Chapter 2
Assisting Refinement in System-on-Chip Design

Hocine Mokrani, Rabéa Ameur-Boulifa and Emmanuelle Encrenaz-Tiphene

Abstract With the increasing complexity of systems on chip, designers have adopted layered 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, based on model transformations—or refinements—in order to guarantee the preservation of functional correctness along the design flow. We also provide experimental results showing the benefits of the approach when property verification is concerned.

Keywords System on a Chip (SoC) · Architecture exploration · Platform-Based Design (PBD) · System modeling · Formal verification · Communication refinement · Property-preservation checking

2.1 Introduction

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

H. Mokrani (✉) · R. Ameur-Boulifa
Institut Télécom, Télécom ParisTech, CNRS-LTCI, Sophia-Antipolis, France
e-mail: hocine.mokrani@eurecom.fr

R. Ameur-Boulifa
e-mail: rabea.ameur-boulifa@telecom-paristech.fr

E. Encrenaz-Tiphene
Sorbonne Universités, Université Pierre et Marie Curie Paris 6,
UMR 7606, LIP6, Paris, France

CNRS, UMR 7606, LIP6, Paris, France
e-mail: emmanuelle.encrenaz@lip6.fr

© Springer International Publishing Switzerland 2015

M.-M. Louërat and T. Maehne (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 311,
DOI 10.1007/978-3-319-06317-1_2

The use of abstraction levels in the SoC design gives another perspective to cope with design complexity. Indeed, the design starts from a functional description of the system, where only the major function blocks are defined and timing information is not captured yet. 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 at Register Transfer Level (RTL).

Furthermore, the 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 the 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 chapter is structured as follows. Section 2.2 summarizes the related techniques in the literature. Section 2.3 describes the major steps of our method for architectural exploration. Section 2.4 details the transformation rules associated with each refinement step. Section 2.5 presents a case-study illustrating the use and benefits one can expect from our approach, concerning behavioral property verification. Section 2.6 concludes and sketches some perspectives.

2.2 Related Works

Nowadays many design methodologies involve formal verification methods to assist the design; generally, verification tools are plugged into the standard (SystemC or SystemVerilog) design flow. These tools are appropriate to perform formal verification at a high level of abstraction, or to derive test-benches generally used for assertions checking on lower design levels. However, there is a lack of design methodologies to assist a designer in the refinement tasks and that offer guarantees about functional properties preservation along the design process. Several frameworks offering design-space exploration facilities have been proposed [3, 9, 15, 18]. However, these frameworks are mostly simulation-oriented and do not formally characterize the relationships between successive abstraction levels. Moreover, formal verification of global functional properties is hard to accomplish when components are described at a low level of abstraction, where many implementation details are provided.

Among design methodologies oriented towards refinement, the B method [2] 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 general and could be applied in the context of SoC design [6]. Although large part of proof obligations can be automatically discharged, the refinement steps are left to the user. Abdi and Gajski [1] defines a model algebra that represents SoC designs at system level. The authors define the refinement as a sequence of model transformations, that allow to syntactically transform a Model Algebra expression into a functionally equivalent expression. The refinement correctness proof is based on the transformation laws of model algebra. Functional equivalence verification is used to compare the values of input and output variables within the models at different levels. Marculescu et al. [11] presents a framework for computation and communication refinement for multiprocessor SoC designs. Stochastic automata networks are used to specify application behavior 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 behavior. 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.

2.3 Our Method

Our approach for design space exploration of SoCs is based on the Y-chart design scheme [14] as shown in Fig. 2.1. We focus on dataflow applications, modelled as a set of abstract concurrent tasks. Application tasks and architectural elements making up 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.

The application is mapped onto the architecture that will carry out its execution: a first platform is available (see Fig. 2.1). 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. This makes effective the communication mechanism, and takes into account constraints imposed by the available resources. Referring to Fig. 2.1, this process leads to Platform2. This process is well established in the simulation-based design exploration tools. The boundedness of the execution support and the synchronizations, which it induces, imposes structural constraints.

