystemC SCoPE.

At the same time, a new communication component has been created in order to support the integration of the code generated by the Eclipse generators from the UML model. As SCoPE runs on top of SystemC, SystemC rules must be followed in virtual platform simulation. In SystemC virtual platforms, all communication master and slave ports have to be connected before the simulation starts. If any port is not connected, SystemC simulation kernel reports and error and ends the simulation. In order to support that, SCoPE ports requires specific indications about master and slave connections.

However, the system platform generated from the UML code, have reported some difficulties in accomplishing these requirements. In order to solve that, a generic communication component has been developed. This component has a variable number of ports, automatically generating as master and slave ports as required. This procedure is not common on SystemC simulations, but, as a result, there is risk on leading master or slave ports unconnected. This capability simplifies the work accomplished by the UML generators.

Finally, the network support has been modified in order to solve a limitation of the previous SCoPE versions. In previous versions, the network was considered as an additional component for certain use cases, but complete support has not been provided. The main limitation of this previous support is that only one network can be integrated in the system to be simulated. This feature has been extended, enabling the management of several networks at the same time. This solution enables correctly modeling COMPLEX Use Case 3, where 2 networks connect nodes 1, 2 and 3.

6.4.8 Support for new SystemC version 2.3.0

The release the 26 of July of 2006 of the new version 2.3.0 of the SystemC open source proof-of-concept library has required some checking and several modifications in the SCoPE+ simulator, in order to be compatible with both the old 2.2.0 version and the new version.

The new version has been developed to be fully compatible with the newly revised IEEE 1666 “Standard SystemC Language Reference Manual”. The library also provides a number of important new features, including an integrated support for transaction-level modeling (TLM). New extensive process control features enable modeling of concepts such as power domains and abstract schedulers. A new first-class events list and a new container for SystemC objects simplify the description of generic and scalable models. In addition, an improved simulation API and a new thread safety mechanism allows better interaction with external tools and support for multi-threading.

Additionally, the library, installation notes and readme files have been updated to support installation on the latest operating systems and compilers.
As a result of these modifications in the SystemC library, two issues have demonstrated incompatibilities with the previous SCoPE+ version. The first modification relies on the integration of a new version of the TLM library with certain differences with respect to the previous version. This modification has requested the adaptation of the SCoPE components that models the HW platform elements to build the virtual prototype. The use of ports and exports for TLM bus channels are a bit different, and have been updated.

The second modification comes from the adaptation of SystemC library to operating systems and compilers. While in previous versions the SystemC library worked on top of the “quick threads” library in order to generate the execution flows required to perform the simulation, the new version has removed its use, replacing it by the direct use of the operating system library, as “pthread” library in the Linux environments where SCoPE+ works.

The problem appears because SCoPE+ has a deep dependency on the “quick threads” library. To solve the problem the library has been integrated into SCoPE slightly modifying how it is used.

6.4.9 Automatic testing support

In order to verify the simulation infrastructure developed, some effort has been spent on automatic testing. This effort has been divided in two parts, extending the simulator to support automatic testing, and generating several tests to check different parts of the simulation engine.

These tests have been performed by adding two new functions to the simulation engine:

```c
get_simulation_return_value();
set_simulation_return_value(int);
```

These functions enable modifying the value that will be returned by the executable at the end of each simulation. Applying that, each test is developed by identifying the number of checking points in the test that have to be correctly reached during the execution. Then, the number of points is initially assigned as return value, and each time a checking point is correctly reached, the number is decreased by 1. As a result, at the end of simulation the return value must be 0 if all checking points have been correctly reached and 0 otherwise.

This return value enables automatically verifying the tests, both by creating a shell script with that specific purpose or by concatenating the execution of the tests by “&” operations. That way, if one test fails, the remaining tests are not executed and an error is reported.

6.4.10 Increment of simulation speed

Considering the large amount of simulations that have to be performed to complete the exploration of the COMPLEX use case, the speed of the simulation engine is a critical aspect to be handled. Experiments done with the use case have reported that the implementation done did not initially achieve the expectations in terms of simulation speed.

In order to solve that, the implementation of the different communication wrappers and the functions developed in the simulation infrastructure to support them has been refined. Minimizing function calls and grouping OS services to avoid multiple calls to the OS
scheduler of the simulation engine have been required in order to speed up the simulation speed.

However, this task is not considered finished and will be continued in T1.3, until the end of the project, in order to optimize the simulation of Use Case 3.

6.4.11 Modifications related to Use Case 3

A last group of modifications have been performed in the simulation engine specifically in order to support the use case of the COMPLEX project.

