Hardware / Software Partitioning of embedded system in OCAPI-xl

G. Vanmeerbeeck P. Schaumont S. Vernalde M. Engels I. Bolsens
IMEC vzw
Kapeldreef 75, 3001 Leuven, BELGIUM

ABSTRACT

The implementation of embedded networked appliances requires a mix of processor cores and HW accelerators on a single chip. When designing such complex and heterogeneous SoCs, the HW / SW partitioning decision needs to be made prior to refining the system description. With OCAPI-xl, we developed a methodology in which the partitioning decision can be made anywhere in the design flow, even just prior to doing code-generation for both HW and SW. This is made possible thanks to a refinable, implementable, architecture independent system description. The OCAPI-xl model was used to develop a stand alone, networked camera, with on-board GIF engine and network layer.

1. INTRODUCTION

State-of-the-art digital platforms use a mix of processor cores and HW accelerators. The design process for such (deeply) embedded SoC devices today starts with a partitioning into chips or chipsets, decided by an architectural guru, and lets hardware and software design teams solve their part of the problem independently **[3]**. Other design environments doing hardware / software co-design **[6]**, **[7], [8], [9], [10], [11], [12],** are indeed following this flow. They all do some high level exploration, partitioning and refinement in that particular order. The resulting system performance however is often greatly depending on this partitioning decision. But none of the current tools is giving exploration possibilities for system partitioning with system performance feedback on an implementable system description. So whenever a certain partitioning turns out to fail performance specifications, a complete new, time consuming, design iteration with another partitioning decision is needed. Most of the current design tools are concentrating on co-verification instead of real co-design. They provide the means to test whether your synthesizable partitioning meets the specifications, rather than shifting the partitioning decision to a later stage in the design flow.

We developed a C++ class library, called OCAPI-xl, as a second-generation version of the OCAPI **[2]** concept that offers this partitioning exploration possibility to the designer. In the OCAPI-xl model, a system is described in a set of concurrent processes, but the partitioning decision on these processes does not need to be made in advance, it can be made anywhere in the design flow. This can be done early in the flow, based on performance analysis results using timing estimates, or later on, on the refined and implementable system description using more accurate performance results. Useful feedback can be obtained throughout the design flow from system simulations. This to evaluate functionality or the chosen hardware/software partitioning decisions and the consequences on system performance.

In our unified model, every block is refined in the same, implementation-independent way. Refinement can be done in a gradual and incremental way, since refined and unrefined blocks can be simulated together. The refinement process itself consists of partitioning a design into a number of concurrent processes that communicate with a well-defined set of communication primitives (being messages, semaphores, and shared variables) and describe functionality using OCAPI-xl objects. The communication primitives can all be implemented in a software way (cfr. messages, semaphores, and shared memory on an OS **[4]**) or in a particular hardware implementation (with automatic protocol and bus expansion on the HDL process entities).

The OCAPI-xl model is providing a complete path down to implementation, featuring code generation for hardware and software. This is possible thanks to the chosen communication primitive semantics that can target both HW and SW. This is also realized thanks to a way to automatically extract the definition of each process based on the implementation decision made on the process and the process's communication scheme and its internal functionality. Also, a mechanism is provided to model accesses to other system components and their behavior via Foreign Language Interface approach (FLI) into the system description.

The OCAPI-xl model was used to develop a stand-alone webcam including an interface to a digital CMOS image sensor, a GIF engine, a network layer and an interface to a 10BaseT ethernet PHY+MAC controller. The synthesized model for this NetCam (with raw-IP sockets) consisted out of 25 concurrent processes, described in about 2Klines of C++ code (taking about 25Kgates on an ASIC), designed from scratch in 14 man-months.

The remainder of this paper is organized as follows: Section 2 describes the OCAPI-xl model. In this section is presented: the design flow, the communication primitives, the timed data model, the way process definitions are extracted upon code generation and the way to include other system functionality via FLI calls. Section 3 describes the NetCam, being the driver application for this methodology. Finally in section 4 some conclusions are drawn.