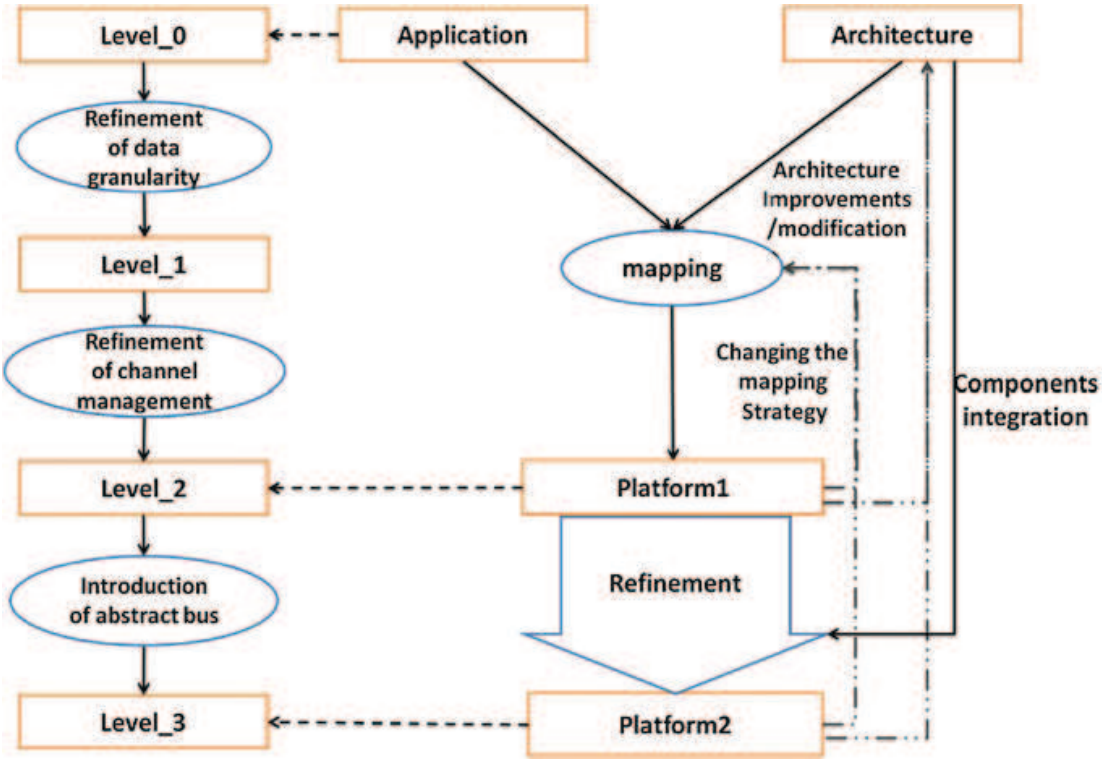

Fig. 2.1 Refinement steps in the design flow

For this reason, the initial set of execution traces of the application is modified along the mapping and refinement process. This means that functional properties that were fulfilled by the initial description of the application may no longer hold once the application has been mapped. For example, deadlocks or livelocks may have appeared, or some good ordering of events may not be respected anymore. These changes are difficult to capture with simulation-only engines, hence formal analysis is required. In order to ensure the preservation of the functional behavior of the application being analysed along the mapping and refinement process, our approach consists of splitting the whole process in defined steps with clearly defined abstraction level (see the left side of Fig. 2.1): Level-0 (application without constraint), Level-1 (application with a defined granularity of the stored data and the transferred data), Level-2 (application with synchronization mechanisms for communication) 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 precision on the initial application and architecture modeling.

2.3.1 Application

The functional behavior of the application is written in Task Modelling Language (TML) [3]. The model of computation of TML is close to the Kahn networks model [8], however TML supports non-determinism and offers different communication styles. A TML model is a set of asynchronous tasks, representing the computations of the application, and communicating through channels, events or requests. In TML, each task executes forever, i.e., the 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 abstract level; they capture the major features of the application to be mapped, without describing precisely the computation of the application and data value being involved on it. Within tasks, precise computation is 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-deterministic, 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. *Requests* in their turn can be seen as one-to-many events. A channel may have a maximal capacity or may be 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 simple example of a TML application is depicted in Fig. 2.2a. It shows two tasks, named TASK1 and TASK2, communicating by FIFO channels, named C1 and C2. TASK1 performs infinite amount of computations and writing actions of a single sample on the channel C1. For each component of the application an abstract model is derived. It captures the component key behavior including both computation and communication aspects. We rely on the *Labelled Transition System* (LTS) formalism [4] for encoding the models.

2.3.2 Execution Platform

The architecture consists of a set of interconnected hardware components, on which the application will be executed. For each *processing element* (e.g., processor or co-processor), the designer provides its number of cores, number of communication

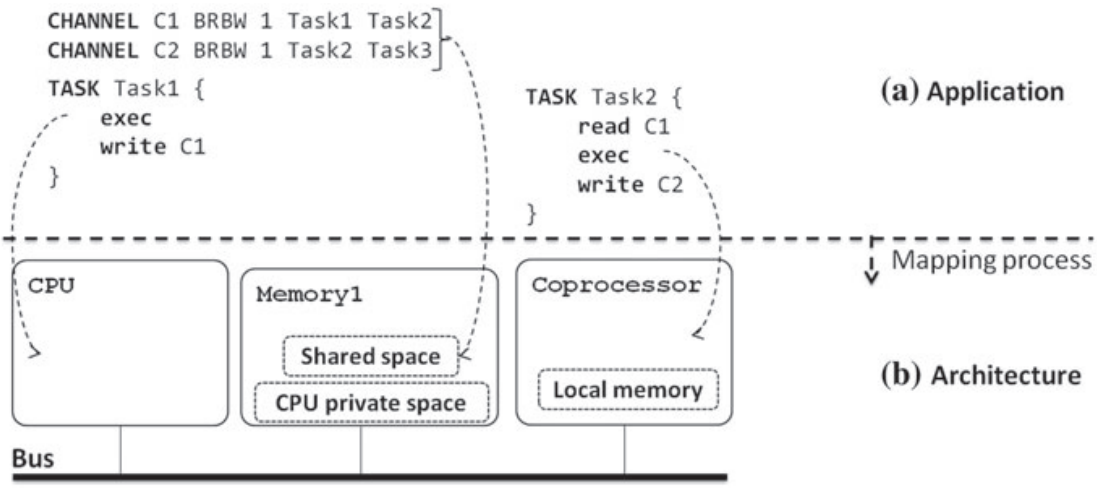

Fig. 2.2 Example of mapping of an application onto an architecture

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*, the designer specifies the type of interconnection (e.g., dedicated buffered line, shared bus, full-crossbar, bridge), transfer granularity, arbitration policy. Referring to the architecture in Fig. 2.2b, it consists of a CPU and a dedicated coprocessor, both connected to a bus and a memory.

