Public

	IST-5- 033511	ANDRES

ANalysis and Design of run-time REconfigurable, heterogeneous Systems

Project Duration	2006-06-01 – 2009-05-31	Type	STREP

	WP no.	Result no.	Lead participant
	WP1	**D1.2b**	**UC**

Modelling of SW. Final Library elements.

Project coordinator name/organisation :	**Dr. Frank Oppenheimer – OFFIS**
Issued by	**F.Herrera, S. Real, E.Villar- UC**
Document Number	**ANDRES/UC/P/D1.2b/1.2**
Classification	**ANDRES Public**
Submission Date	**2009-01-12**

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

History of Changes

ED.	REV.	DATE	PAGES	REASON FOR CHANGES
FHe, SRe, EVi	1.2	2009-01-12	118	Submitted Version.

1. Index

2. Introduction

2.1 *Motivation and Relation to ANDRES*

In this document, the final library elements for software specification and implementation in the ANDRES framework for the design of Adaptive Heterogeneous Embedded Systems (AHES) are described. These library elements correspond to those of the *HetSC* and *SWGen* specification and SW implementation methodologies respectively.

In the ANDRES project, *HetSC* **[HSCW]** and *SWGen* **[SWG08]** methodologies have been extended and integrated in the ANDRES AHES specification and design framework with a specific objective: the modelling and implementation of the software part of the AHES system. *HetSC* is used for the former objective, while *SWGen* is used for the latter one.

Several tasks have been completed to adapt and improve the *HetSC* and *SWGen* methodologies&libraries and to enable their integration in the ANDRES AHES development environment. It includes the following issues:

- Fixing the stable releases of the C/C++ compiler, SystemC library, etc. This figures out in two recent and new releases of *HetSC* and *SWGen* libraries (*HetSC*v1.3 and *SWGen*v1.2) and on their related documentation.
- Defining and enabling interfaces between *HetSC* and the other SystemC-based methodologies involved in the ANDRES project, namely SystemC-AMS and OSSS+R.
- Extending the software generation capabilities of *SWGen* to cover the clocked-synchronous domain
- Extensions defining how adaptivity is specified in *HetSC* have been provided. These extensions provide new methodological guidelines and patterns to specify *HetSC* Adaptive Processes (HAPs). Such patterns are based on the formalization of adaptivity done in **[D11A],** where abstract adaptive objects (AAO) take the shape of *Adaptive Processes (AP)*. As well as new methodological guidelines, additional library elements have been provided to the *HetSC* library. These are the *HetSC* Adaptive Process (HAP) templates.
- Proposing schemes for solving software generation of synchronous reactive specification.
- Identification of new interesting issues related to adaptive software and related to dynamically reconfigurable hardware, proposing new research objectives not covered by ANDRES and even other related projects UC-GIM is currently involved in.

2.2 *Improvements and Extensions from D1.2a*

For the reader aware of **[D12A]**, this document provides the following novelties:

- New separated chapters to explain the extensions of the *HetSC* and *SWGen* methodologies.
- A state of art section provides a global view of adaptive software and places better the ANDRES approach to adaptive software.

- Contribution done by this work to the state of art on the development of adaptive software is more explicitly explained.
- Refined relationship with the ForSyDe metamodel.
- Patterns for specifying HAPs are separately dealt from HAP SystemC templates, now available in the *HetSC* library. A chapter is used for each issue, which clearly distinguish the specification patterns from the available HAP templates.
- Adaptivity is addressed for the untimed and for the synchronous domains (**[D12A]** only covered untimed domain).
- The explanation of HAP patterns has been extended and improved. The different taxonomies of HAPs are now introduced. Moreover, each HAP pattern explanation is more structured, including now a section explaining the intended application of the pattern and a generic pseudo code of the pattern before the *HetSC* example (which already appeared in **[D12A]**).
- Explanation of process-based HAP (prHAPs) is moved to the end of HAP explanation as a special case testing the limits of SystemC language for the specification of adaptive processes. Feasible patterns for the prHAP are presented and possible interesting usages presented.
- The final state of the specific library elements of *HetSC* for specification of Adaptive Processes (HAPs) is reflected.
- Other sections regarding to Motivation, Content and Document Structure, Index, Conclusions and Future Work have been accordingly updated.

2.3 *Content and Document Structure*

This document is structured as follows:

Chapters 3 and 5 are in charge of introducing the *HetSC* and *SWGen* methodologies. In order to limit the extension of this document, the basic fundamentals of the methodologies are explained, while most of details can be found in the related documentation **[HSCW]** and **[SWG08]** updated and released as a result of the work performed in ANDRES.

Chapters 4 and 6 are dedicated to explain the most important ANDRES related extensions of *HetSC* and *SWGen* methodologies respectively.

Chapter 4 reports the most important *HetSC* extensions, which have to do with the connection with the OSSS+R and SystemC-AMS specification methodologies, and with the availability of the *HetSC* documentation. Other minor extensions, related to the host development platform, compiler and SystemC versions supported are already covered by Chapter 3.

Chapter 6 reports, as the most important extensions of *SWGen*, the support of synchronous models, included an already implemented POSIX port for the clocked synchronous domain; the availability of a related user manual; the availability of a µC/OS-II port; and the provision of a more comprehensible and useful structure of *Makefiles*.

Chapter 7 overviews different concepts and approaches to Adaptive Software. This serves to place into context the contributions done and reported by this document.

Chapter 8 fixes and explains the major contributions done in ANDRES in the field of adaptive software. Later on, it provides a brief overview of the general concept of Adaptivity developed in ANDRES, namely the Adaptive Process (AP).

Chapter 9 provides a reflection of how (untimed) APs can be implemented in software by using a software programming language, like C++, and a generic RTOS API. It also relates the formalization presented in Chapter 8 and the requirements of a software implementation technology, like *SWGen*, overviewed in Chapter 5. As well as generality to the discussion, this reflection should clearly show to a reader non familiar with the *SWGen* methodology that the different types of adaptivity described are useful and common (although seldom coded in a ruled or systematic way) in embedded software.

Chapter 10 provides guidelines and patterns to specify APs in SystemC following the *HetSC* methodology. First of all, it explains the types of APs covered, which includes the untimed and the synchronous domains, and the AP types described at the end of Chapter 8. Later on, each case is covered in a regular manner, explaining the intended usage of each HAP, giving specification patterns as SystemC pseudo-code, and providing a related example.

Chapter 11 explains the HAP SystemC templates included in the *HetSC* library, and how they provide complementary advantages to the availability of the specification patterns described in Chapter 10. The advantages of these templates, their capabilities and their related guidelines are explained and shown with some of the examples provided within the last version of the *HetSC* library.

Chapter 12 introduces the concept of Adaptive HdS (AHdS) which arises from the connection of Software with Dynamically Reconfigurable Hardware (DRHW).

Chapter 13 provides the final conclusions.

Chapter 14 addresses future interesting work motivated by the results achieved and reported in this document.

3. *HetSC* methodology

3.1 *HetSC principles*

HetSC **[HSCW]** is a methodology which enables heterogeneous specification of complex embedded systems in SystemC. Thus, it enables the specification of different parts of the system under different Models of Computation (MoCs) **[Jan05]**.

3.2 *Benefits of HetSC in the development of concurrent SW*

Support of heterogeneity is necessary to deal with the growing complexity, concurrency and heterogeneity of electronic embedded systems. *HetSC* is a methodology which aims to the complete design of a HW/SW embedded system. In the scope of ANDRES, it will be focused for the SW design flow. Thus, in this section, the main benefits of employing *HetSC* for the development of concurrent software are overviewed.

HetSC provides a set of advantages that can make the design of concurrent SW for embedded systems more systematic, safer and efficient. *HetSC* defines a set of specification rules and coding guidelines for each specific domain or MoCs. This makes the design task more systematic. Each part of the system is specified at the needed abstraction level, focusing on the important details and optimizing simulation speed. In addition, the fulfilment of MoC specification rules provides useful properties when handling concurrent specifications, such as determinism, deadlock protection, boundness, etc, which can be easily lost when SystemC language (or a programming language) is freely used. *HetSC* focus on determinism. This is a property targeted by every MoC currently supported by *HetSC*. HetSC is a specification methodology, which let reach mentioned properties without consideration of some implementation details, such as scheduling policy. This enables the targeting of the HetSC code to either software or hardware refinement. In any case, *HetSC* also supports a direct flow to SW implementation through a SW generation methodology called *SWGen* **[SWG08]**.

3.2.1 Domains supported

HetSC supports several domains:

- **Untimed domain:** It let specify in an abstract way concurrent models where the time associated to process computation is unknown, but where processes are able to communicate and synchronize among them in order to cooperate and fix certain order relationships between the events, data consumtions and productions. This modelling approach is usual in concurrent software programming, when using threads, processes and primitives for their communication&synchronization like shared variables, mutexes, flags, sockets, etc. With respect to concurrent software programming, *HetSC* enables a more abstract description (for instance, eliminating software implementation details like scheduling policy. priorities, etc. It also enables the usage of the specification for performance estimation, software generation and other system-level related techniques. Moreover, *HetSC* provides more specific groups of rules which match more specific modelling approaches. Each of these approaches provides the advantage of ensuring properties like determinism, but they distinguish among themselves by their ability to make feasible the analysis and/or ensure static schedulability, deadlock protection, etc. The modelling approaches or MoCs currently supported in the untimed domain of HetSC are Bounded Kahn Process Networks (**BKPN**), Kahn Process Networks (**KPN**), Communicating Sequential Processes (**CSP**), Synchronous Data flow (**SDF**) and their derivatives, such as Homogeneous SDF (**HSDF**).

- **Synchronous domains:** *HetSC* provides guidelines and specification facilities to build SystemC-based specifications handling a more detailed level of time handling in the specification. In these specifications, the user specifies concurrent and cooperative computations which have to react to specific local or global events with tight response requirements. At specification level, the computations are assumed to be infinitely fast. This again shifs performance estimation to a later and different design activity where the study of the fulfilment of time constraints must consider the time dimension of process computations (then *infinitely fast* condition has to be substituted by *sufficiently fast* condition). The infinitely fast reaction means that computations cooperating to provide a result for a given stimulus are considered to be synchronous. This approach is familiar to software developers in charge of implementing real-time reactive software. *HetSC* provides solutions for the specification of these kinds of systems, again from a system-level perspective. More specifically, Synchronous Reactive (**SR**) and Clocked Synchronous (**CS**) approaches are supported.
- **Timed domain:** It comprises the timed MoCs already supported in SystemC, such as Discrete Time (**DT**) and Discrete Event (**DE**). Moreover, the last version of the *HetSC* library has been made compatible with the SystemC-AMS library, useful for the support of analogous MoCs in SystemC **[HVG07]**.

3.2.2 Efficient support through MoC Abstraction and *HetSC* Formalization

One of the most characteristic features of the *HetSC* methodology is that it is directly and efficiently built on top of the SystemC standard kernel. This kernel is based on a Discrete-Event (DE) strict-time MoC. *HetSC* performs an efficient and predictable mapping of the events of the supported MoCs over the DE strict time MoC.

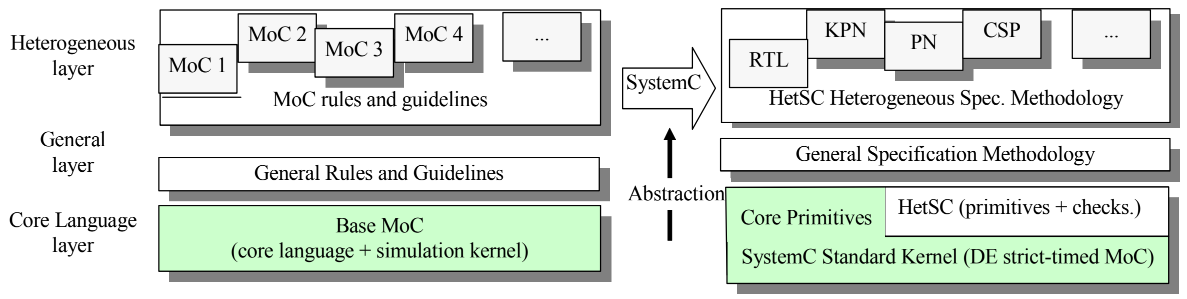

Heterogeneous layer
General layer
Core Language layer
MoC 1
MoC 2
MoC 3
MoC 4
...
MoC rules and guidelines
General Rules and Guidelines
Base MoC (core language + simulation kernel)
SystemC
Abstraction
RTL
KPN
PN
CSP
...
HetSC Heterogeneous Spec. Methodology
General Specification Methodology
Core Primitives
HetSC (primitives + checks.)
SystemC Standard Kernel (DE strict-timed MoC)

Figure 1. The *HetSC* specification methodology is over the DE strict-time MoC of the SystemC kernel.

The immediate advantage of this approach is that there is no need for MoC specific solvers, which provides efficiency to the methodology. Several results for the abstract MoCs supported show that simulation speed is close to those obtained with MoC specific solvers.

The analysis of simulation results requires the abstraction of the time information. This abstraction depends on the MoC. That is, if a specification under an untimed MoC is being simulated, the time information of the simulation results has to be abstracted. This is represented in Figure 2. On its lefts hand side, write accesses to channel *ch1* and channel *ch2* are abstracted as crosses. In the simulation, each write access has associated time information. This has been depicted on the right hand side of Figure 2; where crosses are over its associated SystemC time information, that is a (t,δ) coordinate, where t represents the time stamp and δ the delta cycle (relative to the time stamp).

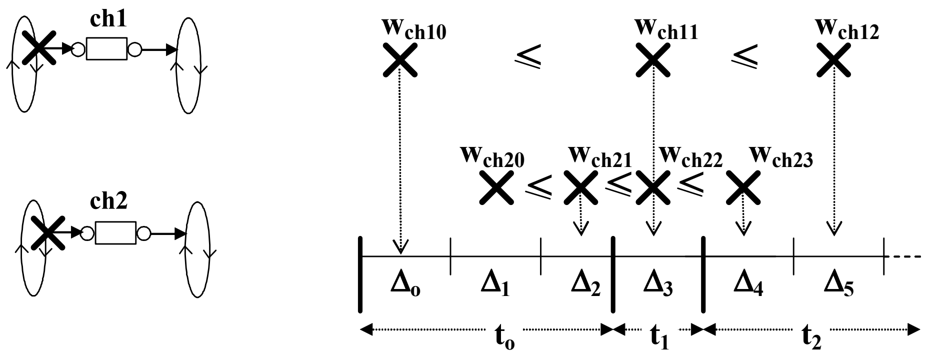


Figure 2. Abstraction of time information.

In the specification and analysis of the simulation results of an untimed model, the *HetSC* user only considers the order relationships between these write events. Untimed specification only fixes such order relationships, while nothing is stated over the rest of time information. Therefore, only order information has to be extracted from the simulation trace (depicted through ≤ symbols on the right hand side of Figure 2). In this figure, each w_{chij} event has a (t,δ) coordinate associated in the simulation trace. However, only order relationships between events is relevant (i.e. $w_{ch10} \leq w_{ch11} \leq w_{ch12} \leq \ldots$, showing that write accesses to *ch1* are sequentially done). More specifically, it has to be considered that write events of *ch1* and write events of *ch2* are unrelated since *ch1* and *ch2* channels belong to decoupled networks. Similar reasoning applies for other untimed and synchronous MoCs in the *HetSC* methodology. This is a feasible reinterpretation of the simulation semantic, consisting in neglecting the information which the MoC does not handle. This flexible interpretation of time information provides enough flexibility for placing the write events in other points of the time axis, as a consequence of further design steps.

This is part of the task of formalization of *HetSC* in ForSyDe, which, as commented, is work in progress and will be reported in further documents of ANDRES.

3.3 *The HetSC library*

The *HetSC* methodology is based on a set of specification facilities. Whenever possible they are taken from the OSCI SystemC library. The *HetSC library*, a proof-of-concept library associated to the *HetSC* methodology, provides a set of facilities to cover the deficiencies of the SystemC core language for heterogeneous specification. Therefore, in order to enable the compilation and execution of *HetSC* specifications, the *HetSC* user only needs to install the OSIC SystemC library and the *HetSC* library.

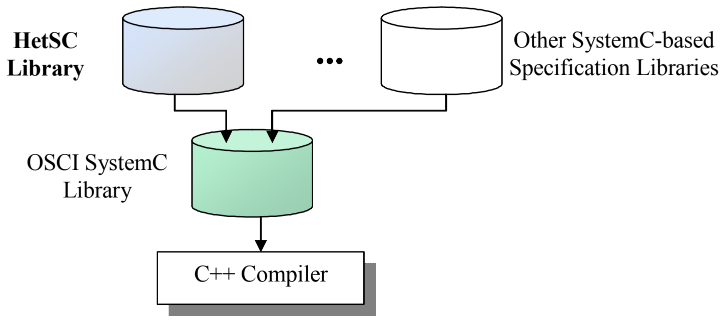


Figure 3. The *HetSC* library is installed over the OSCI SystemC library.

As mentioned, the support of some specific MoCs requires new facilities, not included by the SystemC library. These facilities are:

- Specification facilities, such as interfaces, channels, etc. They provide the specific semantic content and abstraction level required by their corresponding MoC.
- Facilities for specifying HW/SW partition and other information which is used by other system-level design activities (such as software generation or time profiling).
- MoC rule checkers. These are facilities, often transparent for the *HetSC* user, which help to detect and locate MoC rule violations in the specification.
- Facilities for generating MoC specific reports.
- Facilities for assisting the debugging task of concurrent specifications.

The following tables show important specification facilities of the *HetSC* 1.3 library: Table 1 shows *HetSC* interfaces; Table 2, *HetSC* channels; and Table 3, *HetSC* border channels.

MoC	Interfaces
CSP	uc_caller_if<T>
	uc_accepter_if<T>
	uc_rv_if<T>
	uc_sync_if<T>
PN/KPN	sc_fifo interfaces ...
PN(1) (with fifo of sizes=1)	uc_simple_read_if<T>
Equivalent to HSDF (HSDF)	uc_simple_write_if<T>
SDF	uc_arc_introspection_if<T>
	uc_arc_prod_seq_if<T>
	uc_arc_prod_dir_if<T>
	uc_arc_cons_seq_if<T>
	uc_arc_cons_dir_if<T>
	uc_arc_prod_if<T>
	uc_arc_cons_if<T>
	uc_arc_if<T>
SR	*uc_SR*_read_if<T>
	*uc_SR*_write_if<T>
	*uc_SR*_if<T>
KPN/	uc_sig_client_in_if<T>
PN/	uc_sig_client_out_if<T>
CSP-REFINED	uc_sig_server_in_if<T>
	uc_sig_server_out_if<T>
	uc_sigclocked_client_in_if<T>
	uc_sigclocked_client_out_if<T>

	uc_sigclocked_server_in_if<T> **uc_sigclocked_server_out_if<T>**
I/O	**uc_uart_if**

Table 1. *HetSC* interfaces.

MoC	**MoC channels**
CSP	**uc_frv**
	uc_rv
	uc_rv_uni
	uc_rv_sync
PN	**uc_fifo**
KPN	**uc_inf_fifo**
HSDF	**uc_simple_channel**
SDF	**uc_arc_seq**
SR	***uc_SR***
PN-refined	**uc_sigclocked_fifo**
Others	**uc_bucket, uc_protected, uc_shared**

Table 2. *HetSC* channels.

MoC connection	**Border Channels**
PN-CSP	**uc_fifo2rv**
PN-SR	**uc_fifo_SR**
	***uc_SR*_fifo**
KPN-SR	**uc_inf_fifo_SR**
	***uc_SR*_inf_fifo**
RT(HW)-KPN	**uc_signal_inf_fifo**
	uc_inf_fifo_signal

Table 3. *HetSC* border channels.

3.4 *Specification methodology*

The *HetSC* methodology defines a set of specification rules and guidelines about how to use the specification facilities provided by the SystemC and *HetSC* libraries to build a specification under a specific MoC. Moreover, the methodology supports a smooth integration of several MoCs in the same system specification.

The *HetSC* Specification Methodology provides two levels of rules. This has been reflected in Figure 1. There is a first level of rules and guidelines which apply for every MoC supported by the *HetSC* methodology. This level is called *General Specification Methodology*. As well as the specification task, these rules facilitate the application of other design activities to be carried out at a very high abstraction level (system-level). They also facilitate the codesign flow (software generation, hardware synthesis and HW/SW interface generation). The second level enables single-MoC specification and heterogeneous specification. This includes the specification under different MoCs and connection of MoCs through MoC interfaces.

3.4.1 *HetSC* Graphical representation

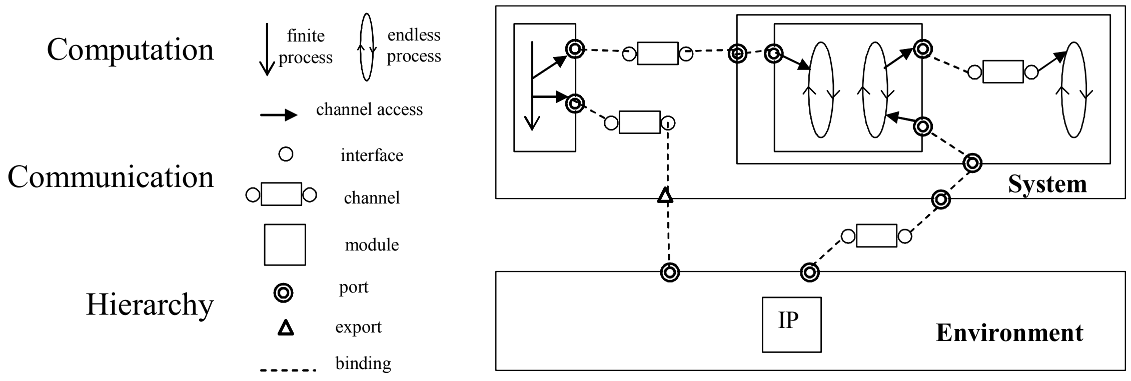


Figure 4. Basic specification facilities of the *HetSC* methodology.

HetSC provides a *graphical representation* of the specification facilities, shown in Figure 4. It enables a quick and intuitive view of the specification. However, the complete information of the specification is in the *HetSC* specification. Further details are found in the *HetSC* user guide **[HSCW]**.

3.4.2 General Specification Methodology

The specification methodology establishes basic rules and methodological guidelines for writing an executable specification which encloses both the test bench and the system modules (instances deriving the *sc_module* class). By default, all these rules are common for each MoC supported. When one rule does not applies (or does it with any nuance) in a specific MoC, it is made explicit. The most important rule of the *General Specification Methodology* is the strict separation between computation and communication. This is done stating that specification processes communicate only by means of channels. Although SystemC channels have, in general, an arbitrarily complex semantic, the aim is that *HetSC* channels have a bounded semantic, limited to what can be understood as necessary for a correct transfer of data. Thus, *HetSC* channel semantic only handle questions as the bufferng size of the channel, synchronization policies, sense(s) of transfer, etc. In other words, any semantic that can be interpreted as functionality or computation is to be specified within processes.

Another important point is that the specification methodology focuses the task of the designer on writing concurrent functionality as C/C++ sequential algorithms, typed as the content of *HetSC* processes (*SC_THREADs* and *SC_METHODs*). Processes are communicated by instancing available SystemC and *HetSC* channels and accessing them from within process code. Further details about the *General Specification Methodology* are found in the *HetSC* user guide **[HSCW]**.

3.4.3 Single-MoC Specification

This level provides a new set of facilities (indeed, most of the facilities provided by the *HetSC* library) and rules that complement and, only eventually and explicitly, override the rules of the general specification methodology.

In a first stage, the rules and guidelines to specify under each single MoC are given. Several untimed MoCs are supported. A process network (PN) can be written as a network of SC_THREADs communicated through *uc_fifo* channels. This *HetSC* channels has a similar semantic to the *sc_fifo* standard channel. However, the *uc_fifo* channel prevents the use of non-blocking and introspection accesses, which is required to ensure the determinism of the specification. The *uc_inf_fifo* channel enables the approach to KPN MoC. It provides and unbounded buffering semantic, thus a non-blocking write access. This channel also reports the maximum buffering size. When time domain constraints are present, this enables a refinement of the KPN specification to a bounded process network (PN). *HetSC* enables the specification of tightly coupled networks through the CSP MoC. This MoC is again a network of SC_THREAD processes which are communicated by up to three kinds of rendezvous channels, the *uc_rv_sync* channel, the *uc_rv_uni* and the *uc_rv* channel. All of them share the rendezvous synchronization semantic, which relates N processes (N= 2 for the latter two rendezvous channels). Under the rendezvous synchronization semantic, any process arrival involves a blocking except for the case when N-1 of the N process involved in the rendezvous has already arrived. In such a case, there is no blocking and every involved process is resumed. *HetSC* supports a dynamic approach to the SDF MoC. The *uc_arc* channel enables the a-priori specification of the consuming and producing rates, which immediately fixes the size of the *uc_arc* channel. SDF nodes are implemented as SystemC processes. SDF process style is constrained in *HetSC* since it has to represent a computation quantum. Each computation quantum is implemented as an iteration of an infinite loop in a SC_THREAD. The input *uc_arc* channels are read at the beginning of the iteration, while the output *uc_arc* channels are written at the end of iteration. *HetSC* provides some specific reports for the SDF MoC, such as the firing schedule of the SDF node graph. In every untimed MoCs several checks are available, such as those restricting the number of processes which access as reader and as writer each channel instance.

Synchronous MoCs are supported. The Synchronous Reactive MoCs is supported through the abstraction of the prefect synchrony hypothesis over the SystemC time axis. This hypothesis dictates that the system instantaneously reacts to the environment stimulus. This instantaneous reaction is interpreted in SystemC as a computation that takes no time in terms of the SystemC time stamp (t coordinate). However, it can involve an advance of one or more delta cycles (δ) in the same time stamp. In this way, a SR specification in *HetSC* is composed of a set of generator processes (SC_THREAD processes), usually representing the environment, which write tokens through *uc_SR* (or *uc_buffer*) channels. These channels present a similar semantic to the *sc_buffer* standard channel, but include some useful checks which constraint the specification style to what it is demanded by the SR MoC in *HetSC*. Specifically, these channels oblige that the generator processes do not write more than once in the same time stamp. That is, they limit up to one domain event per each time stamp. Thus, each time stamp is a slot. The *uc_SR* channel is also a mean to link the processes of a SR specification in what is called a reactive chain. The *uc_SR* channel involves a reaction of the attached reader process, enabling at the same time the transfer of a data token. This reaction is given in the next delta cycle, without time stamp advance, which is coherent with the interpretation of perfect synchrony in *HetSC*. The *uc_SR* channel also enables a dynamic check for detecting the violation of the perfect synchrony hypothesis. Moreover, the perfect synchrony condition can be relaxed. This means letting some time advance in the reaction. In this case, the analysis of the reaction time is automatically done to detect if it was quick enough to cope with the speed of domain events. An activity analysis utility is able to report how many deltas where consumed at each slot of a SR specification in *HetSC*. The CS MoC is conceived in *HetSC* as a particular case of SR where there is only one triggering input, the clock. This clock can be produced through a signal or a buffer channel. Then, system processes need to be sensitive only to the clock signal and the communication among processes is done through *sc_signal* channels. Therefore, behavioural and RTL hardware styles of SystemC are particular specification styles of a CS MoC.

3.4.4 Heterogeneous Specification

MoC integration is smooth in *HetSC* because it employs elements of the same language for the connection: SystemC channels and processes, called in the methodology Border Channels (BCs) and Border Processes (BPs). The *HetSC* methodology gets into less straightforward semantic issues related to MoCs' connection. It establishes a classification of MoC interfaces (untimed-untimed, untimed-synchronous, etc). Each type of connection presents specific issues. Most important ones have to do with time adaptation, specifically when a different level of detail is handled in the MoCs connected (such as in untimed-synchronous connections). The *HetSC* library currently provides some border channels for different kind of MoC couplings.

More details about how to write single MoC specification and how to integrate them in an heterogeneous SystemC specification are found in the *HetSC* user guide **[HSCW]**.

3.5 *Installation of the HetSC Library*

3.5.1 Development Platform

The last version available is the ***HetSC*v1.3** library. This version has been checked on a platform with the next characteristics:

- Linux kernel Linux 2.6.3.
- gcc-4.1.2 compiler.
- SystemC-2.2.0.

SystemC-2.2.0 adapts to the IEEE1666 standard **[IEEE06]**. *HetSC* 1.2 beta does not keep backward compatibility with previous releases of the SystemC library. This has enabled a more efficient and clear implementation of *HetSC*, ready for the development and use by other ANDRES partners. For previous releases of SystemC, the previous version of the *HetSC* library, *HetSC*v1.1**,** is available.

3.5.2 Need of SystemC patch removed for *HetSC*1.2 and SystemC-2.2.0.

The *HetSC* v1.3 library is installed directly over the SystemC-2.2.0 library. *HetSC*-v1.1, the last version before the start of ANDRES needed the previous application of a patch for the SystemC-2.1.v1 release. This patch is provided by the GIM-UC **[HSCW]**. It provides a fix of the count of deltas which enables a correct support of the SR MoC. This patch provides other fixes and features which are useful but not required a correct support of the *HetSC* features. One is a fix for a check related to the *sc_fifo* standard channel. The other is an extension for enabling pseudo-random scheduling, a feature not implemented by the SystemC library yet.

These additional fixes and features are being provided for SystemC-2.2.0. As has been mentioned, its installation is not necessary. It is recommended for a correct use of the standard *sc_fifo* channel and for enabling an improved coverage of system-level verification performed through simulation **[HeVi07]**. For instructions about how installing the UC *patchs* refer to the *HetSC* user guide **[HSCW]**.

3.5.3 Installation

In order to build and install the *HetSC* library, the next instructions are given:

- It is assumed that the *HetSC* folder is untared and placed in the path *$(untar_path).* For instance, if you untar the *HetSC1.3.tar.gz* file in */home/usr/src*, then it creates a */home/usr/src/HetSC* directory, that is, *untar_path=/home/usr/src/HetSC*).

  Make sure that your compiler is a right version. Use the *gcc –v* command for that. Make also sure that the SystemC installation is done with the same *gcc* compiler version that you will use to install *HetSC* library.

- Set and export the SYSTEMC_PATH variable. For instance, in Linux, write:

  $SYSTEMC_PATH=/home/user/soft/systemc-2.2.0/installdir

  $export SYSTEMC_PATH

- Open and edit the *Makefile.defs* file. You only have to edit the next variables:
  - *HETSC_PKGDIR:* This is the path where the *HetSC* library has been untared.

    (For instance, in our example, *HETSC_PKGDIR=/home/usr/src/HetSC*)

- o *INSTALLDIR:* This is the path where the *HetSC* library will be installed

  (For instance, *INSTALLDIR = /home/usr/soft/HetSC*)

  - It is not recommended this directory to be the same as the *HETSC_PGDIR*.
  - *INSTALLDIR* could also be the SystemC installation dir. Then environment variables to set SystemC paths could be reused. The library is prepared for a clean uninstall without touching SystemC installation. However, it is recommended a separated installation directory.

- o *BUILDIR:* This is the path where the *HetSC* library is compiled

  (For instance, *BUILDIR=$(INSTALLDIR)/buildir*)

- o *(ONLY FOR HetSC1.1) SYSTEMC*: This is the path where your SystemC library is installed.

  (For instance, *SYSTEMC=/home/usr/soft/systemc-2.1.v1/installdir*)

- After these variables are set, save the file and compile&install the library through the make command at the $(*HETSC*_PKGDIR) directory:

  $make

Then the compilation is carried out. These two steps can be separately done by executing:

$make build (for compilation of the library)

$make install

Further details on the installation of previous versions of *HetSC* are found in the *HetSC* user guide **[HSCW]**.

3.5.4 Removing build directory

You can remove object files to save disk without perturbing the installation. In order to do this, just execute the next command at the *$(HETSC_PKGDIR)* directory:

$make clean

3.5.5 Uninstall *HetSC* library

The *HetSC* library can be uninstalled just by executing the next command at the *$(HETSC_PKGDIR)* directory:

$make uninstall

3.5.6 How to use the *HetSC* library

In order to use the *HetSC* library you should include the *"general.h"* header (or the "*HetSC.h"* if you are not using other UC libraries) in your specification. This header already includes the "*systemc.h"* header. Before compiling your application, ensure that you include *$(INSTALLDIR)/include* path in the include paths, the static library path, *$(INSTALLDIR)/lib*, and the *HetSC* library it self.

For instance, if you write a specification file called *myspec.cpp*, then you can use the command:

$g++ -I$(INSTALLDIR)/include -L$(INSTALLDIR)/lib -o myspec.x -lHetSC myspec.cpp

to obtain the binary executable *myspec.x*.

3.5.7 Examples

The library comes with a set of simple examples which show some of the features of the *HetSC* library and can be taken as a reference for constructing other examples.

To compile them, move to the */examples* directory in the source directory and edit the *Makefile.defs* file to indicate the installation directory. After that, type the next command:

```
$make –f Makefile.sys
```

Then all provided examples will be compiled. All the examples can be cleaned (removal of object and executable files) through the next command.

```
$make –f Makefile.sys clean
```

In order to compile and clean a specific group of examples, then enter the specific directory and compile them with the same command. For example, to compile only KPN examples, enter the */examples/KPN* directory and type:

```
$make –f Makefile.sys
```

It is possible to compile and clean single examples entering their specific directory and applying a rule of the *Makefile.sys* file. For instance, to compile the first example of KPN directory you should type in:

```
$make –f Makefile.sys example1
```

Most of the examples are ready to, through some simple comment/uncomment of define clauses, to check many different features and behaviour of the *HetSC* facilities. For it, it is necessary to recompile the specific example where you are making such editions.

4. *HetSC* extensions and improvements

The last version released of the *HetSC* library incorporates also:

- A complete documentation available on-line on the *HetSC* website **[HSCW]**.
- Guidelines, specification facilities and examples for the connection of *HetSC* with SystemC-AMS.
- Guidelines and an example for the connection of *HetSC* and OSSS+R.

4.1 *HetSC documentation*

An important feature of *HetSC* is its methodological aspect, independently on the complementary specification facilities provided by the *HetSC* library. Indeed, if SystemC were 100% suitable for all the domains or Models of Computations covered by *HetSC*, no kind of library extension would have been provided. In this sense, fixing a methodology requires a comprehensive and detailed documentation. This aspect of the methodology has been improved by making available in the *HetSC* website a full documentation section which includes the following:

***HetSC* user manual:** The core documentation where the basic principles of the methodology are explained. It includes the general specification methodology, the graphical representation and links to related documentation.

