Computers and Electrical Engineering 110 (2023) 108803

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

System C-based Co-Simulation/Analysis for System-Level Hardware/Software Co-Design☆

Vittoriano Muttillo [a], Luigi Pomante [b,*], Marco Santic [b], Giacomo Valente [b]

[a] Università degli Studi di Teramo– Political Science Department, Italy
[b] Università degli Studi dell'Aquila – DISIM/DEWS, Italy

ARTICLE INFO

This paper is for CAEE special section VSI-eltec. Reviews processed and recommended for publication to the Editor-in-Chief by Guest Editor Dr. Norbert Herencsar.

Keywords:
Electronic system-level design
HW/SW co-design
Embedded systems
Parallel systems
Heterogeneous systems

ABSTRACT

Heterogeneous parallel devices are becoming increasingly common in the embedded systems field. This is primarily due to their ability to improve timing performance, while simultaneously reducing costs and energy. In this context, this study addresses the role of a hardware/software (HW/SW) co-simulation and analysis tool for embedded systems designed on heterogeneous parallel architectures. In particular, it presents an extended System C-based tool for functional and timing HW/SW co-simulation/analysis within a reference Electronic System-Level HW/SW co-design flow. The description of the main features of the tool, and the main design and integration issues represent the core of the paper. Furthermore, the paper presents two case studies that demonstrate the enhanced effectiveness and efficiency of the extended tool. This is achieved through reduced simulation. Thanks to all this, the paper contributes to fully motivate the industrial and research communities to adopt and further investigate system-level approaches.

1. Introduction

The increasing complexity of modern embedded digital systems is causing a significant shift in common industrial design methodologies. This is especially true when they are based on modern *System-on-Chip* (SoC) adopting explicit heterogeneous parallel architectures (e.g., [1,2]) to satisfy challenging timing performance and energy consumption requirements.Traditional design approaches based on independent design of HW/SW components are no longer sufficient to efficiently exploit subparts of such SoCs. For this, system-level HW/SW co-design methodologies, where designers can early check system-level constraints and evaluate cost/performance trade-offs, are of renovated relevance [3]. In fact, these kinds of methodologies can lead the system-level activities by means of proper models, metrics, and tools, supporting the designers in all those tasks that are normally entrusted only to their experience (e.g., HW/SW architecture definition and system-level HW/SW partitioning). System-level HW/SW co-simulation and analysis tools play a crucial role in every system-level HW/SW co-design flow. This is because they enable early and rapid evaluation of system properties. In such a context, this work presents a SystemC-based tool for functional and timing HW/SW co-simulation/analysis at system-level, called HEPSIM2(i.e., *HEPSYCODE Simulator 2*) fully integrated into a reference *Electronic System-Level* (ESL) HW/SW co-design methodology, called HEPSYCODE[1] targeting heterogeneous parallel embedded systems.

The primary focus of this work is to outline the key features of the tool, the approach used to develop them, and the

☆ This paper was recommended for publication by Associate Editor Dr. Norbert Herencsar.

* Corresponding author.
E-mail address: luigi.pomante@univaq.it (L. Pomante).

[1] https://www.hepsycode.com/

https://doi.org/10.1016/j.compeleceng.2023.108803
Received 23 March 2023; Received in revised form 31 May 2023; Accepted 5 June 2023
Available online 12 June 2023

primary design and integration challenges in relation to the reference ESL HW/SW co-design flow. In particular, this work presents improvements and extensions with respect to a previous version (called HEPSIM [4]) that can be summarized as follows:

- **System Modeling: Improved *System Model* with focus on the possibility to exploitan ALT-like (i.e., *Alternative*) construct formulti-Model *of Computation* (MoC)modeling (sez. V);**
- **Simulation Efficiency: Improved *Scheduling Manager* to obtain more simulation efficiency/scalability in the management of complex models (sez. VI);**
- **Analysis Accuracy: Improved Co-*Analysis* and Co-*Estimation* to reduce estimation errors (sez. VII) and provides two case studies (sez. VIII), respectively an extended one and a new one, that show the capabilities of the proposed tool, and its role in the context of the whole reference ESL HW/SW co-design methodology. The final goal is to provide a tool fully suitable for real-world designs and, at the same time, to describe its main internal mechanisms in order to easily allow further extensions and improvements.**

This paper is organized as follows. In Section 2, the most pertinent SystemC-based ESL HW/SW co-simulation and analysis tools available in the literature and market are described. The section also emphasizes the primary distinctions between these tools and the proposed one. Section 3 briefly presents the reference ESL HW/SW co-design flow Sections 4–6 describe the main design and integration issues, while Section 7 presents the main features of the tool. Then, Section 8 presents the two case studies discussed above. Finally, Section 9 draws out some conclusions and outlines the future works.

2. System C-based ESL HW/SW co-simulation/analysis tools

In recent years, the Electronic Design Automation (EDA) industry has been pushing towards the development of ESL tools that can cover the entire design space across hardware and software boundaries [3]. Some of them are placed within a framework of ESL HW/SW Co-Design to perform functional and timing co-simulations/analysis by using *SystemC*[2] as ESL description (i.e., modeling/specification) language. The adoption of such a language is mainly due to its HW/SW unifying nature (i.e., it offers the features of both C++ and common *Hardware Description Languages*), its ESL abstraction mechanisms (i.e., it allows to provide technology independent descriptions of system behavior), and its support to virtual platforms modeling at the transaction-level of abstraction (i.e., *Transaction-Level Modeling*, TLM). The most relevant SystemC-based ESL HW/SW co-simulations/analysis tools, both commercial and academic, and the main differences with the proposed one, are briefly described below.

2.1. State of the art

As a meaningful representative of available commercial tools, *CoFluent Studio*[3]by *Intel* is a modeling and simulation environment for early high-level *Design Space Exploration* (DSE). It allows capturing the application functionality, the HW architecture and their mapping. Application models are specified as networks of communicating processes. Hardware platforms can be constructed graphically using generic processing and interconnection elements. After manually mapping the application and architecture elements, CoFluent can generate a SystemC TLM model of the resulting system for simulation, analysis, and virtual prototyping. Another interesting SystemC-based commercial tool is *SpaceStudio*[4] by *SpaceCodesign*. By using it, designers can create process-based SystemC application models out of predefined library blocks or by importing and wrapping existing C, C++ or SystemC code. Next, system architecture can be graphically assembled, and the application can be manually mapped by dragging application blocks onto previously allocated processors. As a result, SpaceStudio will generate a SystemC TLM model of the defined platform. All SystemC application and TLM models generated through SpaceStudio can be simulated for analysis and performance evaluation. *Virtual System Platform*[5] by *Cadence* allow to model easily extensible TLM virtual platforms for embedded software development. The goal is to allow the software team work in parallel with the hardware one without needing access to physical hardware or to the *Register Transfer Level* (RTL) model. *Platform Architect*[6]by *Synopsys*targets similar features and goals.

Among academic tools, it is possible to find *SystemCoDesigner* (*System-Level Hardware-Software-Co-Design Tool*) [5]. It is a software tool for (semi)automatic design space exploration at system-level. The goal is to allocate resources and bind a task graph onto these allocated resources. The designer has to specify the task graph, the architecture template (still as a graph), as well as all bindings of the nodes in the task graph onto the resources in the architecture template. Among the formal approaches, *ForSyDe*[7] (*Formal System Design*) [6] is a methodology for modeling and design of heterogeneous embedded multi-processor systems. The starting application is modeled by a network of processes interconnected by signals. Then, the model is refined by several design transformations into a target implementation language. It is based on the *Haskell* formal language while functional simulations are based on SystemC. Moreover, as a

[2] https://www.accellera.org/community/systemc
[3] https://www.intel.it/content/www/it/it/cofluent/cofluent-studio.html
[4] http://www.spacecodesign.com/
[5] https://www.cadence.com
[6] https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html
[7] https://forsyde.github.io/

relevant domain-oriented (i.e., space) virtual platforms, it is relevant to cite the SystemC-TLM based one provided by the *European Space Agency*, called *SocROCKET*[8].

Considering the presented SystemC-based tools, it has been possible to classify and compare them (also with respect to HEPSIM2) in a compact way by taking inspiration to the approach presented in Ref. [7]. Table 1 presents such a classification based on application and platform specification, implementation, and DSE support. The *Specification* column considers the *Application Model* (in terms of MoCs) and the *Platform Architecture* (in terms of heterogeneous/homogeneousmulti-processor ones). The *Implementation* column reports the *Model of Structure* (MoS) and the *Model of Performance* (MoP). MoS represent the semantics of the system structure (e.g., a netlist or a transaction-levelmodel). In order to estimate performances, a MoP associates to each individual element some quality numbers (in terms of timing, power/energy, cost/area, etc.) with respect to a specific given implementation. These quality numbers are then used by the DSE step to find and explore different implementation alternatives. Finally, the DSE drives the synthesis of the implementation from the specification, making decisions and refinements in terms of both computational and communication elements. Table 1 considers all reported classification criteria, where a full circle implies that the ESL aspect is fully supported by the corresponding tool, while an open circle indicates partial support (and partial automation). It is worth noting that HEPSIM2 fully supports all the considered criteria.

2.2. Main improvements

The tool presented in this work, called *HEPSIM2* (i.e., *HEPSYCODE Simulator 2*), is based on SystemC and, as shown in Table 1. It presents some relevant similarities and differences with the ones described above. The main improvements provided by HEPSIM2 are then described with more detail in the rest of the section.

First, the HEPSIM2 application model (i.e., *System Behavior Model*- SBM) is based on the *Communicating Sequential Processes* (CSP) MoC [8]. The compliance with such a MoC is relevant for several reasons:

- like some of the cited works (i.e., *CoFluent, SpaceStudio and ForSyDe*), it is a process based MoC. In fact, CSPbelong to such a category and for this it is well suited for HW/SW co-design since it allows the "processes to processors" mapping [9];
- CSPMoC allows a formal modeling: since it is based on a *process algebra*, it can be exploited to perform more formal analysis and automatic model transformations as possible with *ForSyDe*;
- CSP MoC can be easily used to model and integrate other kind of MoC (e.g., *Kahn Process Networks*, KPN; *Concurrent Finite State Machines*,CFSM; *Synchronous Data Flow*, SDF[10]) as better explained in Section 5.

Secondly, HEPSIM2 operates at the ESL level, which is an abstraction level similar to TLM but only considers the behavioral view of the system. It enables the consideration of the effects that mapping on the hardware platform would have on the system behavior without requiring a corresponding TLM structural model to be developed. This is obtained by exploiting an approach inspired to the *native simulation* one [11] but combined with offline statement-level timing estimations to avoid the need for binary code analysis. This feature, as better described in Section 6 and practically shown in Section 8 (*Digital Camera* use case), allows a faster *what if* analysis, since it does not require ISS or HDL integration into virtual platforms. The main drawback is the reduced accuracy in the timing performance estimation with respect to TLM analysis, but the proposed approach can be used to reduce the design space early and fast by selecting the most promising "configurations" which can then be transformed in TLM (or RTL) models for more accurate analysis by means of specific tools.

Thirdly, in addition to conducting functional and HW/SW timing co-simulations, HEPSIM2 is also capable of performing co-analysis and co-estimation activities. In fact, it is able to provide information about communication and potential concurrency in the CSP model (both for processes and channels). More so, it performs estimation of the load that processes execution would impose to processors, and the bandwidth that channels communications would impose to physical links, in order to satisfy imposed timing constraints. Such information, as described below, allows the reference ESL HW/SW co-design flow to exploit in a more effective way the targeted heterogeneous parallel embedded architecture.