2. THE OCAPI-XL MODEL

OCAPI-xl, a C++ class library for refinement and implementation of embedded HW/SW systems, is a second-generation version of the OCAPI **[2]** concept. The main differences with regards to the previous OCAPI tools are:

(a) the system semantics have been revised from data flow to a more general communicating multi-threaded model,

(b) the refinement principle has been reworked from a computational model refinement (Data-Flow → finite state machines+datapaths) into a timed communicating process refinement (functional C → communicating processes).

(c) the integration mechanism has been revised to better support existing architectural resources such as Virtual Component (VC) cores **[2]**.

In this section the following topics of the OCAPI-xl methodology will be presented: the design flow (2.1), the set of communication primitives (2.2), the timed data model (2.3), use based process interfaces, or the way process definitions are extracted upon code generation (2.4) and the FLI call approach (2.5).

2.1 Design Flow

The overall principle of design by programming is that one constructs an executable simulation of both the system and its environment. In order to write the programs in a comfortable way, we use data models, which are commonly used abstractions in the design process. The benefit of designing in an object-oriented language is that it is easy to create those data models, as they can directly be expressed by new objects (e.g. fixed-point number, a finite state machine, a data-flow graph).

The design flow of OCAPI-xl starts with an executable system specification (see figure 1). This is a set of C routines, matlab code or any high level description that expresses the actual task to perform without taking into account architecture, concurrency or timing.

The first major step in the flow is to decompose this functional model into a *concurrent timed model*. This means that a set of concurrent OCAPI-xl processes is to be created. The functionality of each process can be copied and pasted into an FLI object, a calling mechanism allowing external C or C++ functions (see §2.5), that enables maximum code reuse when going from a functional model to the concurrent timed model. These FLI objects can also be used to express high level communications on this concurrent model. This yields in an OCAPI-xl system framework that includes the complete system functionality in several concurrent processes, that can be time annotated for performance analysis, and that acts as a sceleton for further refinement and elaboration.

The next step in the design flow is to refine the concurrent processes into an *implementable concurrent model*. This is done through integration and mapping of the functionality from the copied user code (C or C++) into OCAPI-xl objects. These objects are expressions on specified length integers, as well as control flow statements like *loop* objects and *ifthen* objects. All timing information inside processes also needs to be refined into timing semantics, which indicate boundaries for quantum of computations (see §2.3). All communications must be done using a well-defined set of communication primitives. These primitives are shared variables, messages and semaphores. For more complex communication schemes, these can off coarse be combined into any communication pattern required by the designer (see §2.2). The final result of this step is an implementable concurrent model, consisting of several concurrent OCAPI-xl processes, communicating with OCAPI-xl communication primitives and using FLI calls as interfaces to other system resources.

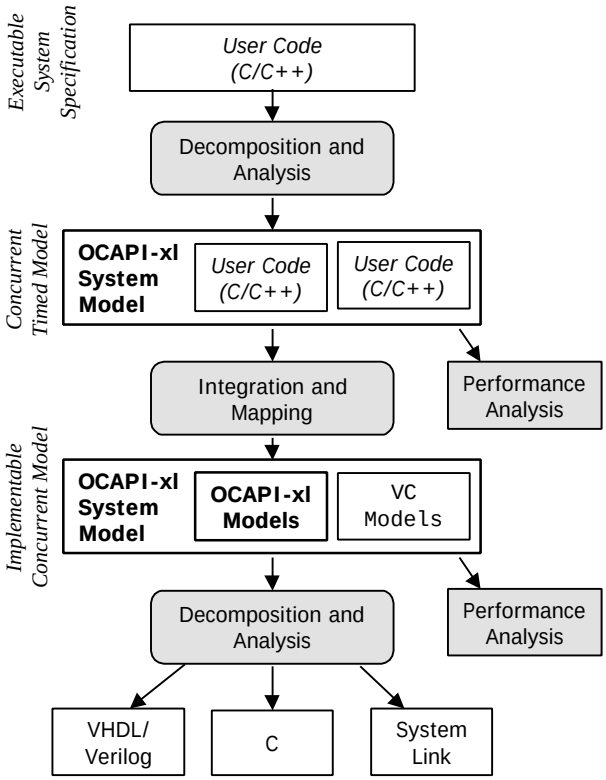

Figure 1: the OCAPI-xl design flow

As a last step in the design flow, we run the code generators on the fully refined model. This creates *'synthesizable code'*, i.e. code that can be fed into a standard tool chain. For hardware targets, this is a tool suite that performs technology mapping of RT-code (described in VHDL or Verilog). For software targets, this is a C-compiler for the specific target core. Additional system link information can be included during code generation also. This can be an FPGA user constraint file for example, when targeting an FPGA for the hardware part of the system implementation.