Annexes: They cover in more detail the different and distinctive aspects of the methodology. The following Annexes are available:

- **Annexe A:** Documentation about the *HetSC* library (installation, specification facilities, compatibility and connection with other SystemC-based libraries, etc).
- **Annexe B:** Addresses Mono-MoC specification in SystemC. It is composed of different sections, each one dedicated to a domain or Model of Computation. Currently, 6 annexes are available.
- **Annexe C:** Addresses MoC interfaces. This annexe, together with Annexe B, provides the methodological support for system-level heterogeneous specification in SystemC.
- **Annexe D:** Documents experimental facilities.
- **Annexe E:** Addresses several aspects about older versions of *HetSC*, compatibility with different versions of SystemC, and about patches provided for SystemC.
- **Annexe F:** Provides and editable document to facilitate the development of HetSC graphical representation of SystemC specifications.

In deed, this document will serve to release an additional annexe about specification of adaptivity in SystemC through the results of Chapter 10. It will also mean the extension of the Annexe A with the facilities reported in Chapter 11.

Additionally, the documentation of the *HetSC* website provides additional documentation in the shape of the presentations done in tutorials, workshops and conferences. Moreover, some of the examples available in the '*downloads'* section of the *HetSC* site have now an associated documentation available.

4.2 *Connection of HetSC with SystemC-AMS*

In **[HVG07]** the connection of SystemC-AMS and *HetSC* was explored. It was shown, that both methodologies can collaborate to supp-ort a wide spectrum of MoCs. Moreover, the collaboration of these methodologies provides an efficient balance between MoCs directly supported over the DE strict-time kernel, and MoCs relying on additional synchronisation and solver layers. The idea is that specific solvers are provided only for a set of MoCs where they provide a significant simulation speed up. In this approach, the set of MoCs relying on solvers corresponds to the analogue domain, where the simulation speed up can be of several orders of magnitude, while untimed and synchronous MoCs can be satisfactorily supported directly over the SystemC kernel. The exception would be fine grained SDF specifications, where the speed up of a static SDF compared to a dynamic SDF could be significant. Thus, in these cases, the static scheduling provided by the T-SDF solver of SystemC-AMS should be favoured. Figure 5 shows how SystemC-AMS and *HetSC* complement each other.

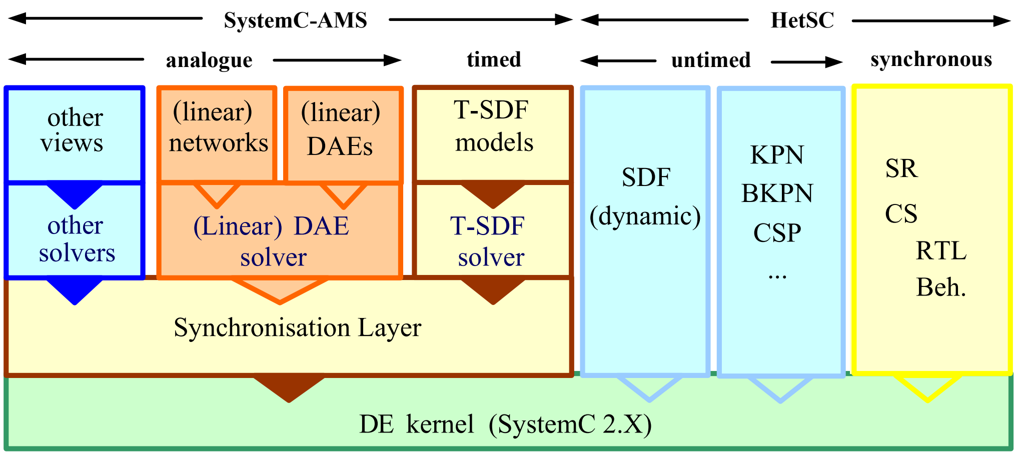


Figure 5. SystemC-AMS and *HetSC* collaboration provides a wide MoC coverage.

In **[HVG07]** has been shown that the connection of SystemC-AMS and *HetSC* can be done by means of *HetSC* border processes and channels, at least in those cases where data type conversion is not necessary.

In the '*downloads'* section of *HetSC* a soundboard example, presented in **[HVG07]** website is now available. The example is represented in Figure 6.

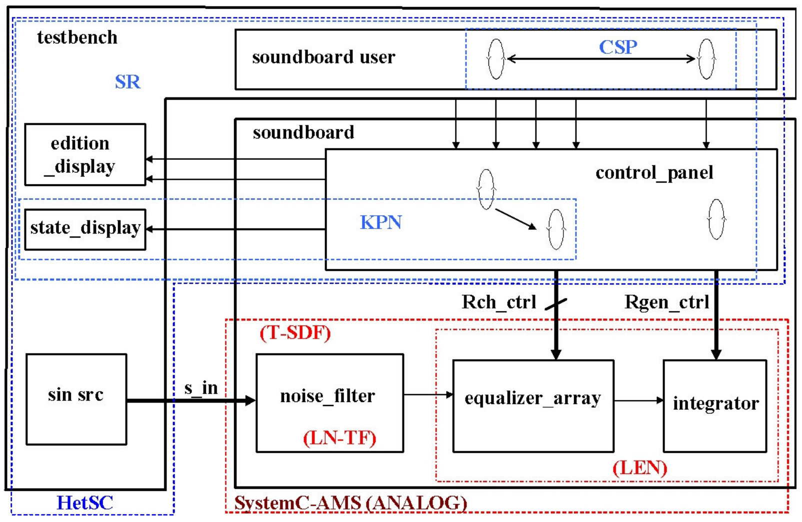


Figure 6. The soundboard example connects *HetSC* and SystemC-AMS parts in the same SystemC model.

Such an example deals with the basic interfacing between untimed fifo-based models and the timed-clocked synchronous models, which is the base of the SystemC-AMS approach. It also deals with the interfacing between a synchronous reactive part and those parts of the system described as linear electrical networks (LEN).

The work in **[HVG07]** has served to provide a more general and compact set of border channels (*tsdf_2_pn, pn_2_tsdf, uc_fifo2v, uc_signal2v, etc*) and related guidelines which are now documented in the Annexe C of the *HetSC* User Manual. For instance, Figure 7 shows one of the border channels, *tsdf_2_pn*, used to communicate the T-SDF domain of SystemC-AMS with blocking fifo-based process networks.

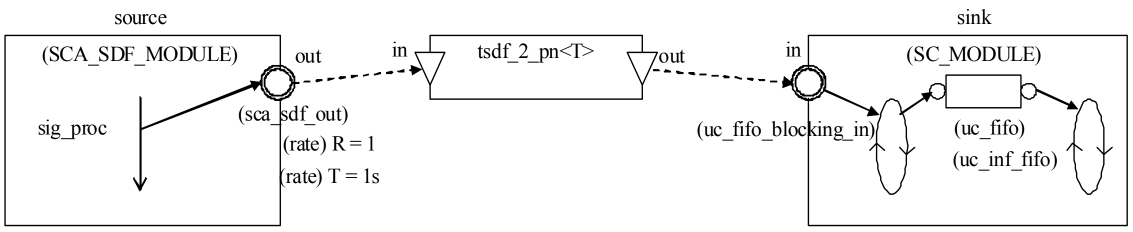


Figure 7. Example of use of the tsdf_2_pn channel.

Moreover, in the ANDRES framework, *converter channels* have been developed **[HDG07]** **[HDG08]**, beyond the *HetSC* methodology. Converter channels incorporate the adaptation in terms of time semantic (as border channels) and the features of *polymorphic* channels **[Sch07]**, which can save communication refinements when exchanging blocks in the refinement of the initial specification. Converter channels also provide the additional feature of automatic data type adaption, while border channels leave data types transferred within channels untouched. Converter channels have been implemented for the KPN/BKPN ↔ T-SDF conversion, where the KPN/BKPN MoCs are represented as FIFO. This implicitly also solves KPN/BKPN ↔ LEN conversion, since LEN is basically synchronized to SystemC's DE-kernel via T-SDF. More detailed documentation about the conversions implemented and the way to use converter channels can be found in **[D15A] [D16A]**.

4.3 *Connection of HetSC with OSSS+R*

The connection of software and digital adaptive models in ANDRES has been and is being tackled in ANDRES in two levels:

At specification level: The goal is to enable a SystemC-based abstract specification of adaptive systems suitable for its implementation in platforms based on one or more microprocessors (that is, on software) and partially dynamically reconfigurable hardware (DRHW). Specifically, this is done by defining the mechanism for connecting *HetSC* **[HSC08]** and OSSS+R **[D13A] [OSS08]** methodologies.

At implementation level: To goal is define a flow for SW and HW implementation on the previously mentioned platform, based on the associated implementation technologies *SWGen* **[SWG08]** and *Fossy* **[FosW]** (OSSS+R).

This section summarizes the connection at SystemC-level, which has been necessary to enable the AHES ANDRES framework integrating the SystemC-based specification libraries (among them *HetSC* and OSSS+R), and which do not integrate actually the implementation tools. The advances done in the implementation level have been reported in **[D24B]**.

In **[HVH08]** the basic structure, guidelines and specification facilities required to enable the connection of the *HetSC* and OSSS+R methodologies are given. The basic structure of a specification including *HetSC* and OSSS+R specification facilities proposed in **[HVH08]** is shown in Figure 8.

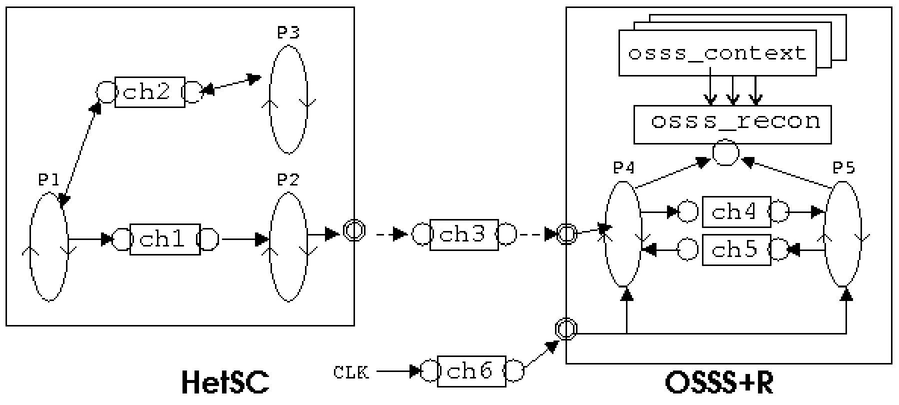


Figure 8. Module and Concurrency structure of a *HetSC*/OSSS+R specification.

A main observation is that it is assumed that the reconfigurable objects and the named contexts are part of the OSSS+R part and handled only by SystemC clocked processes. OSSS+R is under a timed clocked MoC (similar as a Register Transfer Level or RTL from the time domain point of view, plus the abstract capabilities provided by shared and reconfigurable objects). Therefore, the connection of an untimed *HetSC* model with OSSS+R models involves MoC interfaces. The proposal in **[HVH08]** is to use the *HetSC* facilities, like border processes or border channels (BCs), like *ch3* in Figure 8, which can be of a transactor type, with one method call based interface and a signal based interface. In the *HetSC* website **[HSCW]**, an example where *HetSC* an OSSS+R models are connected has been left available. This example starts from a system-level model composed of a set of producer and consumer processes and an adaptive ALU specified as a *HetSC* adaptive process (HAP) as defined in chapter 10. Then, ALU is assumed to be implemented in hardware and refined as an OSSS+R reconfigurable ALU.

Later on, in **[D24B]**, a scheme where untimed processes from the *HetSC* part are able to access methods of an OSSS+R reconfigurable object has been proposed.

5. *SWGen* methodology

SWGen **[SWG08]** is a library-based methodology (*SWGen* library), for automatic generation of embedded software from SystemC, specifically, from *HetSC* code.

5.1 *Advantages of the SWGen methodology*

This methodology has several distinguishing features:

Single Source. The same SystemC (*HetSC*) code used for generating the executable specification is used to generate the software binary code for the target implementation. This is called single-source software generation **[PHF04][FHSV03]**.

Automatic Generation. A few console commands serve to generate the source code and the binary code.

Efficient generation. The generation removes all the implementation code of the SystemC library which is not necessary for the implementation of the target embedded software. The eliminated code corresponds to features for simulation over the discrete event kernel and other system-level features provided by the specification libraries (SystemC and *HetSC*). Therefore, the code generated is smaller and faster than if the cross-compilation of the *HetSC* application included the cross-compilation of the SystemC library.

Based on a Realistic and Reliable SW platform. The software generation library generates code which includes system calls to an embedded RTOS. That is, the RTOS is considered as part of the target platform. This takes advantage of the current state of art in software generation. Any commercial or well established embedded RTOS can be used. Thus, there is no need to write or to synthesize an ad-hoc and untested RTOS.

Low demand for the port of the RTOS and the cross compiler. The *SWGen* library requires a basic support of RTOS services and platform facilities. In addition, since the SystemC library does not need to be cross-compiled, the demands for the port of the cross-compiler are relaxed.

Design for Extensibility to new target platforms. The *SWGen* library is structured in several translation levels. Each of these levels corresponds to different source code packages: sc2cpp (for *HetSC* code that can be directly translated to C/C++), sc2rtos (for *HetSC* code that needs to call RTOS services through a RTOS-API) and sc2platform (for *HetSC* code that need to invoke services out of the RTOS-API).

5.2 *Scope of the SWGen methodology*

5.2.1 Type of Input SystemC code

The library supports basic SystemC features such as modularity, concurrency and separation of computation and communication. More specifically, the general specification methodology of *HetSC* is practically covered. In the current public release this includes the following specification facilities: *SC_MODULE*, *SC_CTOR*, *SC_THREAD*, *wait(double,sc_time_unit)*, *sc_main*, *sc_start*; generic *ports* and basic port binding facilities; and some standard channels. Not every SystemC constructs are supported. The support can be extended in future versions of *SWGen*.

5.2.2 Type of software code generated

The software generation produces C++ code which includes system calls to an embedded RTOS. It includes C++ wrapping code to support an automatic translation from SystemC to C++. There are two generation options. One is to stop at pre-processing time. Then, the source C++ code is generated. The other one includes the application of the rest of the cross tool chain to obtain the target binary code. This code can be downloaded to the platform memory for its execution.

5.2.3 Type of Target and Development platform

The target is any HW/SW platform targeted by a C++ cross-compiler and supporting a RTOS with a minimum set of services (basically, concurrency). The development platform can be a PC or a workstation where the software development kit (SDK) runs. The SDK includes the cross-tool chain (at least, the C/C++ cross-compiler and linker) and the embedded RTOS (shortly named here RTOS, eRTOS or eOS).

5.3 *Basics of software generation in SWGen*

In the *SWGen* methodology, the *HetSC* and SystemC libraries are substituted by the *SWGen* library. This means that the *HetSC* and SystemC specification facilities are substituted by an efficient C++ implementation which, in general, incorporates system-calls to a RTOS-API and low level calls to the underlying drivers. This has been graphically depicted in Figure 9. In the Figure 10, the same has bee represented in terms of code.

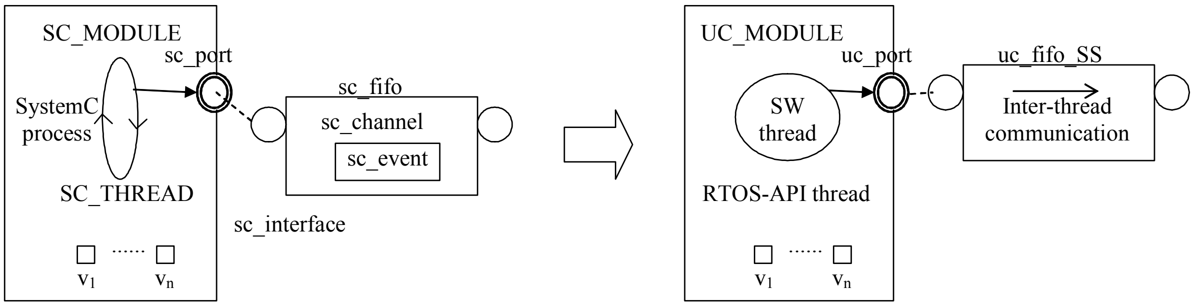
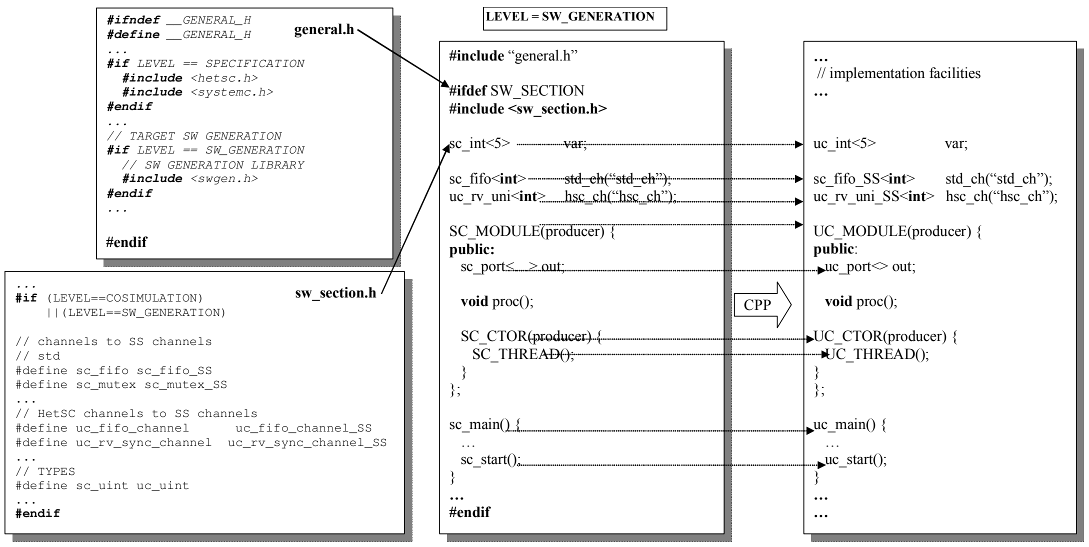


Figure 9. Graphical representation of the *SWGen* generation.



```
#ifndef __GENERAL_H
#define __GENERAL_H
...
#if LEVEL == SPECIFICATION
  #include <hetsc.h>
  #include <systemc.h>
#endif
...
// TARGET SW GENERATION
#if LEVEL == SW_GENERATION
  // SW GENERATION LIBRARY
  #include <swgen.h>
#endif
...

#endif
```

```
...
#if (LEVEL==COSIMULATION)
    ||(LEVEL==SW_GENERATION)

// channels to SS channels
// std
#define sc_fifo sc_fifo_SS
#define sc_mutex sc_mutex_SS
...
// HetSC channels to SS channels
#define uc_fifo_channel      uc_fifo_channel_SS
#define uc_rv_sync_channel  uc_rv_sync_channel_SS
...
// TYPES
#define sc_uint uc_uint
...
#endif
```

```
#include "general.h"

#ifdef SW_SECTION
#include <sw_section.h>

sc_int<5>         var;

sc_fifo<int>      std_ch("std_ch");
uc_rv_uni<int>    hsc_ch("hsc_ch");

SC_MODULE(producer) {
public:
  sc_port<...> out;

  void proc();

  SC_CTOR(producer) {
    SC_THREAD();
  }
};

sc_main() {
  …
  sc_start();
}
…
#endif
```

```
…
 // implementation facilities
…


uc_int<5>            var;

sc_fifo_SS<int>      std_ch("std_ch");
uc_rv_uni_SS<int>    hsc_ch("hsc_ch");

UC_MODULE(producer) {
public:
  uc_port<> out;

  void proc();

UC_CTOR(producer) {
  UC_THREAD();
}
};

uc_main() {
  …
  uc_start();
}
…
…
```

Figure 10. Substitution of the *HetSC* specification facilities by C++ efficient implementation facilities.

As can be appreciated, the concurrency and modular structure of the *HetSC* specification is respected. The modular structure is respected in order to obtain a simple and safe translation mechanism, where the hierarchy of visibility scopes is respected. A flattening of the code would actually require name mangling. In addition, this mapping removes, as mentioned in section 5.1, many SystemC features not necessary in SW generation. For instance, the *sc_port* is substituted by an *uc_port* class. This is an almost empty class which only keeps binding features. However, features such as static port checks are removed since they are only used at system-level.

The concurrency structure is also preserved. SystemC processes are substituted by RTOS threads. The complex data structure maintained for the handling of SystemC processes is substituted by a minimum structure of data required to declare and handle SW threads. These data include at least a thread handler and a stack variable.

The simulation-context class is a main class of the SystemC library in charge of implementing the Discrete-Event (DE) simulation kernel of SystemC. Roughly, this DE kernel divides the SystemC time in an elaboration and a simulation phase. The simulation phase is divided in time advances called time stamps. Time stamps in delta cycles. Finally, delta cycles are divided in evaluation and update phases. The elaboration phase is also subdivided. As can be imagined, this requires a relatively complex set of data structures, classes, etc. The *SWGen* library substitutes the simulation context class by an execution context class, which it is basically in charge of declaring and resuming the threads of the software implementation. The rest of the run-time features are provided by the RTOS itself.

The software generation for the communication is done in a channel basis. That is, the SystemC implementation of system-level channels (which can be *HetSC* and SystemC standard channels) is substituted by a functionally equivalent C++ implementation class. This class has the same interface as the SystemC channel class (but without inheriting the *sc_interface* class). The type of implementation depends on the destination of the processes after the HW/SW partition. In terms of software generation there are two possibilities: SW/SW channels and HW/SW channels. The former ones are in charge of communicating threads corresponding to *HetSC* processes which were kept inside the SW partition. Its implementation includes system-calls to inter-thread communication facilities. Then, for instance, an *uc_fifo* channel admits several implementations depending on the inter-thread communication facilities provided by the RTOS. This can be compared to the SystemC implementation, which is based on SystemC event primitives (*sc_event*). The ability to select among different channel implementations and the automation of such selection is a software synthesis issue. That is, the software generation is driven to produce an optimum implementation. What it is optimum depends on specific criteria (consume, speed, size, etc). This is future work of the GIM-UC, out of the scope of ANDRES. HW/SW channels are implementation classes which, as well as using system-calls to the RTOS-API, employ the services of HW dependent software, like drivers or pieces of code directly accessing the memory map. Implementation of I/O channels has the same requirements since it requires access to I/O drivers.

More details about the supported development and target platforms, how to install and use the library, etc, are found in the referred documentation **[FHSV03] [PHF04]** and in **[SWG08].** In the following chapters, how an adaptive process can be specified in *HetSC* and implemented in software is dealt with. As will be seen, it will be closely related to how the *SWGen* methodology generates software. Indeed, the code generated by the *SWGen* methodology is basically the same kind of code which will be shown, plus the C++ wrapping code used for the automatic translation from SystemC.

6. *SWGen* Extensions and Improvements

6.1 *Introduction*

The support of adaptivity at different domains has involved the development and adaptation of certain aspects of the *SWGen* methodology in ANDRES.

Among them, the support os automatic software implementation of synchronous models can be considered the most innovative aspect. Specifically, the way these models are supported in in *SWGen* has been defined, and an actual support for the clocked synchronous MoC (CS MoC) given. This provides support for the software implementation of CS-HAPs. The SW implementation for the CS-MoC has been done for the POSIX API, which has been included in the last release of *SWGen* (*SWGen v1.2*). This has required and enabled the extension of the *Generation Set* of *SWGen*, as will be explained later.

Additionally, other features of *SWGen* have been improved:

- Generation and Publication of a User Manual.
- Improved handling of Makefiles.
- Additional Port to µC/OS-II embedded RTOS API.

6.2 *SWGen Implementation of Clocked Synchronous (CS) Specifications in SystemC*

6.2.1 Clocked Synchronous SystemC supported

Current support of *SWGen* for clocked synchronous (CS) models is based on the support of the specification structure shown by Figure 11.

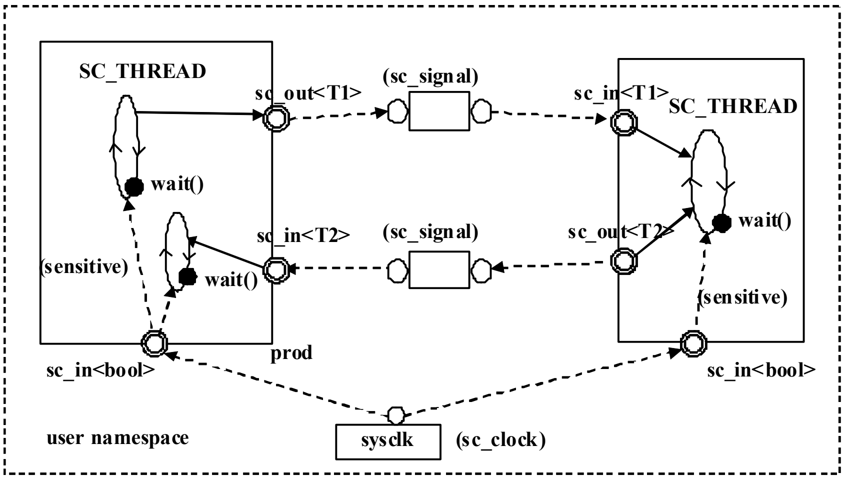


Figure 11. ***SWGen*** **can now generate eSW from a CS SystemC specification.**

Clocked processes are specified by means of SC_THREAD processes, which get blocked on *wait()* statements without any parameter. Such clocked processes have a static sensitivity list, which is associated to a clock input port of *sc_in<bool>* type, and more specifically, to the positive event returned by the *pos()* function associated to the clock input port. The following piece of code shows the basic constructs supported.

```
// module declaration with clocked processes inside
SC_MODULE(m1) {
  sc_in<bool>   in_clk;
  sc_out<T1>    out;
  sc_in<T2>     in;
  void clocked_process() {
     …
    while(true) {
        // statements
         wait();
    }

  }
 SC_CTOR(m1) {
     SC_THREAD(clocked_process);
     sensitive << in_clk.pos();
  }
}
// sc_main
int sc_main(int argc, char *argv[]) {
  …
  // global clock
  sc_clock  sysclk(1.0,SC_SEC);

  // sc_signal communication channels
  sc_signal<T1>  sig1;
  sc_signal<T2>  sig2;
  …
  // binding of sc_signals within the sc_main
  m1.in_clk(sysclk);
  m1.out(sig1);
  m1.in(sig2);
  ...
}
```

Clock input ports are bound to a *sc_clock* instance (*sysclk* in Figure 11). Currently, a single *sc_clock* instance is supported per user namespace. The communication among clocked threads is done by means of *sc_signal* channel instances able to transfer data of any type *T*.

Therefore, as well as the extension of the domains supported for software generation from *HetSC* specifications (now, clocked-synchronous domain adds to untimed domain), the set of specification facilities which have an interpretation as embedded software has been extended. More specifically, the following specification facilities are added to those already supported by the previous release of *SWGen*:

- *wait()* function.
- *sensitive* class.
- *sc_in, sc_out* ports.
- *sc_clock* class.
- *sc_signal* channel class.

Additionally, a software implementation for the *sc_event* class and for its related function *wait(sc_event)* has been provided. The implementation of these constructs is given to the effects of facilitating the structure and general porting of the *SWGen* library, rather than being intended for its support from the system-level. Indeed, *HetSC* methodology does not allow explicit usage of the *sc_event* class, that is, to make notifications and/or waits on *sc_event* instances directly from the process body. In any case, the library will be able produce eSW from specifications with explicit usage of *sc_event* instances (in case it is used out of the boundaries of the *HetSC* methodology).

6.2.2 Implementation fundamentals

In *SWGen*, SC_THREAD processes specifying clocked synchronous processes are mapped to the same eRTOS threads used to map the SC_THREAD processes representing untimed processes. However, a main distinction comes from the availability of a software implementation for the *wait()* statement in clocked synchronous processes. Such statement is implemented as an unconditional blocking statement by means of RTOS facilities, which will let process resumption only by a *software positive event*. The *software positive event* is enclosed and generated by a *software global clock*.

A *software event* is implemented through the *uc_event* class. The *uc_event* class provides an abstract wrapping to the internal eRTOS API calls actually used for the implementation of the *software event* semantic. This class provides a shorted version of the *sc_event* class supporting the immediate *notify()* method. Additionally, an implementation of a *wait* method on the *uc_event* class *wait(uc_event)* is provided. The *uc_event* class and *wait(uc_event)* function provide a direct support for specifications explicitly using the *sc_event* and *wait(sc_event)* statements (thorugh a simple renaming at pre-processing time). However, as mentioned in section 6.2.1, *uc_event* and *wait(uc_event)* are SW implementation facilities mainly oriented to provide support to the rest of SW implementation classes.

The *sc_clock_sw* class is a software implementation class which supports the implementation of the *sc_clock* specification class, used to specify the global clock(s). At the specification (SystemC) level, the *sc_clock* class is a derivation of the *sc_signal* class. However, at the eSW implementation level, the *sc_clock_sw* class is implemented as a *software module* (inheriting the *uc_module* implementation class), which also inherits and implements the *sc_signal_out_if* interface. The *sc_clock_sw* class encloses two *software events*, that is, two instances of the *uc_event* class, and an additional *software thread* declared and instanced through the UC_THREAD macro, which was already used in previous releases of *SWGen* for the porting and support of untimed process implementation as software threads.

The two *software events* of the *sc_clock_sw* class represent the *positive* and *negative* global software events. These events are periodically notified by the internal thread in an alternative and consecutive manner. Following the actual implementation of the inner thread of the *sc_clock_sw* class is shown.

```
void sc_clock_sw::clock_proc() {
  // wait for the initial start time
  wait(start_time_v,start_time_tu);

  while(true) {
    if(clock_value) {
      clock_value = false; //toggle value
      // send positive event to trigger computation
      sw_pos_event.notify();
      wait(posedge_time_v,posedge_time_tu);
      printf("Posedge period of SW clock consumed.\n");
    } else {
      clock_value = true; //toggle value
      // send negative event to trigger SW sc_signal update
      sw_neg_event.notify();
      wait(negedge_time_v,negedge_time_tu);
      printf("Negedge period of SW clock consumed.\n");

    }
}
```

As can be seen, the *sc_clock_sw* class also provides a software implementation for the specification of an initial time and a duty cycle for the software clock.

Such software implementation makes calls to the *wait(double, sc_time_unit)* function. This is a software implementation facility, syntactically similar to its SystemC counterpart, and already present in the *SWGen v1.1* library, which wraps the RTOS calls for delaying (blocking for a given time) the calling process. This implementation had a limit on the resolution of the delay which came from the target platform and its tick timer. Therefore, the resolution of the *sc_clock_sw* class is obviously affected by this resolution. The *sc_clock_sw* class can be instanced once per namespace.

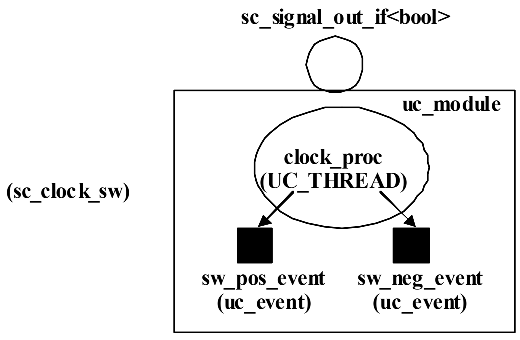


Figure 12. The sc_clock_sw software implementation class.

Since the user software threads implementing SystemC clocked threads are inside *uc_module* instances, the SW thread has to be triggered by a notification traversing a module structure.

Figure 13 reflects the basic scheme to make the association between the clocked process and the software positive event. The UC_THREAD process gets blocked in a *wait()* statement. There it waits on a notification on the software positive event of the software clock, previously associated through a binding between the clok input port and the clock object and a *sensitive* statement associated to the clock input port.

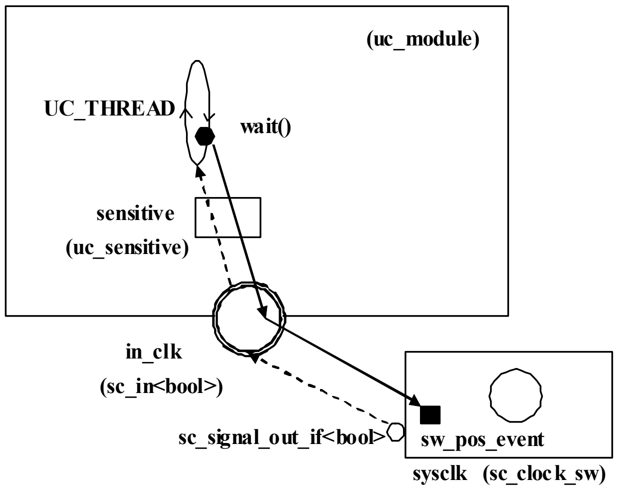


Figure 13. The wait() statement blocks the process till a notification on the software positive event of the software clock happens.

A *sensitive* object of *uc_sensitive* class is instanced within the current *uc_module* instance wrapping the thread. As with the SW implementation of untimed specifications, the *uc_module* class is in charge of keeping the module ambit in the SW implementation, but removing most of the stuff included in the *sc_module* specification class.

The *sensitive* object keeps a reference to an *uc_event* reference enclosed by the port passed to the *sensitive* instance at construction time (specifically, at the construction of the *uc_module* class) through the "<<" operator (implemented by the *uc_sensitive* class). There is only one *sensitive* object per module. Therefore, *SWGen v1.2* currently associates a single triggering event to a module and, thus, to all the clocked software threads within the *uc_module*.

The reference within the clock port is not correctly filled till the binding takes place. *SWGen* currently supports binding of port to signal interface and to a parent port. This also applies for the ports connected to *sc_signal* channels.

SWGen provides an eSW implementation for the *sc_signal* channel class relying on the *sc_signal_SS* class and on the *cs_exec_context* class.

