Assisting Refinement in System-on-Chip Design

Hocine Mokrani, Rabéa Ameur-Boulifa
Institut Telecom,
Télécom ParisTech, LTCI CNRS,
Sophia-Antipolis, France
Email:firstname.lastname@telecom-paristech.fr

Emmanuelle Encrenaz-Tiphene
Laboratoire d'Informatique de Paris6 (LIP6) CNRS
Université Pierre et Marie Curie, Paris, France
Email:emmanuelle.encrenaz@lip6.fr

***Abstract*—With the increasing complexity of systems on chip, designers have adopted layer design methodologies, where the description of systems is made by steps. Currently, those methods do not ensure the preservation of properties in the process of system development. In this paper we present a system on chip design method in order to guarantee the preservation of functional correctness along the design flow.**

I. Introduction

The System-on-Chip (SoC) design faces a trade-off between the manufacturing capabilities and time to market pressures. With the increasing complexity of architectures and the number of parameters, the difficulty to explore a huge design space becomes increasingly harder to address.

An approach to overcome this issue is to use abstract models and to split the design flow into multiple-levels, in order to guide the designer in the design process, from the most abstract model down to a synthesizable model. The use of abstraction levels in the SoC design gives another perspective to cope with design complexity. Indeed, the design starts with a functional description of the system, where only the major function blocks are defined and timing information is not yet captured. During the SoC design process, the system description is refined step by step and details are gradually added. At the end, this process leads to a cycle accurate fully functional system description in Register Transfer Level (RTL).

Furthermore, verification of complex SoCs requires new methodologies and tools, which include the application of formal analysis technologies throughout the design flow. Indeed, in contrast to simulation technique, formal verification can offer strong guarantees because it explores all possible execution paths of a system (generally in a symbolic way); in case of Model-Checking, the verification can be automated but has to face the state explosion problem. This approach is applicable for the first steps of the design process or on elementary blocks of the refined components; it can also help in proving the refinement between two successive steps of the design process.

This paper proposes a method for assisting the process of refinement along the design flow. The approach is based on a set of transformation rules, representing a concretisation step; the transformation rules are coupled with formal verification techniques to guarantee the preservation of stuttering linear-time properties, hence alleviating the verification process on the last steps of the design and paving the way to a better design space exploration.

This paper is structured as follows. Section II summarizes the related techniques in the literature. Section III describes the major steps of our method for architectural exploration. Section IV details the transformation rules associated to each refinement step. Section V presents a case-study illustrating the use and benefits on can expect from our approach. Section VI concludes and sketches some perspectives.

II. Related work

Nowadays many design methodologies incorporate formal verification to assist the design; generally, verification tools are plugged into the standard (SystemC or SystemVerilog) design flow. These tools are adequate to perform formal verification at a high level of abstraction, or to derive test-benches tested generally used for assertions checking on lower levels. There is a lack of design methodologies assisting a designer in refinement tasks and offering guarantees about functional properties preservation along the design process. Several frameworks offering design-space exploration facilities have been proposed [1], [2], [3], [4], but they are mostly simulation-oriented and do not formally characterizes the relationships between two successive abstraction levels, and the formal verification of global functional properties at low level is very hard to accomplish.

Among design methodologies oriented towards refinement, the B method [5] is one of the most famous, due to its rigorous definition, its (partial) mechanization in Atelier-B, and several success stories for transportation devices. This approach is very general and could be applied in the context of SoC design (as in [6]), but the refinement steps are left to the user, (although a large part of proof obligations can be automatically discharged). [7] introduces a model algebra that represents SoC designs at system level. The authors introduce the refinement as a sequence of model transformations, corresponding to manipulation of algebra expressions. The correctness proof is based on the laws of model algebra. Functional equivalence verification is used to compare the values of input and output variables within the models at different levels. [8] presents a framework for computation and communication refinement for multiprocessor SoC designs. Stochastic automata networks are used to specify application behaviour which allows performance analysis and fast simulations.