2.3.3 Mapping and Partitioning

The mapping process distributes application tasks and channels over hardware elements. The mapping determines over which processing elements the tasks will be executed and which memory regions will store data. The *allocation* is static and is described by the designer. The model of the obtained system represents the *combination* of the behavioral models of the application components integrating the constraints imposed by the architecture. Consider the application and the architecture given in Fig. 2.2, TASK1 (resp. TASK2) is mapped see dashed arrows over CPU (resp. co-PROCESSOR) nodes and the channels C1 and C2 are mapped over a shared memory and communicate through the bus. A more complex example of application, architecture, and mapping is presented in Sect. 2.5, Fig. 2.10.

From a formalism's point of view, this combination can be seen as a product of Labelled Transition Systems (LTSs). However, in order to perform this product, one has to adapt the communication granularity, the interface protocols and to manage the shared resources. This is the purpose of the transformation rules described in the following section.

2.4 Transformation Rules

To assist the designer in developing models from Level-0 to Level-3, we provide guidelines for formally refine 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 manipulate orders and substitutions between elementary actions labelling the LTS of the initial model. In Mokrani [13], these transformations have been formalized with partially ordered multisets referred to in the term *pomset* [16] and then translated into LTS formalism. This section presents an intuitive description of the transformations required by the three steps. A more formal and complete description of these transformations is presented in Mokrani [13].

To begin at Level-0, we build behavioral models of TML applications in terms of a set of interacting LTSs. 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 encodes its specific behavior and captures the parameters of interest such as maximal capacity and access policy. For instance, the behavior of TASK2 and channels may be modeled by the LTSs presented in Fig. 2.3a, b, respectively.

2.4.1 Refinement Steps

2.4.1.1 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, which imposes a rescaling of the granularity of data transfer and may also impact the granularity of the computation. The granularity of data measures, both, the atomic amount of computations associated to each EXEC statement and the atomic amount of data associated to each READ or WRITE statement (i.e., the amount of data carried away by a channel). The refinement of data granularity converts the unit of data from the coarse-grained unit into the finer-grained one (e.g., from IMAGE into PIXEL) with a given granularity scaling factor of n (so that total size of IMAGE $= n \times$ size of PIXEL). The models of channels are refined by the transformation 1 (denoted by [T1]). The latter associates

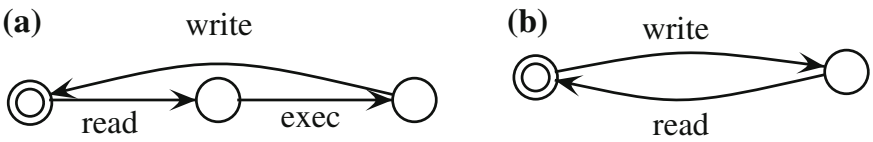

Fig. 2.3 Initial LTSs for TASK2 and channels

to each channel a size bounded by the number of samples *of the new granularity*, which it can transfer, and the maximal memory size of the architecture allocated for the channel given by the *MEMSIZE* function.

Transformation 1 *A channel C is transformed into a channel C' with a granularity scaling factor n such that:*

$$Type(C') = \begin{cases} BR\text{-}BW & if \;\; Type(C) = BR\text{-}BW \vee BR\text{-}NBW \\ NBR\text{-}NBW & otherwise \end{cases}$$

and

$$size(C') \leq min(MEMSIZE(C), n \times size(C))$$

Models of tasks are also impacted by the rescaling of the data granularity. Each initial action is transformed into an ordered set of micro-actions, according to the granularity of a scaling factor. These ordered sets are gradually built and combined by taking into account the parallelism between actions, data dependency and data persistency. The result of this transformation leads to the transformations [T2]–[T6], which are described in the subsequent paragraphs.

Maximal Parallelism Between Actions

For each action of the models derived at Level-0, the order of the corresponding micro-actions depends on the associated data granularity, as well as on the maximal parallelism that the architecture offers. As channels are point-to-point communication media, the generated order for the micro-actions of communication (READ and WRITE) should be *total*. Whereas for the micro-actions of computation (EXEC), it is defined by the maximal parallelism degree *p* offered by the processing unit executing it (e.g., number of cores within processors). Initially, for each action, an order is built by applying transformation 2 *Expansion of actions* (denoted by [T2]).

Transformation 2 *Consider the model of a task characterised by the set of associated actions S. Given a parameter n (granularity scaling factor), and a parameter p (maximal parallelism degree), the transformation* expansion of actions *consists in replacing each action of S by a (n, p)-ordered group of actions with the same label.*

In the case of TML model $S = \{$READ, WRITE, EXEC$\}$, it contains primitives of TML model. Once each group of (micro-)actions being locally ordered, the order of the tasks is reinforced by introducing linear order between the terminal elements of each group. This is performed by the transformation 3 *Global order of actions* (denoted by [T3]).

Transformation 3 *Consider the model of a task and the associated set of the groups of ordered micro-actions. This transformation consists of identifying the terminal element of each group of micro-actions, and building an order relating these terminal actions which respects the order of the initial model.*

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 strengthens the order between actions by the transformation 4 *Data dependency introduction* (denoted by [T4]).

Transformation 4 *Consider the model of a task and the relation R between all its actions. Given a data dependency relation D between its actions, the* data dependency introduction *transformation consists in building the order resulting from the transitive closure of $R \cup D$, so that it produces an order (i.e., no cycle is introduced).*

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 5 *Data persistency* (denoted by [T5]). It enforces that the actions consuming data are executed before the ones removing it.

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

Shared Resources

Another kind of material constraint taken into account along the refinement process is the number of interfaces and the number of cores which are included in the processing unit onto which the task is executed; this is done by transformation 6 (denoted by [T6]).