One of the main modifications is the accuracy of the HW timer provided by SCoPE. This timer is used during the simulation to model all OS services related to time, such as sleeps, timeouts, and SW timers. The simulation engine, supplies a timer with 10 ms accuracy, following the common period of traditional Linux implementations. However, most of the activities of the use case rely on smaller times. Several task periods of the cyclic tasks of the application are in the order of ms. As a result, the entire SCoPE timing handing engine have been modified in order to support a configurable period.

In order to take advantage of this modification, the macros automatically called by the UML generators to start the simulation are in charge of fixing this configurable value to 1 ms. Since all required times are larger than 1 ms, the use of smaller times only result in lower simulation times without providing an accuracy increment.

The second important element modified to support the use case is the tuning of the HW platform parameters in charge of accurately modeling times and power consumption. Since this task is completely dependent on the use case, it has been performing at the end of T3.1 and additional effort is still required. This activity will be continued in T1.3 until the end of the project.
7 SystemC Network Simulation Library (SCNSL)

This section describes the concepts and the solved issues related to the simulation of packet-based networks in SystemC. This investigation leaded to the implementation of the SystemC Network Simulation Library (SCNSL).

![Figure 11 SCNSL internal architecture](image)

Figure 11 shows the main components of SCNSL. Tasks are used to model the devices under development. Thus, tasks shall be implemented by designers either at RTL or TLM level. From the point of view of a network simulator, a task is just a producer or consumer of packets and therefore its implementation is not relevant to the simulation kernel. However, for the system designer, task implementation is crucial and many operations are connected to its modelling, i.e., change of abstraction level, validation, fault injection, HW/SW partitioning, mapping to an available platform, synthesis, and so forth. For this reason, the class TaskProxy has been introduced to decouple task implementation from the backend which simulates the network. Each Task instance is connected to one or more TaskProxy instances and, from the perspective of the network simulation kernel, the TaskProxy instance is the interface of the task. Vice versa, from the point of view of the application, each TaskProxy provides the primitives for network communication. This solution allows keeping a stable interface between the TaskProxy and the simulation kernel and, at the same time, to let complete freedom in the modelling choices for the task, e.g., interconnections of basic blocks or finite-state machines. Two different TaskProxy interfaces are provided depending on the abstraction level of the connected Task (i.e., RTL or TLM). It is worth noting that other SystemC libraries can also be used in task implementation, e.g., reused IP blocks and other libraries such as the well-known SystemC Verification Library. These libraries may simplify designer's work even if they are outside the scope of network simulation. Tasks are hosted on Nodes, which are the abstraction of physical devices. Tasks deployed on different nodes shall communicate by using network communication, while tasks deployed on the same node shall communicate by using standard SystemC communication primitives.

The Channel class represents the network simulation kernel interface. A channel is the abstraction of the transmission medium, and thus it shall simulate all communication aspects including packet collisions. Standard SystemC channels are generally used to model interconnections between HW components and, therefore, they can be used to model network
at physical level. However, many general purpose network simulators reproduce transmissions at packet level to speed up simulations. SCNSL follows this approach, providing models for wired (full-duplex, half-duplex and unidirectional) and wireless channels.

Communication protocols are a key concept for a network simulation. Users could require implementing protocols ex-novo, in case protocols are under design, or they could rely on protocol models provided by the simulator. To allow maximum flexibility, SCNSL supports both these possibilities. In particular, users can implement protocols inside tasks. On the other hand, standard protocol models are provided in optional backend components, namely Communicators. A communicator is a SCNSL component implementing the Communicator interface. New capabilities and behaviour can be easily added by extending this class. Communicators can be interconnected each other to create chains. Each valid chain shall have on one end a TaskProxy instance and, on the other end, a Node; hence transmitted packets will move from the source TaskProxy to the Node, traversing zero or more intermediate communicators, then they shall be transmitted by using a channel model, and finally they will eventually traverse the communicators placed between the destination node and the destination TaskProxy. In this way, it is possible to modify the simulation behaviour by just creating a new communicator and placing its instance between TaskProxies and nodes. Communicators can be used to implement not only protocols, but also buffers to store packets, simulation tracing facilities, etc. Another critical point in the design of the tool has been the concept of packet. Generally, packet format depends on the corresponding protocol even if some features are always present, e.g., the length and source/destination addresses. System design requires a bit-accurate description of packet contents to test parsing functionality while from the point of view of the network simulator the strictly required fields are the length for bit-rate computation and some flags to mark collisions (if routing is performed by the simulator, source/destination addresses are used too). Furthermore, the smaller the number of different packet formats, the more efficient is the simulator implementation. To meet these opposite requirements in SCNSL, an internal packet format is used by the simulator while the system designer can use other different packet formats according to protocol design. The conversion between the user packet format and the internal packet format is performed in the TaskProxy.