Our approach is complementary to these last works since we provide transformation rules, representing the introduction of architectural constraints in the design in order to describe more precisely its functioning. Our rules are tuned to be understandable by the designer, who can select which combination

of rules to apply in order to perform its refinement; at each step, the refinement can be proven by applying automated verification tools, hence guaranteeing the preservation of a large class of functional properties from abstract levels to more concrete ones.

III. Our method

Our approach for design space exploration of Systems-on-Chip is based on the Y-chart design scheme [9] (Figure 1). We focus on data-flow applications, modelled as a set of abstract concurrent tasks. Application tasks and architectural elements composing the underlying execution support (e.g. major features of CPU, memory, bus) are first described independently and are related in a subsequent mapping stage in which tasks are bound to architectural elements.

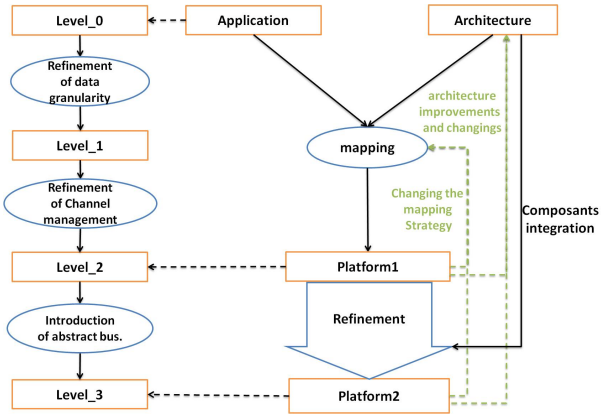

Fig. 1. Refinement steps in the design flow

The application is mapped onto the architecture that will carry out its execution: a first platform is available (cf. Fig. 1). The models derived for both applications and architectures may come with some low-level information from designers They are analyzed to determine whether the combination of application and architecture satisfies the required design constraints introduced at the initial stage. If the design constraints are not met, then the mapping process is reiterated with a different set of parameters until achieving a satisfactory design quality. Once the desired platform is obtained, it is possible to perform communication refinement for optimizing the communication infrastructure, making effective the communication mechanism, and taking into account constraints imposed by the available resources. This leads to Platform2 on Fig. 1.

This process is well established in the simulation-based design exploration tools. However, due to structural constraints imposed by the boundedness of the execution support and some synchronization it induces, the initial set of execution traces of the application is modified along the mapping and refinement process. This means that functional properties that were fullfilled by the initial description of the application may no longer hold once the application has been mapped: some deadlocks or livelocks may have appeared, or some good ordering of events may not be respected anymore. These changes are very difficult to capture with simulation-only engines, and formal analysis is required.

In order to ensure the preservation of the functionality of the application being analysed along the mapping and refinement process, our approach is to split the whole process in defined steps, whose abstraction level is clearly established (cf. Fig. 1, left): Level-0 (application without constraint), Level-1 (Application with a defined granularity of data stored and exchanged), Level-2 (Application with synchronization mechanisms for communications) and Level-3 (Application with synchronization mechanisms for communication transiting through a shared bus). Moreover, we provide formal transformation rules as guidelines for the derivation of a concrete model from an abstract one. Then we can prove the preservation of stuttering linear-time functional properties from two successive representation levels by comparing the set of traces of the two descriptions with a formal verification tool. The remainder of this section gives some precisions on the initial application and architecture modelings.

A. Application

The functional behaviour of the application is written in TML language (Task Modelling Language) [3]. The model of computation of TML is close to the Kahn networks model [10], however TML supports non-determinism and offers different kinds of communication. A TML model is a set of asynchronous tasks, representing the computations of the application, and communicating through channels, events or requests. Each task executes forever: first instruction is re-executed as soon as the last one finishes.