Throughout the design flow, verification is performed by simulating the system description. To make simulation results more comprehensive for the designer print and trace statements are allowed to send internal process information back to the designer. However, since the system expresses parallelism but simulation results will appear sequentially, OCAPI-xl offers the possibility to generate value change dump (VCD) files. VCD is a format that is used to observe the output of HDL simulations. It allows you to get an overall

view of a complex simulation, with a reasonable amount of output file size. When using waveform viewers supporting the VCD format, a graphical view on the system behavior can be obtained. This simplifies the debugging task for the designer when using the OCAPI-xl model.

2.2 Communication primitives

In OCAPI-xl, system behavior is described in a set of concurrent processes. A process can be in three different states: it can be active (running), it can be active but waiting for something to happen (sleeping), or it can be terminated (dead). A process is a collection of actions. An action is a piece of code within a single process that can run without interruption and without interaction with the environment. Actions are described in a specific variable-length integer type (supporting all binary, unary, bit-wise and logical operations) and control objects (supporting loops, conditional control flow and jumps). Processes can communicate with each other, using a well-defined set of communication primitives. These communications can be data based, control based or a combination of both.

Communication between processes can be done in different ways, depending on the nature of the communication. We distinguish three basic characteristics to describe communication from one process to another. These are the flow, the synchronization and the content:

- The *flow* characteristic describes the overall nature of the communication. The flow can be *token* based or *state* based.
- The *synchronization* characteristic describes the behavior of an individual process when participating in a communication. A process can be synchronized on the communication or not (cf. *blocking, non-blocking*).
- The *content* characteristic describes the kind of information passed between processes.

The communication primitives used in OCAPI-xl are software-like, i.e. messages, semaphores and shared variables. They are defined as follows:

- A *message* is a data-pipe that feeds data from sending processes to receiving processes. The data-pipe can have multiple senders and multiple receivers (broadcast, multi-point to multi-point). A message also has a control aspect in the sense that it blocks the receiving process when there is no data in the pipe.
- A *semaphore* is a control primitive that allows expressing resource control in an abstract way. The semaphore of OCAPI-xl is of the binary type, and can be in a locked or in an unlocked state.
- A *shared variable* is a data variable that can be read and written by different processes. It is typically managed as a shared resource by means of a semaphore.

An overview of the communication primitives and their characteristics is given in Table 1. There also the dangers are stated when designing with these communication primitives. This is something the designer should be aware of, but it is also provided as feedback during system simulation when race conditions or deadlock really occur. It is however up to the designer to solve these problems, since they are a result of poor system modeling.

Table 1: OCAPI-xl overview IPC

Type	*Flow*	*Synchronization*	*Content*	*Dangers*
Shared Variable	State Based	Non Blocking Read Non Blocking Write	Data	Race Condition
Message	Token Based	Blocking Receive Non Blocking Send	Data	Deadlock
Semaphore	Token Based	Blocking Wait Non Blocking Post	Control	Deadlock

Since this model is able to target both hardware and software, these primitives are not exactly equal to their software counterparts. The main differences are:

- A message is multi-point to multi-point, so broadcasting a message to a bunch of processes is possible inside an OCAPI-xl system.
- Due to hardware limitations, a message acts like a single slot FIFO queue. Whenever a message is send when the previous was not yet received, the first one will be lost. (during simulation only a warning will be issued when messages queue up)
- Also due to hardware limitations the size of each message send over the same message queue needs to be identical in size every time. In software any message length can be send over the same queue.
- The OCAPI-xl semaphore is also multi-point to multi-point. This means it can be locked and released by multiple processes, however it is not of the counting type but of the binary type. So the state of the semaphore is either *locked* of *unlocked*.
- A variable can be shared by multiple processes (shared variable), or can be a local variable of a single process. This is similar to multithreaded software code. In multi-process software, a variable can only be shared when they reside in shared memory.