7.1 Implementation of network simulation in SystemC

Some issues related to network simulation have been solved during the development of SCNSL:

- Subdivision of functions between Tasks and Nodes.
- Transmission validity: a packet could be invalid for the receiver, because of collisions and out-of-range transmissions.
- RTL simulation: RTL models usually manipulate bit-oriented data, while the simulation is performed at packet-level.
- Packets: SCNSL requires a standard packet to perform correctly the transmission simulation, but a packet is different from a RTL signal or a TLM payload.
• TLM & RTL: the co-existence of RTL and TLM models during the same simulation requires the creation of a sort of transactor.

• Topology: a network scenario is characterized by the concept of topology, i.e. the location of nodes in the physical space. Moreover the scenario could change dynamically during the simulation.

• Communication protocols and channels: System module communication is standardized, since RTL modules shall communicate by using ports and signals, and TLM modules will communicate by using a standard payload and standard communication interfaces and algorithms. Network communication is more complex, since each scenario can use different protocols and channels.

The remaining of this Section addresses these issues and describes the solutions adopted in SCNSL.

7.1.1 Tasks and nodes

From the point of view of network simulation, the main entities are tasks, which are the producers and consumers of packets, and nodes, which create the network topology. Since tasks represent the functionality of the application, their implementation shall be performed by SCNSL users. Tasks can represent both HW and SW components, and thus, it is possible to use both RTL and TLM abstraction levels. Anyway, SCNSL implements some standard tasks, which are common in network simulators. For instance, a Constant Bit Rate task is useful to create some interfering network traffic during a simulation, to test the application in case of packet lost. In SCNSL, each task shall implement a standard interface, RtlTask or TlmTask, according to the task abstraction level. Nodes are the abstraction of physical devices. Thus, they contain physical related properties, like the transmission bit-rate of the network interface, spatial position, etc. Each node can contain one or more tasks, and can be bounded to one or more physical channels. Since SCNSL assumes that the components under design are implemented as tasks, in SCNSL a general implementation of a node is provided by the backend, in the Node class.

7.1.2 Transmission validity assessment

In wireless and half-duplex scenarios an important feature of the network simulator kernel is the assessment of transmission validity which could be compromised by collisions and out-of-range distances. The validity check has been implemented by using two flags and a counter. The first flag is associated to each node pair and it is used to check the validity of the transmission as far as the distance is concerned; if the sender or the receiver of an on-going transmission has been moved outside the maximum transmission range, this flag is set to false. The second flag and the counter are associated to each node and they are used to check the validity with respect to collisions. The counter is used to register the numbers of active transmitters which are interfering at a given receiver; if the value of this counter is greater than one, then on-going transmission to the given receiver are not valid since they are compromised by collisions. However, even if, at a given time, the counter holds one, the transmission could be invalid due to previous collisions; the flag has the purpose to track this case. When a packet transmission is completed, if the value of the counter is greater than one, the flag is set to false. The combined use of the flag and the counter allows covering all transmission cases in which packet validity is compromised by collisions.
7.1.3 Simulation of RTL models

In case of RTL models, the co-existence of RTL events with network events has to be addressed. RTL events model the setup of logical values on ports and signals and they have an instantaneous propagation, i.e., they are triggered at the same simulation time in which the corresponding values are changing. Furthermore, except for tri-state logic, ports and signals have always a value associated to them. Instead, network events are mainly related to the transmission of packets; each transmission is not instantaneous because of transmission delay, during idle periods the channel is empty, and the repeated transmission of the same packet is possible and leads to distinct network events. In SCNSL, RTL task models handle packet-level events by using three ports signalling the start and the end of each packet transmission, and the reception of a new packet, respectively. To each task instance are associated one or more TaskProxies. A TaskProxy is an object able to translate network events into RTL events and vice versa. In particular, each RTL task has to write on a specific port when it starts the transmission of a packet while another port is written by the corresponding TaskProxy when the transmission of the packet is completed. A third port is used to notify the task about the arrival of a new packet. With this approach each transmission/reception of a packet is detected even if packets are equal. The last issue regards the handling of packets of different size. Real world protocols use packets of different sizes while RTL ports must have a constant width set at compile-time. SCNSL solves this problem by creating packet ports with the maximum packet size allowed in the network scenario and by using an integer port to communicate the actual packet size. In this way a TaskProxy or a receiver task can read only the actual used bytes, thus obtaining a correct simulation.

7.1.4 The notion of packet

One key object inside a network simulator is the representation of a packet. While RTL types are not enough flexible to implement a standard packet, the TLM payload could be a valid choice; however TLM communication primitives cannot handle network-related issues such as collisions and out-of-range transmissions. Therefore, in SCNSL the Packet class is an abstraction of an actual packet, and in fact its main fields are a buffer of bytes, storing the actual transmitted data, and an integer describing the size of the buffer. This allows using the Packet class to model also variable-length packets or packets afferents to different protocols. Such a Packet is required to standardize the data exchanged between the SCNSL components, and thus, increasing their modularity and reusability.