The main feature of TML models is *data abstraction*. TML models are built to perform design space exploration from a very abstraction level; they capture the major parameters of the application to be mapped, without describing precisely the computation of the application and data value being involved in it. Within tasks, precise computation are abstracted by an action EXEC whose optional parameter represents the amount of time the computation should take. Channels do not carry data values, but only the amount of data exchanged between tasks. Data are expressed in terms of *samples*. A sample has a type which defines its size. Communications are expressed by actions READ or WRITE whose parameters are the channel being accessed and the amount of (typed) samples to be read or written. Other constructs are provided to perform conditional loops, or alternatives (the guarding condition may be non-determinictic, abstracting a particular computation value).

Channels are used for point-to-point unidirectional buffered communication of abstract data, while *events* are used for control purpose and may contain values, and *requests* can be seen as one-to-many events. A channel may have a maximal capacity or being unbounded, and is accessed through READ or WRITE actions performed by the emitter and receiver tasks. Channel's type describes its access policy and the type of samples it stores. A channel can be either "Blocking-Read/Blocking-Write" (BR-BW), mimicking a bounded queue (its maximal capacity is defined in its declaration), "Non-Blocking-Read/Non-Blocking-Write" (NBR-NBW) to represent a memory element or "Blocking-Read/Non-Blocking-Write" (BR-NBW) to represent an unbounded queue.

A TML application composed of five tasks and five channels of different sizes is presented on Figure 2; it will be further detailed in sec. V.

B. Execution Platform

The architecture describes the number, the interconnexion and the main features of the hardware components on which the application will be executed. For each *processing element* (e.g. processor or co-processor), one provides its number of cores, number of communication interfaces, size of local memory. In case of multitask scheduling, the scheduling policy is specified (fixed-priority, random, round-robin). For each storage element (e.g. RAM, ROM, buffer), the size of the storage element and access policy (random access, FIFO) are given. For each *interconnection* or *interface element*, one specifies the type of interconnection (dedicated buffered line, shared bus, full-crossbar, bridge ...), granularity transfer, arbitration policy.

An abstract behavioral model is attached to each of these components; the behavioral model exhibits the synchronizations and ressources constraints imposed by the corresponding component; this model is described as a *Labeled Transition System* (LTS for short [11]), which is hidden to the designer.

C. Mapping and partitioning

The mapping process distributes application tasks and channels over hardware elements: it determines over which processing elements tasks are executed and the memory regions storing data. The *allocation* is static and described by the designer. A model of architecture, and the mapping of an application is sketched in Fig. 3.

The mapping corresponds to the *combination* of the behavioral models of the application elements with those coming from the architecture according to the allocation chosen. This combination can be seen as a product of LTS, however in order to perform such a product, one has to adapt the communication granularity, the interface protocols and the ressources' sharing resolution. This combination process is described in the following section.

IV. Transformation rules

To assist the designer in developing models from Level-0 to Level-3, we provide guidelines for formally refining the tasks and communication medium from the simple channels to concrete infrastructures. After generating the initial model, the guidelines suggest three steps: *(1)* Refinement of data granularity, *(2)* Refinement of channel management, and *(3)* Introduction of an abstract bus. These transformations manipulates orders and substitutions between elementary actions labelling the LTS of the initial model. In [12], these transformations have been formalized with `pomset`, which is well adapted to represent partially ordered sets, and transposed into LTS formalism. For a sake of brievety, we give an informal presentation of these transformations and refer to [12] for their complete and formal description.

At Level-0, we build behavioural models of TML applications in terms of a set of interacting LTS. For each task, we build an LTS in which the transitions are the atomic actions executed by the TML task. For each channel we generate an LTS which models its specific behaviour (maximal capacity and blocking or non-blocking read/write accesses).

A. Refinement Steps

1) First Step. Refining granularity of data : The first refinement step considers the capacity (or size) of memory elements allocated to each communication channel during the allocation phase. This capacity may be lower than the size of the TML sample to be transmitted, this imposes a rescaling of the granularity of data transfer and may impact the granularity of the computation also. The granularity of data measures the atomic amount of computations associated to each EXEC statement and the atomic amount of data associated to each READ or WRITE statement, hence transiting in each channel. The refinement of data granularity converts the unit of data from the coarse-grained unit (e.g. a type Image) into the fined-grained one (e.g. a type Pixel Block) with a granularity scaling of n (size of type Image = $n\times$ Size of type Pixel Block).