Finally, HEPSIM2 is tightly integrated in the HEPSYCODE ESL HW/SW co-design methodology. From such an integration arises a relevant synergy: all the works cited above rely on a HW architecture directly provided by the designer in the initial step of the HW/SW co-design flow. HEPSIM2 can be used in the same way (i.e., manualDSE) or by exploiting its tight integration with HEPSYCODE (i.e., automatic DSE). In fact, HEPSIM2 is able to evaluate metrics/estimations to be exploited during the DSEstep of the reference HW/SW co-design flow and then, from such a DSE, it can automatically obtain the model of the HW architecture (i.e., the platform model) and the mapping of the application model to be considered for validation purposes by means of a timing HW/SW co-simulation.

3. Reference ESL HW/SW co-design flow

Fig. 1 shows the reference ESL HW/SW co-design flow while its main steps are briefly described below by giving emphasis to the interactions with HEPSIM2 (red elements in Fig. 1). More details about the whole methodology can be found in Refs. [12–14].

As earlier stated, the entry point of the reference HW/SW co-design flow is the *System Behavior Model* (SBM) based on a CSP MoC

[8] https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Microelectronics/SoCROCKET_Virtual_Platform_-_SystemC

Table 1
Classification of SystemC-based ESL co-simulation/analysis tools.

Tool	Specification		Implementation		DSE Support	Decision Making		Refinement	
	Application Model	*Platform Architecture*	*MoS*	*MoP*		*Comp*	*Comm*	*Comp*	*Comm*
CoFluent	Process Network	HeMPES	TLM	TAPM	○	●	○	●	○
SpaceStudio	Process Network	HoMPES	TLM	TAPM	○	●	○	●	○
Virtual System Platform	Component Based	HeMPES	TLM	CAPM	○	●	○	●	○
Platform Architect	Component Based	HeMPES	TLM	CAPM	○	●	○	●	○
System CoDesigner	Dynamic Data Flow	HeMPES	TLM	TAPM	●	●	●	●	○
ForSyDe	Multi-MOC	HeMPES	TLM	T/CAPM	●	●	○	●	○
SocROCKET	Component Based	HoMPES	TLM	T/CAPM	○	●	○	●	○
This work	Multi-MOC	HeMPES	ESL	T/ISAPM	●	●	●	●	●

HoMPES: Homogeneous Multi-Processor Embedded Systems - HeMPES: Heterogeneous Multi-Processor Embedded Systems.
TAPM: Task Accurate Performance Model – CAPM: Cycle Accurate Performance Model – ISAPM: Instruction Accurate Performance Model.

and described by means of SystemC (see Section 5). It is enriched by *Timing Constraints* (TCs) and *Reference Inputs* (RI). The first step of the reference flow, performed by means of HEPSIM2, is the *Functional Simulation*. It verifies the accuracy of SBM outputs with respect to RI. In the following steps, the reference ESL HW/SW co-design flow is supported by a *Technologies Library* (TL), which can be considered as a database that provides the characterization of all HW technologies available to define the so-called *Basic Blocks* (BBs) used in building the final system. The TL contains information about available *General-Purpose Processors* (GPPs), *Application Specific Processors* (ASPs), *Single Purpose Processors* (SPPs) [9], memories and interconnections (i.e., physical links). The next step is the *Co-Analysis & Co-Estimation*. During *Co-Analysis*, SBM is analyzed to evaluate three metrics: *Affinity, Communication* and *Concurrency*. The first represents how much a process is suitable for execution on a specific processor class (i.e., GPP, ASP, SPP)[12]. The second is the evaluation of the number of bits that the different processes pairs have exchanged during simulation over the related channels. It is evaluated by means of HEPSIM2 run in a configuration similar to that used for the *Functional Simulation*. The third is related to how much concurrency is potentially exploitable in the activities of processes and channels. It is evaluated by means of HEPSIM2 run in a specific timing configuration as described in Section 7. *Co-Estimation* is in charge to estimate *Timing, Size* and *Load. Timing* represents the time needed by each processor in the TL to execute an SBM statement (it is based on the statistical distribution of the CC4CS metric [15]). *Size* represents the number of bytes in RAM and ROM needed to store data and instructions for each process implemented in SW. For processes implemented in HW, it is the number of mm^2 (depending on the target HW technology, equivalent metrics like *Equivalent Gates, Look Up Table*, etc. can be used) needed to implement processing, memory, and connection elements. *Load* represents the utilization percentage that each process, when implemented in SW, would impose to each GPP/ASP composing the system to satisfy a timing constraint specified by the designer (i.e., actually it is a*Time To Completion* constraint, TTC). It is evaluated by means of HEPSIM2 run in a configuration similar to the one used for the *Concurrency Analysis* (seeVII). After this step, the flow enters in the

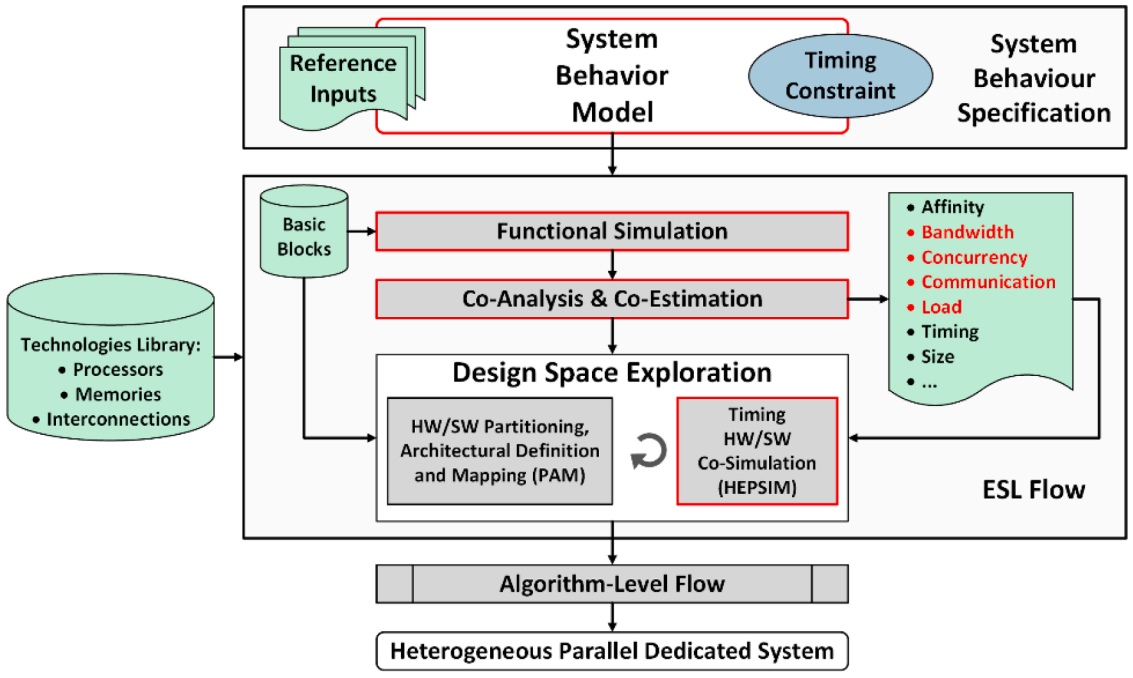

Fig. 1. Reference ESL HW/SW co-design flow.

Design Space Exploration (DSE) one, which is composed by 2 activities: "*HW/SW Partitioning, Architecture Definition and Mapping*" and "*Timing HW/SW Co-simulation*". The first task is to define the hardware architecture of the target system and perform HW/SW partitioning and mapping of processes and channels on available building blocks and physical links. The resulting data is then fed into HEPSIM2 to verify whether the proposed architecture. This kind of simulation exploits all the main features of the proposed tool as described in Section 7. Data exchange among the different steps of the whole ESL HW/SW Co-Design Flow is supported by proper XML files.

4. HEPSIM2: software architecture

This section, together with the next two, describes the main design/integration issues related to HEPSIM2. The description starts by briefly analyzing the *HEPSIM2 SW Architecture* (represented in Fig. 2, where the red elements are those that have been modified with respect to the old HEPSIM) and its main SW components. The large package on the left of Fig. 2 is the standard *SystemC Library*, which also contains the standard *SystemC Scheduler*. The library has been extended by adding a `sc_csp_channel` template class to implement the point-to-point channel semantic (i.e., unidirectional and blocking). The other SW components are *SystemModel, SystemManager* and *SchedulingManager*, supported by the *Technologies Library*. They are briefly described below and, with more details, in following sections.

4.1. SystemModel

SystemModel contains the definition of all the processes and channels used in the SBM (*System Class*) together with their corresponding SystemC code (*SBM Package*). Depending on the kind of co-simulation/analysis to be performed (i.e.,*Functional Simulation, Timing HW/SW Co-Simulation, Concurrency/Communications Co-Analysis, Load/BandwidthCo-Estimation*) SBM code is instrumented by means of some *Macros* defined in the *SystemManager*. It is worth noting that the use of *Macros* has been adopted to simplify automatic instrumentation of code by maintaining its readability.

4.2. SystemManager

This class contains every detail needed to simulate the system. In fact, it manages all the data structures, in addition to those of SystemC kernel, needed to drive the co-simulation/analysis. More so, it defines the *Macros* used for SBM code instrumentation. Depending on the kind of co-simulation/analysis to be performed, they allow to take into account the concept of simulated time (i.e., the time view inside the simulator, i.e., the estimation about how much time will require the target to execute the simulated application), to implement different *Scheduling Policies*, and to evaluate some of the metrics/estimations used in the reference ESL HW/SW co-design flow.

4.3. SchedulingManager

This class implements a second-level scheduler (i.e.,*HEPSIM Scheduler*, HEPSCHED) with respect to the standard SystemC one. HEPSCHED has been implemented as a SystemC `SC_MODULE` containing a dedicated HEPSCHED instance for each instance of GPP/ASP composing the system. Each HEPSCHED instance is implemented as a`SC_THREAD` and manages all the processes allocated to the related GPP/ASP. Processes implemented as SPP do not need a scheduler.

5. System model

The *SystemModel* contains the definition of all the processes and channels used in the SBM (*System Class*) together with their corresponding SystemC code (*SBM Package*). In particular, the whole system behavior is enclosed into a single `SC_MODULE` containing all the processes (`SC_THREAD`) and channels (`sc_csp_channel`). Other `SC_MODULE` and `sc_csp_channel` objects are then used to model the *Test-Bench* (i.e., `STIMULUS` and `DISPLAY`, whose behavior is described by one or more `SC_THREAD`) and connected to the system by means of proper `SC_PORT` objects. A schematic example is shown in Fig. 3.

Processes are modeled by using classic `SC_THREAD` while channels have been modeled by introducing a proper `sc_csp_channel`as better described later. A "CSP `SC_THREAD`" presents an *init* section and an infinite loop behavior while accessing only to its local variables and so communicating with other "CSP `SC_THREAD`" only by means of channels. Finally, in a "CSP `SC_THREAD`", only basic C/C++ statements and C++/SystemC data types are allowed while avoiding a full OOP approach since it could introduce critical issues for estimation and HW synthesis activities (the adopted restrictions have been inspired by Ref. [16]).