If a more complex communication is needed when designing a system, these primitives can be combined to create any communication scheme the designer wants. Here *design patterns* can be applied. A design pattern conveys the essence of a proven solution to a recurring problem. We apply this concept to describe the Inter Process Communication (IPC) problem in our model. By agreeing on design patterns for communication, designers can work on subparts of a system in parallel, without having to know the internals of the other subparts with whom they will communicate.

An example of such a design pattern for communication is a resource manager. In this pattern, one process (the resource manager process) manages all access requests from different processes to that resource. Any number of reading/writing processes can issue requests to the resource manager. In case of parallel access requests to the central resource, accesses are serialized according to process priorities. In figure 2 this design pattern is shown in a communication sequence flow graph. A 'bubble' in the graph represents a combination of actions and one system communication (using a communication primitive or via a FLI object). An octagonal sign repre-

sents a semaphore. A message is represented as an envelope sign. The arrows indicate the direction of flow or the way of communicating (e.g. for a semaphore: post or wait operation, for a message send or receive).

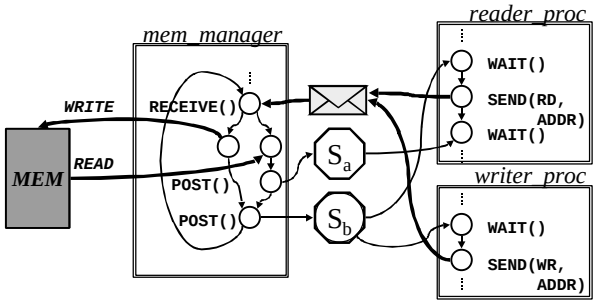

Figure 2: design pattern for communication

Each reader or writer pocess willing to issue a request, must first wait on a sema, which is granted to the process that's being served. That process can then send its request as a message. Each request contains the requested operation (read or write) and the address it should be acted on. After sending the request the issuing process waits on another sema to indicate the termination of the resource access.

2.3 Timed Data Model

The data model in OCAPI-xl is based on timed, communicating processes. The way this timing information is taken into account during simulation of the system model is based on a *parallel virtual time model*. The advanced time of every individual process is kept private in that process. Supplementary to all these local times there is one global time in the system simulator that serves as a reference to align all the local times. While simulating, performance parameters are obtained from comparing the local times of processes or from comparing local times to the global system time. Performance parameters can be the total simulation time or the amount of time a process was blocked. This way the designer can have an idea on system performance or on the amount of parallelism in the system.

The communication behavior is expressed by the use of a well-defined set of communication primitives. When taking timing aspects into account during refinement of the system model, some additional primitives are needed to express this kind of behavior. The way time is modeled in the system is similar to the way it is done in TIPSY **[5]** and POLIS **[10]**. This is a locally synchronous, globally asynchronous time model; meaning time is supported by having a local time in each process, and a scheduler that synchronizes the advancement of time in the system. This scheduler is non-preemptive. For high-level processes and hardware processes, it acts like a Round Robin scheduler. For software processes, the designer can decide on the scheduling himself.

Already in the first system description model, time can be included in the system by annotating timing estimates. The annotation of time is done by the `idle()`-call. These can be based on either the number of clock cycles for a hardware implementation, or the number of instructions for a software implementation. The `idle()`-call will actually increase the local process time by the amount of idle time specified. In this way the process will wait until it is aligned again with the global system time, which results in true process idle time. A computational action takes zero system execution time during simulation. That is why timing annotations are needed for accurate analysis. Otherwise, local process time will only evolve when the process is waiting on a semaphore or on the reception of a message.