The model of channels is refined by transformation 1 (called T1). This latest associates to each channel a bounded size (defined in number of samples *of the new granularity* it transfers), which has to be compliant with the maximal memory size of the architecture, allocated for the channel.

Transformation 1. *A channel C is transformed into a channel C' with a granularity scaling of n such that:*
$Type(C')$ = BR-BW *if* $Type(C)$= BR-BW *or* BR-NBW, NBR-NBW *otherwise;*
and $size(C') \leq min(MEMSIZE(C), n \times size(C))$.

Models of tasks are impacted also: each initial action is transformed into an ordered set of micro-actions, according to its granularity scaling. These ordered sets are incrementally built and combined, while taking into account parallelism between actions, data dependency and persistency, leading to transformations 2 to 6.

a) Maximal parallelism between actions: For each Level-0 action, the generated order of micro-actions representing it depends on its associated data granularity, as well as on the maximal parallelism the architecture offers. As channels are point-to-point communication media, the generated order for READ and WRITE actions is *total*. The EXEC action is split into micro-actions, whose parallelism is defined by the maximal parallelism degree p offered by the processing unit executing it (e.g. number of cores in processors, ...). The initial order for each action is built by transformation *Expansion of actions* (T2). Then, each group of actions being locally ordered, the order of the task is reinforced by introducing linear order between the terminal element of each group. This is performed by transformation *Global order of actions* (T3).

Transformation 2. *Given a task action $u \in \{READ, WRITE, EXEC\}$, a parameter n (associated to granularity scaling), and a maximal parallelism p, transformation* expansion of actions *consists in replacing action u of the model by a (n, p)-ordered set of actions u' with the same label.*

Transformation 3. *Given a model of a task and for each of its action, an ordered set of micro-actions; identify the terminal element of each expansion group, and build an order relating these terminal actions which respects the order of the initial model.*

b) Data dependency and data persistency: A consequence of data abstraction in TML is the loss of information about the data dependency. This information, which can be expressed as a relation between reading-writing, reading-execution, execution-writing and execution-execution actions, is required for an optimal management of the memory space of the architecture. In fact, this relation can be restored from the algorithm targeted by the application, or provided by the designer. Thereby, this relation strenghten the order between actions by the transformation of *Data dependency introduction* (T4). Moreover, the models of tasks have to ensure that the data in the local space remains available until its use has been completed. This is taken into account by transformation called *data persistency* (T5), which enforces that the execution of all the actions consuming a data precedes its removal.

Transformation 4. *Given the model of a task M and the data dependency relation between its actions D, transformation* data dependency introduction *consists in building the order resulting from the transitive closure of $M \cup D$. One has to ensure that this transformation produces an order (i.e. no cycle is introduced).*

Transformation 5. *Given a model of task, a data dependency relation and a size of local memory, the* data persistency *transformation consists in strenghtening the order of the model of the task such that for any execution, a sequence of actions producing data never exceeds the memory size.*

c) Shared ressources: Another kind of material constraint taken into account is the number of interfaces and the number cores which are included in the processing unit in which the task is executed; this is done by transformation 6.

Transformation 6. *Given a model of a task and the number of interface and cores of the processing element associated to the task, the transformation forces the order between all the actions of the task so as to guarantee a mutual exclusion of shared resources.*

Transformations 5 and 6 are performed through a symbolic traversal of the system, constrained by the order of micro-actions already computed via transformations T2 to T4: at each execution step compatible with the input order, one has to ensure that data persistency or ressource exclusion conditions are satisfied. If these conditions are not met, the order is strengthened, which corresponds to force the sequentialization of concurrent actions, up to the satisfaction of these conditions.

At this point, LTS model of each task or channel is built; the model of Level-1 is a synchronous product of these elementary LTS. The number of atomic execution steps and data transfers are fixed and ordered according the maximal parallelism allowed by the architectural description. However, transfers are still abstracted as ordered micro-actions WRITE or READ. The following step produces a detailled view of these actions.