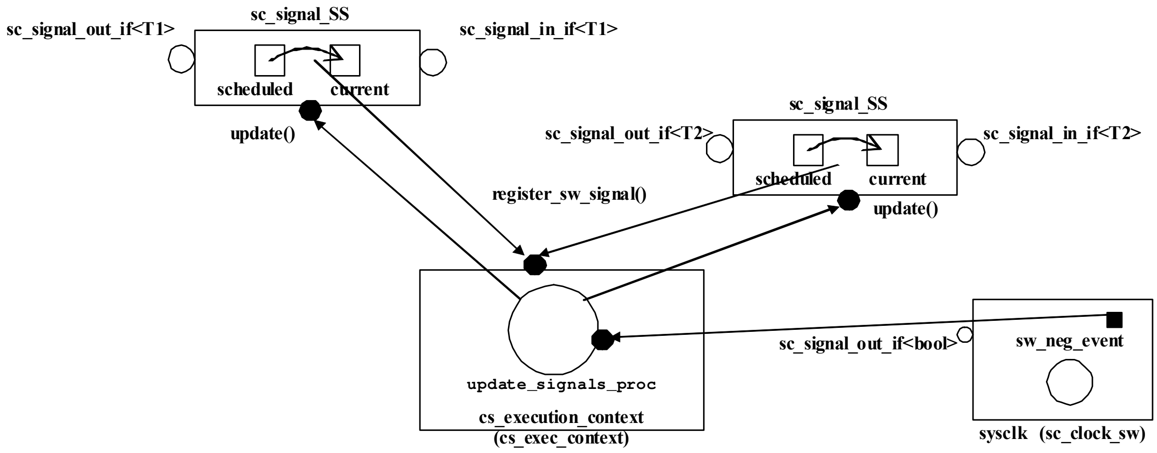


Figure 14. The *sc_signal_SS* class provides a SW implementation for the *sc_signal* channel.

The *sc_signal_SS* class provides an efficient implementation of the SystemC signal interfaces, in the sense that it provides a double value (current and scheduled) semantic without requiring a dedicated software process per channel for updating the current value with the scheduled value. For it, the *SWGen* library instances the *cs_execution_context* global object, of *cs_exec_context* class. This object is in charge of registering every *sc_signal_SS* instance and communicates with the *software clock*. The *cs_execution_context* has an internal process (which, with the software clock process, adds to the mapped clocked processes), called *update_signals_proc*. This process is resumed every time the *software clock* makes a notification to the *software negative event*, and then makes, for each *sc_signal_SS* instance, a call to a stub method called *update*. Such method updates the *current value* with the *scheduled value*.

Summarizing, the *SWGen* CS software implementation alternates signal updates with clocked-process computation. Figure 15 shows how *SWGen* maps the SystemC time (a simulated time) handling of the clocked synchronous specification on the actual time of the eSW implementation. The start of SystemC clock *cycles* is mapped to software *positive* events of the software clock. Software *negative* events are used for software signal updates. Notice that, at sytem-level, such updates where took place at each delta cycle (specifically at its update phase) of the SystemC simulation.

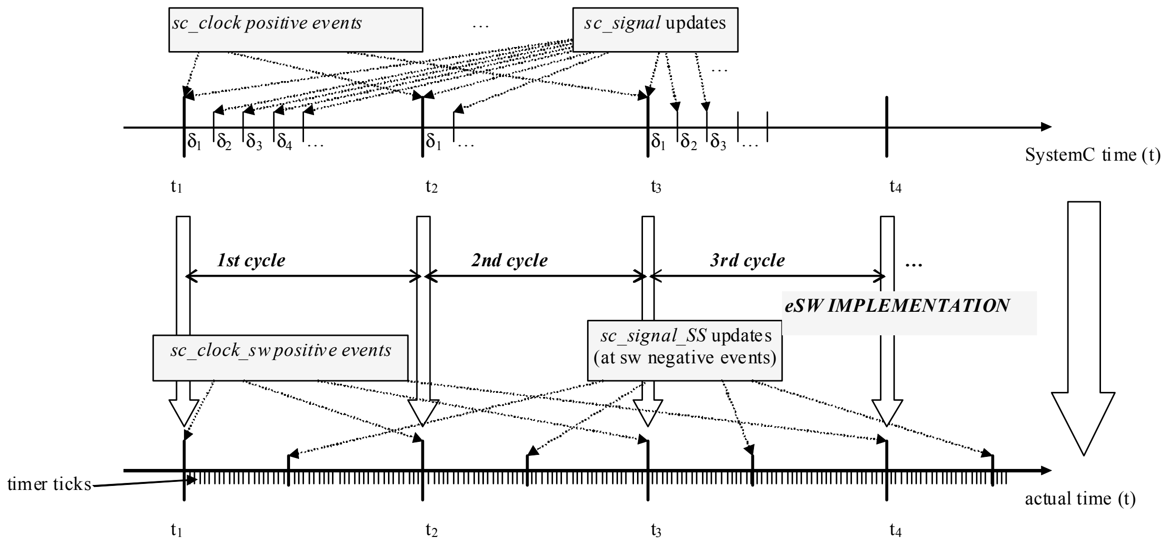


Figure 15. *SWGen mapping* of the Simulated (SystemC) time of a clocked synchronous specification to actual time on its corresponding eSW implementation.

6.2.3 POSIX port

SWGen v1.2 currently provides a port of the CS implementation for the POSIX-API. The structured implementation of the *SWGen* library, and in particular, of its CS extension, explained in section 6.2.2, lets comprise the RTOS-API specific calls in few functions. This facilitates the porting for any embedded RTOS-API. Moreover, a part of the required porting for the CS MoC is common to the untimed MoC support. For instance, the UC_THREAD macro and the *wait(double, sc_time_unit)* macros are required for untimed MoCs and their porting is explained in other *SWGen* related documentation **[FHSV03] [Herr09]**.

The new software implementation facilities involving RTOS-API specific system calls (*syscalls*) for the support of the CS MoC are actually the *uc_event* class and the related *wait(uc_event)* function. The *wait* implementation must let block the process, whose resumption must be conditioned to a notification. In time, the *uc_event* notification must be broadcasted to every pending process. That is, it must support the triggering of two or more processes waiting on the same software event. The solution provided for the POSIX port is based on a POSIX mutex and a POSIX condition variable. The *uc_event* class just declares and instances a mutex and a condition variable. Then, the immediate notification of the *uc_event* class is implemented as follows:

```
void uc_event::notify() {
 pthread_mutex_lock(&event_mutex);
 //printf("SWGen: Notification from SW event.\n");
 pthread_cond_broadcast(&event_cond);
 pthread_mutex_unlock(&event_mutex);
}
```

In POSIX, the calling process can use the *pthread_cond_broadcast* syscall to resume several processes. A POSIX thread waiting on the *uc_event* notification made the *wait(uc_event)* call, which encloses the the following “symmetric” code:

```
void wait(uc_event &e) {
 pthread_mutex_lock(&(e.event_mutex));
 pthread_cond_wait(&(e.event_cond), &(e.event_mutex));
 //printf("SWGen: Notification in SW event received.\n");
 pthread_mutex_unlock(&(e.event_mutex));
}
```

Other implementation could be likely possible. For instance, one immediate one could be based on POSIX signals and their related handlers. Such an implementation has been considered to be likely less stable. However, future work could study such alternative implementations and give more details to deduce if a specific solution is optimum in any case or a software synthesis process should decide the optimum solution for each target.

6.2.4 Contribution in the field of eSW generation from CS specifications

Several works have faced the implementation of synchronous software. For it, synchronous languages like *Signal, Lustre or Esterel* **[BoSi91]**, and optimized compilers able to check the tight conditions of the synchronous models coded have been developed.

A main difference of this work with respect to those works is that it starts from a system-level specification. The starting point is not only an abstract software description, which could be made in any of the previously mentioned synchronous languages, but a system-level model oriented to a HW/SW implementation. A main feature of the system-level *HetSC* model described in section 6.2.1 is that it has an immediate eSW implementation, while its HW refinement is close to the SystemC synthesizable models defined in **[OSCI05]**. Checks related to this synchronous domain are left for the system-level design activities like profiling and DSE, which the system-level SystemC specification enable, instead being a responsibility of the *SWGen* library, which focus on efficient implementation.

More specifically, other novelty of this work is related to the software generation from a synchronous model written in SystemC. Some works have analized the similarities between SystemC and synchronous languages **[BrKl07]**. However, up to now, only a close related work, developed by Virginia Tech (VT) has been found **[SAB02][Sir02]**. The following paragraphs remark the distinctions and contributions with regard to this work.

VT approach **[SAB02][Sir02]** supports embedded software generation from SystemC clocked synchronous models. In this work, C and C++ implementations of the SystemC clocked synchronous specification was provided by means of a substitution of the SystemC host native kernel by an ad-hoc scheduler, targeted to the implementation platform and in charge of implementing the evaluation-update semantic of the *sc_signal* channels.

However, there are main differences between the VT approach and the approach proposed here. First of all, the CS eSW implementation of *SWGen* is just a part of an overall methodology which aims the eSW implementation of a heterogeneous specification, including several domains (i.e untimed and synchronous domains, which in time can include several MoCs). Indeed, previous *SWGen* releases have cover first the implementation of untimed SystemC specifications, a common and widespread modelling paradigm in concurrent software. With this work, *SWGen* is extended to cover clocked synchronous models, another important part of the synchronous approach to software (especially in reactive and real-time applications). However, VT approach is bound to the clocked-synchronous domain, without covering software implementation of untimed specification in SystemC.

Another difference of the *SWGen* approach to CS domain, with regard to the VT approach, is that it relies on the services of an embedded RTOS for solving the implementation issues which cannot be covered by the C/C++ language, as it happened for the untimed specifications (Figure 16a).

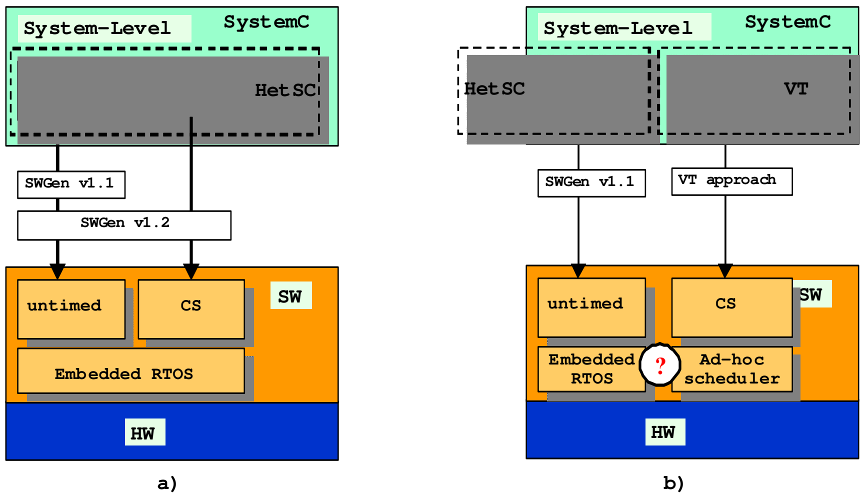


Figure 16. *SWGen* implementation of Clocked-Synchronous SystemC specifications relies on an eRTOS (a) instead of on a methodology-specific scheduler (b).

For instance, in the implementation of untimed MoCs, the concurrency and synchronization services of the eRTOS were used for implementing process networks specified in SystemC under the rules of the *HetSC* methodology. Moreover, software implementation of timed synchronization was also based on RTOS system calls. Now, in order to provide an eSW implementation for CS specifications, this approach is kept. The services required for implementing a clocked synchronization/trigger of threads and a signal-based communication semantic can use again the services of the eRTOS, instead relaying on an ad-hoc scheduler, as it happens in the VT approach (Figure 16b). There were advantages in relying on an eRTOS, like relaying on the own robustness and support of commercial and/or well-known eRTOS and the possibility of exchanging the RTOS for different implementation targets while generating from the same SystemC-based specification.

Therefore, in the *SWGen* approach, shown in Figure 16a, the implementation of multidomain specifications, composed of untimed and synchronous parts, rely on the same basis of a single scheduler implemented by the embedded RTOS, which eases the coherence of this link. This is contrasted with the possibility of merging untimed targeting supported by *SWGen*1.1 library with the VT approach (Figure 16b).

Also regarding this unifying line, the *SWGen* approach and the VT approach are distinct in several syntactical issues. A first one is that *SWGen* approach to CS is currently based on the *SC_THREAD* and the *wait()* statement (while VT approach uses the *SC_CTHREAD*). The *SWGen* approach has the advantage of ensuring the support of the same type and general type of process (*SC_THREAD*) which is suitable and can be used for both types of specifications, untimed and synchronous. Previous releases of *SWGen* discarded the *SC_CTHREAD* since it was not the most suitable and immediate type of process for the support of untimed software models. Therefore, by guaranteeing the usage of the same type of process, the user has a notion or more homogeneity in the usage of the SystemC language and of its related implementation, while being able to specify parts under different domains[1].

Another difference with respect to the VT approach has to do with the communication among processes. Figure 17 shows that the VT approach relies on the usage of *sc_signal* channels for synchronization and shared variables for data transfer.

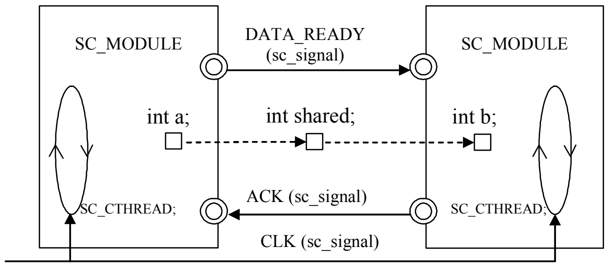


Figure 17. In the VT CS specification Communication is solved through sc_signal channels and shared variables.

In contrast, the *HetSC* specification methodology assumes that both, synchronization and data transfer are both intrinsic parts of the communication, and are responsibilities reserved for the SystemC channels. In the CS *HetSC* specification, such channel is the *sc_signal*. The specification structure proposed in section 6.2.1 holds, therefore, a basic general rule of the General Specification Methodology of *HetSC* **[HUM08]**, consisting in explicitly making communications by means of channels, and only through channels. This keeps the coherence and some homogeneity on the specification methodology.

6.3 *Other SW extensions*

6.3.1 Extension of *SWGen* for eSW generation of eSW Synchronous Reactive Models

This section explains the *SWGen* extension proposed for the implementation of the SR domain. However, a *SWGen* port (i.e. POSIX) has not been implemented yet.

The *HetSC* documentation, available in **[HUM08]**, specifically in the Annexe B.5, defines the rules and guidelines for specification of SystemC models under the SR domain. These guidelines describe how to specify reactive chains composed of generator processes (GP) and reactive processes (RP). Figure 18 provides a *HetSC* graphical representation of a reactive

[1] Additionally, the deprecation of clocked processes (SC_CTHREAD) has been under discussion during the last years, being other motivation for the usage of SC_THREADs in SWGen. Up to date although the SC_CTHREAD appears in the SystemC LRM **[IEEE06]**, some of its related constructs have been deprecated.

chain. Such specification is based on SC_THREAD processes and the *uc_SR HetSC* channel instance, which keeps some semantic similarities with the standard *sc_buffer* channel.

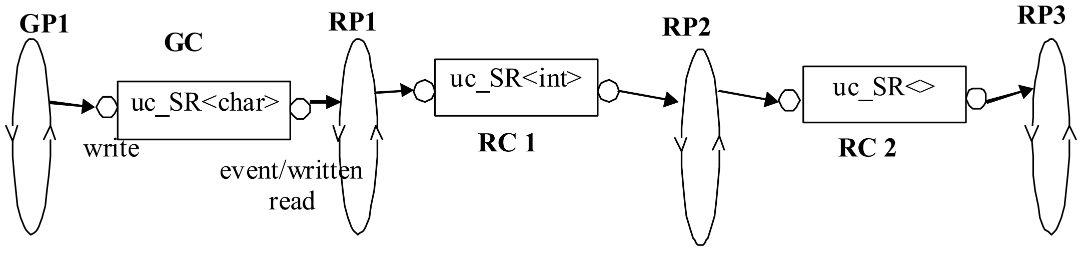


Figure 18. Reactive chain in a *HetSC* SR specification.

As can be appreciated, a SR *HetSC* specification does not explicitly specifies a clock, but actually any *uc_SR* channel instance is able to trigger any reactive process associated to it. This triggering happens whenever an *uc_SR* channel instance is written. Reactive processes (RP) can be triggered by one or more *uc_SR* channel instances and a single *uc_SR* channel instance can trigger several reactive processes (thus several reactive chains). The SR SystemC specification handled the concept of simultaneity by means of time stamps and delta cycles. All the computation on the reactive chain is considered to be synchronous whenever it takes place within the same time stamp, however, delta cycles let apply a causal reaction among the processes which compose the reactive chain.

The *SWGen* implementation is based on a *global implicit software clock*. The period of this clock is defined as configuration parameter of the *SWGen* library. The *global implicit software clock* is implicit because it has no kind of direct correspondence with a SystemC or *HetSC* specification facility. Indeed, the SR model of *HetSC* handles no kind of explicit clock. The period of the global implicit software clock fixes the resolution which the software implementation uses to consider two write accesses of a *uc_SR* channel instance simultaneous (that is, synchronous). Equivalently, it determines the steps for computing reactive computations. Indeed, reactive processes (RPs) are implemented as software threads which can block on *wait(uc_event)* statements, in a similar way as it was done with the CS software implementation. However, these *wait(uc_event)* statements of the RPs are sensitive to the global implicit software event. Thus, as can be seen, the software implicit clock defines a set of implicit cycles which somehow fulfil the role of the delta cycle in the SystemC implementation of SR specifications in *HetSC*. Because of that, they are named software *deltas* (sw-δ). These mapping of SystemC time handling of SR specifications on actual time of the software implementation has been represented in Figure 19. In the *SWGen* implementation, the sensitivity list of the RPs fixes the set of *uc_SR_SS* instances which have to be checked for determining which functionality of the reactive process has to be computed (as appendix B.5 fixes, the functionality depends on the trigger combination).

The *HetSC* generator processes (GPs) of the SR model can and have to call wait on time statements in order to separate and fix the specific slots (identified by specific SystemC time stamps). Indeed, GPs are the autonomous computations which fix when slots are given within the simulated SystemC time. The software implementation of these generator processes is already supported by *SWGen*, since GPs are implemented as “untimed” software threads and the wait-on-time statements already had a basic support on the *SWGen* library.

Notice that, as Figure 19 represents, the software delta is in general an integer multiple of the tick timer and it is periodic. However, the slots do not have to happen as times being an integer multiple of the global implicit software clock period (that is, of the software delta period). In any case, the *SWGen* implementation (as any other feasible implementation) provides an actual time dimension to the reactive computation. Therefore, it is necessary to guarantee that the number of software delta cycles needed to stabilize the computation fit within the actual time between consecutive slots. In this sense, the *HetSC* library provides the activity analysis, which provides data about the number of delta reactions which a reactive chain can take before stabilization. In the software implementation there will be a trade-off in the selection the sw-δ. A bigger resolution (lower sw-δ period) can admit more software deltas and more chance to get the stabilization for a given reaction. However, it requires shorter computation times to the processes of the reactive chain and can lead to an inefficient figure of context changes.

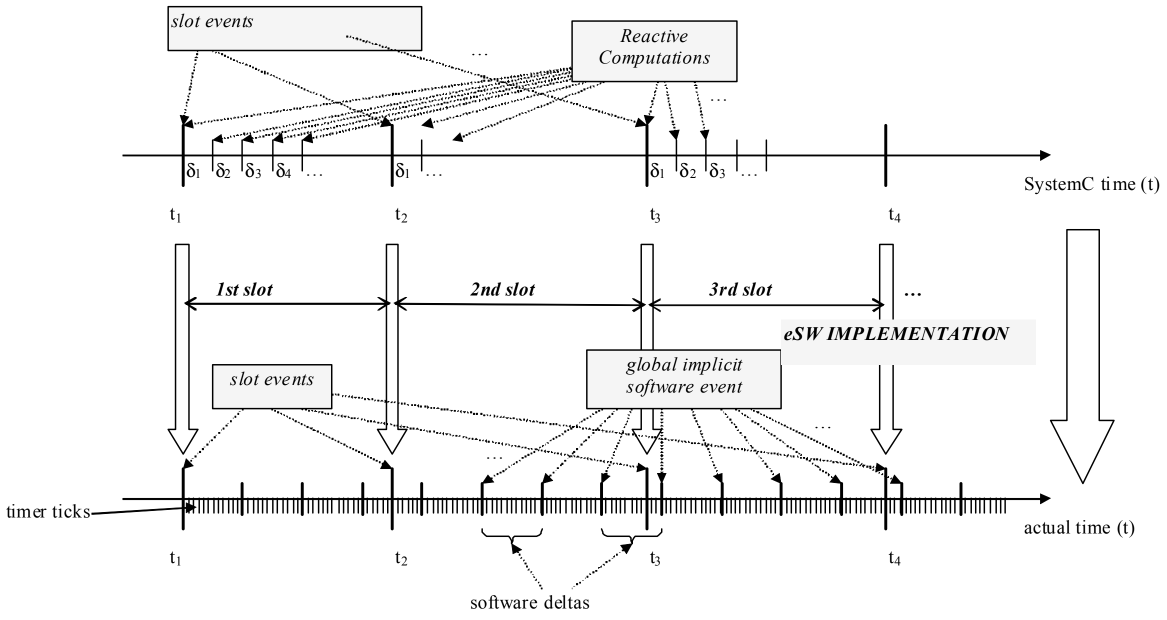


Figure 19. *SWGen* mapping of the Simulated (SystemC) time of a synchronous reactive specification to actual time on its corresponding eSW implementation.

6.3.2 Generation and Publication of a User Manual

A first version of the *SWGen* user manual will be now available on the *SWGen* website **[SWG08]**. The main points included by this manual are:

- Fundamentals and main purpose of the *SWGen* methodology.
- Installation instructions the *SWGen* library.
- Instructions to run the available examples.
- Explain the current *SWGen* generation subset.
- Instructions and clues to setup some of the supported target platforms on the native host machine.

6.3.3 Additional Port to µC/OS-II embedded RTOS API.

The last release of *SWGen* has been provoded with a port (currently covering untimed domain) for the *µC/OS-II* **[Lab02]** API. This demonstrates that *SWGen* technology is able to target a wide range of embedded RTOS (through its related APIS) considering the typical ranges of the size of their footprint. That is, *SWGen* can range from small sized eOS like µC/OS-II to big sized eOS like Embedded Linux distributions (*ELinux*). In the middle, it can target medium sized embedded RTOS like *eCos* **[Mas03]**.

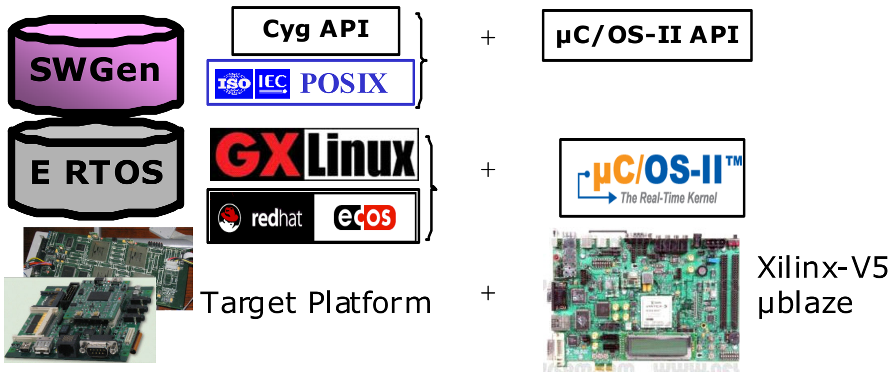


Figure 20. Support of a theµC/OS-II RTOS API has been added to the *SWGen* library.

Another advantage of the port to the *µC/OS-II* API is that it facilitates the connection with the other implementation technologies involved in ANDRES and let fit the application of the *SWGen* technology in the development environments closer to the needs of the industrial partners. This should in time facilitate the development of the ANDRES industrial demonstrators. Figure 21 shows a scheme sent to one of the industrial partners (DS2) about the role that the *SWGen* library can play in the development of their industrial demonstrator.

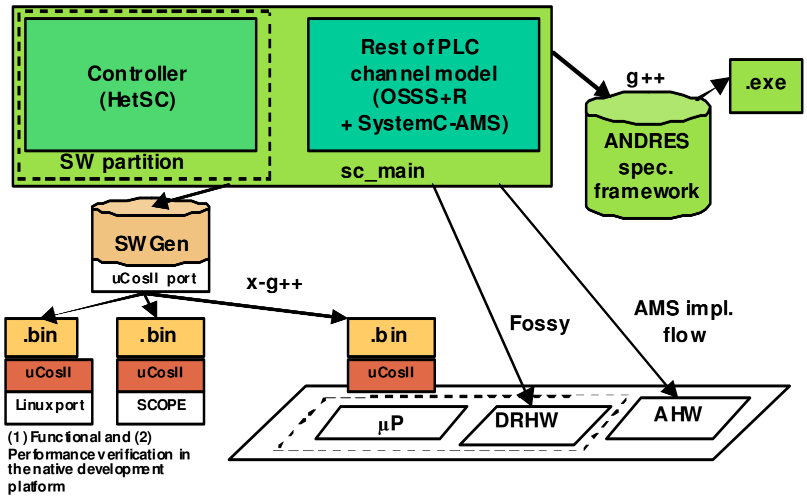


Figure 21. *SWGen* flow in the context of the DS2 use case in ANDRES.

The µC/OS-II RTOS is familiar in their development environment. Additionally, µC/OS-II counts with a Xilinx-microblaze port which can be easily targeted to a Xilinx-based FPGA with dynamic partial reconfiguration capabilities, i.e. as the Virtex V–based platform represented in Figure 20 and sketched in Figure 21 with dashed lines. This has the additional advantage that such platforms are targets currently supported by the OFFIS *Fossy* tool **[D24B]**, able to target DRHW from OSSS+R code.

Figure 21 also shows some of the possibilities enabled by *SWGen* for the progressive validation of the generated software before aiming the target platform. Currently, there are µCos-II ports for Linux targets which let check µCos-II on host machines with a Linux installation. An issue is that this port is for 32 bits, however, some industrial partners currently use 64bits clusters. In order to solve it, *SWGen* makefiles for µCOS-II-Linux target are provided and the documentation provides now guidelines to adapt, install and use the '*Esslingen port*', a Linux port to µC/OS-II available in **[UCP08]**, for 64bits host machines.

Another verification possibility when the eSW generation is targeted to a POSIX-API is to pass the resulting code to SCoPE **[SCo08]**, a performance analysis tool developed by the GIM-UC.

6.3.4 Improved handling of Makefiles

The handling of Makefiles has been improved. Originally, in the initial version of *SWGen*, a big and complicated *Makefile* was handled. Now in *SWGen v1.2*, *Makefiles* have been split in a platform basis. These means that a single *Makefile* serve to keep specific building details of eSW for a specific HW/SW platform, fixing the RTOS API (POSIX, µCOS-II) and software architecture (like ARM, native, etc).

Then, for instance, in order to configure the *SWGen* library and generate an embedded software application for a target platform based on ARM and the *ELinux* eOS, specific *Makefile.arm-linux* files are available. If the generation is targeted to a platform based on the *µblaze* microprocessor and the µC/OS-II RTOS, then, the library provides specific *Makefile.ublaze-ucosii* files for it. If, for instance, the user wants to do a first check of generation for the µC/OS-II target over the host (Linux-based) native platform, specific *Makefile.host-µcosii* files are available. The *Makefile.host-posix* lets check *SWGen* generation for POSIX targets in any Linux based host platform supporting *pthread* library.

7. Adaptive Software

During the last decades, software programming has evolved to cover more abstract and powerful paradigms. Assembly languages left pass to high-level languages. Since1970s, Structured Programming become the usual way of building software, overcoming problems caused by absolute branching. In the 80s, on Object Orientation (OO), incorporated advanced concepts such as inheritance, overloading and polymorphism, enabling a more productive and abstract approach. In OO Programming was created, software is split up into separate classes that are designed to have minimal interaction between them. It makes easier to reorganize the code when the specification changes; however, each change still requires programmer intervention, and thus, an additional cost of money and time **[AS08]**.

A new approach is Adaptive Programming, which addressed the problem which arises when a piece of software has to be reused for new purposes and/or situations it was not originally designed for. Initially, this could be handled by a static adaptation, which automatically solves at the compilation phase, the code which has to be in charge of achieving the proposed goals. However, if the code is working in a changing environment and/or with changing goals, software has to adapt for yielding such changing goals in an optimum way, without the chance of rewriting the program. Therefore, Adaptive Software is about representing the actions that can be taken, the goals that the user is trying to achieve and the way in which the program automatically manages change, selecting the right actions under goals consideration and everything during the run-time without the intervention of a programmer.. Summarizing, adaptive software uses available information about changes in its environment to improve its behaviour within a running program, which involves dynamic adaptation.

There are several lines of work related to the development of adaptive software, in most of the cases, not totally independent, but, to the contrary, rather complementary.

A first line is the proposal of new software architectures. These works propose new sets of specific software components and define their interaction in a way which makes architecture for adaptation more explicit. These approaches break in some sense with some of the assumptions done in previous approaches, like OO and structured programming. For instance, the adaptive functionality cannot be seen like a *black-box*, as it happens with a *function* or a *class*. That is, encapsulating such adaptive functionality in a *class* or a *function* is not longer sufficient. In an architecture of adaptive software, the different ways/algorithms/solutions which can solve a functionality have to be somehow apparent, as well as the goals, and the criteria to enable the automatic decision of which solution is dynamically adopted.

A specific example can be found in **[CHS01]**, which proposes an architecture for modelling adaptive distributed software. A service of a distributed operative system (or of a middleware layer) is spread among different hosts by means of a set of basic components called *adaptive components* or ACs. Each host of the distributed system will run an AC. Figure 22 shows the internal structure of the AC, which consists of two different types of modules: a *component adaptor module* (CAM) and several *adaptation-aware algorithm modules* (AAMs). Each AAM provides a different algorithm that implements the functionality of the component, while the CAM controls the component's adaptive behaviour. When it perceives that any other AAM is a better fit for the current requirements, it initiates the adaptation, replacing current AAM. In this way, either using an AAM or using another one, the system achieves different functionalities and/or performances for the same AC.

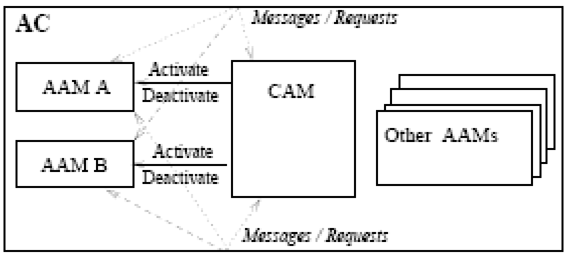


Figure 22. ***Structure of an adaptive component (AC) (taken from*****[CHS01]) .**

There are other proposals of adaptive software architectures. **[SCO02]** proposes an architecture based on *Containment Units* (*CU*). DAS (*dynamic adaptable software*) **[AW98]** is another approach (shown in Figure2) which *describes* a *meta-level* architecture for constructing adaptive software. It is based on *environmental objects* which represent the actual runtime environment. They have different states, each one representing a different set of features of the runtime environment. Each state is, in time, associated to a set of methods. *Event objects* monitor state changes of environmental objects. When they detect the occurrences of the changes, the call to the appropriate methods is done.

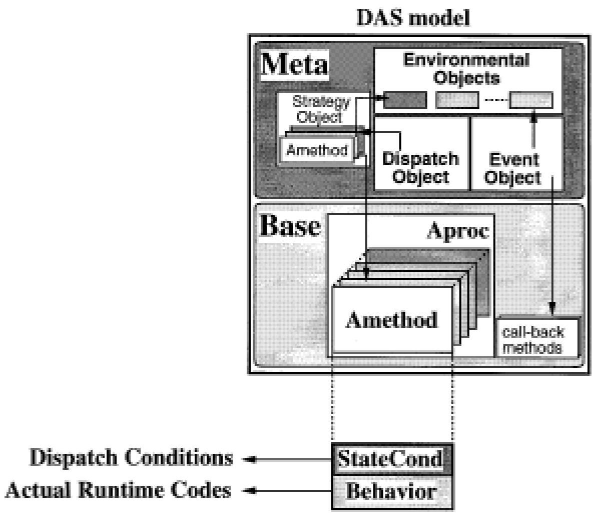


Figure 23. Structure of a DAS model.

Another important line of work on adaptive software is the development of dynamic programming languages. In some cases, these languages are tightly related to the development of the previously mentioned architectures for adaptive software. For example, LEAD++ **[AW98]** is an object-oriented language used to describe the DAS architecture. In **[SCO02]**, *Little-JIL* language **[LJIL09]** is used, enabling the application of static analysis to obtain assurances that the Containment Units can be expected to *demonstrate the robustness for which they were designed.*

In LEAD++ the basic mechanism for dynamic adaptability is called *adaptable procedure.* An adaptable procedure is a special kind of generic function whose internal computation is selected based upon the state of its runtime environment. In **[AW98]**, LEAD++ was implemented as a kind of Java pre-processor composed of a translator and a library of Java classes.

Little-JIL is language that can be used to define the coordination of multiple autonomous agents, describing the control and data flow among those agents and moreover, their usage of available resources during the performance of a task. Such definition is called a *process program*, and provides different ways of achieving the task completion. Each way will have different requirements of resources and agent capabilities. *Little-JIL* enable the definition of Pre- and post-requisites, which are used to dynamically verify that the *process* is being applied correctly, and the modelling of resources, which are reserved and locked during the execution of a process.

There are other languages such as CLOS **[Son88]**, an extension of LISP for Object Orientation, or Dylan **[Dyl09]**. CLOS stands for *Common Lisp Object System*, and it is an extension for OO programming of the LISP language. CLOS is dynamic, which means that both, the contents and the structure of CLOS objects can be modified at runtime. The definition of CLOS classes can be dynamically changed (during run-time), even when there are already running instances of the changed or "adapted" class. Moreover, a special operator (`change-class`) enables the changing of the class membership of a given instance. CLOS also enables the addition, redefinition and removal of methods at runtime. Similarly, in Dylan, a piece of a program which is not *sealed* (that is, explicitly locked to be invariant) can be extended at run time or by additional libraries. Specifically, a running program can add a method to an existing class without accessing to the original source code, or needing its recompilation. Moreover, a library of introspective functions supports the run time examination of objects, including classes and functions. This enables that a Dylan program can indeed debug another Dylan Program.