When further refining the process descriptions, timing semantics need to be incorporated, similar to those used in TIPSY **[5]**. The `sync()`-call depicts the boundary for a *quantum of computation* (QoC). The consequences for the timing behavior of this `sync()`-call is depending on the implementation target:

- *Hardware implementation*: a `sync()` marks the edges of a single clock cycle. Hence, everything between two syncs will be executed within one clock cycle. The amount of time that elapses between two syncs is therefore fixed and is a constant throughout simulation or operation. The actual time of the clock period only depends on the clock frequency used by the hardware. Note however that the use of communication primitives in a hardware implementation can implicitly insert additional clock cycles to the process.
- *Software implementation*: a `sync()` marks the boundaries of a piece of code that is executed without context switching to any other process. In software terms, this corresponds to code between two *yield* calls in a non-preemptive multithreaded or multi process application. The actual time between syncs depends on the processor used (number of cycles per instruction, pipelining…), the clock frequency used by the processor and the amount of time needed for an actual context switch.

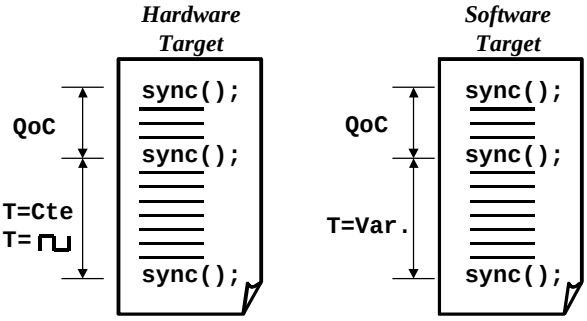

Figure 3 Meaning of time and sync()

Other timing information that can be obtained from system simulation with this virtual parallel time model is the so-called *backlog* and *forwardlog*. This can easily be explained with the example where a message is sent from the sender process to a receiver process. If the local time of the receiving process is larger than the local time of the sending process, the message has to travel some time units 'into the future' to be received. We call this a *backlog*. When backlog occurs upon receiving a message, the sender could write the message into a buffer and immediately continue. The message then

would live in the buffer for the amount of backlog time, after which the receiver will pick it up.
If, on the other hand, the local time of the receiving process is smaller than the local time of the sending process, the message then has to travel 'into the past', which we call a *forwardlog*. When forwardlog occurs, we might consider putting the receiver process asleep, until the sender process is ready to transmit the message. No extra storage would be needed in that case.
Process activity (computation) and buffering are interchangeable commodities at system level. The value of backlog and forwardlog is that they are giving feedback at high level, before computation versus buffering decisions need to be made.

2.4 Processes with Use-Based Interfaces

When writing HDL modules or software functions, interfaces always need explicit declaration. For HDL this means the number of ports, their type and width, whether they are in- or output and the name of the ports. For software it is similar, a function needs a declaration and a definition, with number of parameters, each with a certain name, and with a certain type.
In OCAPI-xl, process interfaces do not need explicit declaration by the designer. Instead, the interface needed for implementation is *use-based*, meaning that it depends on the functionality as is described in the process model and its communications. This spares the designer from specifying obvious process declarations or definitions and allows to speed up switching the implementation decision on a process.