5.1. sc_csp_channel

Since the SBM is based on CSP, the SystemC library has been extended to properly model channels with a `sc_csp_channel` class. This class has been developed according to properties and semantic of CSP MoC and SystemC. It inherits from the SystemC `sc_prim_channel` and uses two interfaces, `sc_csp_channel_in_if` and `sc_csp_channel_out_if`, for reading and writing respectively on the channel with blocking `read()` and `write()` methods. These full handshakes mechanisms have been realized through two `bool` flags, `ready_to_read` and `ready_to_write`, to check the process on the other side of the channel, in

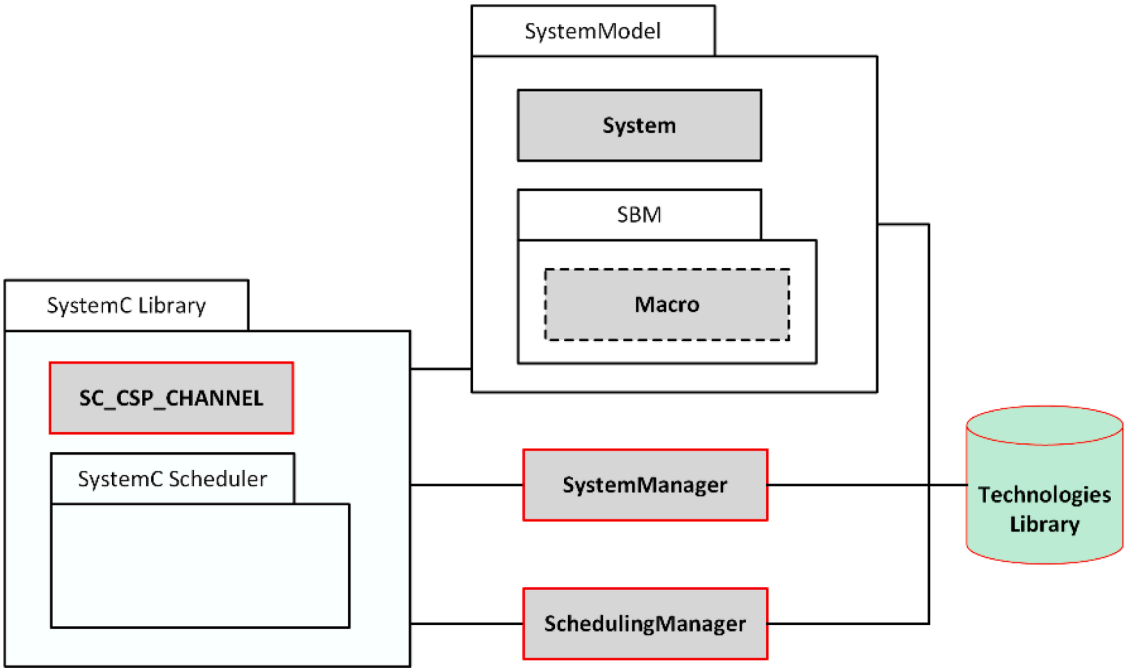

Fig. 2. HEPSIM2 SW architecture.

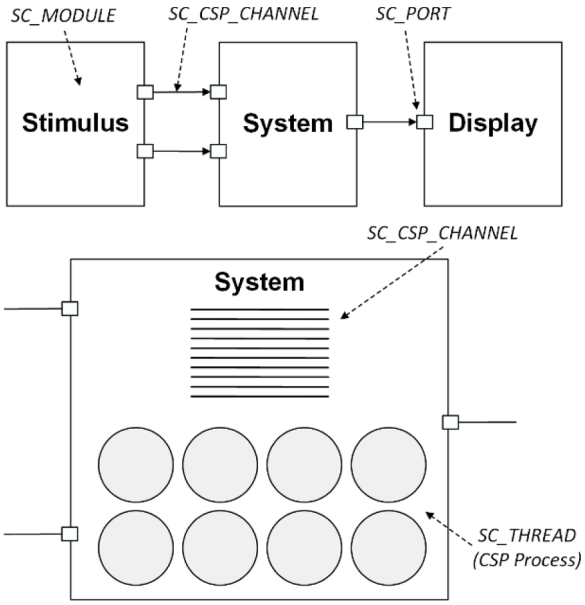

Fig. 3. A schematic example of system and test-bench modeling.

combination with two `sc_event` and the use of `notify()` and `wait()` methods in order to properly return the control, when needed, to the standard SystemC Scheduler.

Then, allowing an `SC_THREAD` check for data from more than a channel at the same time is possible, e.g., with the ALT (i.e., *Alternative*) statement of the OCCAM language, also based on CSP. Other methods have been added to the `sc_csp_channel`interface as a base to build an ALT-like CSP construct. They are:

- `read_test()` and `write_test()`: to check the state of the processes connected to a specific channel avoiding the blocking effect (i.e., they return the value of the `bool` flags `ready_to_read` and `ready_to_write` without calling a `wait()`);
- `register_alt()`: to allow a channel send events targeted to ALT-like CSP constructs;

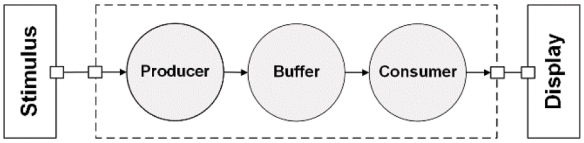

Fig. 4. Producer/consumer example.

- `get_alt_event()`: to allow ALT-like implementation wait for events from channels.

In this way, it has been possible to dramatically improve HEPSIM. In fact, all these elements allow an effective ESL modeling of complex behaviors. Other details about the channel design and implementation can be found in Ref. [14], while a relevant example related to the exploitation of the ALT-like construct is described in the next sub-section. Moreover, all the C++/SystemC code referenced in this paper can be downloaded from Ref. [17].

Finally, it is noteworthy that two versions of `sc_csp_channel` exist: functional and timing channels. The second allows to considering communication times among processes depending on their allocation and interconnection. The current policy is that, if two processes are mapped on the same instance of a GPP/ASP (i.e., local memory access) or on two SPPs (i.e., custom interconnection), then the communication time is considered negligible, otherwise (i.e., non-local memory access) it mainly depends on the amount of data to be transferred and the characterization of the physical link connecting the involved processors. Currently, other than point-to-point physical links, HEPSIM2 also allows the consideration of shared buses by statically considering the latency needed to access the shared bus itself.

5.2. *Buffer process*

In order to highlight the relevance of an ALT-like construct in a CSP MoC, in the following, a relevant example of ALT-like exploitation is presented. It is related to the modeling of a *Buffer* process. Such kind of process is of critical importance since it allows the introduction of "explicit asynchronism" in the CSP MoC.

The listing in Fig. 5 shows a process that contains a buffer of `sc_uint<8>` that can be read and written by two processes (i.e., *Producer* and *Consumer*, see Fig. 4). Thanks to the ALT-like construct. The Buffer process can verify if the Producer and Consumer processes are capable of writing/reading before calling a blocking read/write method. If none of the processes are ready, the *Buffer* process give back the control to the standard SystemC scheduler while waiting for any kind of activity in the connected channels (it is a channel itself that will wake up the *Buffer* process that waits on the logical OR of two *get_alt_event()*(end of Fig. 5). It is worth noting that a more generic *Buffer* process can be easily designed by using a C++ *Vector* in order to obtain both data type independency and virtual unbounded size.

5.3. *Multi-MoC*

The availability of a *Buffer* process enables the adoption of a CSP MoC to explicitly model other kind of MoCs and also to mix them (i.e., *Multi-MoC*) by taking inspiration by similar attempts existing in the literature both focused on SystemC [18] and more general ones [19]. For example, it is straightforward to model KPN MoC by using two channels and a *Buffer* process to model a "KPN channel" and forbidding the use of the ALT-like construct inside the `SC_THREAD`s used to model "KPN processes" (since they have to be *determinate* [20]). The same considerations easily arise for CFSM MoCs where each single FSM can be modeled by means of a switch/case construct inside a process and the communication among them can be made asynchronous or synchronous (by using or not the *Buffer* process). It is worth noting that, in the first case, the resulting MoC is that at the base of the SDL[9] language. Same considerations can lead to the modeling of SDF MoC [21] and also to a Multi-MoC modeling where different processes can embed different MoC-specific behaviors and the communication among different "MoC domains" can be made asynchronous or synchronous (by using or not the *Buffer*process).

6. System manager and scheduling manager

SystemManager and *SchedulingManager* classes represent the main elements in HEPSIM2. In fact, depending on the kind of co-simulation/analysis to be performed (i.e.,*Functional Simulation, Timing HW/SW Co-Simulation, Concurrency/Communications Analysis, Load/Bandwidth Estimation*) SBM code is instrumented by means of two *Macros* defined in the *SystemManager*. The implementation of different analysis approaches and scheduling policies in HEPSIM2 is based on the interaction of such *Macros* with the *SchedulingManager*.

The macro `HEPSY_P(X)` is inserted at the end of the infinite loop body of each `SC_THREAD` representing a process to count the number of times it has been executed (i.e., the number of loops).As shown in Fig. 6, it simply calls the *Profiling()* method of *SystemManager* that updates the dedicated data structures (Fig. 7).

[9] http://www.sdl-forum.org

```
/*****************************************************************************/
#include <systemc.h>
#include "mainsystem.h"
/*****************************************************************************/

#define D 5
#define item sc_unit<8>

void mainsystem::proc_buffer(){

   // Local variables
   item  sample_tmp[D];
   unsigned int wr_index = 0;
   unsigned int rd_index = 0;
   unsigned int elements = 0;

   // Debug info
   cout << "ProcBuffer: \t\t" << "started" << "\t at time \t" << sc_time_stamp() << endl;

   // Registration towards the input and output sp_csp_channel objects
   producer->register_alt();
   consumer->register_alt();

   while(1){
      if (elements == 0){
         // Buffer is empty, so it makes sense to read from input channel (also if blocking)
         cout << "ProcBuffer: \t\t" << "buffer is empty" << "\t at time \t" << sc_time_stamp() << endl;
         sample_tmp[wr_index] = producer->read();
         elements++;
         wr_index = (wr_index + 1) % D;
      }else if (elements == D){
         // Buffer is full, so it makes sense to write to output channel (also if blocking)
         cout << "ProcBuffer: \t\t" << "buffer is full" << "\t at time \t" << sc_time_stamp() << endl;
         consumer->write(sample_tmp[rd_index]);
         elements--;
         rd_index = (rd_index + 1) % D;
      }else{
                  // Just testing the input and output channels (to avoid being blocked)
                  if (producer->read_test() || consumer->write_test()){
                     // If the "producer" is already ready it is possible to read without being blocked
                     if (producer->read_test()){
                  sample_tmp[wr_index] = producer->read();
                  elements++;
                  wr_index = (wr_index + 1) % D;
               }
               // If the "consumer" is already ready it is possible to write without being blocked
               if (consumer->write_test()){
                  consumer->write(sample_tmp[rd_index]);
                  elements--;
                  rd_index = (rd_index + 1) % D;
               }
            }else{// If both producer and consumer are not ready…
               // Waiting for an event from channels (giving the control to the standard SystemC scheduler)
               sc_core::wait(producer->get_alt_event() | consumer->get_alt_event() );
            }
      }
   }
}
```

Fig. 5. Buffer process.

The macro `HEPSY_S(X)` (Fig. 8) is the most "invasive" one. In fact, it is inserted as a prefix to each SystemC statement composing the SBM (see code snippet in Fig. 9) to support the management of the simulated time and the mechanisms for the scheduling of processes.

So, before the execution of the following SystemC statement, `HEPSY_S(X)` first calls the *Increase()* method of *SystemManager* to consider the simulated time for statistical purposes (Fig. 10).