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

Actually, transformations [T5] and [T6] are performed through a symbolic traversal of the model of a task. They are constrained by the order of micro-actions already computed by way of transformations [T2]–[T4]. At each transformation step compatible with the input order, one has to ensure that both data persistency and resource exclusion conditions are satisfied. If these conditions are not met, the order is strengthened, which forces some sequentialization of concurrent actions, up to the satisfaction of persistence and exclusion conditions.

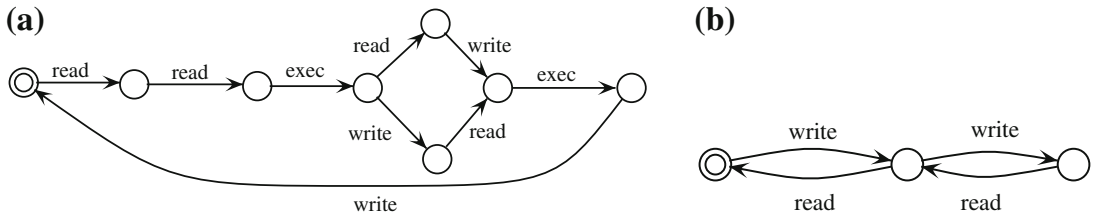

Fig. 2.4 Behavioral models of TASK2 and channels resulting from the first refinement step

The result obtained after applying these transformations is a set of possible execution traces of the considered application, which satisfies imposed design constraints. Referring to the example in Fig. 2.2, we suppose that the written data (resp. read) are refined with a scaling factor equal to 2 (resp. 3). Suppose also that the space allocated to each channel does not exceed two storage compartments. The models of the TASK2 and the channels obtained at the end of this refinement step are shown in Fig. 2.4.

At this stage, the number of atomic execution steps and data transfers are fixed at the granularity offered by the size of memories storing channels, and ordered according the maximal parallelism allowed by the architectural description. However, the communication have to be refined to reflect the access policy of the TML channels. The following step produces a detailed view of these actions.

2.4.1.2 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 selected protocol is inspired from [10]. The reading and writing primitives are expressed by a series of operations that 1. stall a process until data or memory space (named room) in the shared memory is available, 2. transfer data, and then 3. signal the availability of data or room. This protocol uses six primitives: 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 actual transfer of data are the primitives STORE-DATA and LOAD-DATA, the other operations are synchronization primitives. Transformation 7 (denoted by [T7]) replaces the models of channels by the ones depicting this protocol.

Transformation 7 *The model of channel of type* BR-BW *is replaced by the model encoding the selected protocol of communication.*

Furthermore, the introduction of a communication protocol impacts as well the models of tasks. The communication primitives within each task (READ and WRITE) are changed to support the protocol. The transformation is performed in two phases (transformations [T8] and [T9]): the first replaces all the communication

(transfer) actions to a channel of type BR-BW by a communication scheme, which respects the protocol.

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

$$\begin{aligned}
\text{WRITE} &\equiv \text{CHECK-ROOM} \rightarrow \text{STORE-DATA} \rightarrow \text{SIGNAL-DATA} \\
\text{READ} &\equiv \text{CHECK-DATA} \rightarrow \text{LOAD-DATA} \rightarrow \text{SIGNAL-ROOM} \\
\text{EXEC} &\equiv \text{EXEC}
\end{aligned}$$

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 *Consider the model of a task and the selected communication protocol. The transformation* introduction of the protocol *consists in restoring the orders between actions according to the rules given in* Table 2.1.

The patterns in Table 2.1 are modeled with pomsets. The pomset formalism is a compact representation of concurrent actions without expliciting interleavings. The ordering of the operations on each pattern reflects the happens-before relationship between the actions. For instance, the first pattern specifies that the system tests the availability of data before its loading and then issues the room-release after the load operation; all operations of the first instance of reading precede the corresponding ones of the second instance:

$$\begin{array}{ccccc}
\text{CHECK-DATA} & \rightarrow & \text{LOAD-DATA} & \rightarrow & \text{SIGNAL-ROOM} \\
\downarrow & & \downarrow & & \downarrow \\
\text{CHECK-DATA} & \rightarrow & \text{LOAD-DATA} & \rightarrow & \text{SIGNAL-ROOM}
\end{array}$$

This representation leads to the interleaving interpretation, so that a system executing actions concurrently is no different from one that executes them in arbitrary sequential order. With this interleaving interpretation, the system modeled by the pomset above represents five linear traces:

1. CHECK-DATA LOAD-DATA SIGNAL-ROOM CHECK-DATA LOAD-DATA SIGNAL-ROOM
2. CHECK-DATA LOAD-DATA CHECK-DATA SIGNAL-ROOM LOAD-DATA SIGNAL-ROOM
3. CHECK-DATA LOAD-DATA CHECK-DATA LOAD-DATA SIGNAL-ROOM SIGNAL-ROOM
4. CHECK-DATA CHECK-DATA LOAD-DATA SIGNAL-ROOM LOAD-DATA SIGNAL-ROOM
5. CHECK-DATA CHECK-DATA LOAD-DATA LOAD-DATA SIGNAL-ROOM SIGNAL-ROOM

Table 2.1 Patterns of transformation of the actions orders. Each line of the table presents a replacement pattern: the *left pattern* is replaced by the one on the *right*

No	Action pattern	Replacement pattern
(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

Back to our example, the abstract channel is transformed into a shared buffer, which separates data-transfer and synchronisation (see Fig. 2.5).

Because of the interleaving of actions of different operations, the resulting LTS for the TASK2 is large. This makes it highly unreadable. We shall give then its pomset representation (see Fig. 2.6).