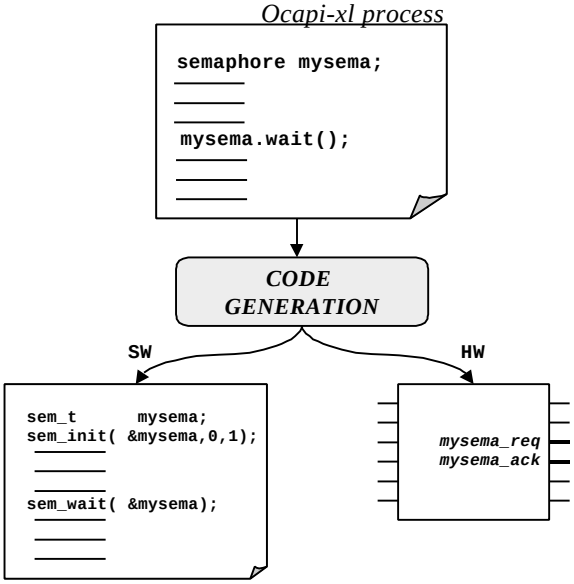

Figure 4: use-based interface for a semaphore

Since interfaces are the result of communication, and communication can only be done using semaphores, messages and shared variables, the process's interface is depending on the communication primitives used inside the process itself. Additionally, for accessing external ports to the system (like VC-cores or external components such as memory), the FLI call is used to give the number of external ports, their names and the directions (see §2.5). When doing code-generation, the declaration needed for the desired target implementation is figured out by the system, so the code generator uses a *lazy declaration* approach to generate an explicit declaration for processes and system interfaces.
If for example a process has to wait on a semaphore before it can continue, only the wait on the semaphore is explicitly described, since it's part of the process's behavior. After software code generation, the wait on the semaphore will be translated and expanded to the declaration and initialization of the semaphore, and the wait-call will be the one of the operating system the target will be run on. When doing hardware synthesis, the wait on the semaphore will result in a protocol expansion on the hardware block of the process with a request and an acknowledge line for the semaphore. (see figure 4) and a library block to implement the semaphore itself in hardware. The same goes for the implementation of messages and shared variables.

2.5 The FLI call approach

The FLI call has a bit of a misleading name. It is not a way to embed other languages like HDL or assembly into the executable model, but it turned out so powerful that it may indirectly be used to do exactly those kind of things.
The basic idea behind an FLI call is that a use-based interface will be created upon implementation to an internal or external component in the system. These components can be memory, a processor, a dedicated HW block or any other interface. The FLI object representing the interface has a method that executes an interface model (see figure 5). This to be able to simulate its behavior together with the system. It also defines the number of ports on the interface, and whether they are incoming or outgoing in relation to the FLI-object itself. In this way abstraction can be made of the physical properties of the interface by applying object oriented techniques.

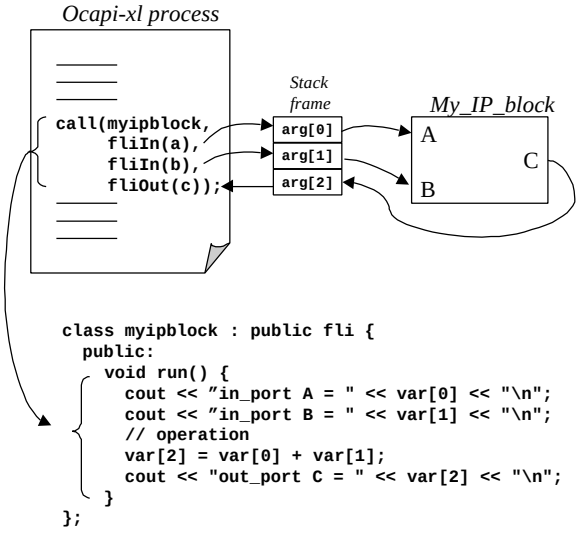

Figure 5: Foreign Language Interface call example

Since the `run` method is not to be synthesized (only the interface to it), it only serves as a model for verification. Therefore the code here can be quite complex and does not need to follow OCAPI-xl semantcs. The model could access device

drivers or network sockets for example, or even access an instruction set simulator (ISS) for a processor. So, this is a very powerful way of making the degree of accuracy of the model very realistic.
For a hardware implementation of an interface, this approach results in the declaration of the specified number of ports in the system top-level entity (VHDL) or module (Verilog). For a software implementation an FLI call results in a simple function call passing the specified parameters. The designer has to complete this call with the implementation specific code (C or assembly) to access the interface as intended.

3. THE NETCAM DRIVER APPLICATION

The driver application and showcase for this unified model is a networked camera, or NetCam. The application consists of several layers (see Figure 6). Network connectivity is provided by the LINK, IP and TCP socket layers **[1]**. The GIF encoder turns the retrieved images into GIF format and the EEprogrammer enables network reconfiguration. From the client side an image request packet is send over the network. The reply is a GIF-encoded image send back over the network to the client.
The implementation platform consists of a 600Kgates FPGA, 2x512K external SRAM memory, a 10BaseT ethernet PHY+MAC controller and an interface to a CMOS image sensor demonstrator board.
Thanks to the OCAPI-xl methodology, this application could be designed in only 14 man-months. It is described in 25 concurrent processes that are communicating with each other using 32 semaphores, 32 messages and are sharing 45 variables. FLI' s are used for every interface to external system components. These are two 512Kbyte SRAM memories, the image sensor demo-board interface, the ethernet controller interface and the configuration memories for reconfiguring purposes. The interface descriptions were abstracted from the system description during the design in a Board Abstraction Library. They where included during simulation for verification reasons, and resulted in the required pin-assignments upon implementation.