Agent technology is another key line on the creation of adaptive software. An agent is an autonomous component that can perceive part of the environment and it can react, taking some actions **[Net08]**. Normally, these actions are not the whole functionality the adaptive system is in charge, but they are in charge of triggering the suitable functionality at a given moment. Therefore, agents are incrusted on a scheme of self-adaptive software, where they are light-weighted computation elements, however also the main responsible for sensing the environment, deciding about the suitability of the current behaviour of the system, and about its adaptation if proceeds. As it happened with the relation of software architectures and languages, agents' technology is closely related to these work lines. For instance, a language like Little –JIL, previously mentioned, is based on agents.

8. Adaptive Software in ANDRES

8.1 *Relation to ANDRES and Contribution*

The analysis done in chapter 7 about the relative recent paradigm of adaptive programming, will serve to first extract some conclusions about what must be understood or taken in common when talking about adaptive software, to later fix which the contributions of ANDRES to adaptive software are.

From the analysis of chapter 7, it seems clear that, when talking about adaptive software *run-time adaptation* is self-understood, without need of re-compilation. This is coherent with the abstract concept of adaptivity handled in ANDRES and its implementation, for instance, in the hardware domain, with the parallel concept of dynamic partial reconfiguration.

When talking about adaptive software, it seems also apparent a specific architecture of self-adaptive system, where some blocks should be able to encapsulate alternative solutions (adaptable computations), while other blocks (agents) are in charge of sensing the environment and adapt computations. As will be seen in section 8.2, this is in much coherent with some of the basic abstract blocks/concepts developed in ANDRES, which serve to build such an adaptive architecture.

However, as well as these common points with the existing work lines in a pure software context, this work will present some distinguishing aspects which provide a new perspective for the specification of adaptive software. These aspects are summarized in the following points, to be later explained in more detail:

1. Orientation to AHES (Adaptive Heterogeneous Embedded Systems)
2. Adaptivity modelled at the System-Level, instead directly at the software domain.
3. Adaptivity in SystemC.
4. Automatic generation of Application-Level Adaptive Software.
5. Under a Formal Framework supporting Heterogeneity and Adaptivity.

A first main distinction is that the work done here is oriented to the system-level specification of Adaptive HW/SW Embedded Systems, equivalently, Adaptive Heterogeneous Embedded Systems (AHES). This contrasts with works reminded in chapter 7, which move more in a pure software domain, independently whether such works address software as complex and needed of computation resources as the distributed software of **[CHS01].**

This orientation to AHES, has direct involvements on the abstraction-level which is considered in the capture of adaptive objects and structures. The design methodologies of embedded systems have moved towards what is called system-level specification, instead of facing a direct capture of the software program. In this line, the structures for adaptivity proposed here are handled at the system-level, thus placed at a more abstract plane than the software domain. For instance, the adaptive specification structures dealt here will not make any assumption or let any dealing of scheduling policies, since this is usually considered a software implementation detail. In one hand this can be perceived as a disadvantage by a software programmer, which looses the capability of, for instance, use or assume a given scheduling policy to guarantee that a given adaptation will take place at certain moment or at a given rate. However, from the perspective of embedded system specification, this has the necessary advantage of enabling the production of a model which can be more easily mapped to both software and hardware. This is reflected in Figure 24.

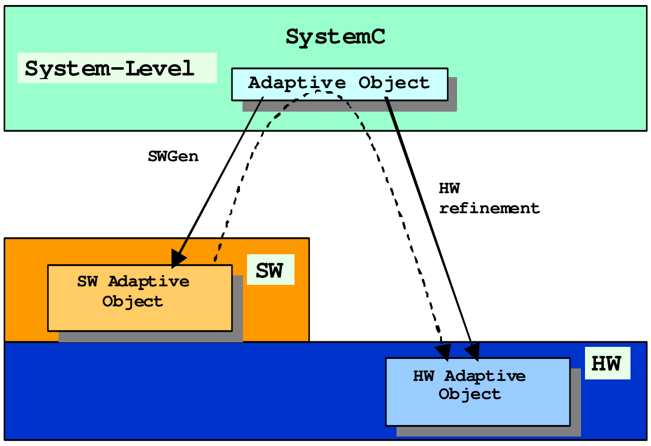


Figure 24. This work addresses the specification of SW Adaptive Objects in a system-level in SystemC.

In case a concurrent specification of the adaptive system where directly code in software assuming a given scheduling policy, then if the design process decides to move it to hardware, it is necessary to remove from the software specification such scheduling information (which can be interpreted as moving up towards the system-level specification detail), to later go down again towards the hardware domain, which usually need to work with real concurrency, thus without assuming a specific sequencing or order fixed by a scheduling policy. This has been represented with the dashed line. However, if the adaptive specification facilities are captured at a system-level, not only system-level verification is possible, but the implementation flows to SW and HW only require the addition of the information and details proper of each domain. This has been represented with the continuous lines.

Again, in a close relationship with the previous points, the work done here is focused on SystemC, which contrasts with the usage of any of the Dynamic Programming Languages mentioned in chapter 7. Being a C++ extension, SystemC does not provide the advanced features which are already inherent to Dynamic Programming Languages. However, SystemC is a more suitable language for the specification of embedded systems for several reasons: like being a system-level language; having a syntax closer to embedded system designers; being an standard language; and counting with a set of extensions (TLM, SystemC-AMS, OSSS+R, *HetSC*, etc) for supporting the different domains involved in the design of an embedded systems. This makes interesting exploring possibilities, as this work does, and eventually the limits of SystemC for specifying adaptivity, and more specifically, adaptive software.

This approach requires and supports (by means of *SWGen*) an automatic software implementation flow directly from the system-level adaptive specification facilities, while such specification facilities still let further and progressive refinement to other domains different from software, like digital or analog hardware. The generated software should be situated at an *application-level* (see Figure 25), which contrasts with some works reviewed in chapter 7, like **[CHS01]**, which addresses the modelling of adaptive architectures at the Middleware and at the RTOS levels.

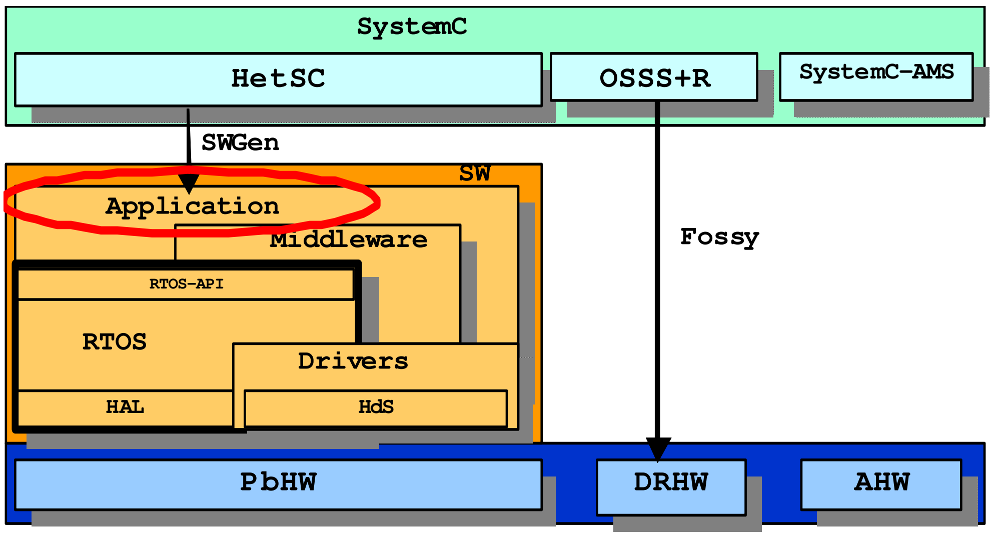


Figure 25. This work focuses on the generation of adaptive software at the application level.

Last, but not least, this approach is based in a metamodel (ForSyDe), which provides a formal framework for supporting both heterogeneity, as well as concepts for Adaptivity. ForSyDe supports heterogeneity by metamodelling up to four basic models of computation (untimed synchronous, discrete and continuous). This is necessary for removing ambiguities in the interactions among the different parts which compose the specification of a whole actual embedded system, which very likely incorporate software, digital and analog hardware. Additionally, ForSyDe has been extended in ANDRES for supporting adaptivity. This extension is briefly reviewed in the next section for the reader convenience, while more details can be found in **[D11A]**. More ForSyDe documentation about support of heterogeneity is available in **[Jan05] [SaJa04]**. As will be seen, the proposal done here is based on these formal concepts for different computation domains and types of adaptation possibilities comprised by the ForSyDe metamodel.

8.2 *Formalization of Adaptivity: Adaptive Processes*

In **[D11A]** (section 1.5), the general term *Abstract Adaptive Object* (AAO) reflected the notion of an object with some kind of adaptivity capability, where its adaptation mechanism is undefined. The different kind of adaptive capabilities and of adaptation mechanisms can define then a wide set of *Adaptive Objects* (AO).

From this general perception of the AAO, the chapter 5 of **[D11A]** develops a more specific concept, the *Adaptive Process* (AP), which extends ForSyDe for modelling of adaptivity. The AP presents an adaptation signal, additional to the rest of input signals, to change the behaviour of the *Adaptive Process*.

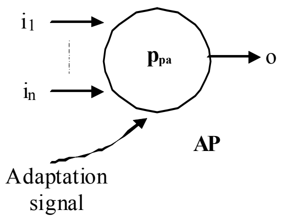


Figure 26. ForSyDe Adaptive Process.

From this general concept, **[D11A]** defines four types of APs, mostly distinguished by the information transferred through the adaptation signal, that is *what* is adapted. Since it is convenient for the rest of the discussion, in *HetSC* these metamodelling concepts are shortly named *parameter-based Adaptive Process* (paAP), *mode-based Adaptive Process* (mAP), *function-based Adaptive Process* (fAP) and *process-based Adaptive Process* (prAP).

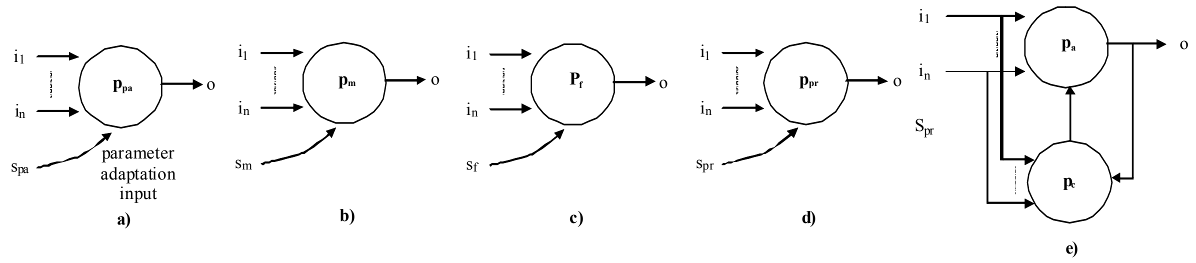


Figure 27. Types of ForSyDe Adaptive Processes.

A fifth category of AP was introduced in **[D11A]**, the *self-adaptive Process* (sAP). The sAP approximates the idea of adaptivity in the software world. In some sense, process p_c of Figure 27 reflects the software *agent*, which, as explained in chapter 7, is an autonomous entity which decides the adaptation the AP p_a must carry out. In the ForSyDe case, the *agent process* p_c, samples the inputs and the outputs delivered by AP. Moreover, in a software context, p_a and p_c inputs can be independent, since *agent* can sample environment inputs which differ from the inputs of the adapted software.

In any case, ForSyDe metamodel is able to capture the sAP as a process network composed of any of the four basic types of APs and the *agent* process. Therefore, the knowledge of how to model those four types of APs in SystemC, and how to implement them as embedded software (eSW), is a basic need which is addressed in the following chapters.

Finally, another important consideration is the typology of APs of Figure 27 is orthogonal to the fact that ForSyDe covers several domains (or MoCs), namely untimed, synchronous, discrete time, and continuous (recently incorporated in ANDRES). Regarding to this work, the former two domains are considered to have some impact in software modelling.

9. Software Implementation of Adaptive Processes

9.1 *Fundamentals*

In this chapter gives an idea about how adaptive processes could be directly described in an embedded software application. It is a sketch of how an eSW programmer could manually capture the formal APs defined by ForSyDe.

Such software implementation of the AP depends on the software development platform, characterized by the implementation language, type of RTOS API (if there is RTOS), etc. For instance, if a concurrent specification is finally implemented in C, without RTOS and without support of concurrency, each AP of the specification has to become a sequential portion of code.

Specifically, in this chapter, it will be assumed that the platform and its related eSW development kit (SDK) includes a C/C++ cross-compiler, and an embedded RTOS with support of multithreading and a generic C-API. This will make easier to later on understand that the eSW generated by *SWGen* from the *HetSC* APs explained in chapter 10 will correspond with the eSW implementation based on the formal ForSyDe APs, introduced in this chapter.

The eSW implementation of the AP also depends on the MoC employed in the specification too. In this chapter, it will be first shown for untimed MoCs since software is often modelled through these MoCs.

Under these assumptions, it can be reasonably assumed that the Adaptive Process will be implemented as a *static thread* (since the AP in ForSyDe is static). By *static* thread is understood that the thread is created once, before the system starts to run, and remains till the end of execution. This involves a conceptualization of two phases in the execution of the software implementation which is not rare in many applications. In an initialization phase (which would correspond to the elaboration phase of the SystemC simulation) threads, among other data structures, are declared and instanced. Later on, threads are executed (which would correspond to the simulation phase of the SystemC simulation).

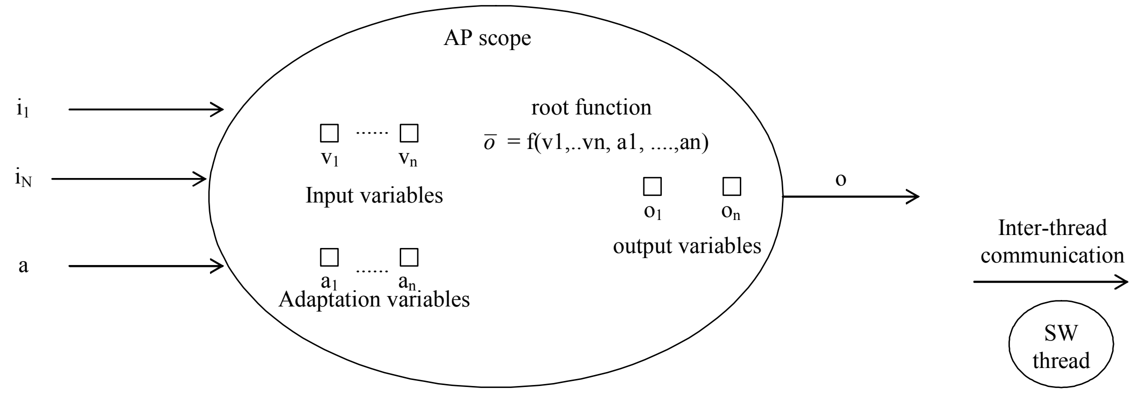


Figure 28. SW implementation of an adaptive process

The thread has an associated C/C++ function, called *root function* in Figure 28. The root function contains C/C++ sequential code which deterministically relates the input data with the output data. Internally, the root function can call other C/C++ functions. The *root function* has neither input nor output parameters and it has a biunivoque association to the *static thread*. The *root function* has, in general, an inner infinite loop (*root loop*) which is in charge of making the process to infinitely compute. The thread communicates with other threads through inter-thread communications system calls, which take the role of input and outputs of the adaptive process implementation. Additionally, there can be a set of related input and output variables whenever a local storage of input data and/or output data is needed. For instance, an input variable is necessary when a token is read more than once in the evaluation of an output token and the read system call is consuming. An input variable is also necessary when it is going to be read in a later evaluation (in this case, it becomes a state variable). Input and output variables are not always necessary and can be overridden. Moreover, in some cases, the output system call can take as a parameter and input system call. However, a functionally equivalent implementation with input and output variables is always possible. Because of this, and for discussion simplicity, input and output variables will be assumed.

The process computation (evaluation) is, in general, a finite set of functional relationships established between each computed output (transferred out of the adaptive process through an output inter-thread communication system call) and its related inputs (read by the adaptive process through input inter-thread communication system calls).

The AP can have a state. For the discussion, it is reasonable to assume that in its SW implementation the state variables will be part of the thread scope. That is, that these variables are local to the *root function*. This approach it is not always practical. A reason is that sometimes state variables are too big to be part of the local scope of the root function, due to stack limits. However, the simplification done is still valid for the analysis if the concept of thread scope is generalized to those global variables, accessed only by one thread, (left out of its local scope due to the size problem). Other state variables are *static* variables local to the root function or to the functions called by the roof function.

The peculiarity which determines this scheme as an implementation of the adaptive process is that, at least one of its input inter-thread communications is considered as *adaptation input* (a), as depicted in Figure 28. If there are input variables associated to this input, they are considered as adaptation variables. This is actually just a useful conceptualization, which will enable a clear identification and distinction of the software implementation of the different types of APs.

The evaluation of the adaptive functionality is given at each output variable evaluation of the *root loop* which is transferred to its correspondent output inter-thread system calls. The adaptation is effective once the *adaptation input* values update the *adaptation variables*.

“When” the adaptation can take place is a matter closely related to the MoC. In this document the examples focus on untimed MoCs. For the software implementation of these MoCs a blocking semantic for the input and output inter-thread system calls is assumed. Then, one adaptation cannot be done while the adaptation input system-call has not received all the adaptation data. In addition, the adaptation also requires the consumption of a certain amount of tokens from the rest of inputs. This implementation scheme keeps coherence with the ForSyDe untimed MoC, where each adaptation cycle can be assimilated as a ForSyDe evaluation cycle. In this implementation, each evaluation cycle requires the consumption of new tokens at the adaptation input.

The following sections show how each type of adaptive process defined in **[D11A]** is implemented in software. In a practical code, several hybrid situations, combining process-based, function-based, mode-base and parameter-base adaptation and self-adaptive adaptation can be present. Nevertheless, identifying each type of adaptivity in SW and stating modelling guides and, eventually, modelling facilities able to capture this kind of constructs is convenient to facilitate the specification and implementation of adaptivity in software.

9.2 *Parameter-based Adaptive Process in Software*

The first type of AP defined in **[D11A]** is the parameter-based AP (paAP). The implementation of the paAP details the SW implementation model given in Figure 28.

The adaptation is done through parameterization. The adaptation variables are a list of internal *parameter variables*. The values read from the *parameter input* update the *parameter variables* (p_i). These parameter variables affect the input-output relationship of the *root function*. The list of variables is of the same type T_p, and matches the type of parameters input. There are no restrictions for T_p, that can be an abstract complex data structure.

The adaptation is effective once the *adaptation input* is completely read and have updated the internal parameter variables. The *root function* reflects a fixed functional relationship between input and output variables once the parameters are fixed.

Lets see an example of SW implementation of a *paAP*. It consists in an adaptive first order IIR low pass filter. This filter has the next transfer function: $H(z) = k_1/ (1 + k_2 \cdot z)$. Its adaptivity consists in which it is possible to change the gain parameter (k_1) and the pole position (k_2). The ForSyDe model is given on the left hand side of Figure 29. On the right hand side the "refinement" of the Adaptive Process with untimed process constructors is given.

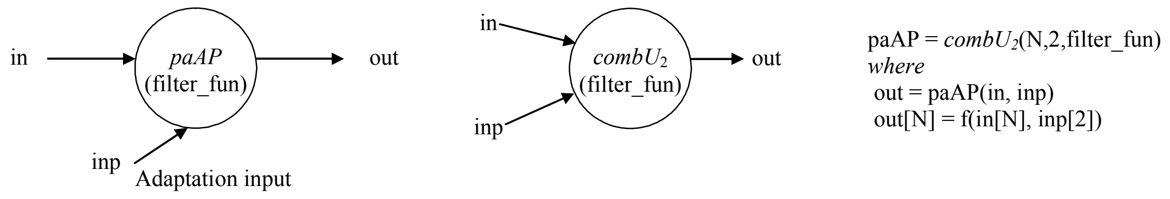


Figure 29. ForSyDe model of the adaptive LPF filter.

Following, an example of a C++ implementation is shown:

```
template<class T, unsigned int N>
class adaptiveLPF : public adaptiveLPF_if<T>  {
public:
  adaptiveLPF();
  void in_param (T&);  // filter parameterization input
  void in(T&);           // samples  input
  void out(T&);          // samples output
private:
   void root_function();
   ...
};
```

The adaptive process is implemented as a thread encapsulated by a class, called *adaptiveLPF*. The interface of the *adaptiveLPF* class is an interface class, *adaptiveLPF_if*, which declares three functions, one for writing input samples, one for reading output samples, and one for the adaptation input. In this case, all of them are of the same generic type (*T)* since the interface and the adaptive class both are actually a C++ class template, but they could be different in other examples.

```
template<class T>
class adaptiveLPF_if {
public:
   virtual void in_param(T&)=0;  // parameters adaptation input
   virtual void in(T&)=0;             // regular input
   virtual void out(T&)=0;          // output
};
```

This interface is the visible part of the c*lass* implementing the paAP. This exemplifies a clean C++ object oriented (OO) software implementation for the Parameter-based Adaptive Process. A C++ programmer can conceive it as a parameter based adaptive and active object.

To understand how the *paAP* is implemented, the insights of the *adaptiveLPF* class are shown. Firstly, the *adaptiveLPF* class constructor is shown:

```
 adaptiveLPF ::adaptiveLPF() {
    …
   // declare the root function as a process
   rtos_api_declare_thread(thandler, convert_to_fp(adaptiveLPF ::root_function), ...);
   // declare message boxes for inputs and outputs
   rtos_api_declare_mb(mbinp, sizeof(float)*2,BLCK,...);
   rtos_api_declare_mb(mbin,sizeof(float)*N,BLCK, ...);
   rtos_api_declare_mb(mbout, sizeof(float)*N,BLCK,...);
   …
}
```

This normally requires some additional declarations in the private part of the adaptive class:

```
template<class T, unsigned int N>
class adaptiveLPF : public adaptiveLPF_if<T>  {
public:
      …
private:
      …
    rtos_api_thread_handler  thandler;        // handlers for the inner process
    rtos_api_mb_handler  mbinp, mbin, mbout; // handlers for message box
}
```

In light red, systems calls to a hypothetical C-based RTOS-API have been used. A system call (*rtos_api_declare_thread*) is in charge of declaring the root function as a static thread. The *convert_to_fp* would be a macro in charge of converting a method pointer to a function pointer. System calls for thread declaration usually take a function pointer instead of a class method pointer as input parameter. The rest of system-calls correspond to the declaration of three message boxes with blocking protocol. The message box is an inter-thread communication service usually available in many RTOS. In case the RTOS would not provide it, other inter-thread communication services could be used, i.e., mutexes. In the example, the message boxes are in charge of buffering the received samples and the output samples. A message box is also available for the adaptation input. The adaptation input is not different from other inputs in terms of synchronization. This implementation could correspond to the SW implementation of a *HetSC* specification under an untimed PN MoC.

Following, the implementation of the root function is shown:

```
void adaptiveLPF ::root_function() {
  T k1, k2;
  T input_var[N], output_var[N];
  unsigned int i;
  while(true) {
     rtos_api_read_mb(mbinp,&k1);                          // parameterization
     rtos_api_read_mb(mbinp,&k2);
     for(i=0;i<N;i++)  rtos_api_read_mb(mbin,&input[i]);      // input samples
    filter_fun(k1,k2,input_var,output_var,N);                    // evaluation
    for(i=0;i<N;i++)  rtos_api_write_mb(mbout,&input[i]);    // output samples
  }
}
```

Notice how the system calls for reading from the message box and the system calls for writing the output message boxes are employed as input and outputs of the static thread which implements the adaptive process. Since it is a first order filter, this example has a functionality which fixes a partition constant of 2 (*Np*=2) for the parameter adaptation input, *in_param*. That is, the adaptation is done by reading two input parameters of type T. Notice also that the type of inter-thread communication mechanism is relevant, regardless it is considered as part or not of the SW implementation of the adaptive process. The interface methods are also implemented making use of the RTOS system-calls for reading and writing message boxes:

```
void adaptiveLPF ::in_param (T& token) {
  rtos_api_write_mb(mbinp,token);
}
void adaptiveLPF ::in(T& token) {
  rtos_api_write_mb(mbin,token);
}
void adaptiveLPF ::out(T& token) {
  rtos_api_read_mb(mbout,token);
}
```

A more general coding of the inner process is possible, for instance, through an array of input parameters (***float*** *k[M];*, being M the partition constant of the parameterization input). In many cases, it will be also possible to write more efficient code avoiding input and output variables whenever dependencies let it. Finally, in this SW implementation, inputs and outputs handle the same type of parameter, *T*.

A way to declare and instance the adaptive LPF class would be the following one:

```
adaptiveLPF<float, 1024>  adapt_LPF;
```

Then, from another part of software code, this adaptive resource could be used, for instance, from three different threads. A control thread could be in charge of setting consecutive adaptations:

```
// control thread
//setting first adaptation
adapt_LPF.in_param (1.2);  // set K1
adapt_LPF.in_param(0.5);  // set K2
// setting second adaptation
adapt_LPF.in_param(3.1);  // set K1
adapt_LPF.in_param(0.6);  // set K2
```

A second thread could be in charge of sending stimulus:

```
// unbounded loop  in producer thread
while(true) {...
  adapt_LPF.in(data);
  // update data...

}
```

and the third thread, would read data processed by the adaptive object.

```
// unbounded loop  in consumer thread
while(true) { ...
  adapt_LPF.out (data); ...
}
```

Notice that there are other alternatives for writing software code implementing this parameter-based adaptive process. In this example, a C++ implementation using a C-based RTOS API has been employed. Another possibility could be, i.e., assuming a C plain implementation using the C-based RTOS API. Then, some kind of encapsulation could be done by putting thread handlers, input and output variables, etc as members of a *struct*.

This example also shows the close relationship between the implementation and the MoC. For instance, in this example, a parameterization is required for each N-set of processed samples. The software does not compute while the next adaptation parameters are pending. This corresponds to an implementation of a variant of a Kahn process network with finite FIFO channels. This could not be always interesting for an implementation. Implementations based on synchronous MoCs would let compute more evaluation cycles without forcing a regular or periodic write of adaptation parameters. In this case either, some kind of MoC refinement (if the specification MoC was untimed) or employing other MoCs at the specification level (if the software implementation preserves the MoC), such synchronous ones, will be necessary. This issue is not in the scope of this document.

9.3 *Mode-based Adaptive Process in Software*

In **[D11A]** the mode-based AP or mAP is defined. All that has been said for the implementation of the paAP applies, considering that the adaptation input updates a single parameter of a countable and finite type. This parameter is called *mode*.

There is a finite and known set of modes for each adaptive process. The mode serves to decode the functional relation to be computed between the input and the output. Each decoded computation can be implemented either through plain code (which can do an arbitrary number of function calls) or by means of one function call which directly relates the input variables with the output variables.

The next example shows the declaration of the SW implementation of a mAP. The mAP is a low pass first order filter of fixed gain and cut-off frequency with four modes. Each mode correspond to a different type of filtering (Butterworth, Chebychev, Bessel and Elliptic), with fixed characteristic parameters.

```
enum mode_t {BUTTER_M=0, CHEBY_M, BESSEL_M, ELLIPTIC_M};
template<class T, unsigned int N>
class adaptiveLPF {
public:
   adaptiveLPF();
  // input ports SW implementation
  void mode (mode_t mode); // mode  input
  void in (T&);                  // samples input
  void out(T&);                  // samples output
private:
   void root_function();
   void butter_LPF(T* in,T* out);
   void cheby_LPF(T* in, T* out);
   void bessel_LPF(T* in, T* out);
   void elliptic_LPF(T* in, T* out);
} ;
```

The *root_function* would be implemented as follows:

```
template< class T, unsigned int N>
void adaptiveLPF<T, N >::root_function() {
  bool mode;
  T  in_var[N],  T  out_var[N];
  while(true) { // root loop
    rtos_api_read_mb (minp,&mode);     //  mode input
    for(unsigned int i=0;  i < N; i++)  rtos_api_read_mb(in_var[i]);
   // functionality computation
    switch(mode) {
      case BUTTER_M:
      case default:
           butter_LPF(in_var, out_var);
     case  CHEBY_M:
           cheby_LPF(in_var, out_var);
     case BESSEL_M:
           bessel_LPF(in_var, out_var);
     case ELLIPTIC_M:
           elliptic_LPF(in_var, out_var);
    }
   for(unsigned int i=0; i < N; i++)  rtos_api_write_mb(out_var[i]);
  }
}
```

Let's see now an example where this kind of adaptivity has been identified in a complex sequential and real code. The following is an extract of the reference code of a H.264 encoder. In the following extract, a mode adaptive object is in charge of doing the entropy encoding. This mode adaptive object is called MB_BLOCK_SIZE (16) times, one for each syntax element (*currSE*).

```
// call to adaptive functionality
//=== encode intra prediction modes ===
 if (intra4)
  for (i=0; i < (MB_BLOCK_SIZE>>1); i++)  {
    // set currSE and dataPart (inputs to the adaptive object)
    …
    //--- encode and update rate ---
    dataPart->writeSyntaxElement (currSE, dataPart);
      ...
  }
```

The call to the adaptive code has been bolded. It can be seen how the input (*currSE*) and the output (*dataPart*) are directly related by the adaptive call. Previously, in other section of the code, the specific entropy encoding mode was set. This is shown in the next extract of the H.264 reference code:

```
  ...
  // mode setting
  if (input->symbol_mode == UVLC)
    dataPart->writeSyntaxElement = writeSyntaxElement_UVLC;
  else
    dataPart->writeSyntaxElement = writeSyntaxElement_CABAC;
  ...
```

where the function pointer *writeSyntaxElement* and the called functions are declared as follows:

```
int (*writeSyntaxElement) (SyntaxElement *se, DataPartition *this_dataPart);
int writeSyntaxElement_UVLC(SyntaxElement *se, DataPartition *this_dataPart) {...}
int writeSyntaxElement_CABAC(SyntaxElement *se, DataPartition *this_dataPart) {...}
```

It can be seen how a function pointer technique has been used to solve in a sequential C which is usually known in object oriented theory as dynamic polymorphism. Thus, if C++ were used in the implementation, this adaptive code can be solved also by means of a *base pointer* technique. That is, a parent class pointer can be used to reference and call common (in terms of declaration) methods of derived class instances. This code is sequential, however this does not remove the generality of the discussion. Indeed, the mode setting and the invocation code can be (with more or less transformation) restricted to the scope of a thread. The declaration and implementation of the invoked functions must be visible. The function pointer technique will be efficiently used in the next section to implement function-based Adaptive processes.

9.4 *Function-based Adaptive Process in Software*

In **[D11A]**, the software implementation of a function-based AP or fAP is proposed. It is similar to the mAP implementation but substituting the mode parameter by a *function pointer* parameter. Therefore, what the software adaptive process receives is indeed the function to execute. Specifically, the *static thread* implementing the adaptive process receives through its adaptation input a pointer to the function to be executed. Under an untimed MoC, the function pointer is received before each evaluation.

This is exemplified now with the adaptive filter. Let's assume that a third party provides a library of different types of LPF functions, whose declarations are the following ones:

```
// Library of LPF filtering
template< unsigned int N, class T>  void butter_LPF(T* in,T* out);
template< unsigned int N, class T>  void cheby_LPF(T* in, T* out);
template< unsigned int N, class T>  void bessel_LPF(T* in, T* out);
// template< unsigned int N, class T>  void elliptic_LPF(T* in, T* out); // in the future
```

Let's assume that the fourth function is not available in the current version of the library, but it will be available in future versions. The implementation of a mAP would require explicitly knowing the different possible mode functionalities. Thus, a current mAP implementation would decode three modes. In the future, after updating the third party library, the mAP should be rewritten to add a fourth mode which comprises the elliptic filtering. The alternative is to implement the adaptive process as a fAP. The declaration of the C++ class implementing such fAP is the next one:

```
template< unsigned int N, class T> void (*FPTR) (T* in, T* out);
template<class T, unsigned int N>
class adaptiveLPF {
public:
    adaptiveLPF();
  // input ports SW implementation
  void func_in(FPTR  fun); // function  input
  void in (T&);        // samples input
  void out(T&);        // samples output
private:
    void root_function();
  ...
} ;
```

where the *root_function* would be implemented as follows:

```
template< class T, unsigned int N>
void adaptiveLPF<Tm, T, N >::root_function() {
  FPTR fun;
  T  in_var[N],  T  out_var[N];
  unsigned int i;
  while(true) { // root loop
    rtos_api_read_mb (func_in,&fun);    //  reads function  input
    for(i =0;  i < N; i++)  rtos_api_read_mb(in_var[i]);
   fun(in_var, out_var);
  for( i=0; i < N; i++)  rtos_api_write_mb(out_var[i]);
 }
}
```

It is interesting to see then how the parameterization input is provided. For instance, the control thread in charge of setting the consecutive function adaptations could have a code like this:

```
// control thread
// declare functions
…
// setting first adaptation
adapt_LPF.func(cheby_LPF);
// setting second adaptation
adapt_LPF.func (butter_LPF);
…
```

Notice that the input parameters are the names of the filtering functions to be executed. Once the third party provides the new version of the filters library, the control thread can also perform the following call without needing to rewrite the fAP:

```
// control thread
…
// setting n-th adaptation
adapt_LPF.func(elliptic _LPF);
…
```

A fAP can provide an additional advantage respect to a mAP, such as speed, since, once the adaptation has been made, there is no need for decoding the function to execute.

9.5 *Process-based Adaptive Process in Software*

Other kind of adaptive object defined in **[D11A]** is the process-based AP or prAP. In ForSyDe, this adaptive process can adapt not only the function to be executed but also the process constructor. This involves that, in the adaptation, the ForSyDe process interface can also change. In terms of ForSyDe, this means changes in several aspects.

- The types of the input and output signals.
- The number of ForSyDe events read from input signals and written in the output signals at each evaluation (input and output partition).
- The number of input and outputs.