7.1.5 RTL & TLM models coexistence

The SystemC allows users to design modules at different abstraction levels, i.e. at RTL and TLM. These modules can be mixed inside a simulation instance by using special modules to interconnect them. Such modules are called transactors and their role is to translate values read from RTL ports and signals into TLM payloads, and vice versa. This idea has been applied also to the design of SCNSL, in order to allow the design of tasks different abstraction levels. This issue has been addressed in two steps:

1. Translation of RTL and TLM exchanged data into an abstract and independent internal format.
2. Standardization of network communication mechanisms, in order to avoid implementing a new transactor for each module. Only two transactors are required:
between RTL and the SCNSL internal format, and between TLM and SCNSL internal format.

Step 1 has been resolved by using SCNSL internal Packet class, as described in Section 4.4. Step 2 has led to the creation of network communication transactors, which have been named TaskProxies. Each TaskProxy can be then considered as the class which separates the user space, from the network simulation backend implemented by SCNSL. Moreover, the combined use of TaskProxies and internal packets has the advantage that SCNSL internal types shall be unrelated to SystemC types. Thus, SCNSL internal classes have been implemented mixing SystemC and C++ code to optimize simulation performances.

### 7.1.6 Simulation planning

In several network simulators, special events can be scheduled during the setup of the scenario, for instance node movements, link status changes, and traffic activation. This feature is important because it allows simulating the model into a dynamic network context. In SCNSL the simulation kernel has not its own event dispatcher, hence this feature has been implemented into an optional class, called EventsQueue. Even if SystemC allows to write in each task the code triggering such events, the choice of managing them in a specific class of the simulator leads to the following advantages:

- **Standard API:** the events queue provides a clear interface to schedule a network event without directly interacting with the Network class or altering node implementation.

- **Simplified user code:** Network events are more complex than System ones; the events queue hides such complexity thus simplifying user code and avoiding setup errors.

- **Higher performance:** the management of all the events inside a single class improves performance; in fact the events queue is built around a single SystemC thread, minimizing memory usage and context switching.

This class can be used also to trigger new events defined through user-defined functions. The only constraint is that such functions shall not block the caller, i.e., the event queue, to allow a correct scheduling of the following events.

### 7.1.7 Implementation of network protocols and channels

Communication protocols are a fundamental aspect of network simulations. Each scenario could require a different protocol, which provides different API’s, while simulator components require a standard API to allow code reuse. These opposite requirements can be satisfied as follow:

- **Each task will interact with the simulator by using standard API’s.** Such API’s will allow performing only the basic and general communication operations, i.e. carrier sensing and packets exchanging. Advanced functionalities can be implemented by encapsulating additional information inside packets. In SCNSL, the API’s are implemented by the TaskProxies, as already explained.

- **In the backend, a standard packet format will be used, to allow component reuse, as described previously described.**
In the backend, components will implement an interface, to standardize the exchanging of packets and carrier status. In SCNSL this interface has been named Communicator. By using the Communicator interface, backend components can be joint into chains, to create complex behaviours.

In SCNSL, there are two options to use a protocol:

1. The user does not adopt any provided protocol model, since its implementation is the design goal. Thus, the protocol will be modelled as any other functionality, at task level. For instance, ad-hoc protocols and protocols that will be synthesized in hardware fall in this case.

2. The user relies on protocol models provided by the simulator. A complete protocol stack can be implemented by using a single simulator component, or by using a component for each protocol layer. Using a single component simplify the scenario creation, since fewer modules shall be instantiated, but using many components allows code reuse (e.g. TCP/IP and UDP/IP stacks have the IP layer in common). SCNSL supports both these implementation strategies.

The use of communicators has also the advantage to allow writing components to change communication behaviour, e.g. queues to buffer packets under transmission. The only constraint about Communicator’s chains is that each chain shall have at one end a TaskProxy, and on the other end a node. Transmission media have been modelled in components named Channels. In SCNSL, there are many channel models, according with the medium type and the physical simulation accuracy. For instance there are unidirectional links, full-duplex links, half-duplex links and wireless channels. Each channel type implements the Channel interface, in order to allow changing the channel model, without changing node implementation. Channels are the most complex objects, because they manage transmissions and, for this reason, they must be highly optimized. For instance, in the wireless model, the channel transmits packets and simulates their transmission delay; it must take into account node movements, check which nodes are able to receive a packet, and verify if a received packet has been corrupted due to collisions. The standard SystemC kernel does not address these aspects directly, but it provides important primitives such as concurrency models and events. The channel class uses these SystemC primitives to reproduce transmission behaviour. In particular, it is worth to note that SCNSL does not have its own scheduler since it exploits the SystemC scheduler by mapping network events on standard SystemC events.
8 References