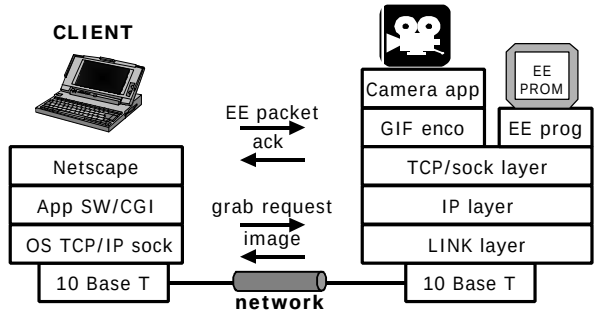

Figure 6: Application structure

The initial implementation was a stand-alone all hardware implementation on the FPGA, and thanks to the unified description the implementation is now being migrated to a processor based platform to optimize the resource usage in the system.

4. CONCLUSIONS

In this paper OCAPI-xl, a C++ class library for true unified Hardware / Software modeling that allows taking partitioning decisions anywhere in the design flow was presented. This is achieved by describing the system model in a refinable, a target implementation independent way. Communication is expressed using a software-like approach, using messages, semaphores and shared variables. To keep processes target independent, it uses a lazy definition approach, or generates use-based interfaces, depending on the chosen implementation target. To model external system components, and to get system ports to these components, a powerful FLI method was introduced. It also provides code generation possibilities to get synthesizable code out of your system model both for hardware and software.
As a proof of concept for the OCAPI-xl methodology, a standalone NetCam was designed implementing an interface to a digital CMOS image sensor, a GIF engine, a network layer and an interface to an off-the-shelf ethernet controller. This was achieved in only 14 man-months, thanks to our methodology.

REFERENCES

[1] Stevens, R.: *TCP/IP Illustrated: Volume 1 & 3*. Addisson-Wesley, 1994 & 1996.

[2] Schaumont, P.; Vernalde, S.; Rijnders, L.; Engels, M.; Bolsens, I.: *A Programming Environment for the Design of Complex High Speed ASICs*. Proceedings Design Automation Conference 1998.

[3] Chang,H.;Cooke,L.;Hunt,M.;Martin,G.;McNelly,A.;Todd, L.: *Surviving the SOC Revolution, A Guide to Platform-Based Design.* Kluwer Academic Publishers, 1999

[4] Stevens, R.: *Advanced programming in the UNIX environment*. Addisson-Wesley, 1999.

[5] Verkest, D.; Cockx, J.; Potargent, F.: *On the use of C++ for system-on-chip design.* Proceedings of the IEEE Workshop on VLSI (IWV), 1999.

[6] SpecC Technology Open Consort. : www.specc.org

[7] The Open SystemC Initiative: www.systemc.org

[8] Cadence VCC : HW/SW Co-design environment: www.cadence.com/datasheets/vcc_environment.html

[9] Arnout,G.: *C for System Level Design*. Proceedings Design, Automation and Test in Europe Conference, 1999

[10] Balarin, F. et al.: *Hardware-Software Co-design of Embedded Systems – The POLIS experience.* Kluwer Academic Publishers, 1997.

[11] EECS UCBerkeley: *Heterogeneous Modeling and Design:Ptolemy Project*: http://ptolemy.eecs.berkeley.edu

[12] Jersak, M.; Ziegenbein, D.; Wolf, F.; Richter, K.; Ernst, R.; Cieslok, F.; Teich, J.; Strehl, K.; Thiele, L.: *Embedded System Design using the SPI Workbench.* Proceedings 3rd International Forum on Design Languages, 2000.

[13] Lavagno, L.; Sentovich, E.M.: *ECL: A Specification Environment for System-Level Design.* Proceedings Design Automation Conference 1999.

[14] Virtual Socket Interface Alliance (VSIA): www.vsi.org