This has implications in terms of SW implementation. A simplification of the SW implementation is to consider a fixed interface, in terms of the set of method interfaces available. However, the inputs and outputs which are employed at each evaluation cycle and the number of tokens read and written can change at each evaluation cycle. To exemplify it, a new version of the adaptive low pass filter class is shown. Its declaration is the next one:

```
template<class T>
class adaptiveLPF {
public:
    adaptiveLPF();
   // input ports SW implementation
   void adapt_in(adapt_struct as); // function  input
   void in (T&);                   // samples input
   void out(T&);                   // samples output
private:
    void root_function();
   ...
} ;
```

In this version, the N template parameter, which determines the number of input tokens read and output tokens written, has disappeared. It is because, now, this is passed as parameter of the adaptation input. Notice the difference respect to the software implementation of the previous adaptive processes, where the adaptation parameters did not change the number of input or output tokens received and transferred in the inter-thread communications (N). N was settled at compilation time, since it was a template parameter. However, in this example, N is part of the adaptation parameter, of the *struct* type *adapt_struct*, which is declared as follows:

```
template<class T> void (*FPTR) (T* in,T* out, unsigned int N);
struct adapt_struct {
   FPTR  fun;
   unsigned int N;
};
```

The struct carries the number of samples to be evaluated after the adaptation and the function pointer with the function to be evaluated. Now, the function pointer is a little bit different from those shown for the implementation of the fAP. Now the declaration of the function pointer corresponds to functions that take as parameter N. Therefore, these functions are more general since they can process a different amount of samples each time they are called.

```
template<class T>  void butter_LPF(T* in,T* out, unsigned int N);
template<class T>  void cheby_LPF(T* in, T* out, unsigned int N);
template<class T>  void bessel_LPF(T* in, T* out, unsigned int N);
template<class T>  void elliptic_LPF(T* in, T* out, unsigned int N);
```

Then, the *root function* would be written as follows:

```
template< class T>
void adaptiveLPF<Tm, T, N >::root_function() {
  adapt_struct  adapt_var;
  T  * in_var,  T  *out_var;
  unsigned int i;
  while(true) { // root loop
    rtos_api_read_mb (minp,&adapt_var);     //  read adapt_var struct
    reallocate(in_var, adapt_var.N*sizeof(T));
    reallocate(out_var, adapt_var.N*sizeof(T));
    for(i =0;  i < adapt_var.N; i++)  rtos_api_read_mb(in_var[i]);
   adapt_var.fun(in_var, out_var,adapt_var.N);
   for( i=0; i < adapt_var.N; i++)  rtos_api_write_mb(out_var[i]);
  }
}
```

As can be seen, this time, the number of input and output system calls directly depends on the adaptation input. Then, the thread which feeds the adaptation input of this implementation of process-based adaptive object could present a code like the next one:

```
      // control thread
      adapt_struct adapt_conf
      // filters first 1000 samples with a butterworth filter
      // setting first adaptation
      adapt_conf.fun = butter_LPF;
      adapt_conf.N = 1000;
      // performing first adaptation
      adapt_LPF.adapt_in(adapt_conf);
      // filters next  5000 samples with a cheby filter
      // setting second adaptation
      adapt_conf.fun = cheby_LPF;
      adapt_conf.N = 5000;
      // performing second adaptation
      adapt_LPF.adapt_in(adapt_conf);
      …
```

This example represents a slight change on the process interface (referred to the partition of the inputs and outputs). From the ForSyDe point of view, this already involves a change on the ForSyDe process constructors at each evaluation. More complex implementation schemes can be found if it is assumed that the number and data types of input/output signals is dynamic. A possibility for such implementations is, for instance, to pass different class instances at each evaluation of the static thread. Each class instance can present a different interface and functionality. In this case, the number of inter-thread communications of the process (somehow, the thread "interface") would be fixed, since, static inter-thread communication instances are assumed.

9.6 *Self-Adaptive Process in Software*

An immediate possibility for implementing the self-adaptive Process is to introduce a second thread for the calculation of the next adaptation. This thread can have the same inter-thread communication with the producer threads writing the adaptive thread (the thread implementing the adaptive process), to have the same inputs. It can also receive the output tokens of the adaptive thread from the threads reading such output.

Figure 30 proposes an equivalent scheme where the inputs of the next adaptation thread are generated by the adaptive thread. Notice that, under the assumption of implementing untimed MoCs with blocking synchronizations and consuming reads in the inter-thread communication, a retransmission of the inputs and outputs of the adaptive thread is necessary. All of them become inputs of the next adaptation thread, used to compute the adaptation parameters, mode, function, etc, which are provided to the adaptive thread. Therefore, the set of adaptive thread plus new adaptation thread, together with their inter-thread communication is the implementation of the self-adaptive process.

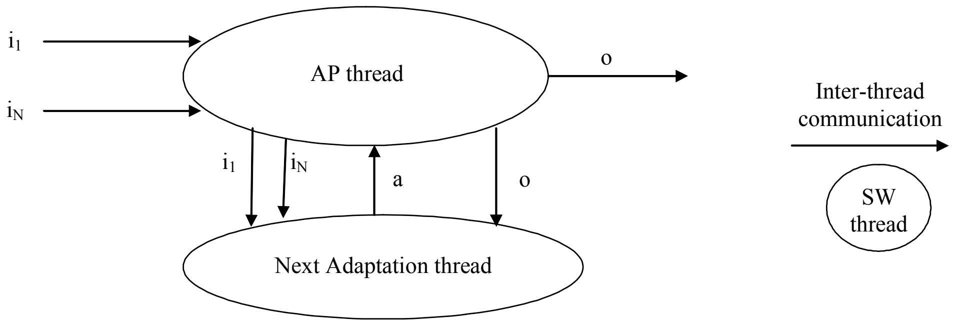


Figure 30. A SW implementation of a self -adaptive process

An alternative implementation does not use the new thread. In such a case, a the thread implementing the adaptive process has to dedicate part of its internal code to calculate the value of the next adaptation parameters, mode, function or process to be applied in the next evaluation. This is more efficient whenever the evaluation of the next adaptation input cannot be easily parallelized from the input/output computation or when the computation of the next adaptation thread is light-weighted with respect to the overhead of the inter-thread communication between adaptive and next adaptation thread.

9.7 *Implementation of Synchronous APs*

In chapter 6, the implementation as synchronous software will be introduced. It actually constitutes an additional extension of the *SWGen* methodology, which covered untimed MoCs, but not synchronous MoCs before ANDRES. This implementation has a direct link with the semantic of *HetSC* synchronous MoCs, where the synchronous APs shown in chapter 11, are framed.

10. Adaptive Processes in *HetSC*

This chapter shows how adaptivity is supported by the *HetSC* methodology. This proposal is based on the ForSyDe formal adaptive processes developed in **[D11A]** and on the way they can be implemented as eSW, as explained in the previous chapter. Specifically, the chapter provides guidelines for the specification of APs in *HetSC* (HAPs). These guidelines state the types of APs supported; focus the effort of the user in identifying which type of AP is most suitable for a given specification problem; and finally, explain how to specify such an AP in SystemC following the *HetSC* methodology. The SystemC structures generated can be then later either manually refined to software following the scheme shown in the previous chapter, or automatically generated by means of the *SWGen* methodology.

10.1 *Introduction*

As explained in chapter 8, the AP (Figure 31a) is a metamodelling concept which serves to unify the understanding of the AO among the different SystemC-based methodologies of ANDRES, among them, *HetSC*. This chapter defines how an AP is specified in *HetSC*, using *HetSC* and SystemC constructs. For the rest of the discussion, the general implementation of the AP in *HetSC* is shortly called HAP (*HetSC Adaptive Process*). Patterns to tell how to specify HAPs will be given in sections 10.3, 10.4 and 10.5. These patterns can be considered design patterns, as well as specification patterns, since they have been developed to support the immediate application of the *SWGen* methodology for automatic generation of eSW. Before, the specification of a HAP is addressed in a general way in this section, to later introduce the different types of HAPs (and thus, their corresponding patterns) in section 10.2.

A HAP is specified as a SystemC static process (Figure 31b). Since SystemC processes are associated to member functions of SystemC modules, a HAP will be necessary enclosed by a SystemC module (as represented in Figure 31b).

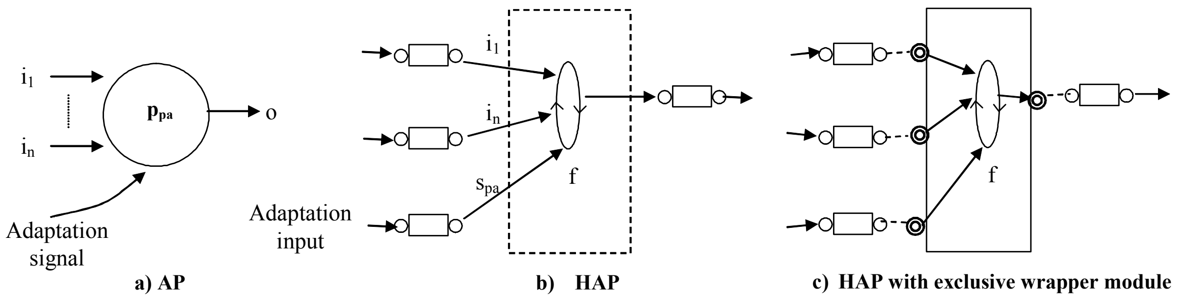


Figure 31. Specification of an Adaptive Process in *HetSC*.

The HAP can have associated a set of context variables. It is recommended these variables to be local to the function associated to the SystemC process, although they can be out of it if there are stack limitations. In such a case, it is recommended these variables to be inside the 'wrapper' module. If that is not possible, then the variables can be global (out of the module), but never shared by other processes[2]. Global variables are considered to be shared in *HetSC* if they are accessed by a second process with involvements in its control or data path or in its outputs. The communication of the HAP with the rest of the system is through channels, as

[2] This takes into account a basic rule of the *HetSC General Specification Methodology*, to obtain a strict separation between computation and communication, the modularity of the specification, and the feasibility of the SWGen flow among other design activities.

the *HetSC* methodology involves. The MoC of the specification fixes the type of *HetSC* channels employed.

The main feature which enables the consideration of this process-based structure as a HAP is that at least one of the inputs is considered as an adaptation input. The rest of input channels are considered as *regular* inputs. Then, an adaptation is considered to be a change in the internal state of the HAP which specifically changes *adaptation context* variables. This, in time, changes the functional relationship between the regular inputs and the output. Moreover, it can even change the HAP interfaces. HAPs with more than one output channel can be also specified. In such a case, a ForSyDe adaptive process is necessary for abstracting each functional relationship fixed between the regular and adaptation inputs and each specific output. In order to focus and simplify the discussion, single-output patterns will be presented.

The *HetSC* MoC fixes the level of detail of time information handling. This means that the specification can fix the *adaptation time* only at a specific-level of detail: the one handled by the MoC. In *HetSC* the MoC is strongly characterized by channel semantics. Therefore, the usage and semantic of HAPs are involved by the *HetSC* MoC, and thus by the channels and other specification facilities related to that MoC.

It is recommended, although not mandatory, to reserve an associated *wrapper* module for the HAP (Figure 31c). In such a case, the application of the *HetSC* methodology involves that the communication of the adaptive process with the rest of the specification has to be done by means of ports. Guidelines given in the chapter will be based on this assumption.

HAPs are design patterns based on ForSyDe. In ForSyDe, process constructors determine the MoC, which can be related to *HetSC* structures. Thus, for instance, in a ForSyDe untimed adaptive process, the information available about adaptation time just considers the relationship given between the amount of ForSyDe events that have to be read at the regular inputs and the amount of ForSyDe events that are read from the adaptation input (for a single adaptation), which is determined by the partition functions of the adaptive process constructor. In a similar way, in *HetSC*, synchronization semantic and buffering size of the input/output channels and the internal functionality of the HAP, and the MoC will determine the specific relationship between the amounts of tokens (data units) consumed in the regular inputs and the tokens consumed in the adaptation inputs. The tokens transferred in the SystemC model correspond to ForSyDe events at the metamodelling level.

10.2 *Types of HAPs*

Two typologies of APs are distinguished in *HetSC*. One comes from the consideration of the specification domain (that is, of the MoC). The other one comes from the typology reminded in Chapter 8, considering the information adapted. This work covers the APs under two types of specification domains: Untimed and Synchronous, for the different types of information adapted (parameter, mode, function, process). This is shown in Table 1. Obviously, the cross-product of these sets produces till 10 types of APs.

AP Type	Value
Specification Domain	Untimed, Synchronous
Information Adapted (Adaptation Context)	Parameter, Mode, Function, Process, Self-Adaptive

Table 1 Types of APs currently covered in *HetSC*.

10.2.1 Untimed HAPs

Untimed HAPs handle no notion of strict-time about its inner computation. Therefore, the only information about *adaptation time* is the relationship between the amount of data units (tokens) consumed at the adaptation input and the amount of tokens consumed at the regular inputs. In the most general case, such a rate can be fixed at each adaptation. Then, the HAP is inscribed in dynamic data flow model. In a more specific case, it can remain the same for each adaptation during the whole simulation time. Then, the HAP can be part of a static data flow model.

In any case, in an untimed HAP, the *adaptation time* comes as a function of consumed data, both at regular and adaptation inputs. An adaptation happens only after a given amount of data tokens have been consumed at adaptation input. Additionally, an adaptation cannot happen till a minimum amount of tokens has been consumed from the regular inputs for each adaptation.

Untimed HAPs alternated adaptation and computation. An adaptation precedes its associated computation. This sequence of adaptation-computation is synchronized with the producer and consumer processes. An HAP can involve the blocking of any of the connected processes, depending on the size capabilities of the rest of connected processes.

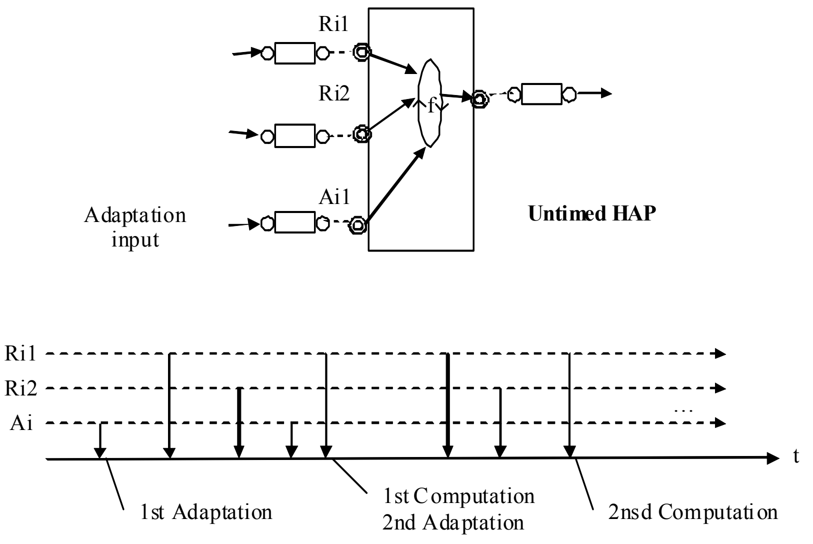


Figure 32. Adaptation time in untimed HAP comes as a relationship of consumed data.

As an example, Figure 32 shows an untimed HAP with two regular input ports (Ri1 and Ri2) and an adaptation input (Ai1). Lets assume that this untimed HAPs can be metamodelled are as an untimed ForSyDe APs, whose input partitions are fixed to be $\gamma_{Ri1}=2$, $\gamma_{Ri2}=1$ and $\gamma_{Ai1}=1$. The bottom part of Figure 32 shows the occurrence of write accesses to the input channels. It can be seen first that in order to deduce the adaptation time is not necessary (not even possible, if the whole model does not consider time) to consider or to distinguish an adaptation strict-time (that is, a specific SystemC time stamp). What the untimed HAP of Figure 32 states is that an adaptation will not take place till a given amount of tokens at adaptation input ($\gamma_{Ai1}=1$ in this case) have been consumed. It also states that a further adaptation will not take place while the HAP do not consume $\gamma_{Ri1}=2$ tokens from the first regular input and $\gamma_{Ri2}=1$ tokens from the second regular input, as well as the necessary tokens from the adaptation input. Figure 32 shows for instance, that despite a second adaptation token can be read, the 2nd adaptation is not done till the 3 required tokens at the two regular inputs are consumed. Then, the 1st computation is done and the 2nd adaptation can take place.

Untimed HAPs are suitable when they have to be employed in untimed models, which do not handle explicit strict-time information on the function/process computation. This is quite usual in software models which are written to preserve functionality independently on the time conditions established for the target architecture.

10.2.2 Synchronous HAPs

Synchronous HAPs handle a more detailed notion of time. Two types of synchronous APs are distinguished. Synchronous Reactive APs and Clocked-Synchronous APs. In the former case, the adaptation *can* happen in a *slot*, while in the second case in a *cycle*. In the Synchronous HAP, there is a fixed relationship between the amount of data consumed from the regular inputs and from the adaptation input. Just one data unit is read from each regular input and from the adaptation input at each slot or cycle. This abides the ForSyDe meta-modelling of synchronous processes, where a fixed partition of 1 is assigned to every input and output. Complex data type of data can be still transferred through the inputs. Then, for instance, burst of data can be transferred through the inputs of the synchronous model. A facility like the *uc_burst* class integrated in the *HetSC* library is used in the *HetSC* examples to show this possibility. More information about this can be found in chapter 11.

Further consideration can be done about the *adaptation time*.

Clocked Synchronous APs have a clock input, that is, a global event, in such a way that the AP computation is triggered at the same time (thus synchronized with) the rest processes of the specification (or the CS domain of the specification) by this global event. In Clocked Synchronous APs or CS APs, adaptation input is read at each clock event, before performing the computation. The information about *adaptation time* is therefore more detailed than in the case of untimed APs, since adaptation takes place at each clock input event whenever there is a change of value in the adaptation input.

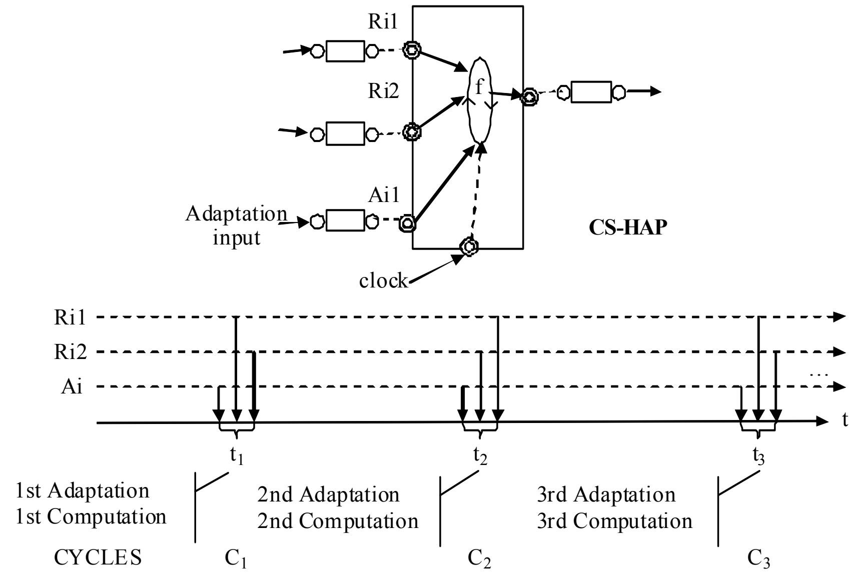


Figure 33. Adaptation in the CS-HAPs takes place at cycles.

Figure 33 shows a clocked-synchronous HAP with two regular inputs and one adaptation input. An additional input port transfers the global clock trigger event to the CS-HAP. The read of the regular inputs and of the adaptation inputs take place at specific time stamps ti. Their actual value is not relevant, but the fact they are distinct and consecutive time stamps, defining a total order. In the CS-HAP, the adaptation input Ai is always read the first. Then, the regular inputs can be read in any order (and as many times a required during the cycle).

Clocked Synchronous HAPs will be used when they are incrusted in a system-level model handling a clocked notion of time. In a software implementation, such a clock is a global *software event*, in charge of triggering all the clocked part of the software partition of the system. The implementation of the *software clock* will be based on the implementation of the *software event*. The *software event* can be solved in different ways (by means of software signals, or other inter-thread primitives). The *software clock* can define an actual time period (by using the tick timer) or not. In the former case is useful when the related analysis tools are able to structure the code in a reactive way and analyze the delay taken by the critical path (in a manner similar to SW). For instance, audio application will require at some point the production of samples at a fixed rate, i.e. 8KHz. In such a case, the model, and further the SW application, can be designed to respect such latency of 125μsec. Very roughly, an embedded system working at 80MHz, would have a margin of 10000 cycles (instructions assuming a perfect pipelined architecture) in the works case to compute the required functionality per sample. However, the *HetSC* understanding of the clock does not necessarily relates the *clock* to a periodic clock. With respect to the software implementation of the CS-HAP it has been necessary an extension of the *SWGen* methodology. The specific implementation supported by *SWGen*, showing *SWGen* automatically provides an eSW implementation of the *software clock*, the *software event*, and other primitives supporting the CS domain is further explained in chapter 6.

The **Synchronous Reactive HAPs**, is a SR process of *HetSC*. Such a process can be triggered by any write in any of its inputs, provoking its computation. Therefore, in the SR-HAP, the adaptation input can "asynchronously" adapt the inner functionality which relates the regular inputs with the output. This means that one or N adaptations can be performed (in one or N slots/cycles) without involving any processing of regular outputs (in contrast to the CS-HAP, where the adaptation and the computation takes place at the same cycle[3]). For a given computation, only the last adaptation will be effective. Symmetrically, several reactions can take place with no kind of adaptation in between. Each reaction will be due to a trigger (or, what is the same in the Synchronous Reactive (SR) approach of *HetSC*, a write) of any of the input channels of the SR-HAP. Both adaptations and computations take place at given and different time stamps, which, as in the CS case, establish a total order on events. Such time stamps are in this case *slots* (instead *cycles*). The write/trigger of the adaptation inputs and regular inputs can take place at the same time slot. In such a case, in coherence with the other types of HAPs, the adaptation takes precedence over the computation.

[3] In CS-HAP there is computation at every cycle, which does not mean that there is adaptation take place at every cycle in CS-HAP. However, if there is adaptation, it takes place at the same cycle, before the computation.

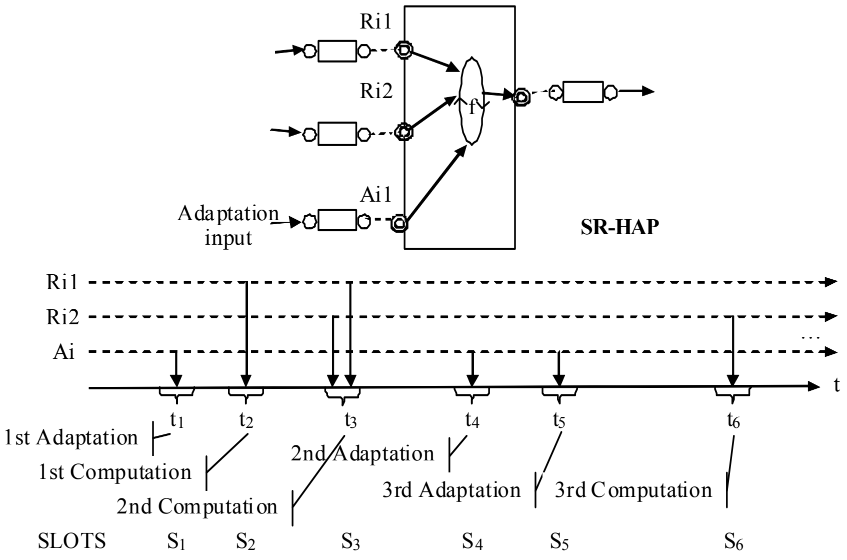


Figure 34. Adaptation in the SR-HAPs takes place at slots.

Figure 34 shows a SR-HAP of two regular inputs. In this case, a first trigger of the HAP takes place at time stamp t_1 due to a write to the adaptation input channel Ai. Then an adaptation taking 0 time takes place at t_1 (1st slot). Later on, there is a computation at t_2 (2nd slot) due to a write to the first regular input. At a later time stamp t_3 (3rd time slot), there is a SR-HAP trigger due to a simultaneous write of both regular inputs. As can be seen, computation at second and third slot use the state left by the first adaptation, without needing any adaptation in between. Later on, two consecutive adaptations are forced in the 4th and the 5th time slots. A new computation is done in the 6th time slot (in t_6), which uses the state left by the 5th adaptation.

Two types of SW implementation are foreseen for SR-HAPs (and SR specifications in general). The former one can be based on asynchronous events (i.e., exceptions) in charge of triggering software computation. The second one consists in generating an implicit SW periodic clock whose period is a basic implementation parameter, unlike in the CS case, where the global clock was explicitly specified. Then, this implicit global clock is in charge of defining the granularity of "software time" (which can be tuned to be bigger than the processor clock and make software more efficient). Then, the simultaneous software events are defined to be those relying on the global software period. This global software period is a design parameter. In one side, it has to be big enough to keep the greater reaction time lesser than it. On the other side, it would be interesting it to be short enough to let reactions to single events (fundamental mode), which would simplify HAP coding.

Synchronous Reactive APs are suitable for specifying processes whose functionality has to be changed by an asynchronous event, such as a keyboard keystroke, etc, and the adaptation data can be atomically passed to the adaptive process.

The different types of HAPs (regarding the type of adaptation context or adaptation information) will be explained first for the untimed HAPs (section 10.3), to later extend their explanation to synchronous HAPs.

10.3 *Patterns for Untimed HAPs*

10.3.1 Untimed Parameter-based Adaptive Process

Figure 35a reminds the formal representation for the untimed paAP. There, i_1 to i_n and s_{pa} are untimed ForSyDe signals ($\dot{s}$).

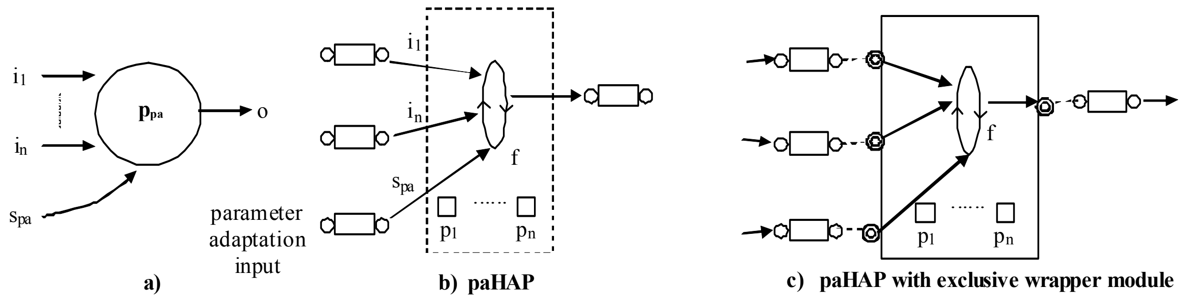


Figure 35. Untimed paHAP.

The Figure 35b shows the general *HetSC* structure of an untimed *parameter-based HAP (paHAP)*. Figure 35c does the same using an exclusive *wrapper* module.

The peculiarity of the paHAP is the set of parameterization variables associated to the SystemC process. They can be within the local scope of function (f) or, if not possible, they are recommended to be within the ambit of the *wrapper* module, as in Figure 35.

When to use it:

The paHAP is useful when the different functionalities performed among adaptations can be done by means of the same computation structure, changing only a set of fixed and well defined parameters. Then, the HAP root function can include a single parameterizable function.

A typical case could be an adaptive FIR filtering, where each adaptation only needs to modify the filtering coefficients. Then, external processes just pass different sets of coefficients through the adaptation input to, for instance, switch from a low-pass filter (LPF) to a high-pass filter (HPF), to change the cut-off frequency of the LPF, etc. In any case, the function structure remains the same and can be reused by the HAP.

Design Pattern:

The untimed paHAP class can be declared as follows:

```
(1)    class user_paHAP : public sc_module {
(2)    public:
(3)      sc_port <IF<T_1> >        in_1;
(4)      sc_port <IF<T_2> >        in_2;
(5)      ...
(6)      sc_port <IF<T_N> >        in_M;
(7)      sc_port <IF<T_P> >        in_param;
```

```
    sc_port < IF<T_O> >     out;


    void root_function();


    SC_CTOR (user_paHAP) {
      SC_THREAD(root_function);
    }
  private:
    T_P     param[N_param];
    T_1 input_var1[N_in1],  T_2 input_var1[N_in2], ..., T_N  input_var[N_inM];
    T_O     output_var[N_out];
    ...
  };
```

T_i stands for the specific type of data to be transferred at each input/output port. This data type defines the specific untimed interface used **IF**. Therefore, IF can be substituted for the different interfaces employed in the untimed MoCs of *HetSC*, for instance, *sc_fifo_blocking_in_if*, for the input, and *sc_fifo_blocking_out_if*, for the outputs. Equivalently, specific ports can be used to simplify the syntax. For instance, the untimed *HetSC* ports *uc_fifo_blocking_in* for the inputs and *uc_fifo_blocking_out* for the outputs.

N_param stands for the number of adaptation parameters. It can be generalized if different parameters of different types where needed. N_in1, N_in2, …N_inM stand for the partition which the internal functionality will require from each input. The meaning of each N_in depends on the type of untimed MoC. Under a static data flow, each N_in figure means the actual and fixes partition of the input. Under a dynamic data flow, each N_in is the maximum partition. In any case, such a partition is fixed by the structure of the inner functionality, which fixes how many input tokens have to be read from each input in order to compute the output.

For a static dataflow approach, the internal root function can be then coded as follows:

```
void user_paHAP::root_function() {
  unsigned int i;
  while(true) {
     // read adaptation parameters
     for (i=0;i<N_param;i++) {
         in_param->read(param[i]);
     }
     for(i=0;i<N_in1;i++) in_1->read(input_var[i]);       //
     ...
     for(i=0;i<N_inN;i++) in_N->read(input_var[i]);       //
(11)        param_fun(N_param, param,
                          N_in1, input_var1,
                          N_in2, input_var2,...,
                          N_inN, input_varN,
                          N_out, output_var);         // evaluation
(12)        for(i=0;i<N;i++)  ++) out->write(output[i]);   // output samples
(13)      }
(14)    }
```

An important and distinguishing fact here is the blocking semantic of these input/output interfaces. They let code the paHAP under an untimed MoC. The pattern clearly separates the adaptation parameters (concentrated in (5) to (7) statements), from the read of regular inputs (from (8) to (10)), from the evaluation (11), and from the output (12). The parameter input has to be read as many times as parameters of *T_P* type have to be read, fixing the parameter input partition.

The coding has to respect the completion of parameter and regular inputs reading, before the evaluation. This ensures the adaptation to be performed before the evaluation. Regarding to the rest of aspects of the code there is flexibility to modify much of this structure in several aspects. For instance, the read of regular inputs can be rearranged. It is possible to exchange the order of loops in (8) and (10). It is also possible to interleave input port reads, for instance, reading the first token from input *in_1*, then the first token from input two, and so on till reading and amount of tokens from each of the M-th inputs given by its partition. Moreover, the evaluation can be also spread among the inputs and outputs, then reducing the need of buffering, done in the shown code by means of *param*, *output_var*, *input_var1*, *input_var2*, …, and *input_varN* array variables.

As well as such flexibility, the code shown also admits a set of simplifications. Obviously, the template defined could work with homogeneous data types for the regular inputs, adaptation data and output data. In the most simple case, T_1 = T_2 =…= T_N = T_O = T_P. In some cases, the number of inputs can be also reduced. The function call in charge of the evaluation (11) can be also simplified. In the general case shown there, *param_fun* is a function which works with M input arrays for the collected regular inputs, every of them of a different length, and additional array of a given length for the parameters input and finally and output array with its own length for dumping results. Obviously, this function call will get simpler if working with arrays of the same length. Depending on the algorithm it can also happen that the function can work on input and output partitions of length 1. In a simple 1-input case, the root function can get as simple and compact as follows:

```
(1)     void user_paHAP::root_function() {
(2)       unsigned int i;
(3)       while(true) {
(4)         // read adaptation parameters
(5)         for (i=0;i<N_param;i++) {
(6)             in_param->read(param[i]);
(7)         }
(8)        // evaluation
(9)        out->write( param_fun( param, in_1->read() ) );
(10)     }
(11)   }
```

In this case, the read of regular inputs is compacted in the same sentence of the *param_fun* call. There is no need to buffer regular inputs, since functionality operates on inputs of partition 1. The same can be told for the output. This structure is maybe not the most efficient one since it obliges one adaptation for each data unit processed. Because of this, for untimed HAPs it can make sense evaluating on longer partitions for regular inputs.

Example:

Following, an example of a *HetSC* specification of an untimed paHAP is presented. It consists in an adaptive Low-Pass Filter (LPF), based on the same computation structure (i.e. either FIR or IIR), which only changes the filter coefficients (b_k):

```
template<class T, unsigned int N>
class adaptiveLPF :  public sc_module  {
public:
  sc_port <sc_fifo_blocking_in_if<T> >  in_param;  // filter parameterization input
  sc_port <sc_fifo_blocking_in_if<T> >  in;            // samples  input
  sc_port < sc_fifo_blocking_out_if T>    out;           // samples output
  void root_function();
  SC_CTOR(adaptiveLPF) {
     SC_THREAD(root_function);
  }
private:
    T  k1, k2;
    T input_var[N], output_var[N];
    ...
}
```

where the root function has the next implementation:

```
template<class T, unsigned int N>
void adaptiveLPF <T,N>::root_function() {
  unsigned int i;
  while(true) {
     in_param->read(k1);                              // gain parameterization
     in_param->read(k2);                              // pole parameterization
```

```
    for(i=0;i<N;i++) in->read(input_var[i]);        // input samples
    filter_fun(k1,k2,input_var,output_var,N);         // evaluation
    for(i=0;i<N;i++)  ++) out->write(output[i]);  // output samples
  }
}
```

As can be seen, it is a structure quite similar to the structure of the software implementation presented in the previous chapter, but substituting inter-thread communication system calls by *HetSC* channel accesses. Obviously, the corresponding channel instances have to be declared and instanced to connect the adaptive SystemC process with other processes of the specification. As in section 9.2, the inner function called from the root function, *filter_fun*, provides a functional relationship between every output sample variable (*output_var[i]* for $0 \leq i \leq N$) transferred to the output and the set of input variables read from the regular inputs (*input_var[i]* for $0 \leq i \leq N$) and the adaptation parameters, *k1* and *k2*. The blocking read semantic of the channels bound to the inputs ports ensures the synchronization between the computation and the adaptation. Therefore, the functionality is computed with both, adaptation parameters and input samples updated before each evaluation.