Then, if the process is not allocated on a SPP, it waits for a "token" from the related HEPSCHED instance and the control goes back to the standard SystemC scheduler. At a given point, the standard SystemC scheduler will give the control to the HEPSCHED instance (i.e., a `SC_THREAD`) associated to the GPP/ASP that executes the process. Then, such a HEPSCHED instance, depending on the adopted *scheduling policy* (see Sections 3.2–3.4 for some examples), will select the next ready process to be executed (i.e., by giving it the "token" by means of a *notify()*) and will wait for the token back (this will happen after the following SystemC statement execution). Then, when selected by the standard SystemC scheduler, the control passes again to `HEPSY_S(X)` that advances the simulated time by calling *wait (upSimTime())*and, if needed, give back the token (i.e., `notify()`).

upSimTime (Fig. 11) evaluates the time depending on CC4CS and *Affinity* metrics, and the frequency related to the involved processor. It is worth noting, as another extension with respect to HEPSIM, that the considered CC4CS metric is now explicitly dependent also from the data types used in the instrumented process, and that the *Affinity* is used to compute the final timing value by means of an

```
// Macro P for Profiling (to be used at the end of while(1)
#define HEPSY_P(X) pSystemManager->Profiling(X);
```

Fig. 6. Macro `HEPSY_P`.

```
/* For a given processID, increments the profiling variable */
void SystemManager:: Profiling(int processId){
   VPS[processId].profiling++;
}
```

Fig. 7. `Profiling()` method.

```
// Macro S for managing the increase of simulated time while also interacting with the scheduler
#define HEPSY_S(X)                                                                    \
   pSystemManager->Increase(X);                                                       \
   if(!pSystemManager->checkSPP(X)) wait(pSchedulingManager->schedule[X]);            \
   wait(pSystemManager->upSimTime(X));                                                \
   if(!pSystemManager->checkSPP(X)) pSchedulingManager->release[X].notify(SC_ZERO_TIME);
```

Fig. 8. Macro `HEPSY_S`.

interpolation between the *CC4CSmin* and *CC4CSmax* values available in the TL for each processor (see Section 8.1 for an example). At this point, the following SystemC statement is executed and, at the beginning of the next `HEPSY_S()`, the control goes back again to the standard SystemC scheduler where it is repeated from the beginning. It is worth noting that inside the HEPSCHED instances, it is also possible to insert an additional *wait()*(see Section 3.2 for an example) that allow to consider the overhead of the scheduling activities (i.e., *Context Switch Overhead*).

Based on the `HEPSY_S(X)` macro and its interactions with the *SchedulingManager*, three different scheduling policies have been implemented, i.e., *First-Come First-Served* (FCFS),*Round-Robin* (RR) and *Fixed-Priority* (FP), as described below. Adding new ones is straightforward since it is needed only to code the desired algorithm inside a HEPSCHED instance. However, before providing more details about the available scheduling policies, it is important to describe the approach adopted to model the state of processes and the scheduler itself.

6.1. StateModeling

In order to allow an effective scheduling, the concept of state has been added to processes by considering classical *undefined, waiting, ready* and *running* values. The *undefined* state is the default one used until processes are not mapped to processors. Processes implemented in SW start in *ready* state, become *running* when selected by the related schedulers (i.e., when the token is given to `HEPSY_S(X)`) and are *waiting* during a channel communication (read and write operations on a channel are practically blocking system calls). Processes implemented in HW start directly in *running* state and become *waiting* during a channel communication. State changes from *ready* to *waiting* (*running* to *waiting* for HW processes) and vice versa are managed directly inside the read/write channel operations. State changes from *ready* to *running* (only for SW processes) and vice versa are managed directly inside the related HEPSCHED instance (see Fig. 12).

If the processes state modeling described above represents only a slight variation with respect to HEPSIM, a disruptive innovation is the introduction of the scheduler state. In fact, when a HEPSCHED instance detects that all the allocated processes are *waiting* (see the bottom of Fig. 12), it changes its state to *blocked*. It will be *unblocked* directly from a channel at the end of a read/write operation, when the processes related to those channels are back to the *ready* state. This mechanism allows reducing the overhead associated to the scheduling management and preserve usability and scalability (see Section 8.1 for an example). In fact, in HEPSIM, instead of blocking itself, the HEPSCHED instance started to cyclically call minimal `wait(ε)` (in the order of nanoseconds) until it found some *ready* processes (i.e., practically introducing a time-driven approach). This led to the generation of a relevant number of events that slowed down the simulation performance, especially when the time intervals among different inputs from `STIMULUS` were much greater than the ε in the scheduler. In such a case, to preserve usability, there was the need to increase ε (manually or automatically) implicitly accepting reduced accuracy in the simulated time estimation. Moreover, the presence of such a mechanism seriously affected the scalability of the number of HEPSCHED instances concurrently simulatable. All these problems have been completely solved by introducing the concept of scheduler state and the related mechanisms described above.

```
// CSP process (#id 5) neverending loop
   HEPSY_S(5) while(1)
   {HEPSY_S(5)
      // Main input from channel
      HEPSY_S(5) sample_tmp=stim2_channel_port->read();

      //fir16e

      // fill datatype
      HEPSY_S(5) fir16e_p.acc=acc;
      HEPSY_S(5) for(unsigned j=0; j<TAP16; j++) fir16e_p.coef[j]=coef[j];
      …
      …
```

Fig. 9. Instrumented CSP process.

6.2. *First-come first-served*

Based on the `HEPSY_S(X)` macro, its interactions with the *Scheduling Manager*, and the concepts of processes and scheduler state, the *First-Come First-Served* (FCFS) scheduling policy has been implemented as follow. It is worth noting that, in such a policy, a *ready* process keeps the processor until it is *ready*, i.e., it will give back the control to the related HEPSCHED instance only when starting a read/write operation on a channel. Such a policy is particularly efficient for systems that are not strictly interactive or not subject to real-time requirements since it minimizes the number of context switches and related overheads. Fig. 12 shows the most interesting part of the code. Particularly relevant are the process and scheduler state management (with related block/unblock of the scheduler), and the token exchange with the `HEPSY_S(X)` macro previously described. More so, it is possible to model different overheads for full and partial context switches: the `full_oh` parameter is related to a situation in which a context switch is actually happening, while `part_oh` is related to a situation in which the scheduler checks for new processes but, at the end, the running one is not replaced. Finally, it is worth noting that the algorithm can also model an ideal situation in which the context switches have no overhead (i.e., by setting both `full_oh` and `part_oh` input parameters to 0). Such a feature is extremely useful for *Load Estimation*, as will be described in Section 7.5 and it was not possible in HEPSIM due to the mandatory need for cyclical `wait(ε)`.

```
// Increase processTime for each statement
void SystemManager:: Increase(int processId){
   // Cumulated sum of the statement execution time
   VPS[processId].processTime += upSimTime(processId);
}
```

Fig. 10. `Increase()` method.

```
sc_time SystemManager:: upSimTime(int processId){
   // Average number of clock cycles needed by the PU to execute a C statement
   int datatype = VPU[allocationPS[processId]].getDT();
   int CC4CSmin = VPU[allocationPS[processId]].getCC4CSmin(datatype);
   int CC4CSmax = VPU[allocationPS[processId]].getCC4CSmax(datatype);

   // Frequency of the processor
   float frequency = VPU[allocationPS[processId]].getFrequency();

   // Affinity
   float Affinity = VPU[allocationPS[processId]].getAffinity();

   // Average time needed to execute a C statement
   sc_time value = (interp(CC4CSmin, CC4CCSmax, Affinity, frequency*1000), SC_MS);

   return value;
}
```

Fig. 11. `upSimTime()` method.

```
// Fist Come First Served (FCFS)
// A process run without preemption until it is ready
// It is blocked only when it starts using a channel (i.e., a kind of system call)
void SchedulingManager::policy_FCFS(int pinstance, float full_oh, float part_oh){
   …

   //// MAIN BEHAVIOUR

   // Get a local process (i.e., a process allocated on the PU)
   ps = vps[local_PS[k]];

   // First execution: partial overhead to set the context (no need to save an old one).
   // Increase the simulated time for the overhead of the considered PU
   wait(p.getOverheadCS()*part_oh);
   …

   do{
      // Until the considered local process is ready it will run without scheduling overhead
      while(process_state[ps.id]==ready){

         // If the local process is different from the last one that has been run
         // there is the need for a context switch, so we need to pay an overhead
         if(k!=last_served_ps_id){
            // Increase the simulated time for the overhead of the considered PU
            wait(p.getOverheadCS()*full_oh);
         }

         // Exchange token with the current ready process and change its state in run
         schedule[ps.id].notify(SC_ZERO_TIME);
         process_state[ps.id]=running;
         wait(release[ps.id]);

         // If the process has not been blocked (i.e., no channel operations)
         // go back in ready state
         if (process_state[ps.id]==run){
            process_state[ps.id]=ready;
            last_served_ps_id = k;
         }

         // The process is no more ready, move in a circular way to next allocated process
         k = (k+1)%(local_PS.size());
         ps = vps[local_PS[k]];

         // if a "loop" on the allocated processes is completed without serving any process
         // then block the scheduler
         if((k==last_served_ps_id) && (process_state[ps.id]==waiting)){
            blocked[pSystemManager->allocationPS[ps.id]]=true;

            // Increase the simulated time for the partial overhead of the considered PU
            wait(p.getOverheadCS()*part_oh);

            wait(unblock[pSystemManager->allocationPS[ps.id]]);

            // Increase the simulated time for the partial overhead of the considered PU
            wait(p.getOverheadCS()*part_oh);
            blocked[pSystemManager->allocationPS[ps.id]]=false;
         }
      }while(1);
   }
}
```

Fig. 12. FCFS policy implementation.

6.3. *Round robin*

By following an approach similar to that adopted for FCFS policy, a *Round Robin* (RR) has been implemented also. The main difference with FCFS is that context switches among *ready* processes allocated to the same HEPSCHED instance are forced at each statement execution (it is like if the *quantum* would be the time needed to execute a statement). Such a policy is not often useful in an

embedded context since it is more oriented to interactive systems.

6.4. Fixed priority

By following an approach similar to that adopted for FCFS and RR policies, a *Fixed Priority* (FP) has been implemented also. In this case, after the execution of each SystemC statement, the process with the highest priority within the *ready* ones allocated to the same HEPSCHED instance is selected for the execution of the next SystemC statement. Practically, it is a *Fixed-Priority with Statement-Level Preemption*. Such a policy is fundamental for reactive systems, especially if subjected to real-time constraints, but it introduces relevant overhead due to both the need of a context switch after each SystemC statement execution (as in the case of RR) and to the management of a data structure to identify the highest priority *ready* process each time.

7. HEPSIM2: main features

This section describes the major features of HEPSIM2 and their role and synergy within the whole HEPSYCODE methodology. They are presented by following an increasing tool complexity order instead of the order in which they are used in the reference ESL HW/SW co-design flow (see Section 3 and Fig. 1).

7.1. Functional simulation

During the Functional Simulation step, HEPSIM2 allows the validation of SBM by means of a functional simulation. RI is of critical importance since they have to be as much as representative of the possible operating conditions of the system. Such a simulation allows to consider timed inputs (i.e., there is a concept of time in the *TestBench*), but it does not consider the time needed to execute the statements composing the processes, i.e., statements (both computations and communications ones) are executed in 0 simulated time. If SBM is not correct (i.e., wrong outputs or critical conditions such as, e.g., deadlocks), it should be properly modified and simulated again.

Since the SystemC model representing SBM is executable by construction, this step is straightforward. It is based on the simulation kernel provided by the standard SystemC library (commercial simulators can be used as well). *TestBench* and *System* are modeled as described in Section 5.

7.2. Communication analysis

During the Co-Analysis step, HEPSIM2 allow to evaluate the *Communication* metric. It consists in the number of bits that the different processes pairs have exchanged (i.e., the number of bits sent/received over each channel) during a *Functional Simulation*.

As for the *Functional Simulation*, it is based on the simulation kernel provided by the standard SystemC library and it exploits SBM and RI. The `sc_csp_channel` class is directly able, by means of proper internal data structures, to collect needed data when performing read/write operations. Such data are then written to a proper XML file.

From a DSE perspective, such values are used to identify which processes can benefit from an allocation on the same processor (i.e., to limit communication costs).