2.4.1.3 Third Step: Introduction of Abstract Bus

Once the Level-2 models are available, we introduce information of sharing communication infrastructures. We define an abstract protocol for bus management. The proposed protocol targets a wide family of centralized buses, it contains an arbitration component, interface modules (to depict initiator and target interfaces). It provides a

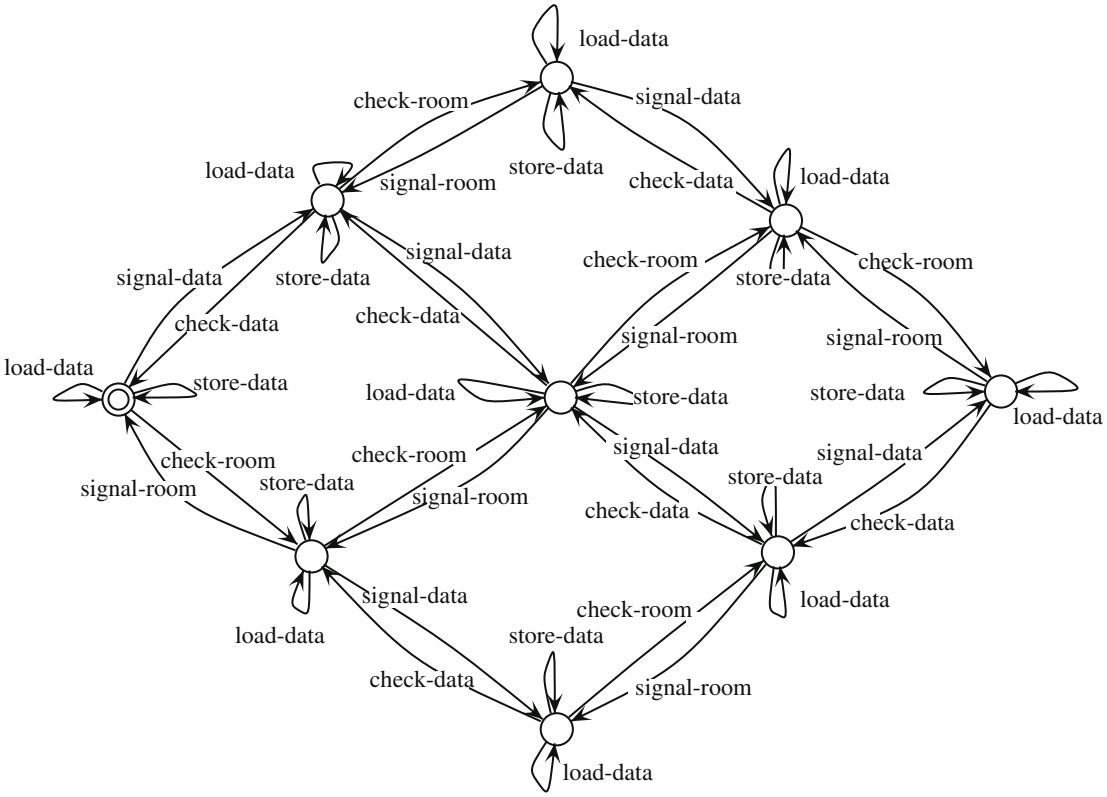

Fig. 2.5 Behavioral model of a two-element channel resulting from the second refinement step

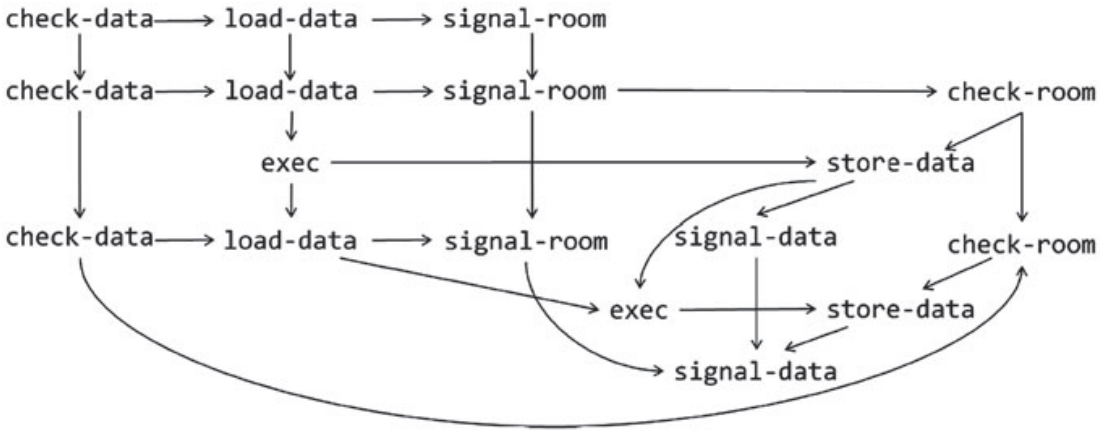

Fig. 2.6 Pomset representation of TASK2 resulting from the second refinement step

transfer policy abstraction which does not distinguish between atomic, burst or split transfers. At this stage, the models of channels remain unchanged, whereas, those of tasks are transformed to incorporate the interface modules. Moreover, a generic model of bus arbiter is introduced. This is performed by transformations [T10] and [T11], which we shall not detail here.

2.4.2 Generation of Models for Level-1, Level-2, and Level-3

As usual in the setting of distributed and concurrent systems, we give behavioral model of the application and the application-architecture combination in terms of a set of interacting finite state machines, called Labelled Transition Systems (LTSs). An LTS is a structure consisting of states with transitions, labelled with actions, between them. The states model the system states; the labelled transitions model the actions that a system can perform.