2) Second Step. Refining channel management: In the second step, channels are replaced by communication media equipped with an abstract protocol respecting the blocking-read/blocking-write semantic. The protocol is inspired from [13]: the reading and writing primitives are expressed by a serie of operations that stall a process until data or space (room) is available, then transfer data in the architecture and finally signal the availability of room or data. This protocol uses six actions: CHECK-ROOM and SIGNAL-ROOM to test and inform a room is available for writing; STORE-DATA and LOAD-DATA to perform the transfer; and CHECK-DATA and SIGNAL-DATA to check and inform the availability of a data to read. The model of channels are replaced by the media, according to transformation 7 and within each task, READ and WRITE micro-actions are transposed according to transformations 8 and 9.

Transformation 7. *Given a channel of type* BR-BW*, the model of channel is replaced by the model describing the above protocol of communication.*

The introduction of a communication protocol impacts the models of tasks. The transformation is performed in two phases: the first one replaces all the transfer actions in the channels of type BR-BW by the details of the protocol:

Transformation 8. *Given a model of a task, and the selected communication protocol, the actions of transfer to a channel* BR-BW *are transformed by the following rules:*

WRITE $\equiv$ CHECK-ROOM $\rightarrow$ STORE-DATA $\rightarrow$ SIGNAL-DATA
READ $\equiv$ CHECK-DATA $\rightarrow$ LOAD-DATA $\rightarrow$ SIGNAL-ROOM

In fact, the actual transfer of data are the operations STORE-DATA and LOAD-DATA, the other operations are synchronization primitives. In the second phase the orders between the primitive of transfer and the primitive of synchronization are calculated according to the order established in the abstract model, in a way to preserve a maximal parallelism.

Transformation 9. *Given a model of a task and the selected communication protocol. The transformation* introduction of the protocol *consists in restoring of the orders between actions according to the rules given in Table I (Each line presents a replacement pattern. Left : Level-1 ordered actions to be replaced, Right : Level-2 ordered actions replacement).*

(1)	READ→ READ	CHECK-DATA→LOAD-DATA→SIGNAL-ROOM ↓ ↓ ↓ CHECK-DATA→LOAD-DATA→SIGNAL-ROOM
(2)	WRITE→ WRITE	CHECK-ROOM→STORE-DATA→SIGNAL-DATA ↓ ↓ ↓ CHECK-ROOM→STORE-DATA→SIGNAL-DATA
(3)	WRITE→ READ	CHECK-ROOM→STORE-DATA→ SIGNAL-DATA ↓ ↓ ↓ CHECK-DATA → LOAD-DATA →SIGNAL-ROOM
(4)	READ→ WRITE	CHECK-DATA → LOAD-DATA →SIGNAL-ROOM ↓ ↓ ↓ CHECK-ROOM→STORE-DATA→ SIGNAL-DATA
(5)	READ→ EXEC	CHECK-DATA→LOAD-DATA→SIGNAL-ROOM ↓ EXEC
(6)	WRITE→ EXEC	CHECK-ROOM→STORE-DATA→SIGNAL-DATA ↓ EXEC
(7)	EXEC→ READ	EXEC ↓ CHECK-DATA→LOAD-DATA→SIGNAL-ROOM
(8)	EXEC→ WRITE	EXEC ↓ CHECK-ROOM→STORE-DATA→SIGNAL-DATA
(9)	EXEC→EXEC	EXEC→EXEC

TABLE I. TRANSFORMATION OF THE ACTIONS ORDER

3) Third Step. Introduction of abstract bus: In this step we introduce information of sharing communication infrastructures. We define an abstract protocol for bus management. The selected protocol targets a wide family of centralized buses: it contains an arbitration component and interface modules to mimick initiator and target interfaces, and provides a transfer policy abstracting atomic, burst or split transfers.