In this example, the adaptation input is of the same type as the rest of the inputs. The partition of the adaptation input is, in this case, 2, since each adaptation requires reading 2 tokens, the gain parameter (*k1*) and the pole parameter (k2). The body of the *root_function* determines when adaptation is carried out respect to the number of input samples computed, N. Since N is a template parameter, the number of tokens read and written does not change among different adaptations.

Following, an extract of the code of a *sc_main* function where the paHAP is declared, instanced and bound to channel instances is shown:

```
…
// channel declaration and instances
uc_fifo<float>  in_param_fifo("in_param_fifo",2);
uc_fifo<float>  in_fifo("in_fifo",1000);
uc_fifo<float>  out_fifo("out_fifo",1000);
// paAP declaration and instance
adaptiveLPF<float,1000>   adaptLPF;
// paAP binding
adaptLPF.in_param(in_param_fifo);
adaptLPF.in(in_fifo);
adaptLPF.in(out_fifo);
…
```

Instances of *uc_fifo* class of a sufficient size have been used, thus a BKPN MoC has been employed. However, schemes under different untimed MoCs can achieve the same result. For instance, the *uc_fifo* instances of this example can be substituted by *uc_inf_fifo* instances. The use of other types of channels of *HetSC* untimed MoCs will be illustrated in the following sections.

10.3.2 Untimed Mode-based Adaptive Process

Figure 36b proposes the structure for the *HetSC* specification of a mode-based Adaptive Process (mHAP).

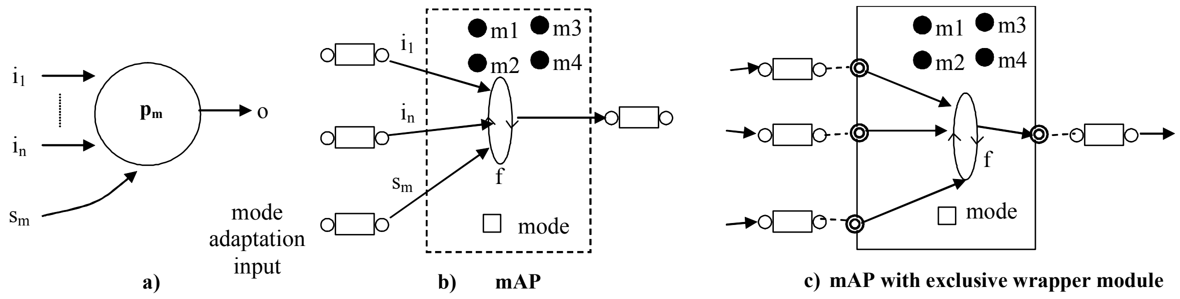


Figure 36. Specification in *HetSC* of the mode-based Adaptive Process.

As can be seen, there can be a *mode* parameter updated at each adaptation or the mode input can be directly read for it. In any case, the functionality to be computed is determined by the mode read. Such functionality can be encapsulated in methods (represented as black dots within the module class in Figure 36) or C functions which are visible, thus within the scope of the mHAP. Each functionality (or computational mode) to be executed is associated to a discrete countable mode.

When to use it:

The mHAP is suitable when the set of computational modes is bounded, thus ranged by a finite countable type, i.e. an *unsigned int* C type. With respect to the paHAP, mHAP advantage is that it does not require the structure of the computations associated to each mode to be the same. To the contrary, each computational mode can present a very different computation structure. Each of those computation structures can be wrapped in a specific (C/C++) function.

A typical case could be switching between a LPF and a HPF M-order filtering, where, in both cases, the order and coefficients of the filter have been prefixed. In the mHAP, the LPF and the HFP computation structure can be different. For instance, the LPF filter can be of FIR type, while the HPF filter of IIR type. Another clear case can be an encrypter which can employ very different crypto algorithms depending on the mode.

Design Pattern:

An untimed mHAP class can be declared as follows:

```
(1)     class user_mHAP :  public sc_module  {
(2)     public:
(3)       sc_port <IF<T_1> >                   in_1;
(4)       …
(5)       sc_port <IF<T_N> >                   in_N;
(6)       sc_port <IF<COUNTABLE_TYPE> >       in_mode;
(7)       sc_port < IF<T_O>  >                 out;
(8)       void root_function();
```

```
    SC_CTOR(user_mHAP) {
      SC_THREAD(root_function);
    }
private:
    T_1 input_var1[N_in1], ..., T_N  input_varN[N_inM];
    T_O     output_var[N_out];
    void  f1(...);
     ...
    void  fn(...);
...
};
```

As can be seen, the template is similar to the *paHAP*. It includes N input ports of an untimed blocking interface and an internal root function. It also can include in the private part variables for storing input and output variables, in case the internal computation needs it. The main differences with respect to the *paHAP* pattern come on the adaptation input and on the internal structure of the *root_function*. The adaptation input transfers a countable data type (i.e. an *unsigned int*, and *enum*, etc) which represents the mode.

Then, the root function can be written as follows:

```
void user_mHAP::root_function() {
  unsigned int i;
  while(true) {
    switch(in_mode->read()) {
      case 0: // mode 0
        // read input samples
        for(i=0;i<N_in1;i++) in_1->read(input_var[i]);
        ...
        for(i=0;i<N_inN;i++) in_N->read(input_varN[i]);
        f1(input_var1, ..., input_varN, output_var);
        for(i=0;i<N_out;i++) out->write(output_var[i]);
        break:
      case 1: // mode 1
        // statement 1
        // statement 2
        // statement n
       break:
      case 2: // mode 2
        // f2
```

```
            // unfolded  interleaved inputs, computation and outputs
            out->write(in1->read() + 1);
            out->write((in1->read() + in2->read()+1);
           break:
    ...
          case Nmodes: // mode Nmodes
             out->write(fn(in1->read(),in2->read()));
            break:
        } // end switch
      } // end loop
    } // end root function
```

According to an untimed model, the root function specifies a strict order, where the adaptation input is read first and once (mode input partition is thus 1). Later on, the mode is decoded to perform the reading of the inputs, the computation and the writing the outputs in each *case* branch.

This order between mode input read and regular inputs reading can be altered if each computational mode to performs the same readings (which means same partition for each regular input). The pattern previously shown shows a more general case where each branch presents a different input partition. For instance, mode 0 computation involves reading *N_in1* tokens from input *in1*, *N_in2* tokens from input *in2*, etc to finally read *N_inN* tokens from input *N_inN*. However, mode 2 computation reads two tokens from input *in1* and one token from input *in2* to produce and write two output tokens.

Moreover, the pattern shows that in a general case, each more or branch computation does not need to be encapsulated within a C function, as it is generically shown in mode 1 of *mHAP* pattern. Therefore, it is possible to interlace the reading of inputs, computation and outputs and even avoid intermediate variables for data read and output data. The *mHAP* pattern shows such interlacing for mode 2, where the process first reads once input *in1* to compute and write the 1st output. Then a second read from input *in1* and the first read from input *in2* take place after the first output write.

Mode computation can be encapsulated either in mHAP class methods (as it is shown by the previous pattern) or be in external functions visible to the mHAP class. When the mode computation is encapsulated within a function or within a class method, the internal variables are needed to interface such method/functions with input/output ports. The following example shows the declaration and implementation of a mHAP class using this approach.

As with the paHAP, the mHAP pattern previously presented admits several simplifications if the number of inputs and/or the types of input and output data can be reduced.

Example:

The following example presents the declaration of an adaptive LPF based on the example of section 9.3.

```
enum mode_t {BUTTER_M=0, CHEBY_M,BESSEL_M, ELLIPTIC_M};
template<class T, unsigned int N>
```

```
class adaptiveLPF :  public sc_module  {
public:
  sc_port <sc_fifo_blocking_in_if<mode_t> >      in_mode;  // mode input
  sc_port <sc_fifo_blocking_in_if<T> >            in;         // samples  input
  sc_port <sc_fifo_blocking_out_if<T> >           out;        // samples output
  void root_function();
  SC_CTOR(adaptiveLPF) {
     SC_THREAD(root_function);
  }
private:
    void butter_LPF(T* in,T* out);   // method m1
    void cheby_LPF(T* in, T* out);  // method m2
    void bessel_LPF(T* in, T* out);  // method m3
    void elliptic_LPF(T* in, T* out); // method m4
    mode_t  mode;
    T input_var[N], output_var[N];
    …
};
```

where the root function has the next implementation:

```
template<class T, unsigned int N>
void adaptiveLPF<T,N> ::root_function() {
  unsigned int i;
  while(true) {
     in_mode->read(mode);                              // mode adaptation
     for(i=0;i<N;i++) in ->read(input_var[i]);         // input samples
    switch(mode) {                                     // evaluation
      case BUTTER_M:                                       // mode 1
      case default:
           butter_LPF(input_var, output_var);
     case  CHEBY_M:                                        // mode 2
           cheby_LPF(input_var, output_var);
     case BESSEL_M:                                       // mode 3
           bessel_LPF(input_var, output_var);
     case ELLIPTIC_M:                                     // mode 4
           elliptic_LPF(input_var, output_var);
    }
    for(i=0;i<N;i++)  out ->write(output_var[i]);   // output samples
  }
}
```

In order to show other possible untimed implementations, in this example, a solution with infinite *HetSC* FIFO channels will be shown. Then, the extract of the *sc_main* code where the communication channels and the adaptive process is declared and instanced would be as follows:

```
…
// channel declaration and instances
uc_inf_fifo<float>  mode_fifo("mode_fifo");
uc_inf_fifo<float>  in_fifo("in_fifo");
uc_inf_fifo<float>  out_fifo("out_fifo");
// mAP declaration and instance
adaptiveLPF<float,1000>   adaptLPF;
// mAP binding
adaptLPF.in_mode(mode_fifo);
adaptLPF.in(in_fifo);
adaptLPF.out(out_fifo);
…
```

Notice that, although infinite fifos are used, the N template parameter fixes the amount of samples processed for each mode adaptation. Notice also that, as it happened in the software implementation proposed for the mode-based AP (section 9.3), the executed mode function directly relates the rest of inputs with the outputs, without further influence of the mode value. The mode parameter is only used to select (decode) which function (or, equivalently, which branch of inner code) to compute in the current evaluation.

10.3.3 Untimed Function-based Adaptive Process

Figure 37b proposes the structure for the *HetSC* specification of a function-based Adaptive Process (fAP).

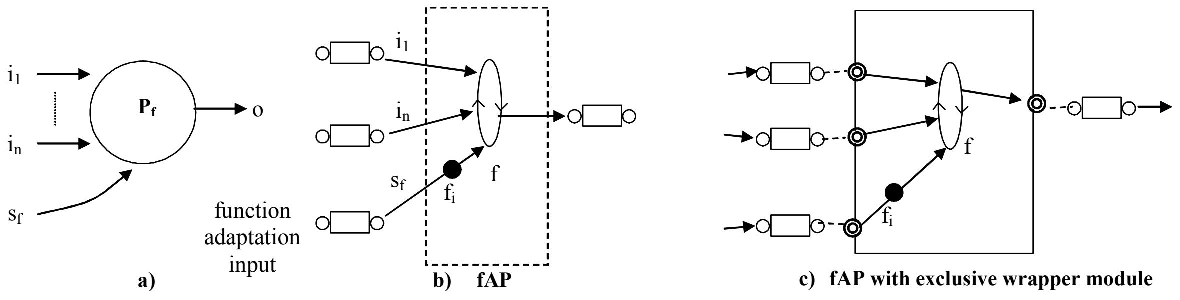

i_1
i_n
s_f
P_f
o
a)
function adaptation input
i_1
i_n
s_f
f_i
f
b) fAP
f
f_i
c) fAP with exclusive wrapper module

Figure 37. Specification in *HetSC* of the function-based Adaptive Process.

It follows a similar idea as that developed for the SW implementation of a fAP, assuming that the functionality to be executed by the HAP is out of its scope of visibility, that the set of functions to execute can change in the future or that is unknown within the context of the HAP. The only thing known is the prototype of the functions to be executed, which is fixed. Then, the tokens which feed the adaptation input are function pointers, representing such functions. This is a new feature incorporated to the *HetSC* methodology. The development of *HetSC* channels was mostly intended for the transfer of data, (usually simple data structures), which are consumed and transformed by processes. However, thanks to the generality of *HetSC* channels (they are actually templates which can transfer abstract data types), they can also be used to transfer the functionality to be executed in the shape of function pointers.

When to use it:

The fAP is useful, when, as with the mHAP, the structure of the different computations performed among adaptations is different, and furthermore, the set of different functionalities performed among adaptations is undetermined, since it comes from the environment or it could be changed during run time (thus it cannot be characterized by a finite countable type, i.e. *an unsigned int*). Then an fHAP can be reused, while the mHAP would require an extension to cope with the new computational modes.

A typical case can be a process that can dynamically get updates, for instance of a routing algorithm, or other kind of algorithm which could be improved in the future, etc…The new functionality could be provided by dynamic libraries or libraries which can be updated or added during run-time.

Design Pattern:

An untimed fHAP class can be declared as follows:

```
class user_fHAP :  public sc_module  {
public:
   sc_port <IF<T_1> >         in_1;
   sc_port <IF<T_2> >         in_2;
   …
   sc_port <IF<T_N> >         in_N;
   sc_port <IF<FPTR> >        in_function;
   sc_port < IF<T_O>  >       out;


   void root_function();


  SC_CTOR(user_fHAP) {
     SC_THREAD(root_function);
  }
private:
  FPTR     fp;
  T_1 input_var1[N_in1],  T_2 input_var1[N_in2], …, T_N  input_var[N_inN];
  T_O    output_var[N_out];
```

```
    ...
};
```

The root function can be implemented as follows:

```
void user_fHAP::root_function() {
  unsigned int i;
  while(true) {
    in_function->read(fp);                              // function adaptation
    for(i=0;i<N_in1;i++) in_1->read(input_var[i]);   // input samples
     ...
   for(i=0;i<N_inN;i++) in_N->read(input_var[i]);
   fp( N_in1, input_var1,
       N_in2, input_var2,...,
       N_inN, input_varN,
       N_out, output_var);                          // evaluation

    for(i=0;i<N;i++) out->write(output[i]);          // output samples
  }
}
```

FPTR stands for the type of function pointer to be transferred through the adaptation input port. After each adaptation, fp is updated with the incoming read function pointer, which points a function which can be out of the scope of the fHAP instance. FPTR prototype fixes the interface of the inner computation performed by the fHAP. This, in time fixes the expected input ports and input partitions. It also demands the intermediate *input_var* and *output_var* variables to interface input/output ports with the inner function call via *fp*.

As in sections 8.3.1 and 8.3.2, simplifications on the number of inputs and on the output port types can be done.

Example:

Following, a *HetSC* example based on the example of section 9.4 is presented. Now, the *adaptiveLPF* class defined by the user admits the adaptation of an unbounded number of types of LPF filtering, whenever all of them are encapsulated within a function of the type *void LPF_fil_fun(unsigned int N, T *in, T *out);*, where T stands for the data type of the samples filtered.

```
template<class T> void (*FPTR) (T* in,T* out);
template<class T, unsigned int N>
class adaptiveLPF :  public sc_module  {
public:
  sc_port <uc_arc_cons_if<FPTR> >     in_fun; //function  input
  sc_port <uc_arc_cons_if <T> >        in;        // samples  input
  sc_port <uc_arc_prod_if< T> >        out;       // samples output
  void root_function();
  SC_CTOR(adaptiveLPF) {
     SC_THREAD(root_function);
  }
private:
     FPTR  fun;
     T input_var[N], output_var[N];
     ...
};
```

where the root function has the next implementation:

```
template<class T, unsigned int N>
void adaptiveLPF ::root_function() {
  while(true) {
     in_fun->read(fun);                  // functionality adaptation
     in->full_read(input_var);           // input samples
     fun(N, in_var, out_var);            // computation
     out->full_write(output_var);        // output samples
  }
}
```

Then, each function pointer transferred through the adaptation input can address a function of the library of LPF filter functions declared in section 9.4. If this library is extended or enhanced with a new one, then, this template can still be used, whenever the new filter functions fulfil the same prototype.

An extract of the code in charge of declaring and instancing the fAP and the channels which connect it with the rest of the specification is shown:

```
    …
    // channel declaration and instances
    uc_arc<1,1,FPTR >      fun_arc("fun_arc");
    uc_arc<N,N,float>      in_arc("in_arc");
    uc_arc<N,N,float>      out_arc("out_arc");
    // fAP declaration and instance
    adaptiveLPF<float,1000>   adaptLPF;
    // fAP binding
```

adaptLPF.in_fun(fun_arc);

adaptLPF.in(in_arc);

adaptLPF.out(out_arc);

…

Notice that this adaptive process is written under a *HetSC* SDF MoC. The *HetSC* specification of the fAP uses accesses to *uc_arc* channels with equal input and output rates (1 for the adaptation input, which reads the function pointer of the next function to be evaluated, and N for the input and output arcs). Thus, in this example, the fAP is actually a node of a *HetSC* SDF graph. In this case, the root function does not have to include loops for reading input data and write output data thanks to the *full_read* and *full_write* methods provided by the *uc_arc*, which perform the transfer in terms of the consuming and producing rates.

10.3.4 Untimed Self-Adaptive Process

Once the specification of untimed paHAPs, mHAPs and fHAPs has been shown, the specification of self-adaptive processes in untimed domain is depicted in Figure 38b. As in ForSyDe, the sAP in *HetSC*, that is, the sHAP consists in a process network composition. Such composition includes one of the previously shown HAPs and an additional SystemC process, the feedback or *agent* process (aP), in charge of deciding the adaptation. The agent process reads through feedback channels the necessary information to calculate the data for the adaptation input. In the general case, this information can be taken from the input and from the output of the HAP. In the Figure 38b, the adaptive process is a mHAP. Then, the feedback/agent process generates a single token which represents the next working mode, transferred through a channel proper of an untimed MoC.

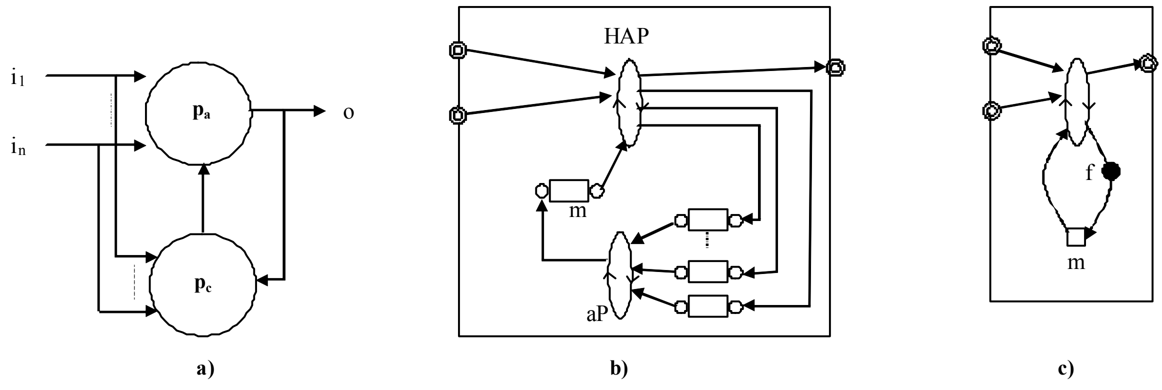


Figure 38. Specification in *HetSC* of the self-Adaptive Process.

In the case of Figure 38b, notice that the HAP has to retransmit the feed-backed input and output data tokens. For it, it also needs a new channel instance for each feed-backed input and output. This is because in the untimed domains of *HetSC* reads are consuming. However, in the correspondent ForSyDe metamodel, in Figure 38a, the same ForSyDe signal can be "sampled" by the correspondent untimed ForSyDe process as many times as desired within the same ForSyDe evaluation cycle.

A simplification where the feedback process disappears has been represented in Figure 38c. There, the evaluation of the AP generates, as well as the outputs, the next adaptation data. The black point represents the function in charge of calculating the next adaptation variable (a mode variable in the case of the Figure 38c). As it was seen in section 9.6, the cost of this structure in a software implementation was that the calculation of the next adaptation variable is sequentially added to the output evaluation time. However, this structure is efficient in terms of saving inter-process communication. In adaptive functionalities where there is no dependency between output calculation and the calculation of the next adaptation data, the Figure 38b is able to capture such concurrency and facilitates later the generation of concurrent SW. Its cost is the necessary inter-process communication that is related to the time consumption of system calls for the inter-thread communication associated to the software implementation, shown in section 9.6.

When to use it:

The sAP is useful when the adaptive object also includes the functionality in charge of deciding next adaptation. This matches the most extended notion of adaptive system in a software context.

10.4 *Patterns for Synchronous HAPs*

From the synchronous HAP structures shown in section 10.2.2 and the description of the parameter-based, mode-based and function-based patterns applied to untimed HAPs shown in section 10.3, the user can foresee a regular building of the different types of general patterns. Because of this, these sections will limit to present the pattern associated to each case and to highlight the main differences with respect to previous patterns.

10.4.1 Clocked Synchronous HAPs (CS-HAPs)

CS-paHAP:

```
class user_CS_paHAP :  public sc_module  {
public:
   sc_in <bool >                      in_clk;
   sc_in <T_1 >                       in_1;
   ...
   sc_in <T_N >                       in_N;
   sc_in <T_P >                       in_param;
   sc_out < T_O  >                    out;
   void root_function();
   SC_CTOR(user_CS_paHAP) {
       SC_THREAD(root_function);
       sensitive << in_clk.pos();
   }
private:
    T_P  param;
    T_O  output_var;
    void  f_param(...);
     ...
};
```

The clocked synchronous patterns proposed include SystemC ports (*sc_port*) enclosing SystemC signal interfaces (*sc_signal_in_if*, *sc_signal_out_if*,…), or equivalently their related specialized ports (*sc_in* and *sc_out*). An input port common to all the CS templates and already graphically reflected in Figure 33 is the clock input port, of *sc_in<bool>* type. The rest of the template declaration is similar to the untimed paHAP declaration.

As it was mentioned, in section 10.2.2, the input and output partitions of CS HAPs are 1. This is coherent with the usage of signal input and output ports, which only let read/write a data unit at each clock cycle. An advantage is that the input can be sampled as many times as desired.

The following pattern for the coding of the *root_function* is provided for the case of handling single adaptation parameter of *T_P* type.

```
void user_CS_paHAP::root_function() {
  while(true) {
     wait();
     param = in_param->read();
     f_param(param, in1->read(), ..., inN->read(), output_var);
     out->write(output_var);
  } // end loop
} // end root function
```

The *f_param* method encloses a parameterizable computation structure. As in the untimed paHAP case, such computation could be enclosed also as a function external to the *user_CS_paHAP* class or unfolded in a sequence of statements within the *root_function*. The usage of buffering variables for the inputs has been omitted since, at a given cycle, they can be read as many times as wanted. Another difference with regard to the untimed patterns is the *wait* statement in the *root_function* and its correspondent *sensitive* statement in the HAP declaration. The wait statement blocks functionality till the next cycle. Summarizing, the *CS_paHAP* is basically another clocked synchronous process, with a style closer to HW description, but admitting a SW implementation, as it will be shown in section 6.

If the HAP template has to handle several parameters, then T_P can be a complex type (i.e., an array). An alternative is to use the following pattern for the root function:

```
void user_CS_mHAP::root_function() {
  unsigned int i;
  while(true) {
     for(unsigned int i=0;i<N_p;i++) {
        wait();
        param[i] = in_param->read();
     }
     f1(param, in1->read(), ..., inN->read(), output_var);
     out->write(output_var);
```

```
    } // end loop
} // end root function
```

In this case, several *N_p* cycles are taken in order to get the adaptation parameters. The last read is used to compute and write to the output. This pattern admits other possibilities by including wait statements, for instance, between the last read and computation and between computation and output write.

CS-mHAP:

```
class user_CS_mHAP : public sc_module {
public:
  sc_in <bool >                    in_clk;
  sc_in <T_1 >                     in_1;
  …
  sc_in <T_N >                     in_N;
  sc_in <COUNTABLE_TYPE >         in_mode;
  sc_out < T_O  >                 out;
  void root_function();
  SC_CTOR(user_CS_mHAP) {
     SC_THREAD(root_function);
     sensitive << in_clk.pos();
  }
private:
   T_O  output_var;
   void  f1(...);
    …
   void  fn(...);
  …
};
```

Where the *root_function* can be coded as follows:

```
void user_CS_mHAP::root_function() {
  while(true) {
     wait();
     switch(in_mode->read()) {
        case 0: // mode 0
        // read input samples
        f1(in1->read(), ..., inN->read(), output_var);
        out->write(output_var);
        break;
      case 1: // mode 1
        // statement 1
        // statement 2
        // statement n
        break;
        ...
    } // end switch
  } // end loop
} // end root function
```

CS-fHAP:

```
class user_CS_fHAP :  public sc_module  {
public:
  sc_in <bool >              in_clk;
  sc_in <T_1 >               in_1;
  ...
  sc_in <T_N >               in_N;
  sc_in <FPTR >              in_function;
  sc_out < T_O  >            out;
  void root_function();
  SC_CTOR(user_CS_fHAP) {
    SC_THREAD(root_function);
    sensitive << in_clk.pos();
  }
private:
   T_O  output_var;
   FPTR  fp
   ...
};
```

Where the root function can be coded as follows:

```
void user_fHAP::root_function() {
  while(true) {
    wait();
    fp= in_function->read();     // function adaptation
    fp( in1->read(),
       in2->read(),...,
       inN->read(),
       output_var);              // evaluation
    out->write(output);          // output samples
  }
}
```

Again, as it happened with other CS HAP patterns, the code of the root function gets a bit simpler due to the 1 partition of each input and output.

CS-sHAP:

Regarding the specification of self-adaptive CS-HAPs, the most important differences have to do with the way the adaptive and agent functionality can be composed in the CS domain. Figure 39 shows that the composition is simplified when the decision functionality, in charge of calculating the adaptation parameters/mode/function, and the adaptive functionality are split into different process, the agent process (aP) and the adaptive process (HAP). In effect, as Figure 39b shows, the composition gets simpler than in the untimed case (Figure 38b), since the adaptive process does not need to retransmit the data units, but they can be directly sampled from the input and output signal ports. SystemC signal channels are only necessary for transferring the parameters/mode/function from the agent process to the adaptive process.

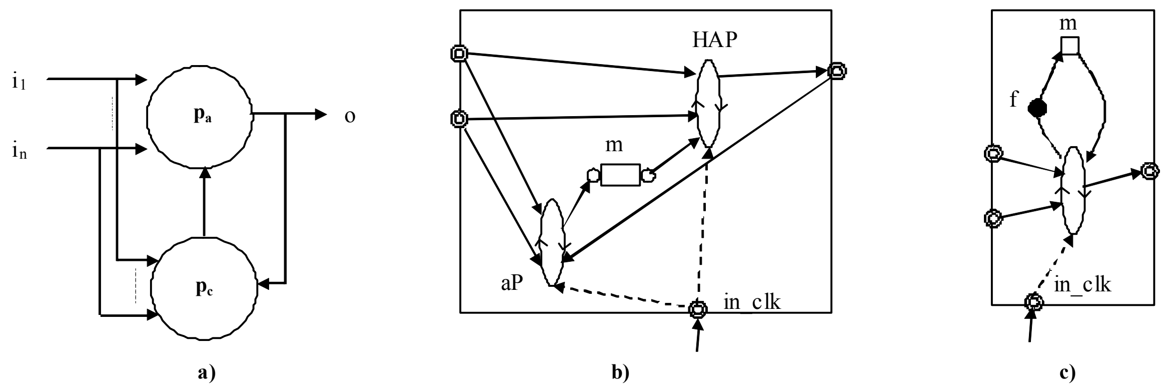


Figure 39. Self-Adaptive Clocked-Synchronous HAP.

Notice that in the scheme of Figure 39, the aP decides the adaptation for the next cycle, while the HAP computation refers to the current cycle. If the adaptation has to refer to the current cycle, the scheme of Figure 38c is recommended, where the decision is based on a functionality which is computed before adaptive computation.

10.4.2 Synchronous Reactive HAPs (SR-HAPs)

SR-paHAP:

```
class user_SR_paHAP :  public sc_module  {
public:
   sc_port <uc_SR_in_if<T_1> >       in_1;
   sc_port <uc_SR_in_if<T_2> >       in_2;
   …
   sc_port <uc_SR_in_if<T_N> >       in_N;
   sc_port <uc_SR_in_if<T_P> >       in_param;
   sc_out<T_O>                       out;

   void root_function();
   SC_CTOR(user_SR_paHAP) {
      SC_THREAD(root_function);
      sensitive << in_1;
      sensitive << in_2;
      …
      sensitive << in_N;
      sensitive << in_param;
   }
private:
    T_P param;
    T_O output_var;
     …
};
```

The synchronous reactive patterns proposed use the specific input SR port of *HetSC*, necessary to access through modules to *uc_SR* input channels able to trigger the HAP. As can be appreciated, there is no kind of clock input. The SR-HAP is a reactive process (RP in the SR domain of *HetSC*). Therefore, each input is able to trigger process computation at a given time slot. The same happens with the adaptation input, which, at a given slot, can be the only input triggering process. In such a case, as mentioned in section 10.2.2, only an adaptation takes place at the given slot. That is, the SR-HAP does not make any regular computation at such slot. Moreover, several consecutive adaptations at consecutive slots are possible.

The following pattern for the coding of the *root_function* is provided for the case of handling single adaptation parameter of *T_P* type.

```
void user_SR_paHAP::root_function() {
  while(true) {
     SR_WAIT_POINT();
     if(in_param->written()) {
       param = in_param->read();
     }
     // check trigger combination
     if(in_1->written() && in_2->written() && ...&& in_N->written()) {
       f_param_N(param, in1->read(), ..., inN->read(), output_var);
     }
     ...// other trigger combinations
     }  else if(in_1->written()) {
       f_param_1(param, in1->read(), output_var);
     }
     ...
     } else if(in_N->written()) {
       f_param_N(param, inN->read(), output_var);
     }
     out->write(output_var);
   } // end loop
} // end root function
```

As synchronous HAPs, and as happened with CS-HAPs, the input and output partitions of SR HAPs are 1. This is coherent with the access to *uc_SR* channels, which only let read/write a data unit at each clock cycle. In a similar way as CS-HAPs, *uc_SR* channels responsible for the triggering of the reactive process can sampled as many times as desired. This helps to simplify the coding of the *root_function*.

Notice that a non-strict reactive HAP requires several functional relations (represented as several function calls in the pattern) to cover all the trigger combinations. This code gets simpler in those cases where trigger combinations can be merged, and if old input values can be used for the coding of the non-strict reactive HAP. The, following pattern for the root function reflects this extreme:

```
void user_SR_paHAP::root_function() {
  while(true) {
     SR_WAIT_POINT();
     if(in_param->written()) {
       param = in_param->read();
```

```
        }
        // update triggered inputs
        if(in_1->written()) {
          in_var1 =  in1->read();
        }
        if(in_2->written()) {
          in_var2 =  in2->read();
        }
        ...
        if(in_N->written()) {
          in_varN =  inN->read();
        }
        f_param_N(param, in_var1, in_var2, ...in_varN, output_var);
        out->write(output_var);
      } // end loop
    } // end root function
```

An equivalent code to this could be done by directly accessing input ports, but in such a case the SR *HetSC* check CHECK_OLD_VALUE_READ_UC_SR_CHANNEL must be disabled.

The SR-HAP can be also specified as a strict reactive process. This will be shown in for the case of the SR-mHAP. Once the SR-paHAP has been shown, the rest of SR-HAP will follow a similar variants as in the different cases of CS-HAPs.

SR-mHAP:

```
enum mode_t {mode1, mode2, ...modeN};
class user_SR_mHAP :  public sc_module  {
public:
  sc_port <uc_SR_in_if<T_1> >          in_1;
  sc_port <uc_SR_in_if<T_2> >          in_2;
  ...
  sc_port <uc_SR_in_if<T_N> >          in_N;
  sc_port <uc_SR_in_if<mode_t> >       in_mode;
  sc_out<T_O>                          out;

  void root_function();
  SC_CTOR(user_SR_mHAP) {
     SC_THREAD(root_function);
     sensitive << in_1;
```

```
      sensitive << in_2;
      ...
      sensitive << in_N;
      sensitive << in_m;
   }
private:
    ...
};
```

```
void user_SR_mHAP::root_function() {
  while(true) {
     SR_STRICT_WAIT();
     switch(in_mode->read()) {
       case mode1:
         out->write( f1(in_1->read(), in_2->read(), ...,in_N->read()) );
         break;
       case mode2:
         out->write( f2(in_1->read(), in_2->read(), ...,in_N->read()) );
         break;
       ...
       case modeN:
         out->write( f2(in_1->read(), in_2->read(), ...,in_varN->read()) );
         break;
     } // end switch
   } // end loop
} // end root function
```

SR-fHAP:

```
class user_SR_fHAP :  public sc_module  {
public:
   sc_port <uc_SR_in_if<T_1> >          in_1;
   sc_port <uc_SR_in_if<T_2> >          in_2;
   ...
   sc_port <uc_SR_in_if<T_N> >          in_N;
   sc_port <uc_SR_in_if<FPTR >          in_function;
   sc_out<T_O>                          out;
```

```

    void root_function();
    SC_CTOR(user_SR_fHAP) {
       SC_THREAD(root_function);
       sensitive << in_1;
       sensitive << in_2;
       …
       sensitive << in_N;
       sensitive << in_f;
    }
  private:
     FPTR  fp;
     …
  };
```

```
  void user_SR_fHAP::root_function() {
   while(true) {
     SR_STRICT_WAIT();
     fp= in_function->read();
     out->write( fp(in_1->read(), in_2->read(), …,in_varN->read()) );
    } // end loop
  } // end root function
```

The implementation of the *root_function* for the non-strict case of the SR-fHAP must rely on the old values of the inputs, since, otherwise, it would be necessary for the agent process to predict the type of the next trigger combination in order to pass the suitable m-inputs function. (with 1≤m≤N). Instead, it is assumed that the adaptation input will transfer N-input functions. Then the root function would be coded as follows.

```
   void user_SR_fHAP::root_function() {
    while(true) {
      SR_WAIT_POINT();
       if(in_function->written()) {
          fp = in_function->read();
       }
       // update triggered inputs
       if(in_1->written()) {
          in_var1  =  in1->read();
       }
```

```
        if(in_2->written()) {
          in_var2 =  in2->read();
        }
        ...
        if(in_N->written()) {
          in_varN =  inN->read();
        }
        out->write( fp(in_var1, in_var2, ...,in_varN) );
      } // end loop
    } // end root function
```

Notice that in this patters it is assumed that the adaptation functions pass the output value as the return value. The patterns are flexible in this sense. Notice also that, in case of having to perform adaptation and computation at the same s*lot*, adaptation takes place before computation. Therefore, SR-HAP, CS-HAP and untimed-HAP patterns work in the same way in this sense.