At each level i, we build an LTS for each component (LTS_t^i and LTS_c^i for resp. task and channel) of the system obtained at this level and by synchronous product (denoted by ||) of elementary LTSs we build the global model of the overall system (M^i):

$$\forall i \in \{0, 1, 2\}.\ \text{M}^i = ((||_{t \in Task}\ \text{LTS}_t^i)\ ||\ (||_{c \in Channel}\ \text{LTS}_c^i))$$

The LTS models of the highest level (Level-0) are generated automatically from the source code TML application. The intermediate models (of Level-1 and Level-2) up to the most concrete one (Level-3) are generated by applying the transformations of the channel models and the task models. The models of Level-1 are built by applying transformation [T1] to each channel and transformations from [T2]–[T6] to each task:

$$\forall c \in Channel : \text{LTS}_c^1 = \text{T1}(\text{LTS}_c^0) \text{ and } \forall t \in Task : \text{LTS}_t^1 = \text{T6}(\text{T5}(\text{T4}(\text{T3}(\text{T2}(\text{LTS}_t^0)))))$$

The models of Level-2 are built by applying transformation [T7] to each channel and transformations from [T6]–[T9] to each task:

$$\forall c \in Channel : \text{LTS}_c^2 = \text{T7}(\text{LTS}_c^1) \text{ and } \forall t \in Task : \text{LTS}_t^2 = \text{T6}(\text{T9}(\text{T8}(\text{LTS}_t^1)))$$

Notice that the transformation [T6] is reused at this level. Indeed, it consists in guaranteeing the exclusive access to resources.

Finally, the global LTS of the Level-3 (M^3) is obtained by the synchronized product of the models of channels and of tasks, plus the models of components introduced in the third step:

$$\text{M}^3 = ((||_{t \in Task}\ \text{LTS}_t^3)\ ||\ (||_{c \in Channel}\ \text{LTS}_c^2)\ ||\ (||_{i \in Interface}\ \text{LTS}_i)\ ||\ (||_{a \in Arbiter}\ \text{LTS}_a))$$

In the same way as previous steps, LTS_t^3 is obtained by applying transformations [T10] and [T11] to the models of tasks.

2.4.3 Proof of Property Preservation

Once the behavioral models have been generated, we prove complete and infinite trace inclusion between lower levels and higher levels by proving the existence of a simulation relation between two successive models (e.g., $\forall i : \text{M}^{i+1} \sqsubseteq \text{M}^i$); this

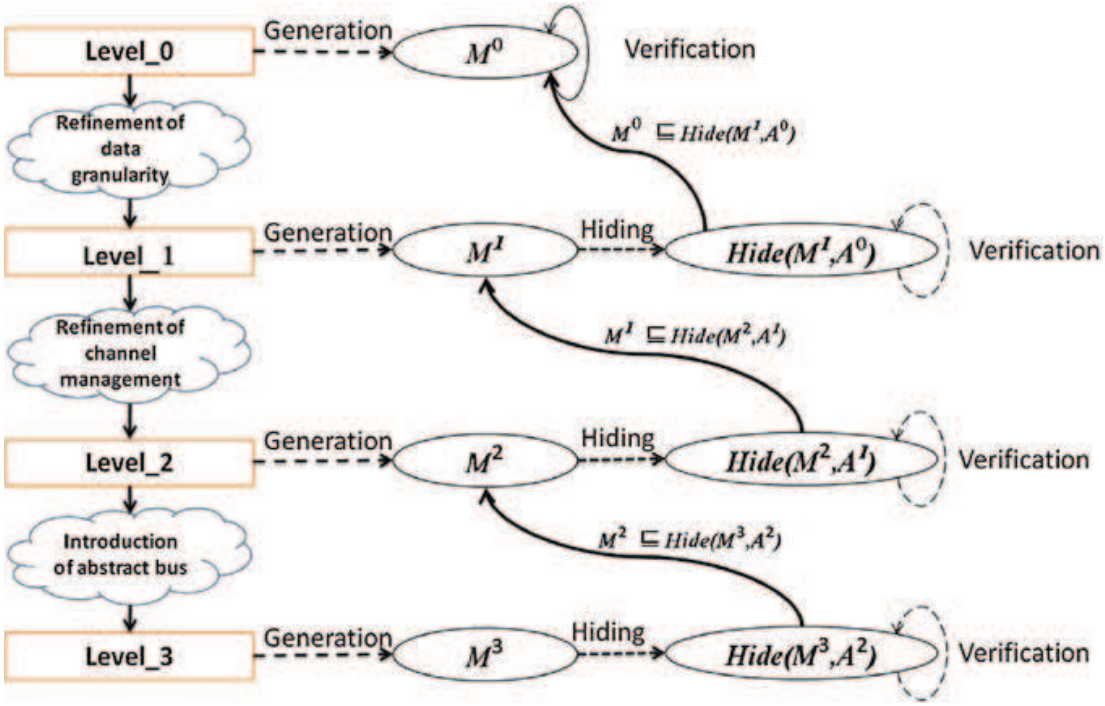

Fig. 2.7 Verification process with the first strategy of hiding operation

result ensures the preservation of stuttering linear-time properties from M^0 down to M^3. Actually, the refinement process introduces new actions, so that the set of actions of abstract level is included in the set of concrete level, $A^i \subseteq A^{i+1}$. Refinement checking between successive levels requires the hiding of the additional details about its behavior. To perform this, we used two strategies:

1. we kept the transitions of M^i (for $i > 0$) labelled over A^{i-1} but the new ones (from $A^i \backslash A^{i-1}$), introduced by the refinement, were considered as *non observable* τ actions. In terms of traces, we can find the trace of the path σ_{i-1} by removing the transitions labelled by new actions from the path σ_i (see Fig. 2.7).
2. We kept the transitions of M^i (for $i > 0$) labelled over A^0. So we can find the trace of the path σ_0 embedded into the trace of the transitions in σ_i (see Fig. 2.8).