7.3. Timing HW/SW co-simulation

During the *Design Space Exploration* step, HEPSIM2 allows to perform the*Timing HW/SW Co-simulation* activity. It is responsible for checking if the architecture/mapping item proposed by the *HW/SW Partitioning, Architecture Definition and Mapping* activity is able to satisfy the specified TTC. This kind of simulation exploits all the mechanisms described in Section 6 (i.e., *Macros* and scheduling policy) by considering some info about the HW architecture (i.e., BBs, scheduling policies, and physical links) and the mapping of SBM to it (all is provided by means of proper XML files), other than SBM, RI and the *Timing* metric. It is based on the simulation kernel provided by the standard SystemC library, extended by the `sc_csp_channel` class and supported by the *System Manager* and *Scheduling Manager* components as described in Section 6. The SystemC model representing SBM is previously automatically instrumented with the macros `HEPSY_P()` and `HEPSY_S(X)` used to interact with such components and manage scheduling and simulated time.

7.4. Concurrency analysis

During the Co-Analysis step, HEPSIM2 is used to evaluate the *Concurrency* metric. It represents the potential degree of concurrency among processes and channels within the system. The goal is to obtain an indication about “how much” concurrency can be found in the dynamic activities of processes and channels pairs. As a relevant improvement with respect to HEPSIM, the potential concurrency is evaluated by means of HEPSIM2 run in a configuration similar to that used for Timing HW/SW Co-Simulation instead of using a Functional Simulation based approach. Practically, HEPSIM2 evaluates a *Timing Concurrency* instead of a *Functional Concurrency*. In fact, if the latter is one of the most used approach (typically associated to functional simulations or to the concept of abstract units of time and batch processing, e.g., [22]), it can be overly pessimistic for reactive systems in correspondence of particular RI. To elaborate further, when inputs are delayed by only a minimal interval of time between each other (rather than being provided to the system concurrently), a significant amount of concurrent processing may not overlap at all.

HEPSIM2, relying on the *Timing HW/SW Co-Simulation* previously described, by adding to the *System Manager* some supporting data structures, evaluates two matrixes of potential concurrency, for processes pairs and channels pairs. Values in the matrixes represent the number of times two processes or two channels have been concurrently *active* (normalized with respect to the number of checks performed during the co-simulation). In more detail, the strategy adopted in HEPSIM2 is the following.

First, it has been stated that a data structure to also represent the state of a channel (i.e., *waiting, ready, running*) which is updated directly inside the read/write channel operations. The channel is *waiting* when it is not used, *ready* when one of the related CSP processes starts a read/write operation, and *running* during the data transfer (the change from *ready* to *running* and vice versais managed only inside the `write()` method). Then, a proper SC_THREAD, called `concurrency_sampling()` (Fig. 13), has been added to the *Scheduling Manager* to periodically call a set of methods that check the state of processes and channels, and consequently update the concurrency matrixes. In particular, `checksStatesProcesses()` update the processes concurrency matrix when two processes are found *running* concurrently. Similar concepts apply to the checks related to channels. It is worth noting that the best value to be assigned to `sample_period`in `concurrency_sampling()` depends on several factors as described below.

The *Concurrency Analysis* is then performed by running several *Timing HW/SW Co-Simulation* by allocating each process to a dedicated processor instance (i.e., max concurrency), for each considered processor. The final concurrency matrixes; one for the processes and one for the channels, will be the ones reporting the average values obtained by the different simulations. Depending on the considered processor, a proper value must be assigned to `sample_period`. As a rule of thumb, to obtain meaningful results within an acceptable time, `sample_period` has to be around *10/100 ns* for SPP/GPP/ASP with very high/high frequency, and around *100/1000 ns* for SPP/GPP/ASP with low/medium frequency. Not withstanding, if the simulation is too slow (due to oversampling), it is possible to manually tune the sample period. An autotuning approach, depending also on the amplitude of the time intervals among different inputs from `STIMULUS` is currently being investigated.

From a design space exploration (DSE) perspective, the concurrency matrices are utilized to assess which processes and channels could benefit from allocation on different processors and physical links, respectively.

7.5. Load estimation

During the Co-Estimation step, as described with more detail in Ref. [4], HEPSIM2 is used to perform the *Load Estimation*, i.e., to estimate the utilization percentage that each process, when implemented in SW, would impose to each GPP/ASP to satisfy a timing constraint specified by the designer (i.e.,*Time to Completion*, TTC).

Load is estimated by allocating all the processes to a single instance of each available GPP/ASP and performing *Timing HW/SW Co-Simulations* by applying an ideal RR scheduling policy (i.e., no overhead RR with `full_oh=part_oh=0`). So, for each GPP/ASP used to define the BBs, three parameters are computed: *FRT* (*Free Running Time*), i.e., the total ideal simulated time required for the processes to provide all the expected outputs (i.e., to complete the simulation); *t*, the average ideal net simulated time (i.e., that doesn't consider communication times) needed to each process to make a single loop; *N*, the number of loops performed by each process. Starting from these parameters, in the hypothesis of periodic processes, it is possible to evaluate the so-called *Free Running Load* for each process on each GPP/ASP by the equation $FRL = (t*N)/FRT$, where *FRT/N* is the average period of a process on the considered GPP/ASP. At this point, by imposing the required execution time (i.e., TTC), it is possible to estimate the *Load* that each process would impose to the GPP/ASP processor to satisfy TTC. In fact, by setting $TTC=x*FRT$ (with $0 < x < 1$, otherwise TTC is already satisfied by FRT), the value of the estimated *Load* to satisfy TTC is *FRL/x*.

If, for a process, the estimated *Load* is greater than 1, it is possible to assert that the allocation of such a process on the considered GPP/ASP is not able to satisfy TTC and therefore must be avoided.

From a DSE perspective, by considering the sum of the *Load* for all the processes allocated to a GPP/ASP, it is possible to check if the total imposed load is acceptable (i.e., less than 1 or a lesser threshold to consider the overhead of a possible OS, e.g., 0.7 [23]).

7.6. Bandwidth estimation

During the Co-Estimation step, HEPSIM2 is used to also perform the *Bandwidth Estimation*, i.e., to estimate the utilization percentage that each channel would impose to a physical link to satisfy a timing constraint specified by the designer (i.e., *Time to Completion*, TTC). It is performed by exploiting the data obtained during the *Communication Analysis* (i.e., the number of bits sent/received over each channel). Then, considering the requested TTC is possible to estimate the average bandwidth (bit/s) requested to each channel to satisfy the TTC.

```
void SchedulingManager::concurrency_sampling(){
   while(stop_concurrency_sampling==false){
      // Concurrency check
      checkStatesProcesses();
      checkStatesChannels();
      wait(sample_period, SC_NS);
   }
}
```

Fig. 13. Concurrency checks.

8. Case studies

This section presents two case studies. The first shows the use of HEPSIM2 in performing co-simulations/analysis by means of the same application considered in Ref. [4] for the old HEPSIM. This allows an easy understanding of the involved HEPSIM2 features, and it has also been used to perform a scalability analysis. The second illustrates co-simulations/analysis activities by considering the *Digital Camera* (DC) application proposed in Ref. [9]. It has been modeled in HEPSYCODE and used to check the consistency and accuracy of the simulation results, and to motivate the adoption of a system-level approach. All the C++/SystemC code related to the case studies can be downloaded from Ref. [17].

8.1. FIR-FIR-GCD case study

The application considered in this case study is called *fir8-fir16-gcd* and an overview of its SBM is represented in Fig. 14. *STIMULUS* generates 10 pseudo-random *sc_uint<8>*pairs every 1 ms as input to the system. Such pairs are sent to *fir8* and *fir16* processes (where *fir* is the *Finite Impulse Response*) that respectively interact with other two processes. In fact, both the *fir* computations are decomposed in 2 parts: the first executes multiplications with proper coefficients (*evaluation*) while the other executes needed shifting operations (*shifting*). The outputs of the first computations are sent to the *gcd* process (where *gcd* is the *Greatest Common Divisor*) that cooperates with the *evaluation* process to provide the greatest common divisor between each received data pairs. In the end, the output of the *gcd* process is sent to *DISPLAY* to be visualized.

In summary, the application is composed of 8 processes and 12 internal channels. Finally, there are 3 external channels used to make connections with the *TestBench* (i.e.,*STIMULUS* and *DISPLAY*). In this case study, the target form factor is a *System-on-Board*. This means that the considered BBs are in the form of COTS discrete ICs. The processors selected from the TL to build such BBs are: max 2 instances of *Intel 8051* (16 MHz, CC4CS 297, Context Switch Overhead 50us, relative cost 10), max 1 instance of *Microchip DSPIC* (24 MHz, CC4CS 260, Context Switch Overhead 30us, relative cost 20) and max 1 instance of Xilinx *Spartan 3* (50 MHz, CC4CS 10, Context Switch N/A, relative cost 400).

It is worth noting that, in this case, *upSimTime()*(see Section 6) has been evaluated as described in Ref. [4], i.e., without considering affinity-based interpolation and data type dependency for CC4CS. Moreover, still as in Ref. [4], processors are able to communicate only by means of a single shared bus: it has been selected from the TL an I^2C bus with a bandwidth of 400 Kbps since it is a multi-master one and it is available for all the considered BBs. The main motivation behind these choices is to exploit such a case study and also to check if the provided outputs are consistent with those obtained using the old HEPSIM (when applicable), while hopefully improving the simulation time (i.e., how much time takes the simulator, running on the host, to perform the simulation).

Processes are identified by using the following *id*: 0 and 1 for the *TestBench*, 2 for *fir8*, 3 for *fir8 evaluation*, 4 for *fir8 shifting*, 5 for *fir16 main*, 6 for *fir16 evaluation*, 7 for *fir16 shifting*, 8 for *gcd*, 9 for *gcd evaluation*. Channels are identified by a proper *id* and the *ids* of the connected processes.

To identify BBs, *id* 1 is for *Intel 8051 instance 1,* 2 for *Intel 8051 instance 2*, 3 for *DSPIC* and 4 for *Xilinx Spartan3*. Starting from such a case study, HEPSIM2 exploitation in the different steps of the reference ESL HW/SW co-design flow is described below.

During the *Functional Simulation*, HEPSIM2 is used to verify the correctness of SBM with respect to *Reference Inputs* looking for situations like wrong outputs or *deadlocks*, without considering communication and computation times. The output is shown in Fig. 15.

During the *Co-Analysis and Co-Estimation* step, HEPSIM2 is used to perform *Communication Analysis, Concurrency Analysis, Load Estimation* and *Bandwidth Estimation*.

The output related to the *Communication Analysis* is a couple of data structures that reports information about *id* and data exchanged by processes pairs and channels (Fig. 16). This information is conveyed to the design space exploration (DSE) step through an XML file, which will balance this metric with the others defined in the reference ESL HW/SW co-design flow.

The output related to the *Concurrency Analysis* is a couple of triangular matrixes (Fig. 17) obtained as described in Section 6.4. First rows and columns of both matrixes refer to processes and channels *id*, and each internal value represents, in average, “how much”

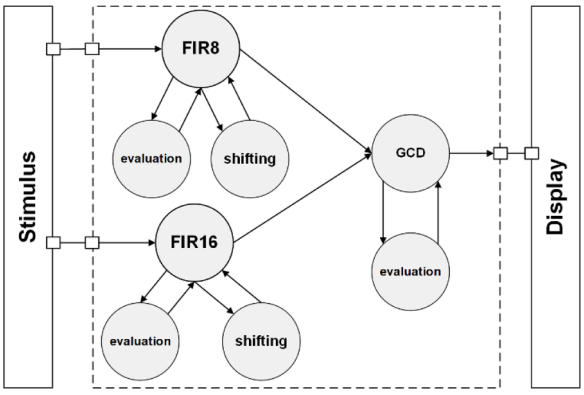

Fig. 14. FIR-FIR-GCD SBM.

processes and channels pairs have been concurrently active during timing simulations with all the processes allocated on dedicated instances of the considered processors (i.e., GPP 8051, ASP DSPIC, and SPP Spartan3). This allows discovering which processes and channels pairs could be usefully allocated respectively on different BBs and physical links. Such information is then provided to the DSE steps (by means of an XML file) that will trade-off such a metric with the others as defined in the reference ESL HW/SW co-design flow. It is worth noting that, in this case study, channels concurrency is not exploited since, by definition, it is available only to a single shared bus.