10.5 *Patterns for process-based HAPs*

This section presents how the different types of prAPs can be specified in SystemC. This is considered a special case since it is a more complex concept which runs into some limitations of SystemC for its specification. It is also special since it can lead to the interesting feature of an adaptation involving a dynamic change of the current domain.

10.5.1 Process-based Adaptive Process

The prHAP enables the access of a different set of input channel accesses (potentially of different types and with different partitions) after an adaptation. This follows the ForSyDe formalism for the prAP, which enables the adaptation of process interface (as well as process functionality).

In SystemC, a kind of “process interface” is not defined, however it can be considered to be the set of input accesses performed by the processes (focusing on the input interface), and which in a prAP can vary during run time. In the case of wrapping the HAP within a module (as most of patterns shown till here) it also means the enabling changing the access to different input ports.

An immediate approach to the ForSyDe formal concept of prHAP could be thought as an adaptation of the process interface by substitution, which would enable the reuse of specification resources. For instance, it could be thought that since some of the channels/ports will not be accessed after a given adaptation, they could be removed and created or recovered and bound again during simulation time if they have to be used again.

However, creation and destruction of such several SystemC elements, usually associated to process interface, it is not possible during the simulation. In SystemC, channels and ports cannot be dynamically (during simulation time) created and destroyed, but they are declared, instanced and bound at elaboration time (before simulation start). An apparent alternative could seem to perform the adaptation at a module instance level. That is, dynamically change the module instance addressed and representing the prHAP and thus redoing its associated bindings. However, again, SystemC handle module creation and binding in a static way (SystemC module creation cannot be done within a process, thus at simulation time).

Therefore, the interface of the prAP in terms of the whole set of channels, ports and exports accessed from the prHAP during the whole simulation has to be fixed in advance. In any case, SystemC can be used for the specification of prHAPs, since the SystemC specification is not the actual implementation. Quite the opposite, having all possible interfaces statically defined at the beginning of the simulation can help to left clear the boundaries of the adaptation and provide information to the implementation steps for getting and optimum result.

Therefore, the specification of prHAPs in SystemC is (and has to be) specified as a kind of adaptation by selection **[D11A]**. Figure 40b sketches a feasible general structure for the *HetSC* specification of a process-based Adaptive Process.

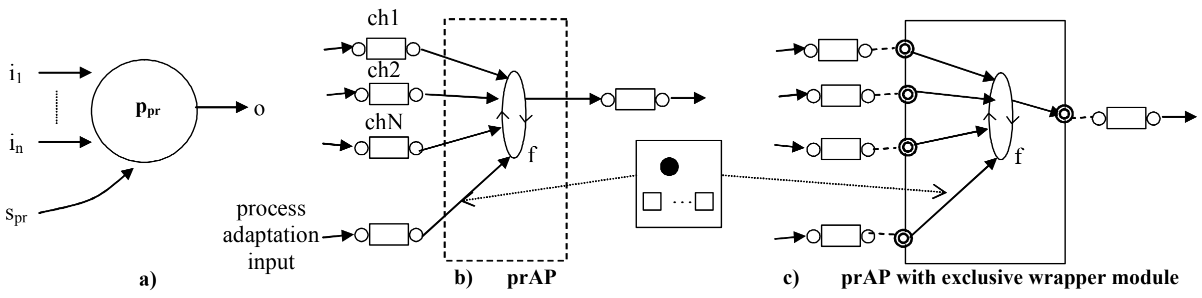


Figure 40. Specification in *HetSC* of the process-based Adaptive Process.

In this scheme, all the alternative sets of channels accessed after each adaptation have to be instanced and bound at elaboration time. Then, each adaptation defines the set of channels accessed. Then, it follows a similar idea as that developed for the SW implementation of a prHAP.

In the case of using an exclusive wrapper module, similarly, ports are created and bound at elaboration time. Therefore, the port interface of the prAP is fixed at elaboration time and must cope with any access combination defined by each adaptation.

In some sense, the prAP means a generalization of the patterns previously shown for the case in which an adaptation can mean not accessing every input port. This is specially useful in untimed HAPs, where, without the ability to select the set of input ports/channels and each input partition after one adaptation, the untimed HAP would be tied to scheme where would be reading every input at fixed rates even if there is nothing useful to read.

The following pattern shows this case for two inputs *prHAP* which receives through the adaptation input a structure with a mode parameter and the input partition for each input. Thus, this is an extension of the mHAP case, which depending on the mode, reads different inputs with a different input partition.

```
enum mode_t {mode1, mode2,...};
struct adapt_info_t {
  mode_t mode;
   unsigned int partition[N]; // partition of each input
};
class user_prHAP : public sc_module {
public:
    sc_port <IF<T_1> >                    in_1;
    ...
    sc_port <IF<T_N> >                    in_N;
    sc_port <IF<adapt_info_t> >           in_process;
    sc_port < IF<T_O>  >                  out;
    void root_function();
    SC_CTOR(user_prHAP) {
      SC_THREAD(root_function);
    }
  private:
     adat_info_t     adapt_info;
     T_1 input_var1[MAX_N_in1], ..., T_N  input_varN[MAX_N_inN];
     T_O     output_var[N_out];
     void  f1(...);
      ...
     void  fn(...);
  ...
  };
```

```
  void user_mHAP::root_function() {
    unsigned int i;
    while(true) {
      adapt_info = in_process->read();
      switch(adapt_info.mode) {
         case 0: // mode 0
           // for instance,  reads only from the first input
           for(i=0;i< adapt_info.partition[0];i++) in_1->read(input_var[0]);
           f1(input_var1, output_var);
           for(i=0;i<N_out;i++) out->write(output_var[i]);
           break:
         case 1:
           // for instance,  reads only from the first input and N-th input
           for(i=0;i< adapt_info.partition[0];i++) in_1->read(input_var[0]);
           for(i=0;i< adapt_info.partition[0];i++) in_1->read(input_var[0]);.
           f2(input_var1, input_varN, output_var);
           for(i=0;i<N_out;i++) out->write(output_var[i]);
```

```
          break:
        case 2: // mode 2
          …
          break:
        …
      } // end switch
    } // end loop
  // end
```

This pattern can be obviously particularized to more simple cases, like for instance, assuming that the same partition will be applied to every input at each adaptation.

Synchronous prHAPs eliminates the factor of input partitions from consideration, since, as mentioned, they are 1 for every input. Therefore, the patterns already defined for synchronous HAPs already cover the prHAP by selection, since there is no need to consider a case where partition is transferred through the adaptation input. For instance, a mHAP can involved for different modes reading different inputs. This will not involve the process blocking before completing its computation for the rest of the *cycle*/*slot* in any of the synchronous cases, CS and SR.

An interesting case is that of the heterogeneous prAPs, explained in the following section.

10.5.2 Heterogeneous prHAPs

Heterogeneous prHAPs are those prHAPs where adaptation involves change on the model of computation. More specifically, the adaptation involves accessing different types of input ports, and different paths within the reactive process, involving a change on the domain (model of computation) after an adaptation.

When to use it

A potential application of this kind of HAPs is foreseen in less explored fields. For instance Heterogeneous prHAPs could be used to specify, model, analyze and design systems which capabilities for HW/SW context change. That is, systems where DRHW enable the same piece of silicon to compute some software functionality at some time, while some hardware functionality at other time. This systems would have an additional flexibility degree for its design with respect to any software concurrent system (supporting SW context changes), or DRHW (supporting HW context changes).

Design Pattern

Again, as untimed and synchronous prAPs, the specification requires static instantiation of the input ports and channels accessed. The process will count with different execution paths depending on the type of synchronization done. The adaptation will take place considering the current domain the prAP is computing in. That is, if the prAP is within an untimed domain, there must be a blocking synchronization with an "untimed" input channel (i.e. of fifo type), to produce de adaptation. If the current domain were the CS one, the following clock event will determine the next adaptation,

Example:

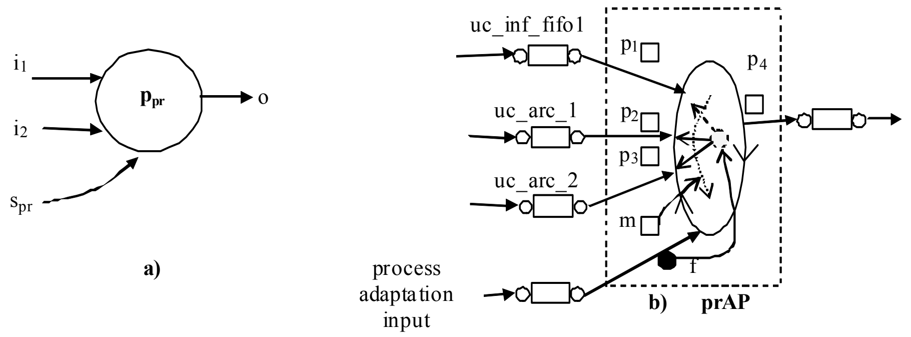


Figure 41. This prAP changes the MoC after the adaptation.

The example is represented in Figure 41. The adaptation input transfers tokens of *adapt_info_type* type, a struct whose members are a function pointer, a set of parameters which set the input partition (number of tokens to read from each input), and the output partition (number of tokens generated for the output), and a mode variable which defines which channels are read and written in the computation. Actually, this example reflects the building of a prHAP as a combination of the paHAP, mHAP and fHAP, where the adaptation information has been multiplexed in the same adaptation inputs through the *adapt_info_type*. Other equivalent combinations can be found. The question is that each adaptation can modify which input and output channels are accessed and the number of tokens read from each input channel and written to each output channel in the next evaluation. Moreover, the MoC varies in this example since one evaluation the process can perform an access to one type of channel (i.e. to some *uc_inf_fifo* channels) while in the next evaluation, it accesses other type of channels (i.e. to some *uc_arc* channels) are done. Thus, while in the former case, the prHAP can be part of a Kahn process network (KPN MoC), in the second case can be considered as a node of SDF domain.

Following, the code of a simpler example, based on the example of section 9.5 is provided. This example presents combines a change of the input and output partition and of the function to be executed.

```
template<class T> void (*FPTR) (T* in,T* out, unsigned int N);
struct adapt_struct {
  FPTR fun;
  unsigned int N;
};
template<class T>
class adaptiveLPF : public sc_module {
public:
  sc_port <sc_fifo_blocking_in_if<adapt_struct> >   in_as;  // adaptation input
  sc_port < sc_fifo_blocking_in_if <T> >             in;    // samples input
  sc_port < sc_fifo_blocking_in_if <T> >            out;  // samples output
  void root_function();
  SC_CTOR(adaptiveLPF) {
    SC_THREAD(root_function);
  }
private:
     adapt_struct  adapt_var;
     T  *input_var,* output_var;
     ...
};
```

where the root function has the next implementation:

```
template<class T>
void adaptiveLPF ::root_function() {
  while(true) {
     as->read(adapt_var);                                          // functionality adaptation
     reallocate(input_var, adapt_var.N*sizeof(T));                 // reallocates room
     reallocate(output_var, adapt_var.N*sizeof(T));                // for inner vars
     for(i=0;i< adapt_var.N;i++)  in_fifo->read(input_var[i]);     // input samples
     adapt_var->fun(adapt_var.N, input_var, output_var,);          // computation
     for(i=0;i< adapt_var.N;i++)  out_fifo->write(output_var[i]);  // output samples
  }
}
```

where the *reallocate(T* p, unsigned int M)* function template ensures the allocation of M bytes addressed from *p* pointer. Depending on the dependencies among input and output tokens, the reallocation can be optimized. For instance, let assume that the functions passed produce one token per each input token. Then, the root function can be written as follows:

```
void adaptiveLPF ::root_function() {

  input_var = new T;

  output_var = new T;
  while(true) {
     as->read(adapt_var);                        // functionality adaptation
     for(i=0;i< adapt_var.N;i++) {
       in_fifo->full_read(input_var);              // input sample
       adapt_var->fun(input_var, output_var );   // computation
       out_fifo->write(output_var[i]);             // output sample
     }
  }
}
```

These degrees of flexibility keep coherence with all the possibilities contemplated by the prAP in **[D11A]**. Since the prAP can change process constructors in the adaptation, the input and output partition, the number of input and output signals and even the MoC could change after the adaptation.

11. *HetSC* Templates for Adaptive Processes

In the previous section, guidelines and examples for the specification of APs in SystemC have been given. Such guidelines include design patterns for supporting an abstract modelling of adaptive processes, which in time has a direct implementation as adaptive software by means of the *SWGen* methodology.

Additionally, the *HetSC* library has been extended to provide a set of SystemC templates (and their related guidelines) for the specification of HAPs. These are the *HetSC* Adaptive Process (HAP) templates. Using these templates provides the SystemC user:

- A mean to capture an adaptive process quickly and in a compact way. The user does not need to code the adaptive class under the patterns defined in section
- Fast reuse of adaptive sequential C/C++ code with a clean decoupling between the C/C++ part and the SystemC constructs, which provide concurrency, heterogeneity and adaptivity.
- Support of eSW implementation by means of *SWGen*.

HetSC has been extended with the HAP templates summarized by Table 2:

Domain	**HAP Template (class name)**	**Description**		
		1-output, Tout type [=Tin] 1-adapt-in, Tada type [=Tin]		
		inputs	**partition**	**Additional features**
Untimed	paAP	1	ri, ai, o,	
	paAPn	N	ri,ai,o	Let state preservation
	mAP	1	ri,o	
	fAP	1	ri,o	
	prAP	1	ri,o	Adapt ri at each adaption
Synchronous Reactive	sr_mAP1	1	1,1,1	
	sr_mAP2	2	1,1,1	Tin1, Tin2
Clocked Synchronous	cs_mAP1	1	1,1,1	1-input
	cs_mAP2	2	1,1,1	Tin1, Tin2
	cs_mAPn	N	1,1,1	Same input type

Table 2. HAP templates included in the *HetSC* v1.3.

The usage of the HAP templates requires an identification of the specific adaptation patterns comprised in Table 2. As can be seen, these templates work for one output and one adaptation input port. Specific data types can be transferred for each one, while the default type is the same as the regular input. Then different templates are provided depending on the number of inputs, type of adaptation information and domain. Two input templates can deal with different input data types, while N-input templates require handling the same input data type.

Untimed HAP templates let in any case specify different partitions for each regular input, for the adaptation input and for the output. These partitions are fixed for the adaptation (thus mostly suitable for specifications under a static data flow domain). An exception is the prAP case, since the prAP template let changing the partition among adaptations.

For cases not covered by this table (for instance, an adaptive process of two inputs handling different data types) the user has to rely on the guidelines and patterns provided in the previous chapter. HAP templates have also some requirements on the function prototypes passed to the HAP templates which can be easily overcome by means of wrapping functions, as will be explained.

Following, the structure of the documentation of the HAP templates will be explained by means of one representative template. This includes showing the template declaration as it will be found in the *HetSC* associated documentation, its usage and results of a related example. Additional details will be given for other HAP templates. Some details about the internal implementation of the HAP templates will be given, which is actually not part of the *HetSC* documentation of the templates.

11.1 *Using a HAP template*

The first information which is given to the user about a HAP template is its public declaration. For instance, for an untimed parameter-based HAP of *N_inputs* inputs, the *paAPn template* is available. Its declaration is as follows:

```
template<class FPTR, class Tin, unsigned int N_inputs=1,
              class Tout=Tin, class Tpar=Tin, , class Tout=Tin >
class paAPn :  public sc_module  {
public:
  sc_port <sc_fifo_blocking_in_if<Tpar> >        in_param;
  sc_port <sc_fifo_blocking_in_if<Tin> >         in[N_inputs];
  sc_port < sc_fifo_blocking_out_if< Tout> >     out;
  paAPn(sc_module_name name,
            FPTR fp,
            unsigned int  Nin[N_inputs],
            unsigned int  Nparams=1,
            Ts              *state= NULL);
  SC_HAS_PROCESS(paAP);
private:
     ...
};
```

Notice that the user actually does not need to know the private members of the class, not even defining this class. What the user needs to know this template exists, its limitations and applicability and public declaration. This template enables the specification of untimed (static data flow) parameter-based APs of several inputs, with recovering of state among adaptations.

The template also assumes a structure for the parametric function which is associated to it as well as for the input data which process and output data which returns. Specifically, for the case of the *paAPn* template, the structure of the function is assumed to be the following one:

```
//          void param_f_name()(
//                      unsigned int   Nin[],
//                      Tin            **in,
//                      unsigned int   Nout,
//                      Tout           *out,
//                      unsigned int   Nad,
//                      Tparam         *in_param,
//                      Ts             *in_state);
```

That is, a function which receives as parameters an array Nin[N_inputs], with the partition of the regular inputs; a pointer to the array of input variables (which are in time arrays for the each specific input partition); the output partition; a pointer to the output variable; the partition of the adaptation input; a pointer to the input parameter; and finally, a pointer to a state variable.

Following the usage of this template is shown with an example available in the *HetSC* v1.3 library. First, a snippet of the *sc_main* function is shown:

```
…
// Initial state for the first call to fir_fil_5_wrap call
float state[5] = {0.0,0.0,0.0,0.0,0.0}; // state variable plus initial state
unsigned int Nin[1] = {100};        // 100 samples from the first input

// paAP declaration and instance
paAPn<FLOAT_FILP, float> ap("ap",
                         fir_fil_5_wrap,
                         Nin,
                         100,  // Nout
                         6,    // a FIR 5th order filter needs 6 parameters!
                         state);
...
```

In this piece of code, an instance named *ap* of the *paAPn* template is being declared and created. As it is shown, although the template supports a very generic use, it can be simply used for instancing a more specific HAP. Specifically, the template is being used to instance an adaptive process which performs a 5th order FIR filtering and which is able to adapt its 6 filtering coefficients (b_k) after every 100 processed samples. The instantiation has been graphically represented in Figure 42, where the paAPn template is used to instance a HAP of a single input (and a single output and adaptation input). All the fifo blocking input and output interfaces handle the same data type. Because of this, the template instance can be in this case very simplified, by selecting *Tin=float*, while the rest of template values remain at their default values (N_inputs=1 and Tout=Tpar=Ts=Tin, that is, float for every of them). The first parameter passed to the template is the function pointer which tells the template the specific prototype of parametric functionality associated to it. In this example, the specific declaration done, was the following one:

typedef void (*FLOAT_FILP) (**unsigned int** *,**float** **,**unsigned int**,**float** *,**unsigned int**,**float** *, **float***);

which, as can be seen, suits to the basic structure of the prototype required by this template.

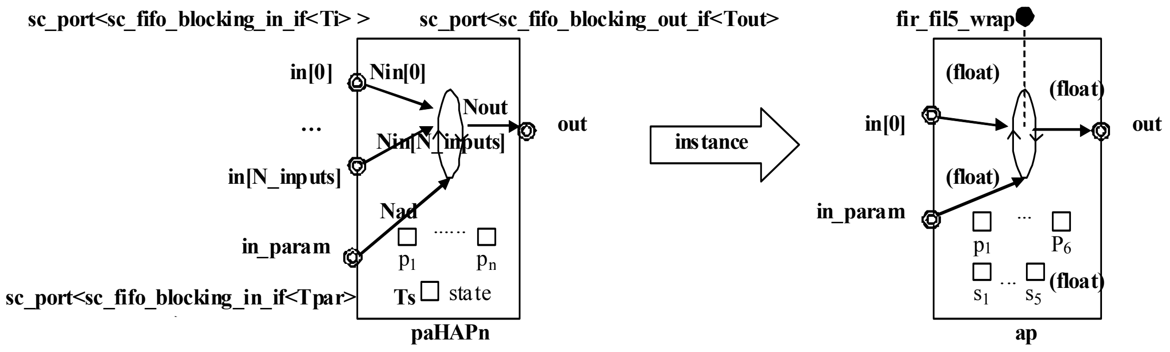


Figure 42. Instantiation of the paHAPn template.

As can be see, the instantiation of the template, and it is represented in the right hand side of Figure 42, makes a static association or binding between the HAP template and the specific functionality, in the shape of a C/C++ function (*fir_fil5_wrap* in this case). In this way, a clean decoupling between the pure functional code and the SystemC code in charge of providing structure, concurrency, heterogeneity and adaptivity to the specification is got.

The HAP template instance has to be bound to the modules of the rest of the specification. In this case, since it is an untimed HAP, the example builds up a bounded Kahn process network. The next code snippet shows the binding statements of the template (also represented in Figure 43), where, *param_fifo, samples_fifo* and *out_fifo* are instances of fifo channels (the example has been checked for both *uc_fifo HetSC* channels and *sc_fifo* standard channels).

```
ap.in_param(param_fifo);
ap.in[0](samples_fifo);
ap.out(out_fifo);
```

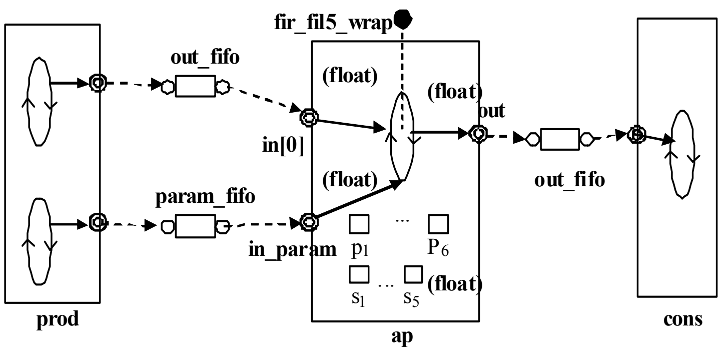


Figure 43. Binding of the paHAPn template instance to the rest of the specification.

In order to illustrate the most complicated usage these templates can require and how it is easily solved, the example assumes that the template is reusing a kind of filters library, which includes a function to perform a 5th order FIR filtering with the following declaration:

```
void fir_fil_5(unsigned int N,
        float *coeff,
        float *output_var,
        float *input_var,
        float state[5]);
```

As can be seen, this function, does not strictly matches the generic function prototype demanded by the HAP template *paAPn*. This is easily solved by means of a *wrapping function*, whose pointer is the parameter finally passed to the HAP template instance. Specifically, for this example, the wrapping function has the following coding:

```
void fir_fil_5_wrap(unsigned int  *Nin,
                    float         **input_var,
                    unsigned int  Nout,
                    float         *output_var,
                    unsigned int  Nparam,
                    float         *coeff,
                    float         *state
) {
  fir_fil_5(*Nin,coeff,output_var,*input_var, state);
}
```

In this case, the wrapping function provides a general prototype to a more specific functionality (since the library function is written to work for a fix partition for the parameters input, and variable, but equal partitions for the input and output). In other cases, the wrapping function can simply have to adapt the type of parameter passing, for instance, convert a pass-by-return-value in a pass-by-pointer, as the HAP template requires for the output.

The HAP template of this example also lets specify state preservation among adaptations. As can be seen, the template instance *ap* has been provided with an initialized state variable (an array of 5 units of Ts=float type), which represents the memory and initial state of the adaptive filter. In this way, the simulation of this example provides the result of Figure 44.

In the upper part of Figure 44, the input to the *ap* instance of the HAP, consisting in Gaussian noise, is represented. What is not represented is the input to the adaption input. The example provides different sets of adaptation parameters, specifically the b_0-b_5 FIR parameters corresponding to a:

- 1st adaptation : a null filter ($b_0 = b_1 = b_2 = b_3 = b_4 = b_5 = 0.0$)
- 2nd adaptation : a bypass filter ($b_0 = 1.0$, $b_1 = b_2 = b_3 = b_4 = b_5 = 0.0$)
- 3rd adaptation : a LPF filter of fc=0.5

- 4th adaptation : a LPF filter of fc=0.2
- 5th adaptation : a LPF filter of fc=0.01

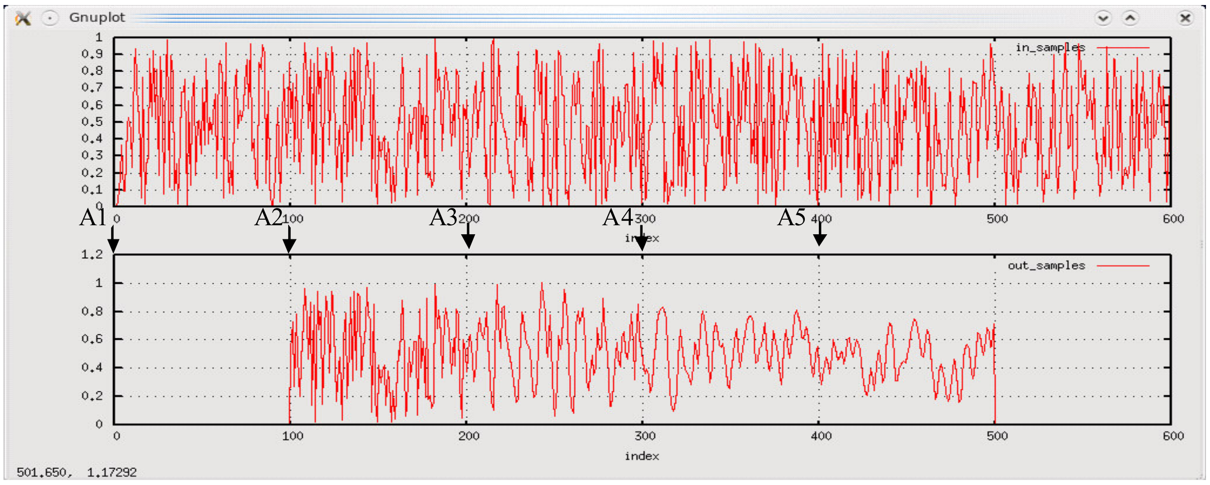


Figure 44. Results of the paAPn example delivered in the *HetSC*v1.3 library.

As an untimed HAP, each adaptation required to pass an update the set of 6 b_k coefficients. It can be appreciated that, after the 5th adaptation, the HAP instance no longer produce output tokens since it does not receives further adaptation parameter, which blocks current computation, this further production of output data.

The effect of the state context can be notice when the results of Figure 44 are contrasted with the results when the *paAPn* template is not provided with a state (shown in Figure 45).

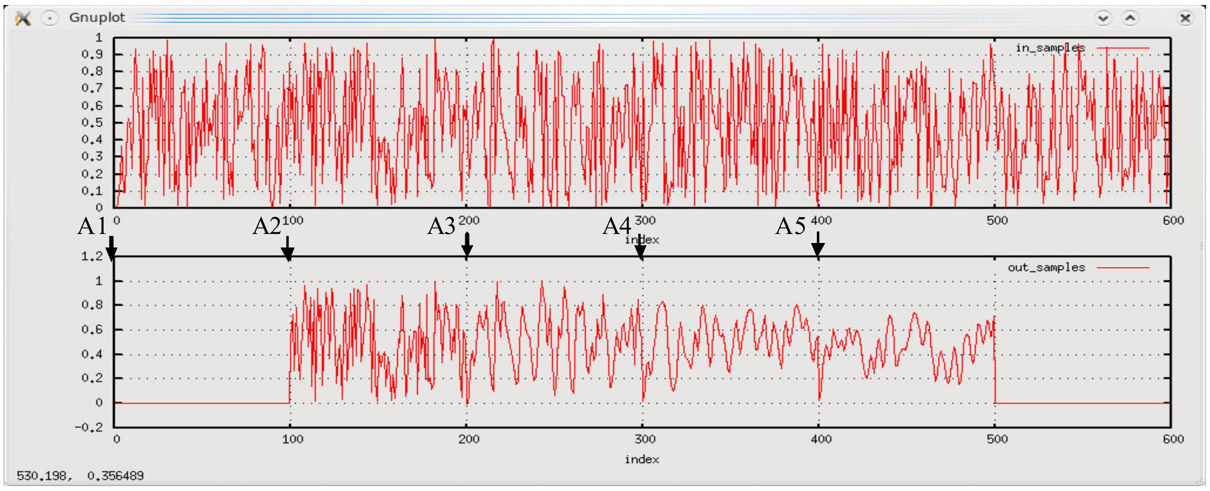


Figure 45. Results of the paAPn example without using state.

In Figure 45, the glitches to 0 after each adaptation contrast with the smooth transitions given in the results of Figure 44. Obviously, the need for state depends on the application. Applications can be found where holding state is not necessary. In such a case, the instance of the HAP template becomes even simpler. In the case of the paAPn template, would be as follows:

```
// paAP declaration and instance
paAPn<FLOAT_FILP, float> ap("ap",
                              fir_fil_5_wrap,
                              Nin,
                                   100,   // Nout
                                   6);    // a FIR 5th order filter needs 6 parameters
```

With this invocation, the template adopts the NULL value for the state variable pointer by default and then, internally, the template will not make any usage of it.

For an example like this, less general HAP template can be used, for instance, the *paAP template*, which works for a single input and do not keep state. Then the usage of the HAP template gets a bit simpler. For instance, the instantiation will only require to pass an *unsigned int* variable to the input partition parameter (instead of an input partition array), and will be only necessary to bind an input port *in* (instead having to bind each *in[i]* port).

The usage of other untimed HAP templates will be, in general, quite similar, with some peculiarities regarding the type of adaptation. For instance, the following code snippet illustrates the instantiation of a single-input untimed mHAP template.

```
typedef  void (*FLOAT_FILP) (unsigned int,float *,unsigned int,float *);

FLOAT_FILP modfun_array[N_MODES] = {fir_fil_5_lpf_wrap, cheb5_hpf_fil_wrap};

mAP<FLOAT_FILP,float> ap("ap", // Name for the paAP instance
                               100,    // # of input samples per adaptation
                               100,    // # of output samples per adaptation
                                2,     // # of modes
                                modfun_array); // mode functions array
```

As can be seen, the instantiation of this template requires the static association of a set of function pointers (in the shape of an array of function pointers) to the template instance. In the most general case, as this example illustrates, these function pointers address wrapping functions. The template also requires to be passed by constructor the number of modes. Then, the mAP template will present an adaptation input port of a fixed data type (which contrast with the parameterizable data type of the parameter input of the paAP templates).

As well as these peculiarities, other differences among HAP templates fit to those remarked in chapter 10 when guidelines comprising the intended usages and shapes of the design patterns have been explained. For instance, the mHAP is suitable when the set of associated functionalities, even sharing the same prototype, enclose different computation structures (in contrast to the paHAP templates, more useful when having the same parameterizable computation structure. For instance, in this mHAP usage example, comprised within the *HetSC*v1.3 library, under the first mode (mode 0) the HAP works as a low-pass filter (LPF), while the second mode (mode 1) performs as a high-pass filter (HPF). Initially, this could have been solved by means of a paHAP if it is decide to apply a FIR filtering in both modes. However, in this example, the LPF filtering is done by means of a FIR filtering, while the HPF filtering is done by means of an IIR (infinite impulse response) filter. The algorithms in each case present several structural differences, and, within the C/C++ library are each one encapsulated in a different function, *fir_fil_5* and *cheb5_hpf_fil*, which in time are encapsulated in their corresponding wrapping functions, *fir_fil_5_lpf_wrap* and *cheb5_hpf_fil_wrap*.

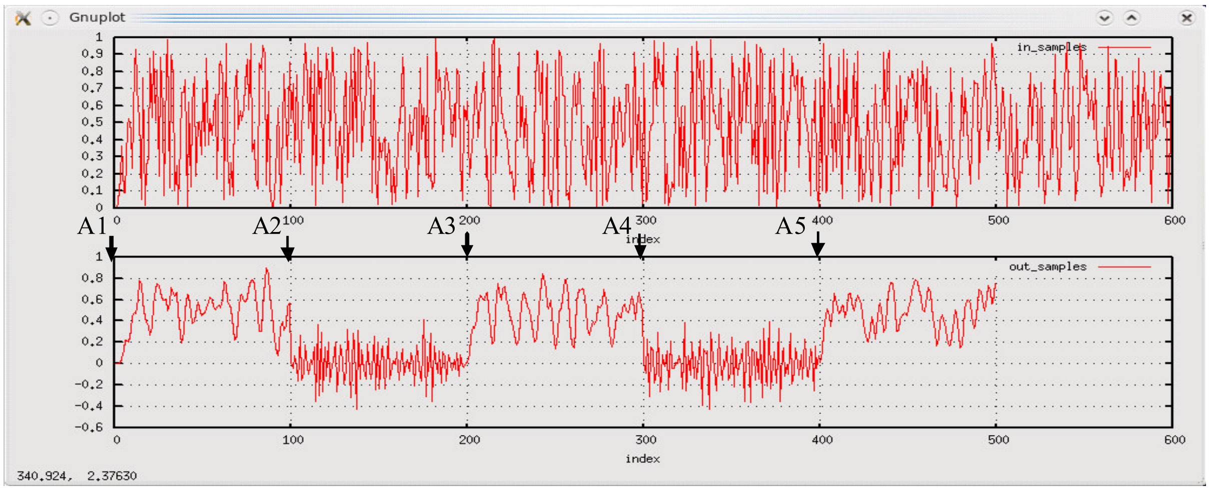


Figure 46. Results in the example of mode adaptation.

Figure 46 represents the results of the example of mHAP usage.