B. Generation of models for Level-1, Level-2 and Level-3

From an initial model (Level-0), the elementary refinement steps are applied to build the intermediate models (Level-1 and Level-2) up to the most concrete one (Level-3).

Let M^0 be the LTS resulting from the synchronized product of LTS representing the tasks (LTS_t^0 for each task t) and channel (LTS_c^0 for each channel c) at Level-0. In the following, $||$ denotes the synchronized product.

$$M^0 = (||_{t \in Task}\ LTS_t^0) \ ||\ (||_{c \in Channel}\ LTS_c^0)$$

The LTS of Level-1, M^1, is a synchronized product of the transformed channels (LTS_c^1) and tasks (LTS_t^1). Each channel is modified by application of transformation 1: $\forall c \in$ Channel: $LTS_c^1 = T1(LTS_c^0)$. Each task is modified by application of transformations 2 to 6: $\forall t \in$ Task: $LTS_t^1 = T6(T5(T4(T3(T2(LTS_t^0)))))$.

$$M^1 = (||_{t \in Task}\ LTS_t^1)) \ ||\ (||_{c \in Channel}\ LTS_c^1))$$

In the same way, the LTS of Level-2, M^2, is a synchronized product of the transformed channels (LTS_c^2) and tasks (LTS_t^2). Each channel is modified by application of transformation 7: $\forall c \in$ Channel: $LTS_c^2 = T7(LTS_c^1)$. Each task is modified by application of transformations 8, 9 and 6: $\forall t \in$ Task: $LTS_t^2 = T6(T9(T8(LTS_t^1)))$

$$M^2 = (||_{t \in Task}\ LTS_t^2)) \ ||\ ((||_{c \in Channel}\ LTS_c^2))$$

And finally, the LTS of Level-3 is a synchronized product of the transformed channels and tasks and components introduced in the third step, by applying similar transformations.

V. Case Study

This section illustrates the use of the approach for the design and functional verification of a digital camera initially presented in [14]. The functional specification is partitioned into five modules, namely CCD, CCDPP, CODEC, TRANS, and CNTRL.

The digital camera captures, processes, and stores pictures into an internal memory. This task is initiated when the user presses the shutter button to take a picture. The CCD model simulates the capture of a picture and the transmission of pixels. The CCDPP module performs the luminosity adjustment on each pixel received from CCD module. The CODEC module applies the DCT (Discrete Cosine Transformation) algorithm to each bloc transmitted from CNTRL before being retransmitted into the CNTRL. The CNTRL module serves as the controller of the system, it also executes the quantization and Huffman compression algorithm after receiving the transformed bloc from CODEC. The camera is able to upload the stored picture to external permanent memory. The TRANS module takes care of this task when it receives data from CNTRL.

```
CHANNEL  CI1  Image1 BRBW  1  CCD   CCDPP
CHANNEL  CI2  Image2 BRBW  1  CCDPP  CNTRL
CHANNEL  CI3  Image2 BRBW  1  CNTRL  TRANS
CHANNEL  CB1  Bloc   BRBW  1  CNTRL  CODEC
CHANNEL  CB2  Bloc   BRBW  1  CODEC  CNTRL

TASK CCDPP{
   read CI1
   exec
   write CI2
   }
TASK CCD {
   write CI1
   }
TASK TRANS{
   read CI3
   }

TASK CNTRL{
   read CI2
   Repeat N times
        write CB1
        read CB2
        exec
   Endrepeat
   write CI3
   }

TASK CODEC
   {
   Repeat N times
        read CB1
        exec
        write CB2
   Endrepeat
   }
```