The output related to the *Load Estimation* are the values obtained for each process (as described in VI.E) that represent the *Load* which would be imposed to each available GPP/ASP to satisfy a given TTC in the case of single-instance allocation. Table 2 shows the *Load Estimation* results with respect to an *Intel 8051*and TTC equal to 0.25*FRT, where FRT results equal to 0.0566908 s.

Intuitively, if estimated load is greater than 1 for one or more of the processes, these cannot be allocated on the considered GPP/ASP instance since it will lead to TTC violation. Such information, evaluated for each available processor, is provided to the DSE steps that will trade-off such a metric with the other ones.

Finally, the output related to the *Bandwidth Estimation* is obtained, for each channel, by dividing the number of exchanged bits (see BIT*NUM in Fig. 16) by the TTC. Due to the limited number of exchanged bits, bandwidth is not an issue for the single shared bus available for this example. Anyway, such information is provided to the DSE steps that will trade-off such a metric with the other ones.

All the activities described above have been performed in less than 1 min on a PC equipped with an Intel Xeon CPU E3-1225 v5 @ 3.30 GHz, 32 GB system memory, 128 KB LI cache, 1 MB L2 cache, 8 MB L3 cache.

Finally, HEPSIM2 is exploited during the DSE step, when a *Timing HW/SW Co-Simulation* is required to verify if the architecture/mapping items proposed by the *HW/SW Partitioning, Architecture Definition and Mapping* activity are able to satisfy TTC.

To give an example of the results of such an activity, some simulations have been performed by considering different architecture/mapping items (with FCFS scheduling policy and scheduling overhead) and different TTCs, as shown in Table 3. The first column represents the TTC, the second, allocation (expressed by using processes *id* and BB*id*), the third, relative cost of the proposed solution, the fourth, simulated time, and the last, scheduler overhead (OVH). It is worth noting that, as TTC is reduced, the DSE step (mainly the one described in Ref. [4] since the communication architecture is fixed) always provides new architecture/mapping items able to satisfy them. Apart from the first row which reports info about the worst-case scenario (i.e., all the processes on a single instance of Intel 8051), used to evaluate FRT, the other rows can be read as follow. By asking to satisfy a TTC=0.75*FRT (second and third rows), two solutions are provided (at the same cost). The first is based on a dual 8051 (note as the most concurrent processes pairs, i.e., 3–6 and 6–9 in Fig. 17, have been kept on different 8051 instances while the communication costs are mainly not relevant in this case study). The second adopts a single instance of DSPIC. Both are below the required TTC but the second is slightly better. By further decreasing TTC, new architecture/items (more expensive) are suggested, always able to satisfy the constraint (except for 0.5*FRT where the simulated time is slightly over the related TTC). Finally, it is also possible to note that no mixed HW/SW solutions are provided due to the fact that the case study is too small to be completely implemented on a single FPGA. Therefore, if the required TTC leads to the adoption of an FPGA, the DSE step exploit it to maximum (for cost reasons).

Such co-simulations have been performed in less than 1 min on a PC equipped with an Intel Xeon CPU E3-1225 v5 @ 3.30 GHz, 32 GB system memory, 128 KB LI cache, 1 MB L2 cache, 8 MB L3 cache. The whole DSE steps (one for each row) have required less than 5 min.

By exploiting the presented case study, it is possible also to perform a scalability analysis to show the effectiveness and efficiency of the improvements introduced in HEPSIM2.

The first analysis is related to the scalability towards the time intervals among different inputs from `STIMULUS`. This task was easy to perform, as changing the time intervals from 1 ms to 10 ms, 100 ms, and 1000 ms resulted in consistent outputs and simulated times

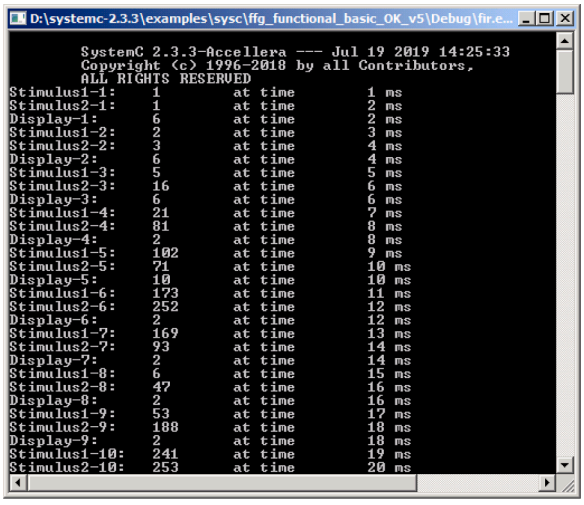

Fig. 15. Output of functional simulation.

PROCESSES COMMUNICATIONS MATRIX (#written bits from W to R)

	0	1	2	3	4	5	6	7	8	9
0	0	0	80	0	0	80	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1630	720	0	0	0	80	0
3	0	0	190	0	0	0	0	0	0	0
4	0	0	640	0	0	0	0	0	0	0
5	0	0	0	0	0	0	2990	1360	80	0
6	0	0	0	0	0	190	0	0	0	0
7	0	0	0	0	0	1280	0	0	0	0
8	0	80	0	0	0	0	0	0	0	160
9	0	0	0	0	0	0	0	0	80	0

CHANNELS PROFILING

ID-W-R	BIT	NUM	BIT*NUM
0-2-3:	163	10	1630
1-3-2:	19	10	190
2-2-4:	72	10	720
3-4-2:	64	10	640
4-5-6:	299	10	2990
5-6-5:	19	10	190
6-5-7:	136	10	1360
7-7-5:	128	10	1280
8-2-8:	8	10	80
9-5-8:	8	10	80
10-8-9:	16	10	160
11-9-8:	8	10	80
12-0-2:	8	10	80
13-0-5:	8	10	80
14-8-1:	8	10	80

Fig. 16. Output of communications analysis.

PROCESSES NORMALIZED CONCURRENCY (#TEST: 50715)

	2	3	4	5	6	7	8	9
2	0	0.04	0.04	0.05	0.08	0.05	0.04	0.03
3	0	0	0.007	0.06	0.2	0.1	0.02	0.06
4	0	0	0	0.03	0.1	0.1	0.02	0.1
5	0	0	0	0	0.04	0.03	0.06	0.04
6	0	0	0	0	0	0.008	0.04	0.3
7	0	0	0	0	0	0	0.02	0.06
8	0	0	0	0	0	0	0	0.03
9	0	0	0	0	0	0	0	0

CHANNELS NORMALIZED CONCURRENCY (#TEST: 50715)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0.4	0.2	0.3	0.3	0.4	0.2	0.4	0.2	0.3	0.3	0.6	0.5	0.7
1	0	0	0.3	0	0.1	0.1	0.2	0.08	0.03	0.2	0.2	0.04	0.1	0.2	0.3
2	0	0	0	0	0.3	0.4	0.5	0.2	0.4	0.3	0.4	0.3	0.5	0.6	0.7
3	0	0	0	0	0.1	0.1	0.1	0.1	0.02	0.1	0.1	0.08	0.2	0.2	0.2
4	0	0	0	0	0	0	0.1	0.3	0.2	0.3	0.4	0.06	0.3	0.4	0.5
5	0	0	0	0	0	0	0.5	0	0.2	0.1	0.2	0.3	0.4	0.4	0.5
6	0	0	0	0	0	0	0	0	0.3	0.2	0.3	0.3	0.5	0.5	0.7
7	0	0	0	0	0	0	0	0	0.09	0.3	0.3	0.05	0.2	0.3	0.3
8	0	0	0	0	0	0	0	0	0	0.04	0.2	0.2	0.4	0.3	0.4
9	0	0	0	0	0	0	0	0	0	0	0.4	0.0007	0.2	0.4	0.4
10	0	0	0	0	0	0	0	0	0	0	0	0	0.4	0.5	0.6
11	0	0	0	0	0	0	0	0	0	0	0	0	0.4	0.3	0.4
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0.6	0.7
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.8
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 17. Processes and channels concurrency matrixes.

Table 2
Load estimation for *Intel 8051* and TTC=0.25*FRT.

Process ID	#loops	t (s)	T=FRT/#loops (s)	FRL=t/T (s)	Load estimation: FRL/0.25 (s)
2	10	0.000347119	0.00566908	0.061230217	0.244920869
3	10	0.00069238	0.00566908	0.122132692	0.488530767
4	10	0.00054388	0.00566908	0.095937965	0.383751861
5	10	0.000404663	0.00566908	0.071380718	0.285522871
6	10	0.00128638	0.00566908	0.226911598	0.907646391
7	10	0.00098938	0.00566908	0.174522145	0.698088579
8	10	0.0002673	0.00566908	0.047150508	0.188602031
9	10	0.00113788	0.00566908	0.200716871	0.802867485

for both simulator versions. However, while the simulation time with HEPSIM2 remained constant, the situation was different for the older version of HEPSIM. In fact, the comparison about the simulation times reported in Table 4. Table 3 clearly highlights that the new blocking scheduler allows to fully preserve the advantages of event-driven simulations. With the old HEPSIM, a time interval of 1000 ms is not feasible due to the duration of the simulation. Simulation times have been obtained on the same PC described before, by repeating the executions five times for each test under similar workload conditions and then averaging the times measured by means of the `clock()` function of the `time.h` library.

Table 3
Results from the DSE on the case study.

TTC (s)	Allocation: processes ID (processor ID)	Cost	Simulated Time (s)	OVH FCFS (s)
N/A (WC)	All (0)	10	0.0704158 FRT = 0.0566908	0.013725
*0.75*FRT = 0.0425181*	2379 (0) 4568 (1)	20	0.0420892	0.004875 0.006075
*0.75*FRT = 0.0425181*	All (2)	20	0.0413209	0.008235
*0.5*FRT = 0.0283454*	234 (0) 56789 (2)	30	0.0286933	0.002175 0.004755
*0.4*FRT = 0.02267632*	234(0) 567(2) 89(1)	40	0.0213916	0.002175 0.001125 0.001305
*0.3*FRT = 0.01700724*	All (3)	400	0.0100521	N/A

The second analysis is related to the scalability with respect to the number of processes and channels in the SBM, and processors and schedulers in the HW/SW architecture. Performing it was a bit comples but the obtained results are satisfactory. The strategy has been to create more complex SBMs, by replicating (up to 5 times) the *fir8-fir16-gcd* case study presented above, and simulating it by considering some of the architecture/mapping items listed in Table 3 plus an additional one (i.e., the sixth row in Table 5) built to mix GPP/ASP and SPP. Table 5 reports the results about the simulation times with HEPSIM2 while Fig. 18 (left) shows the corresponding graph. They have been obtained on the same PC described before, by repeating the executions five times for each test under similar workload conditions and then averaging the times measured by means of the `clock()` function of the `time.h` library.

By analyzing the results, it is possible to note that the first architecture/mapping item that significantly moves away from a linear increase in simulated time is the 0|1 in 30|45. It is due to the relevant number of CSP processes mapped to only two interacting schedulers. However, the observation that the value for All (0) always increases linearly leads to the hypothesis that it is not solely a matter of the number of CSP processes or the number of schedulers, but rather a combination of both factors. In fact, the next item to significantly increase (in 40|60) is the 0|1|2 followed by 0|1|2|3 (that has one CSP process on SPP, i.e., no scheduler for it). Finally, the All (3) is the item that is less affected by the increasing complexity since it has no schedulers at all. However, something unusual is happening in 50|75: all other items are going to be worst (with a similar time) apart from All (3), as expected, and, surprisingly, 0|1 (that improves its performances!). A deeper analysis of the internal simulator data structures has shown that, in the 50|75 scenario, the rate of context switches in the two scheduler instances of the 0|1 item is decreased with respect to the previous scenarios, leading to a reduced overhead for the simulator itself and, consequently, to a better simulation performance.