The second-solution is more scalable. Indeed, the full system size obtained with the second strategy is much smaller than the one obtained with the first strategy since we hide more actions. However, the first solution appears more interesting. It allows us to prove a large set of properties: properties related to the different levels not only those related to application level. In our methodology, first, we try the experiment with the first strategy. If it fails, we apply the second one.

2.5 Case Study

This section illustrates the use of the proposed methodology for the design and functional verification of a digital camera initially presented in Vahid and Givargis [17]. The functional specification is partitioned into five modules, namely Charge-Coupled

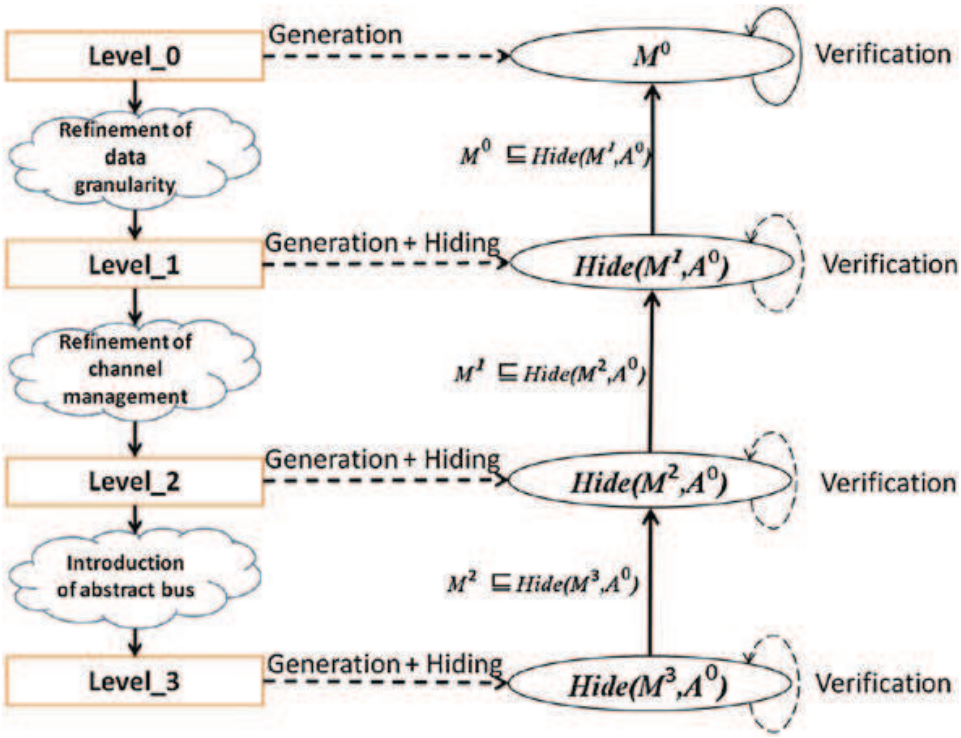

Fig. 2.8 Verification process with the second strategy of hiding operation

Device (CCD), CCD PreProcessing (CCDPP), Discrete Cosine Transformation + quantization (CODEC), transmitter (TRANS), and controller (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 Discrete Cosine Transformation (DCT) 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.