To finish this subsection, the declaration of a synchronous HAP template will be shown and its usage illustrated. Specifically, the declaration of the clocked synchronous template for a case of two inputs of different types, *cs_mAP2*, is shown following:

```
template <class FPTR, class Tin1, class Tin2=Tin1, class Tout=Tin1>
class cs_mAP2 : public sc_module {
public:
  sc_in<bool>             in_clk;
  sc_in<unsigned int>   in_mode; // Input mode
  sc_in<Tin1>             in1;
  sc_in<Tin2>             in2;
  sc_out<Tout>            out;

  void root_function();

  SC_HAS_PROCESS(cs_mAP2);

  cs_mAP2(
    sc_module_name    name,
    unsigned int         N_mod,            // Number of modes
    FPTR                  *mod_fun_array  // array of mode-functions of two inputs
  );
private:
      …
};
```

As can be seen , this template handles a signal ports and a clock input port. The usage and instantiation of the template becomes even simpler that in the untimed case, since the input and output partitions are 1 for all the inputs (regular and adaptation inputs) and for the output. This does not remove the possibility of transferring complex and/or big structures of data among adaptations, since the type transferred at each input is generic. In order to facilitate the specification task, the *HetSC* library has been extended to offer the *uc_burst_t* class. The declaration of this class is as follow:

```
class uc_burst_t {
public:
  T              *data;
  unsigned int  length;
  uc_burst_t();
  ~uc_burst_t();
  uc_burst_t(unsigned int N);
  uc_burst_t(const uc_burst_t<T> &brst); // copy constructor

  // Assignment
  void operator=(uc_burst_t<T> brst);
  bool operator==(uc_burst_t<T> &brst);
};
```

The *uc_burst_t* class comes also with an overload of the *"<<"* operator and of the *sc_trace* function for this class. This class let declare burst of data of N tokens and makes it possible to declare and create an instance of the *cs_mHAP* template like the following one (included also in the examples of the *HetSC*v1.3 library):

```
typedef  uc_burst_t<unsigned char> (*BURST_CRYPTP) (uc_burst_t<unsigned char>,
const char *);

BURST_CRYPTP mod_cryptfun_array[N_MODES]= {crypt1_wrap, crypt2_wrap};

cs_mAP2<BURST_CRYPTP, uc_burst_t<unsigned char> , const char *>
adapt_encrypt("adapt_encrypt",
                2, // # of modes
                mod_cryptfun_array);
```

This sentence declares a clocked synchronous mode-based adaptive encrypter. This adaptive encrypter read at each cycle a burst of data from its first input and the crypto-key from its second input and produces a result by using one of two possible crypto-algorithms. The selection of the specific crypto algorithm applied comes from the value read at the adaptation input at the given cycle.

A code snippet of the binding of this template instance is given as follows:

```
adapt_encrypt.in_clk(sysclk);
adapt_encrypt.in_mode(mode_crypt_sig);
adapt_encrypt.in1(data_crypt_in1_sig);
adapt_encrypt.in2(data_crypt_in2_sig);
adapt_encrypt.out(data_decrypt_in1_sig);
```

where *sysclk* is a *sc_clock* object, *mode_crypt_sig* is a *sc_signal<unsigned int>* channel, *data_crypt_in1_sig* and *data_decrypt_in1_sig* are *sc_signal<uc_burst_t<unsigned char> >* channel, and *data_crypt_in2_sig* is a *sc_signal<const char*>* channel.

Figure 47 represents the example provided by the library.

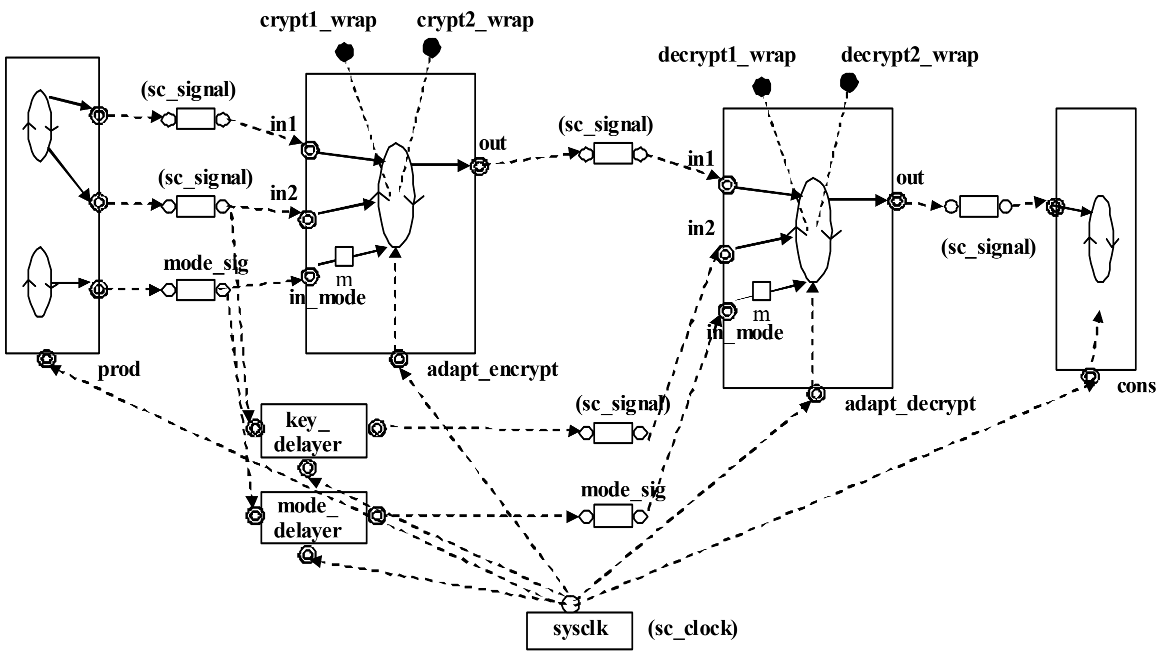


Figure 47. Example of usage of the cs_mAP2 template distributed with the *HetSC*v1.3 library.

The example makes actually two instantiations of the *cs_mHAP2* template, one for instancing an adaptive encrypter and another for instancing an adaptive decrypter. Notice that, in order to transfer the right mode and keys to the adaptive decrypter, the delay modules have been instanced. This would have been actually unnecessary if the specification would have been untimed.

More specifically, the example sends each cycle a burst of characters, alternating the number of characters and the key. That is, the first cycle, *prod* sends 100 characters to be coded with key 1, the second cycle, 50 characters to be coded with key 2, the third cycle, 100 characters to be coded with key 1, and so on (this “synchronization” betweens would not be actually necessary, that is, the third burst could be, for instance, coded with key 1). The example also alternates the modes, but keeping 3 cycles the first mode and then 2 cycles the second mode. However it can be changed, for instance, to have one mode working each consecutive cycle and the example will still work, producing at the *cons* module the same characters stream fed at the input. Obviously, the bursts are appearing at the output at periodic times, marked by the global clock of the specification. A snippet of the log of this example is given:

```
[nando@Becagim5 mAP2]$ make -f Makefile.sys run
./sys/cs_mAP2_ex.x
Instantiating blocking access list.
        HetSC 1.3alpha - Built Nov 20 2008
Copyright (c) 2005-2007 by GIM-UC - All rights reserved
        University of Cantabria, Spain.
          www.teisa.unican.es/HetSC
            SystemC 2.2.0 --- May 23 2008 22:14:00
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED
Mode Adaptation 0 sent.
1st burst&key sent.
in1: "P, r, o, c, l, a, i, m, s,  , t, h, i, s,  , U, n, i, v, e, r, s, a, l,  , D, e, c, l, a, r, a, t, i, o, n,  , o, f,  , H, u, m, a, n,  , R, i, g, h, t, s,  , a, s,  , a,  , c, o, m, m, o, n,  , s, t, a, n, d, a, r, d,  , o, f,  , a, c, h, i, e, v, e, m, e, n, t,  , f, o, r,  , a, l, l,  , p, e, o"
in2: "16_bytes_key_01"
 at 0 s
2nd burst sent.
in1: "p, l, e, s,  , a, n, d,  , a, l, l,  , n, a, t, i, o, n, s, ,,  , t, o,  , t, h, e,  , e, n, d,  , t, h, a, t,  , e, v, e, r, y,  , i, n, d, i, v, i"
in2: "16_bytes_key_02"
 at 0 s
                out: ""
                at 0 s
1st burst&key sent.
in1: "d, u, a, l,  , a, n, d,  , e, v, e, r, y,  , o, r, g, a, n,  , o, f,  , s, o, c, i, e, t, y, ,,  , k, e, e, p, i, n, g,  , t, h, i, s,  , D, e, c, l, a, r, a, t, i, o, n,  , c, o, n, s, t, a, n, t, l, y,  , i, n,  , m, i, n, d, ,,  , s, h, a, l, l,  , s, t, r, i, v, e,  , b, y,  , t, e, a, c, h, i"
in2: "16_bytes_key_01"
 at 10 ms
                out: ""
                at 10 ms
2nd burst sent.
in1: "n, g,  , a, n, d,  , e, d, u, c, a, t, i, o, n,  , t, o,  , p, r, o, m, o, t, e,  , r, e, s, p, e, c, t,  , f, o, r,  , t, h, e, s, e,  , r, i, g, h"
in2: "16_bytes_key_02"
 at 20 ms
Mode Adaptation 1 sent.
```

```
          out: "P, r, o, c, l, a, i, m, s,  , t, h, i, s,  , U, n, i, v, e, r, s, a, l,  , D, e, c, l, a, r, a, t, i, o, n,  , o, f,  , H, u, m, a, n,  , R, i, g, h, t, s,  , a, s,  , a,  , c, o, m, m, o, n,  , s, t, a, n, d, a, r, d,  , o, f,  , a, c, h, i, e, v, e, m, e, n, t,  , f, o, r,  , a, l, l,  , p, e, o"

          at 20 ms

1st burst&key sent.

in1: "t, s,  , a, n, d,  , f, r, e, e, d, o, m, s,  , a, n, d,  , b, y,  , p, r, o, g, r, e, s, s, i, v, e,  , m, e, a, s, u, r, e, s, ,,  , n, a, t, i, o, n, a, l,  , a, n, d,  , i, n, t, e, r, n, a, t, i, o, n, a, l, ,,  , t, o,  , s, e, c, u, r, e,  , t, h, e, i, r,  , u, n, i, v, e, r, s, a, l,  , a
"

in2: "16_bytes_key_01"

 at 30 ms

          out: "p, l, e, s,  , a, n, d,  , a, l, l,  , n, a, t, i, o, n, s, ,,  , t, o,  , t, h, e,  , e, n, d,  , t, h, a, t,  , e, v, e, r, y,  , i, n, d, i, v, i "

          at 30 ms

2nd burst sent.

…
```

As can be seen, there is latency for the output results quite characteristic from the CS domain.

12. Adaptive HdS

It has been explained that the development of Adaptive Software in ANDRES has been focused on software at the application-level (see Figure 25 and section 9.1).

The work on ANDRES has served to estate initial fundamentals and proposals which have to do with the generation of Adaptive Software at other levels, most specifically with what can be called Adaptive HdS or AHdS (shown in Figure 48).

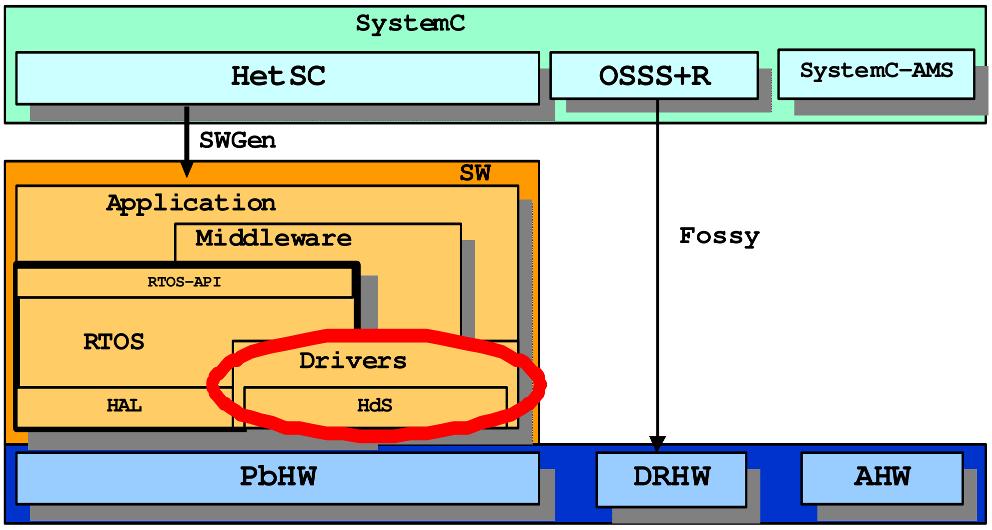


Figure 48. Adaptive HdS is an open field.

In a great part, these implications come from the need of considering the involvements of (partial dynamically) reconfigurable hardware on the software stack. The dynamic change of hardware has then some implications that have to be handled from the software side, automatically and implicitly (without the intervention of the programmer at design time or of the user at run time).

Further extensions of the ANDRES concepts for adaptivity can be then foreseen. Following, some of them are enumerated:

Adaption of the level of concurrency:

For instance, an application can split it self in more or less processes at a given time, depending on the available execution resources. This could be interesting on multiprocessor platforms or even in reconfigurable platforms which could vary the number of processor cores dynamically. It is foreseen that this could be handled at the RTOS level instead of being addressed at the application level.

Adaptive HdS:

HdS stands for the software layers which are heavily dependent on the underlying hardware. In this context, this mostly stands for drivers. That is, RTOS Hardware Abstraction Layers (HAL) are excluded from this category. Therefore, Adaptive HdS comes from the need of drivers to dynamically (during run-time) adapt to the changes of Dynamically Reconfigurable Hardware (DRHW). Then, some automation is needed to make easier and efficient the development of the HW/SW interface of such drivers.

When complete HW slices (thus pieces of HW functionality) change, then the HdS must consider its adaptation in several aspects. The foreseen scenarios where automation can be necessary are the following ones:

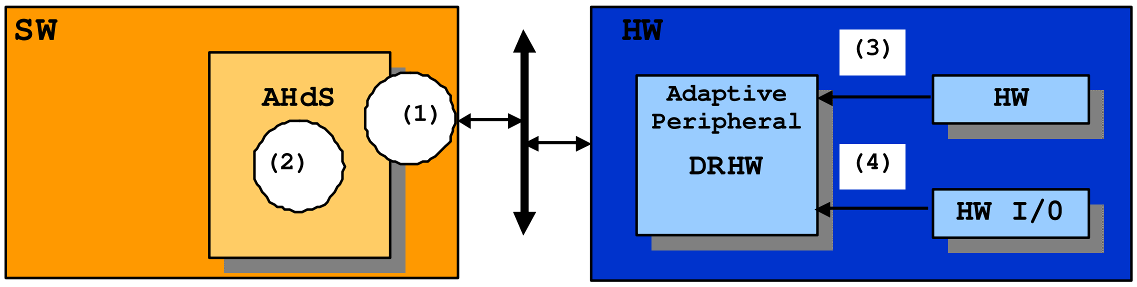


Figure 49. AHdS reacts and performs *in time* the SW actions required by Adaptive Peripherals.

Adaptation to a new memory map ((1) in Figure 49): The software adapts to a change in what is usually called the programmer's view of HW. For instance, the set of memory mapped registers of a reconfigurable peripheral could change in size. The base address of the register set could also be changed in the reconfiguration (i.e. to remap a bigger data register). Moreover, even if the set, size and base address of the memory mapped registers remain fixed, a reconfiguration of the hardware device could just change the semantic of the registers (for instance, the meaning and size of the different flags of the control register). In any of these cases, the SW part would need to consider these changes during run-time.

State Initialization/Recovering from software ((2) in Figure 49): The reconfigured hardware could need either recover its state or to perform a set of initialization steps, which vary depending on the type of reconfigured hardware functionality and that cannot be managed in hardware (i.e. due to its size). Therefore, the HdS must adapt or react to provide the suitable initialization or state recovering steps.

The solutions proposed will require the proposal of a scheme to enable the *SW reaction to HW changes*, with involvements on software solving HW/SW interface (mostly *drivers* in this context). Such *HW* changes are transparent to the application software and account for signalling from specific HW parts ((3) in Figure 49) or from HW I/O events ((4) in Figure 49), which provoke the adaptation of the adaptive (or reconfigurable) peripheral.

The SW reaction can mean the load of a completely new *driver* or the load of the HdS part of the driver, if it is found that a single driver can be efficiently separated in a common part and an HdS part (Figure 48 show this case). It is expected that the first approach will be more practical in general (meaning driver = HdS, instead of HdS ⊂ driver), based on the assumption that HdS usually lacks functionality. In any case, the study of practical cases could clarify this point. Assuming the fomer case, that is, the adaptation as a driver selection, AHdS initially solves a kind of *Plug-and-Play (PnP)* for HW/SW interfaces. Moreover, adaptive software does more than a PnP. Assuming that the PnP has been covered and, for instance, two drivers (D1 and D2) have been selected and loaded for two reconfigured functionalities (RF1 and RF2), the context change between RF1 and RF2 affects somehow D1 and D2. Assuming the feasibility of HW bein master too, if some RF stops and "disappears", its associated driver should aware of it and tell the software accessing it is happening. That is, an *Automatic-UnPlug (AUP)* must be supported.

An additional challenging and distinctive aspect of AHdS is that the consideration of the adaptation and selection/loading of the appropriate *drivers* for dynamic reconfigured hardware (DRHW) has to get the expected functionality, right initialization activities, etc under the strict time constraints which rule the design of DRHW. Therefore, AHdS reaction must be a reaction *in time* and *quick enough*. In this sense, this kind of AHdS should provide some kind of Real-Time Automatic Plug and UnPlug (RT-APU) driver scheme. Therefore, the HW and SW architectures able to provide such RT-APU drivers on Dynamically Reconfigurable peripheral hardware have to be defined and experimented.

Currently, two schemes are foreseen as feasible for the Hardware to Software notification:

- **Interruption-based**. An interruption (IRQ) is associated to the peripheral adaptation. The IRQ handler triggers the adaptation actions at software side.
- **SW task polling a HW register**. The software side uses a specific task (ensuring its computation at a given frequency) which polls a specific register where hardware adaptations are notified.

AHdS is not going to be covered by ANDRES. ANDRES mostly addresses the development of the HW/SW interface between the SW partition generated by *SWGen* and the HW generated by *Fossy*. Some advance has been already reported in **[D24B]**. For it, a fixed interface is assumed for the programmer view. The ANDRES approach provides a kind of non-reactive scheme. That is the ANDRES SW/HW interface will solve a case where SW triggers HW reconfiguration, but the case of SW being adapted because of changes of DRHW (where hardware is the master, that is the active part provoking the adaptation) remains open.

In the incipient *SofSoC* project **[Soft08]**, where GIM-UC is involved, a step forward to solve the software adaptation for reconfigurable hardware devices *at design time* will be done. However, the need for *dynamic* adaptation leaves clear that AHdS is an open field. The concepts and proposals advanced here in ANDRES, and potentially in SofSoC, should help to face these issues in future work.

13. Conclusions

This document reports the extensions of the *HetSC* and *SWGen* methodologies performed within the ANDRES project. These extensions have enabled the possibility for *abstract* system-level specification of Adaptive Heterogeneous Embedded Systems (AHES) in SystemC with a direct link to embedded software (eSW) implementation. It serves to fulfil one of the main roles of these UC-GIM methodologies within the ANDRES design framework: to enable the specification and implementation of the software part of the SystemC AHES specification.

For it, the *HetSC* methodology has been extended in several ways. In one side, guidelines and facilities for the connection of *HetSC* with the other SystemC-based specification methodologies of ANDRES, OSSS+R and SystemC-AMS, have been provided. Other actions for the integration of the last *HetSC* library (v1.3) in the AHES specification and design framework have been taken.

A main innovative part of this work is the enabling of abstract specification of adaptivity in *HetSC*, and more specifically, of adaptive embedded application software. More specifically, design patterns for specifying the different types of Adaptive Processes (the formalism provided by the ForSyDe metamodel in **[D11A]**) in *HetSC* have been provided. Such design patterns of *HetSC* APs (or HAPs) are defined for the different types considered in **[D11A],** and moreover for two main domains with impact on software programming: untimed and synchronous. Additionally, in order to give a more practical focus to the work, guidelines are given to let the user identify the suitable use cases for each HAP, and examples for clarifying its application. A set of HAP templates extends the *HetSC* library and enhances the productivity in the specification of some particular cases of HAPs.

The system-level approach to APs by means of HAP patterns and templates has the advantage of enabling the abstract specification of AHES whose implementation is targeted to a HW/SW heterogeneous platform (while keeping an automatic eSW implementation flow through *SWGen*). This is a distinguishing aspect with regard to previous works on adaptive software. The *HetSC* AHES specification can be just considered a system-level model, and it is implementation-independent. However at the same time, this specification is suitable for refinement, i.e., to hardware, and of course, for automatic software implementation through the single-source *SWGen* flow. Not in vain, HAP patterns have been proposed after a reflection about how adaptive embedded software could be manually written in case of relying on a C/C++ cross-compiler and a generic embedded RTOS API.

The *SWGen* methodology has been also extended for the support of synchronous models, with provides innovative aspects with regard previous SystemC-based approaches. More specifically, a clocked synchronous POSIX port has been already completed and released with the last version of *SWGen*. Other improvements have been oriented to an easier and more efficient handling of the library and its application to industrial flows and demonstrators.

The type of software generated by *SWGen* has been located at the application layer (including calls to the RTOS and to the drivers). It has been seen that adaptivity is a useful conceptualization, where some specific inputs are considered to be of a special kind since they change the relationship between the rest of *regular* inputs and the outputs. The adaptive process is modelled as a SystemC process and implemented in SW as a *thread*. In the specification level, the inputs and outputs are accesses to SystemC/*HetSC* channels, where at least one is dedicated to adaptation. In the software implementation, the inputs and the outputs are inter-thread communication system calls, where at least one is in charge of adaptation.

Part of the SystemC process or thread context is an adaptation context, that is, a set of parameter variables, a mode variable, a function pointer, etc, which is updated by the adaptation input(s).

At the same time, the work has served to explore some of the capabilities and limitations of the SystemC language to express adaptive systems. The specification of HAP patterns does not require a significant extension of the *HetSC* library, except for the ability to transfer through the channels function pointers and other information bound to the adaptation input. This information can change the functionality executed by the adaptive SystemC process and even which channel instances are accessed, which is related with the advanced formalism of APs adapting the own process or prAPs. The prAP brings the possibility to change the MoC after an adaptation during simulation time.

A limitation of SystemC has seemed apparent if specification schemes for prHAPs as an exact reflection of an optimum implementation procedure are aimed. This limitation would come from the fact that some of the basic SystemC structures have to be statically created (like ports, modules or channels) in terms of the SystemC simulation phases. The support of such short of dynamic structures (or of others, like the dynamic addition of a method to a class) is a requirement for *dynamic* programming languages, which actually deal with software implementation. Therefore, this limitation would make the SystemC language less suitable if it is applied and competing with *dynamic* languages in a context of pure software development. However, in an AHES specification&design context, it is not an actual limitation since SystemC is used for specification. Therefore, the altenative specification patterns proposed should be sufficient to capture prHAPs and still let optimum software, and moreover, hardware implementation of them in the remaining codesign activities.

The websites, with the last versions of the libraries (*HetSC*-v1.3 and *SWGen*-v1.2) and related documentation of *HetSC* and *SWGen* have been extended and improved, reflecting the advances done in ANDRES. This documentation is alive and dynamic, having a continuous update and improvement during the live of the ANDRES project.

This work has also served to reflect on other type of adaptive software which arises and becomes necessary because of the fact of having changing hardware peripherals implemented as Dynamically Reconfigurable Hardware. This type of adaptive software has been named Adaptive Hardware dependent Software (A-HdS) and addresses the need of some kind of Real Time Automatic Plug&Unplug (RT-APU) mechanism and Real Time Adaptive Driver (RT-AD) scheme, with associated suitable HW architectures which let software react and adapt *in time* and *fast enough* to changes in hardware.

It has been concluded that ANDRES, even after completion of ongoing work will not overcome this issue. Moreover, other related projects, like *SoftSoC*, where GIM-UC is involved, will provide hints for the solution, without addressing the problem of dynamic adaptation of software as a response to the changes in DRHW yet. Therefore, this is still an open, interesting and challenging issue which should be solved in future work and projects.

14. Future Work

It has been pointed out that *HetSC* and *SWGen* methodologies, libraries, documentation and websites are dynamic and are being continuously updated during the project. In deed, this will become one main activity of the end period of the project, which will provide an additional support to the industrial partners. Such period can be used to provide additional ports of synchronous domains, for instance to *µC/OS-II* API.

The connection of *HetSC* with OSSS+R at implementation level can be improved by refining the advances done in **[D24B]**, in order to get an actual implementation of the connection of *SWGen* and *Fossy* results.

An additional goal of the *HetSC/SWGen* library is the improvement of the link with *SCoPE* **[SCo08]**, by enabling a *SWGen* output able to directly feed SCoPE, without manual transformation. This will enhance the analysis capabilities at the system-level, in ANDRES currently focused on the global analysis of AHES schemes and architectures provided in **[D11C]**.

Also in a close future within the ANDRES framework, GIM-UC is working on the publication of the fundamentals on the formalization of SystemC, and specifically of *HetSC*, by means of the ForSyDe metamodel. This will undoubtedly complement the formal basis of the *HetSC* methodology, provided that the *HetSC* extensions for specification of adaptivity proposed in this work already rely on ForSyDe formalisms.

Beyond the ANDRES timeline, as it has been pointed out at the end of the previous section, *A-HdS* is a challenging and innovative research area which will be surely required by many applications using DRHW.

Finally, another work line could be the extension of SystemC to support dynamic creation (during simulation) of specification facilities for module hierarchical structure (modules, ports and exports) and dynamic creation of channels. It would enable making a *HetSC* a language closer to *dynamic* programming languages. It would enable the creation and deletion of channels and ports during run time to support a more compact implementation of process-based HAPs. However, as it was explained in section 10.5.1, it does not grow the specification capabilities of SystemC. The interest of such an extension would be limited to enable a SystemC specification closer to its optimum software implementation (in this sense, *SWGen* could deal this case as a mapping) and to optimize simulation resources demanded by the system-level simulation. On the other hand, it could even slow down the simulation.

Lilkely, a more interesting feature could be the formalization of SystemC specifications with dynamic creation of processes and looking into the possibilities which the dynamic adaptation of the concurrency level, that is, the amoung os processes used at a given time for the solution of a given functionality, could provide.

15. References

[AS08] Website. http://norvig.com/adapaper-pcai.html

[AW98] N.Amano, T.Watanabe. *"LEAD++: An object-Oriented Reflective Language for Dynamically Adaptable Software Model",* Special Section of Papers Selected from ITC-CSCC '98. 1998.

[BoSi91] F.Boussinot and R. de Simone. "*The Esterel Language*" In Proceedings of the IEEE, September 1991.

[BrKl07] J. Brandt and K. Schneider. *"How Different are Esterel and SystemC?"*. In Proceedings od FDL'07. Barcelona. Sept., 2007.

[CHS01] W.Chen(a), M.A.Hiltunen, R. D.Schlichting. *"Constructing Adaptive Software in Distributed Systems"*, Proc. of the 21st Int. Conf. on Distributed Computing System, Phoenix, AZ. Apr 2001.

[Dyl09] A. Shalit. *"The Dylan Reference Manual. The Definitive Guide to the NewObject-Oriented Dynamic Language"*. Addison-Wesley.1996. Available in html format in http://www.opendylan.org/books/drm/ .

[D11A] A. Jantsch. "*Methodology for Specification of Adaptivity*". Deliverable document D1.1a, Release 1.5 of the project "*Analysis and Design of run-time Reconfigurable, Heterogeneous System (ANDRES)*". November, 2006.

[D11C] I. Sander and J. Zhu. "*Overall Methodology for Partitioning and Performance Analyisis of AHES*". Deliverable document D1.1c, Release 1.0 of the project "*Analysis and Design of run-time Reconfigurable, Heterogeneous System (ANDRES)*". July, 2008.

[D12A] F.Herrera, EVillar. "Modelling of SW. Initial Library Elements". Deliverable document D1.2a, Release 1.5 of the project "*Analysis and Design of run-time Reconfigurable, Heterogeneous System (ANDRES)*". June, 2006.

[D13A] A. Herrholz, P.A. Hartmann. OFFIS. "*Modelling run-time reconfigurable hardware – Initial library elements"*. Deliverable document D1.3a of the project "*Analysis and Design of run-time Reconfigurable, Heterogeneous System (ANDRES)*". January, 2007.

[D15A] M. *Damm*. "*Modelling extensions for polymorphic signals, initial library elements"*. Tech. Report ANDRES/OFFIS/P/D1.5a/1.0, ANDRES project deliverable, May 2007.

[D16A] J. Haase, P. A. Hartmann, F. Herrera. *"Initial Version of Integrated Framework"*. Deliverable ANDRES/TUV/R/D1.6a/1.0 of ANDRES project. 2008-06.

[D24B] A. Herrholz, A. Schallenberg, C. Brunzema, K. Grüttner. "Synthesiser P2 (Optimisation techniques implemented)". Deliverable document D1.3a of the project "Analysis and Design of run-time Reconfigurable, Heterogeneous System (ANDRES)". January, 2007. 11-2008.

[FosW] http://fossy.offis.de/**.**

[FHSV03] V.Fernández, F.Herrera, P.Sánchez and E.Villar. "*Embedded Software Generation From SystemC For Platform Based Design*" in "*SystemC: Methodologies and Applications*". Ed. W.Mueller, W. Rosenstiel, J.Ruf. Kluwer Academic Publishers. March 2003. ISBN 1-4020-7479-4.

[HDG07] J. Haase, M. Damm, C. Grimm, F. Herrera, E. Villar "Using Converter Channels within a Top-Down Design Flow in SystemC" The 15th Austrian Workhop on Microelectronics. Graz, Austria. 2007-10

[HDG08] J. Haase, M. Damm, C. Grimm, F. Herrera, E. Villar. *"Bridging MoCs in SystemC Specifications of Heterogeneous Systems"*. EURASIP Journal on Embedded Systems. Special Issue "C-Based Design of Heterogeneous Embedded Systems". Volume 2008, Article ID 738136, 16 pgs. doi:10.1155/2008/738136. 2008-05.

[HSCW] www.teisa.unican.es/*HetSC*

[HUM08] F.Herrera, E. Villar. "*HetSC Users Manual*". Universidad de Cantabria. Santander. 2008. Disponible en www.teisa.unican.es/*HetSC*/documentation.html.

[Herr09] F.Herrera. *"Especificación Heterogénea y Generación Automática de Software desde SystemC para Sistemas Embebidos"*. PhD thesis. Submitted to the University of Cantabria. Nov. 2008. Draft available in www.teisa.unican.es/~fherrera.

[HeVi07] F. Herrera, E. Villar "*Extension of the SystemC kernel for Simulation Coverage Improvement of System-Level Concurrent Specifications*". Proceedings of the Forum on Design Languages (FDL'06), Darmstadt, ECSI. 2006-09.

[HVG07] F.Herrera, E.Villar, C.Grimm, M.Damm and J.Haase. "A general approach to the interoperability of *HetSC* and SystemC-AMS". In Proc. of the Forum of Design Languages, FDL'07. Barcelona. September. 2007.

[HVH08] F. Herrera, E. Villar, P. A. Hartmann. "*Specification of HW/SW adaptive Embedded Systems in SystemC"*. In Proc. of the Forum on specification and Design Languages 2008, FDL08. Stuttgart. Germany.

[IEEE06] IEEE Computer Society. "*IEEE Standard SystemC Language Reference Manual*". IEEE Standard 1666-2005. March, 2006.

[Jan05] A. Jantsch. *Models of embedded computation*. In R.Zurawski, Ed., *Embedded Systems Handbook*. CRC Press, 2005. Invited Contribution.

[Lab02] Lean J. Labrosse. *MicroC/OS-2". The real-Time Kernel. CMP Books*. 2nd edition. 2002.

[Lee97] E.A.Lee. *A denotational semantics for dataflow with firing.* Technical Report UCB/ERL M97/3, Department of Electrical Engineering and Computer Science, University of California, Berkeley, January 1997.

[Mas03] A.Massa. *"Embedded Software Development with eCos"*. Prentice Hall PTR. Nov 25, 2003. ISBN: 0-1303-5473-2.

[Net08] P.Netinant. "An extensible and adaptable model for system software", Proceedings of the 7th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems, 2008. Computer Science Department, Bangkok University, Bangkok, Thailand.

[OSCI05] OSCI. "*SystemC Synthesizeable Subset*". 2005. En www.systemc.org.

[PERFW] www.teisa.unican.es/perfidix

[PHF04] H.Posadas, F.Herrera, V.Fernández, P.Sánchez & E.Villar. "*Single Source Design Environment for Embedded Systems based on SystemC*". Journal on Design

Automation for Embedded Systems. Vol. 9. Number 4. pp.293-312. Springer. December, 2004.

[SaJa04] I. Sander and A. Jantsch. *System modeling and transformational design refinement in ForSyDe*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23(1);17-32, January 2004.

[SCO02] B.Staudt Lerner, J.M.Cobleigh, L.J.Osterweil, A.Wise. *"Using Containment Units for Self Adaption of Software",* Proc. of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, 2002.

[LJIL09] Website. http://laser.cs.umass.edu/tools/littlejil.shtml

[SAB02] B.Sirpatil, J.Armstrong, J.Baker. "*Using SystemC to Implement Embedded Software*". International HDL Conference and Exhibition (HDLCon 2002), March, 2002.

[Sch07] R. Schroll. Design komplexer heterogener Systeme mit Polymorphen Signalen. Dissertation, Institut für Informatik, Universität Frankfurt am Main, 2007.

[SCo08] http://www.teisa.unican.es/gim/en/scope.php.

[Sir02] B.Sirpatil. "*Software Synthesis of SystemC Models*". Master Thesis. Virginia Polytechnic Institute and State University. Blacksburg, Virginia. July, 2002.

[Son88] Keene, S.. "*Object-oriented Programming in Common Lisp: A Programmer's Guide to CLOS"*. 1988, Addison-Wesley. ISBN 0-201-17589-4.

[Soft08] http://www.medeaplus.org/web/projects/appli_phase2.php?request_temp=softsoc.

[SWG08] www.teisa.unican.es/*SWGen*.

[UCP08] http://www.it.hs-esslingen.de/~zimmerma/software/index_uk.html