In summary, by considering the average and its linear increase with respect to 10|15 (last two rows of Table 5), it is possible to note (Fig. 18- right) that the whole simulated time increase is sublinear until 30|45 and then start to seem in some way exponential (without considering the anomaly highlighted above for the item 0|1). So, considering the limited absolute times (always < 3 s) with respect to SBM of quite relevant complexity, it is possible to state that the timing performance of the proposed simulator allows to exploit also it for real-world projects.

8.2. Digital camera case study

The application considered in this case study is called *Digital Camera* (DC) [9]. It has been modeled in HEPSYCODE and used to check the consistency and accuracy of the results provided by HEPSIM2, and to motivate the adoption of a system-level approach by showing the overall benefits that is possible to obtain by working at such an abstraction level.

In the original case study, a simplified JPEG compression is considered, and four successively improved implementations are described and simulated. The main advantage to refer to such a case study is that other than a system level model, several RTL are already available. The main functional requirements of the application are: acquisition of a 64 × 64 pixels image data from a CCD; zero-bias adjustment; compression of the image (*Direct CosineTransform* – DCT, 8 × 8 pixels and quantization); compressed image transmitted serially to a PC.

As previously stated, Ref. [9] proposes four different *on-chip* implementations, starting from a single GPP (i.e., Intel 8051), connected to flash memory and an external RAM, and mapping all the functionalities to such a processor. Although this implementation

Table 4
Simulation Times comparison (with all the processes mapped on a single instance of 8051).

Simulated time intervals among stimulus pairs (ms)	Simulation Times (s)	
	Old HEPSIM	HEPSIM2
1	0.31	0.33
10	22.94	0.33
100	1740.76	0.33
1000	130,781.25	0.33

Table 5
Scalability analysis on the case study.

Architecture/mapping items / #CSP Processes|# CSP channels	10|15 (s)	20|30 (s)	30|45 (s)	40|60 (s)	50|75 (s)
All (0)	0.33	0.52	0.82	1.03	2.92
2379 (0)	0.39	0.61	1.47	2.19	
4568 (1)					2.15
234(0)	0.35	0.75	1.00	2.08	
89(1)					2.92
567(2)					
All (3)	0.26	0.43	0.46	0.74	0.89
234(0)	0.31	0.52	0.89	2.01	2.96
8(1)					
567(2)					
9(3)					
Average	0.33	0.57	0.93	1.61	2.37
Linear increase of average	0.33	0.66	0.99	1.32	1.65

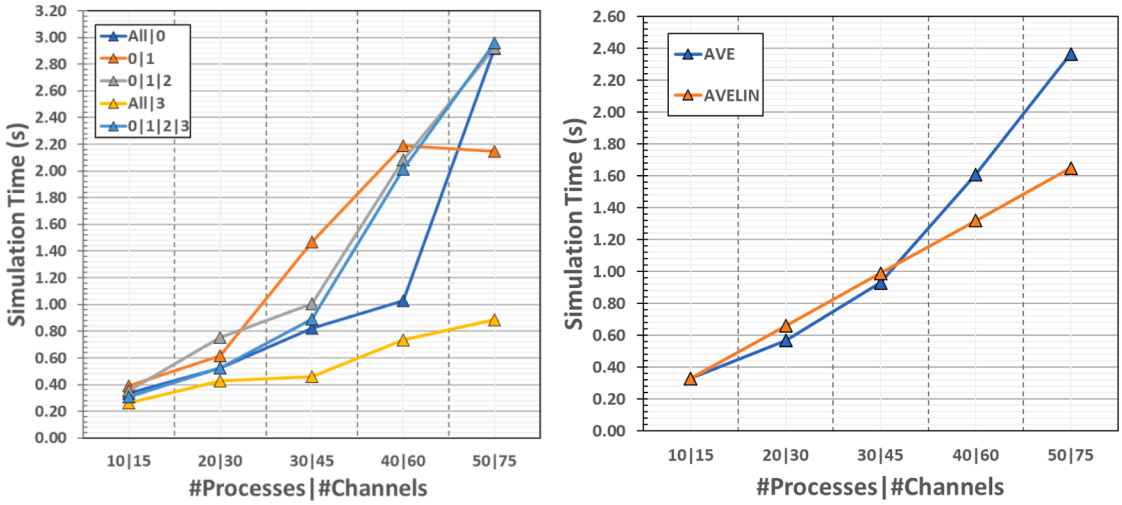

Fig. 18. Scalability analysis graph.

allows satisfying power, size, and time-to-market constraints, it does not meet timing performance requirements when imposing that the DC must process an image in 1 s. For such a reason, the implementation has been modified in order to speed it up by using different approaches that involve the exploitation of several SPPs and/or the adoption of fixed-point arithmetic for time-critical processes. Table 6 summarizes the different implementations with relative issues and notes.

The SBM of such an application is shown in Fig. 19. As requested by HEPSYCODE, it has been modeled by means of the CSP MoC and SystemC.

It is worth noting that the majority of the code needed in the processes have been taken directly by the original C-based system-level model [9]. One minor difference is the decomposition of some complex statements in multi-line simpler ones, in order to better exploit the CC4CS metric for timing estimation. Moreover, to further identify the best CC4CS value to be used for each processor, a dominant data type for each process composing the SBM has been selected according to the features of the process itself. The chosen dominant data types for the processes are the following:

- *ccdpp: int8*, since the mainly used data type is *char*
- *cntrl: int16*, since the mainly used data type is *short*
- *codec: float* in Implementation 1 and 2, since the mainly used data type is *float, int32* in Implementation 3 and 4, since fixed-point is adopted
- *uat: int8*, since the mainly used data type is *char*

Finally, in order to compute the final value of CC4CS to be used for each process, as described in Section 6, the *Affinity* has been manually defined as reported in Table 7 by considering the nature of the processes (i.e., data oriented vs control oriented) and the data types (it is worth noting that, in order to consider a fixed point implementation of *codec*, it has been possible just to change the related *Affinity* values without the need to actually modify the system-level code). In this way it is also possible to estimate the time required to execute a generic C statement of a process on the processor on which it has been allocated.

Table 6
Reference DC implementations.

#id	Description	Components				Issues/Notes
		ccdpp	*cntrl*	*codec*	*uat*	
1	1 uC 8051 (All SW)	uC 8051	uC 8051	uC 8051	uC 8051	Timing constraint not satisfied (ccdpp alone requires near 0.5 s)
2	1 uC 8051 and some SPPs	SPP	uC 8051	uC 8051	SPP	Timing constraint not satisfied (9.1 s)
3	1 uC 8051 (fixed point) and some SPPs	SPP	uC 8051	uC 8051 (fixed point)	SPP	Significant improvement (1.5 s) but still a bit over the requested time
4	1 uC 8051 (fixed point) and some SPPs	SPP	uC 8051	SPP (fixed point)	SPP	Well under the requested time (0.099 s)

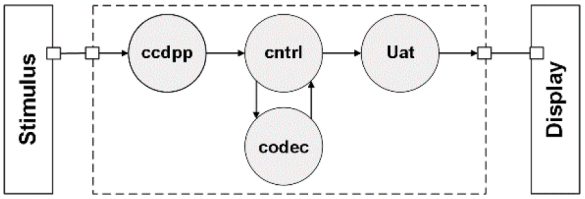

Fig. 19. Digital camera SBM.

Table 7
Affinity values.

Process	GPP	ASP	SPP
ccdcpp(int8)	1	1	1
cntrl(int16)	0.5	0.5	0.75
codec(float/int32)	0 (floating point) 0.25 (fixed point)	0.25 (floating point) 1 (fixed point)	0.25 (floating point) 1 (fixed point)
uat(int8)	1	1	1

Starting from the SBM, it is then possible for HEPSIM2 to evaluate the performance of the different implementations. The goal is to consider implementations #2-#3-#4 analyzed in Ref. [9] and compare the results about timing performance estimation. For this, proper mappings of processes to processors have been manually performed, according to the different implementations, to simulate them at system-level. It is worth noting that implementation #1 has not been considered since [9] reports simulated time only with respect to *ccdpp* process and is therefore of limited relevance.

As previously stated, in this case study, the target form factor is a *System-on-Chip*. This means that the considered BBs are in the form of COTS IP cores. The considered Technologies Library is composed of the following elements:

- **Processing Units**
  - *Intel 8051*[10] *(GPP)*
    - Frequency: 20 MHz; CC4CSmin: 125.5; CC4CSmax: 293.3; Context Switch Overhead: 243 us
  - *Xilinx Artix7 (FPGA used to implement SPPs)*
    - Frequency: 20 MHz (in Ref.[9]SPPs have the same frequency of8051); CC4CSmin: 0.9; CC4CSmax: 1.9; Context Switch Overhead: N/A (it does not execute SW)
- **Physical Links**
  - *RAM (internal to 8051 core):* to allow communications among processes implemented on the same 8051
    - Communication time is supposed to be negligible
  - *GPIO_PORT (8 bit)*: to connect *Stimulus* and *ccdpp* when *ccdpp* is implemented on 8051
    - Communication time: transfer of 8 bits in 24 clockcycles
  - *CUSTOM (8 bit)*: to connect *Stimulus* and *ccdpp*, when *ccdpp* is implemented with a SPP
    - Communication time: transfer of 8 bits in 1 clock cycle

[10] http://www.ann.ece.ufl.edu/i8051/i8051syn/

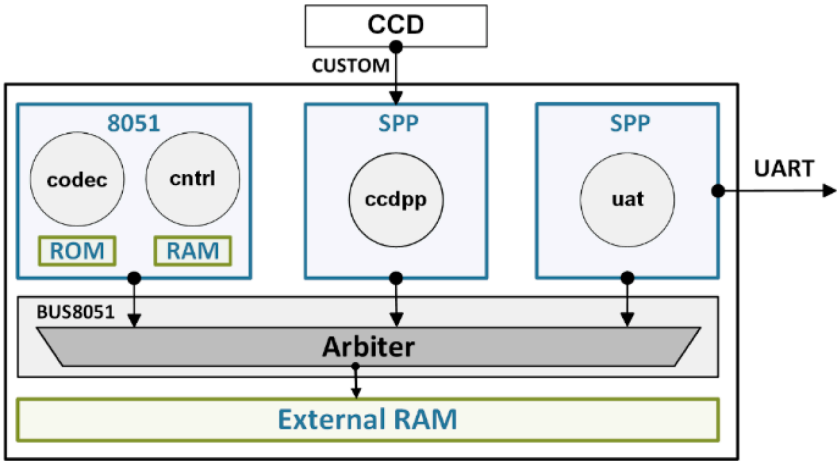

Fig. 20. Implementation #2.

```
<mapping>
    <allocation PSid="0">
        <processId PSid="0" PRname="NA" value="NA" />
    </allocation>
    <allocation PSid="1">
        <processId PSid="1" PRname="NA" value="NA" />
    </allocation>
    <allocation PSid="2">
        <processId PSid="2" PRname="ARTIX7@20" value="3" />
    </allocation>
    <allocation PSid="3">
        <processId PSid="3" PRname="MPU8051@20" value="1" />
    </allocation>
    <allocation PSid="4">
        <processId PSid="4" PRname="MPU8051@20" value="1" />
    </allocation>
    <allocation PSid="5">
        <processId PSid="5" PRname="ARTIX7@20" value="3" />
    </allocation>
</mapping>
```

```
<mapping>
    <allocation CHid="0">
        <channelId CHid="0" Lname="BUS8051" value="6" />
    </allocation>
    <allocation CHid="1">
        <channelId CHid="1" Lname="BUS8051" value="6" />
    </allocation>
    <allocation CHid="2">
        <channelId CHid="2" Lname="RAM" value="0" />
    </allocation>
    <allocation CHid="3">
        <channelId CHid="3" Lname="RAM" value="0" />
    </allocation>
    <allocation CHid="4">
        <channelId CHid="4" Lname="CUSTOM" value="4" />
    </allocation>
    <allocation CHid="5">
        <channelId CHid="5" Lname="UART" value="10" />
    </allocation>
</mapping>
```

Fig. 21. Mapping files for implementation #2.