Fig. 2. TML code of Digital Camera

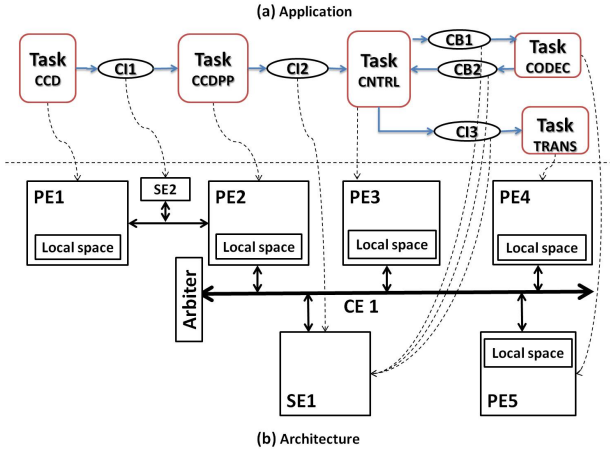

Fig. 3. Architecture and Mapping of Digital Camera

Based on the code given in [14], we model this application in TML (Fig. 2), provide an architecture and a mapping (Fig. 3). The architecture is made of five processing elements (PE1 to PE5) equipped with local memory. PE1 and PE2 communicates through a dedicated buffered line SE2; PE2 to PE5 and a share memory SE1 are connected through a bus, whose access is controlled by an arbiter ARBITER. The allocation is represented with dashed lines from the task graph on the upper part of Fig. 3; it associates one TML task per processing element; channel CI1 will be implemented on buffered line SE2 while all other channels are implemented into the shared memory SE1. A maximal capacity is associated to each memory space.

We specified the models M^0, M^1, M^2 and M^3 of the four levels of refinement from Level-0 upto Level-3 following the generation scheme described in Sec. IV-B. The code of the LTSs in each model M^0 downto M^3 is written in Fiacre [15], and the property verification and refinement checking between successive levels are performed using CADP version 2008 [16]. We prove complete and infinite trace inclusion between lower levels and higher levels by proving the existence of simulation relations between two successive models (e.g. $\forall i : M^{i+1} \sqsubseteq M^i$); this result ensures the preservation of stuttering linear-time properties from M^0 downto M^3.

The properties are the following, expressed in logic MCL, one of the input format for CADP. In these expressions, square brakets represent universal quantifier of the internal regular expression representing a path, and chevron brackets represent existential quantifier. Atomic propositions are Level-0 action labels, which are preserved along the refinement process.

- Absence of deadlock state in the system.
  P0: $[true^*] < true >$
- For each data written on a channel **CI1** the data must be read in the future.
  P1: $[true^* \ . \ write\_CI1 \ . \ true^* \ . \ read\_CI1]$
- The actions sequence for task **CCDPP** is always executed.
  P2: $[true^* \ . \ read\_CI1 \ . \ true^* \ . \ exec\_1 \ . \ true^* \ . \ write\_CI2]$
- For each data written by task **CCD**, the data is read by the task **TRANS** in the future.
  P3: $[true^* \ . \ write\_CI1 \ . \ true^* \ . \ read\_CI3]$
- A data written into channel **CI2** is never overwritten before having been read:
  P4: $[true^* \ . \ write\_CI2 \ . \ (not \ write\_CI2)^* \ . \ read\_CI2]$

These properties are preserved from M^0 downto M^3. We compared the timed needed for the verification of these properties in a basic strategy were no formal relation is established between levels, and in our refinement-based strategy. In the basic strategy, the properties have to be verified at Level-3. In the refinement-based strategy, once the validity of the properties has been established in M^0 and the refinement are checked, the properties are guaranteed to be true in subsequent levels.

Table II presents quantitative information about the size of the models M^0 downto M^3, the time to verify the properties for each level, and the time to verify the trace inclusion between two successive models. These results show the benefits of the refinement-based strategy (compare *sum of times of column LEVEL-0 + sum of times of line "Refinement"* with *sum of times of column LEVEL-3*).