- *BUS8051 (8 bit)*: to connect 8051 and External RAM (to store images) and SPPs
  - Communication time: 1 clock cycle to take the control of the bus (there is only one master, i.e., the 8051) and transfer of 8 bits in 24 clock cycles
- *GPIO_PIN (1 bit)*: to connect *uat* and *Display*, when uat is implemented on 8051
  - Communication time: 9600 bit/s
- *UART*: to connect *uat* and *Display*, when uat is implemented with a SPP
  - Communication time: 115,200 bit/s

Based on such a TL, all the mappings needed to model implementations 2-3-4 have been created by means of proper XML files. Figs. 20 and 21 show implementation 2 and the corresponding files needed to map processes on BBs and channels on physical links.

HEPSIM2 simulation results are reported in Table 8. Finally, to compare the results with real data, the different implementations have been synthesized considering as target an AMD/Xilinx Artix7 XC7A35T-1CPG236C FPGA by using Vivado 2017.4. Table 8 reports both the actual execution times, measured by means of the integration of an unobtrusive HW on-chip monitoring system ([24]) and the FPGA resource utilization.

HEPSIM2 estimations are characterized by an average relative error of near 29% with respect to [9] and the actual implementations. The error is due to the fact that simulations in Ref. [9] are based on cycle-accurate RTL models (in fact the results are practically equal to the actual implementation times), while the simulations with HEPSIM2 are performed at system-level and are based on the CC4CS metric that presents a limited accuracy as reported in Ref. [15]. However, the use of HEPSIM2 allows to rapidly model different configurations (it is sufficient to modify two XML files), and perform faster simulations, as reported in Table 9. Moreover, it is worth noting that all the estimations are reliable, i.e., they correctly preserve the ordering obtained by means of RTL simulations and actual measurements.

Table 8
Simulated and actual results comparison.

#id	Simulated Times (s)		Actual FPGA Implementation (AMD/Xilinx Artix7 XC7A35T-1CPG236C FPGA by using Vivado 2017.4)	
	VHDL RTL [9]	*HEPSIM2*	*Execution Times (s)*	*Resource Utilization (without the monitoring system)*
2	9.1	8.633	9.1	LUT: 6871 (33%) FF: 11,858 (28%) BRAM: 1 (2%)
3	1.5	0.872	1.5	LUT: 6871 (33%) FF: 11,858 (28%) BRAM: 1 (2%)
4	0.099	0.059	0.099	LUT: 7621 (37%) FF: 13,301 (32%) BRAM: 0.5 (1%)

Table 9
Simulation and modeling times.

#id	Simulation Times (Intel Xeon CPU E3-1225 v5 @ 3.30 GHz, 32 GB RAM, 128KB LI cache, 1 MB L2 cache, 8MB L3 cache)		Modeling Times	
	[9] *(by using Vivado 2017.4)*	*HEPSIM2*	[9]	*HEPSIM2*
2	15 min	10 sec	Some hours	Few minutes
3	30 min	10 sec		
4	15 min	5 sec		

Table 10
HEPSIM and HEPSIM2 general comparison.

Features	HEPSIM	HEPSIM2
System Modeling		
-ALT-like construct	N/A	Fully supported
-Multi-MOC	Partially available (i.e., limited to some MoCs)	Fully supported
Analysis Accuracy		
-CC4CS	Fixed value	Affinity and data types based (more accuracy)
-Concurrency	Functional (possible under estimation)	Timing
Simulation Efficiency		
-Scalability	Limited(due to mixed time/event-driven approach)	Good (fully event-driven approach)
-Simulation times	Dependent on stimulus timing(due to mixed time/event-driven approach)	Good (fully event-driven approach)

9. Conclusions

This work has presented a SystemC-based tool, called HEPSIM2, for functional and timing HW/SW co-simulation/analysis at system-level, targeting heterogeneous parallel embedded systems. It improves and extends a previous version (called HEPSIM) as summarized in Table 10.

The ultimate objective is to create a tool that is fully applicable to real-world designs. Furthermore, the aim is to describe the tool's primary internal mechanisms to encourage and facilitate the adoption of system-level approaches, such as those supported by HEPSYCODE and HEPSIM, by both research and industrial communities. This will also enable further extensions and improvements to be made. As previously noted in the preceding sections, the following possibilities will be examined: the ability to handle Multi-MoC modeling, additional types of physical links (such as hierarchical buses and meshes) with potential related arbiters, and hypervisors. These features are intended to enable the modeling of more intricate scenarios and HW/SW architectures. Lastly, additional real-world case studies will be modeled and analyzed, including the actual implementation, to conduct even more meaningful validation activities.

CRediT authorship contribution statement

Vittoriano Muttillo: Conceptualization, Methodology, Software, Validation, Writing – original draft, Writing – review & editing, Visualization. **Luigi Pomante:** Conceptualization, Methodology, Software, Validation, Resources, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration, Funding acquisition. **Marco Santic:** Software, Validation, Writing – original draft. **Giacomo Valente:** Validation, Resources, Writing – review & editing, Visualization.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work has been partially supported by the ECSEL RIA 2017 FITOPTIVIS and ECSEL RIA 2018 COMP4DRONES projects.

References

[1] AMD/Xilinx Zynq/VERSAL SoCfamilies. https://www.xilinx.com/products/silicon-devices/soc.htm ; 2023.
[2] Intel/Altera Cyclone V family. https://www.intel.com/content/www/us/en/products/details/fpga.html; 2023.
[3] Teich J. Hardware/Software Codesign: The Past, the Present, and Predicting the Future. In: in Proceedings of the IEEE, Special Centennial Issue. 100. Elsevier; 2012. p. 1411–30.
[4] Ciambrone D, Muttillo V, Pomante L, Valente G. HEPSIM: an ESL HW/SW co-simulator/analysis tool for heterogeneous parallel embedded systems. In: Proceedings of the 7th Mediterranean conference on embedded computing, MECO 2018 - including ECYPS 2018; 2018. p. 1–6. Best Paper awards.
[5] Haubelt C, Schlichter T, Keinert J, Meredith M. SystemCo-designer: automatic design space exploration and rapid prototyping from behavioral models. In: Proceedings of the design automation conference; 2008.
[6] Sander I, Jantsch A. System modeling and transformational design refinement in ForSyDe [formal system design]. IEEE Trans Comput Aided Des Integr Circuits Syst 2004;23(1):17–32. Jan.
[7] Gerstlauer A, Haubelt C, Pimentel AD, Stefanov TP, Gajski DD, Teich J. Electronic System-Level Synthesis Methodologies," in IEEE Transactions on Computer-Aided Design of Integrated. Circuits Syst 2009;28(10):1517–30.
[8] Hoare CAR. Communicating sequential processes. New York, NY: Springer; 1978. p. 413–43.
[9] Vahid Frank, Givargis Tony. Embedded system design: a unified hardware/software introduction. John Wiley & Sons, Inc; 2001.
[10] Marwedel Peter. Embedded system design (Chapter 2). Fourth Edition. Springer; 2021.
[11] Nicolas A, Sanchez P. Parallel native-simulation for multi-processing embedded systems. In: Proceedings of the 2015 Euromicro conference on digital system design; 2015. p. 543–6.
[12] Brandolese C, Fornaciari W, Pomante L, Salice F, Sciuto D. Affinity-driven system design exploration for heterogeneous multiprocessor SoC. IEEE Trans Comput 2006;55(5):508–19.
[13] Pomante L. System-level design space exploration for dedicated heterogeneous multi-processor systems. In: Proceedings of the international conference on application-specific systems, architectures, and processors; 2011.
[14] Pomante L, Vittoriano M, Marco S, Serri P. SystemC-based electronic system-level design space exploration environment for dedicated heterogeneous multi-processor systems. Microprocess Microsyst 2020;72:102898.
[15] Muttillo V, Valente G, Pomante L, Stoico V, D'Antonio F, Salice F. CC4CS: an off-the-shelf unifying statement-level performance metric for HW/SW technologies. In: Proceedings of the companion of the 2018 ACM/SPEC international conference on performance engineering (ICPE '18); 2018.
[16] "SystemC synthesis subset language reference manual". https://www.accellera.org/community/system ; 2023.
[17] Pomante L, Muttillo V. HEPSIM2 code snippets. Zenodo; 2023. https://doi.org/10.5281/zenodo.7880067.
[18] Patel HD, Shukla SK. SystemC Kernel Extensions For Heterogenous System Modeling: A Framework for Multi-MoC Modeling & Simulation. USA: Kluwer Academic Publishers; 2004. 10.5555/1024213.
[19] Ptolemaeus C. System design, modeling, and simulation using ptolemy II. Ptolemy.org; 2014. http://ptolemy.org/books/Systems.
[20] Kahn G. The semantics of a simple language for parallel programming. In: Proceeding of the international federation for information processing (IFIP); 1974. p. 471–5.
[21] Lee EA, Messerschmitt D. Synchronous data flow. Proc IEEE 1987;75:1235–45.
[22] Stuijk S, Basten T, Ypma J. CAST - a task-level concurrency analysis tool. In: Proceeding of the third international conference on application of concurrency to system design, 2003; 2003. p. 237–8. https://doi.org/10.1109/CSD.2003.1207721.
[23] Liu CL, Layland JW. Scheduling algorithms for multiprogramming in a hard-real-time environment. J ACM 1973;20(1):46–61. Jan. 1973.
[24] Valente G, Fanni T, Sau C, Di Mascio T, Pomante L, Palumbo F. A Composable Monitoring System for Heterogeneous Embedded Platforms. ACM Trans Embed Comput Syst 2021;20(5):42.

Dr. Vittoriano Muttillo, Vittoriano Muttillo received his Ph.D. in Computer Science Engineering from the University of L'Aquila (Italy) in 2019. He currently holds the position of Assistant Professor at the Department of Political Science at the University of Teramo. His-research interests are primarily focused on embedded systems, with a particular emphasis on Electronic Design Automation and Model-Based System Level HW/SW Co-Design.

Dr. Luigi Pomante, Luigi Pomante received his Ph.D. degree in Computer Science Engineering from "Politecnico di Milano" (Italy) in 2002. Since 2008, he has held the position of Assistant Professor at "Università degli Studi dell'Aquila" (Italy). His-activities primarily focus on Electronic Design Automation and Networked Embedded Systems.

Dr. Marco Santic, Marco Santic received his MSc degree in Computer Science Engineering in 2011 and his PhD degree in Electric and Information Engineering in 2015, both from "Università degli Studi dell'Aquila" (Italy). Since 2015, he has held a Post-Doc research position at the Center of Excellence DEWS at "Università degli Studi dell'Aquila". His-activities primarily focus on Embedded Systems and Wireless Sensor Networks.

Dr. Giacomo Valente, Giacomo Valente received a Laurea degree (equivalent to a BSc+MSc) in Electronic Engineering in 2014 and a Ph.D. degree in Information and Communication Technology in 2018 from the University of L'Aquila (Italy). He currently holds the position of Assistant Professor at the University of L'Aquila, where he focuses on developing EDA tools for on-chip run-time monitoring of embedded systems and reconfigurable technologies for real-time systems.