	LEVEL_0	LEVEL_1	LEVEL_2	LEVEL_3
♯state M^i	336	123088	437782	173546709
♯transition M^i	872	431622	1646712	472413097
verif time P0	$0,4s$	$12s$	$7mn$	$25mn$
verif time P1	$0,5s$	$16s$	$9mn$	$32mn$
verif time P2	$0,7s$	$23s$	$14mn$	$47mn$
verif time P3	$0,6s$	$17s$	$10mn$	$33mn$
verif time P4	$0,7s$	$24s$	$13mn$	$45mn$
Refinement	$\times$	$0.14s$	$3s$	$17s$

TABLE II. TIME OF ANALYSIS

VI. CONCLUSION

We presented a refinement-based methodology for design-space exploration of SOC. Our approach provides guidelines to assist the designer in the refinement process, focusing on communication refinement. We established well-identified abstraction levels and transformation rules to derive a more concrete model from a more abstract one. Each abstraction level can be associated with a verification environment, in order to prove functional properties or refinement properties between different abstraction levels. This last point allows us to establish the validity of functional properties of concrete descriptions by testing the property on the most abstract level and proving the refinement, which is less costly than verifying the property on the concrete model directly; we examplified this fact on a digital camera case study.

These encouraging results draws several perspectives. A first direction consists in proving that the transformation algorithm always produces a refinement, for *any* initial (deadlock-free) model; up to now, the refinement is established when the transformations are applied to a particular initial model. Another perspective concerns the extension of the approach to task computation refinement.

REFERENCES

[1] V. Zivkovoc, E. Deprettere, P. Van Der Wolf, and E. De Kock, "Design space exploration of streaming multiprocessor architectures," in *proc of SIPS'02, San Diego, CA*, 2002.

[2] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout, "A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms," in *Proc. of DATE'05, Munich, Germany*, 2005, pp. 876–881.

[3] L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet, "A UML-based environment for system design space exploration," *13th IEEE International Conference on Electronics, Circuits and Systems, 2006. ICECS 06.*, 2006.

[4] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Trans. Computers*, vol. 55, no. 2, pp. 99–112, 2006.

[5] J.-R. Abrial, *The B-book: assigning programs to meanings*. New York, NY, USA: Cambridge University Press, 1996.

[6] J. L. Colley, "Guarded atomic actions and refinement in a system-on-chip development flow: Bridging the specification gap with event-b," in *Thesis*, 2010.

[7] S. Abdi and D. Gajski, "Verification of system level model transformations," *Int. J. Parallel Program.*, vol. 34, pp. 29–59, February 2006.

[8] R. Marculescu, Ü. Y. Ogras, and N. H. Zamora, "Computation and communication refinement for multiprocessor soc design: A system-level perspective," *ACM Trans. Design Autom. Electr. Syst.*, vol. 11, no. 3, pp. 564–592, 2006.

[9] H. Mokrani, R. Ameur-Boulifa, S. Coudert, and E. Encrenaz, "Approche pour l'intégration du raffinement formel dans le processus de conception des socs," in *Journal Européen des Systèmes automatisés, MSR'11*. Lavoisier, Hermès, 2011, pp. 221–236.

[10] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing '74: Proceedings of the IFIP Congress*, J. L. Rosenfeld, Ed. North-Holland, 1974.

[11] A. Arnold, *Finite transition systems - semantics of communicating systems*, ser. Prentice Hall international series in computer science. Prentice Hall, 1994.

[12] H. Mokrani, "Assistance au raffinement dans la conception de systèmes embarqués," *PHD Thesis (in French), LTCI / Telecom-ParisTech, to appear*, 2013.

[13] P. Lieverse, P. van der Wolf, and E. Deprettere, "A trace transformation technique for communication refinement," in *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, 2001.

[14] F. Vahid and T. Givargis, *Embedded system design - a unified hardware / software introduction*. Wiley-VCH, 2002.

[15] B. Berthomieu, J. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat, "Fiacre: an Intermediate Language for Model Verification in the Topcased Environment," in *ERTS 2008*, Toulouse France, 2008.

[16] H. Garavel, F. Lang, R. Mateescu, and W. Serve, "Cadp 2010: A toolbox for the construction and analysis of distributed processes," in *TACAS'11*, ser. LNCS, M. L. P. A. Abdulla, K. Rustan, Ed., vol. 6605. Saarbrücken, Germany: Springer, Heidelberg, 2011.