Based on the SystemC code of the application given in Vahid and Givargis [17], we encoded it into TML language (the code is shown in Fig. 2.9), we provide a target architecture, which can support the application and a mapping relationship between them (Fig. 2.10). The architecture consists of five Processing Elements (PE1 to PE5) equipped with their own local memory. PE1 and PE2 communicate through a dedicated buffered line SE2; PE2 to PE5 as well as a shared memory SE1 are connected through a bus, which access is controlled by an arbiter. The allocation is represented with dashed lines from the task graph given on the upper part of Fig. 2.10; 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 mem-

```
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. 2.9 TML code of digital camera

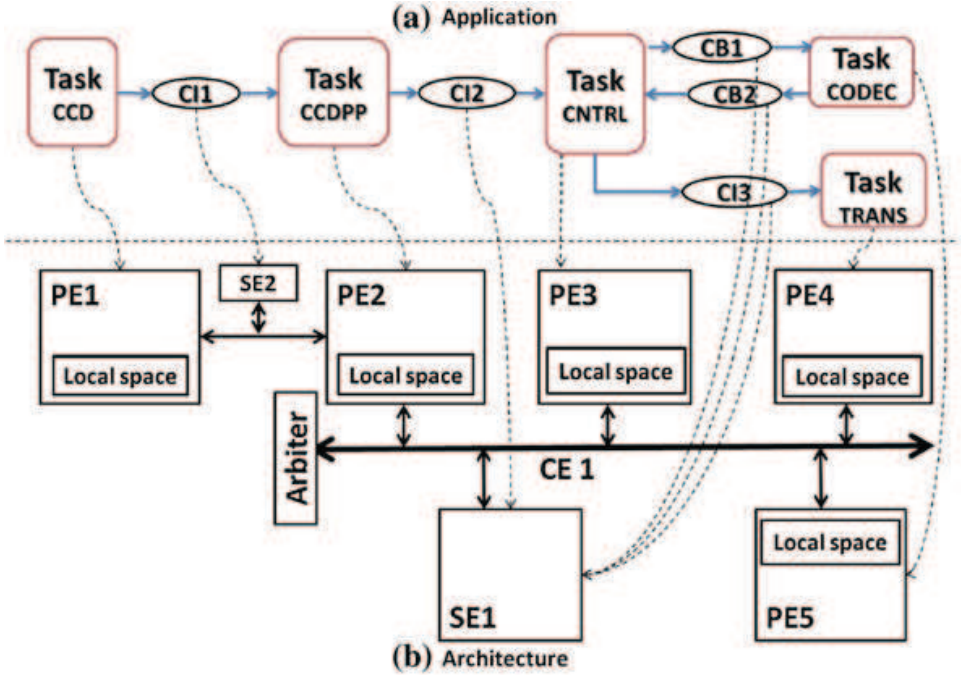

Fig. 2.10 Architecture and mapping of digital camera

ory SE1. We consider that a maximal capacity is associated to each memory space. We built the global models M^0, M^1, M^2, and M^3 of the platform corresponding to the different levels of refinement from Level-0 up-to Level-3 by following the generation scheme described in Sect. 2.4.2. In the current state of our research, we encode manually the models (LTSs) into Fiacre language [5].

We ran the use case with different architectural parameters summarized in Table 2.2: each PE has 1 core, each PE except PE2 has 1 interface, PE2 has 2 interfaces, and each local memory size equals to 2 UNITS. The size of shared memories

Table 2.2 Summary of architectural parameters with different sizes

	TML					PE1	PE2	PE3	PE4	PE5	SE2	SE1
	IMAGE1	IMAGE2	IMAGE3	BLOC1	BLOC2	MEMORY					SPACE	
Case 1	2×3	2×2	2×2	2	2	6	4	2	2	2	$size(CI1)$	$\sum_{c \in \mathcal{C}} size(c)$
Case 2	16×18	16×16	16×16	64	64	16×18	18	16×16	64	16×16	$size(CI1)$	$\sum_{c \in \mathcal{C}} size(c)$
Case 3	32×34	32×32	32×32	64	64	32×34	34	32×32	64	32×32	$size(CI1)$	$\sum_{c \in \mathcal{C}} size(c)$

are defined by the required size for all the channels, which are mapped onto it. So, for SE2, the required size is defined as $\sum_{c \in \mathcal{C}} size(c)$ such that $C = \{CB1, CB2, CI3\}$. We experimented the use case with different sets and different sizes of data, which are expressed by the number of UNITS.

2.5.1 Refinement Checking

At each refinement step, we built the corresponding models. By using the equivalence checker BISIMULATOR of the CADP toolbox [7], we compared the models at successive levels of abstraction. The refinement preorder relation, which we used, takes finite stuttering into account. It verifies the non-introduction of new traces. Moreover, for verifying the inclusion of complete and infinite traces, we also verify at each level the non-introduction of new blocking state (deadlock freedom) and the non-introduction of τ-cycles (livelock freedom).

The full state generation fails with the two last test cases. We chose the second solution for the hiding operation to further reduce and to generate the state space of the system. Then, we verified the trace inclusion between successive models, so $M^0 \sqsubseteq M^1 \sqsubseteq M^2 \sqsubseteq M^3$. Table 2.3 summarizes quantitative results obtained from the experiment with the first and the last test case: they show the system sizes (states/transitions, after minimization) as well as the time consumption for the refinement verification.

2.5.2 Properties Verification

We also verified several properties that express various facets of the system correctness. They are expressed using the Model Checking Language (MCL) logic [12], which is an extension of the alternation-free regular μ-calculus and supported by CADP toolset. The MCL formulae are logical formulae built over regular expressions using boolean operators, modalities operators (necessity operator denoted by "[]" and the possibility operator denoted by "<>") and maximal fixed point operator (denoted by "mu"). Notice that atomic propositions are the actions labels of the Level-0, which should be preserved under the refinement process.

- Deadlock freedom: Absence of states without successors.

  P0: **[true*] <true> true**

- Initially, no reading action on channel **CB1** can be reached before the corresponding writing:

  P1: **[true*.(not write_CB1)*.read_CB1] false**

Table 2.3 Computation times of refinement and verification analysis

	Test case 1				Test case 3			
	Level-0	Level-1	Level-2	Level-3	Level-0	Level-1	Level-2	Level-3
♯ States	336	123088	437782	173546709	2542	141302	516326	2369408
♯ Transitions	872	431622	1646712	472413097	7212	419758	1038037	6593033
Verif time P0	0.8 s	13 s	9 min	26 min	3 s	3 min	11 min	40 min
Verif time P1	0.5 s	8 s	5 min	16 min	2 s	2 min	5 min	25 min
Verif time P2	0.7 s	9 s	5 min	15 min	2 s	2 min	6 min	25 min
Verif time P3	0.9 s	21 s	13 min	41 min	5 s	4 min	16 min	66 min
Verif time P4	0.8 s	14 s	11 min	39 min	5 s	3 min	16 min	64 min
Refine time	n/a	8 s	3 min	13 min	n/a	3 min	25 min	75 min

- No more than two actions of writing on channel **CB1** is possible before the corresponding reading:

```
P2:[true*.write_CB1.(not read_CB1)*.write_CB1.
                        (not read_CB1)*.write_CB1] false
```

- A writing action on channel **CI1** will be eventually reached:

```
P3:mu X.[true*] (<true> true and [not write_CI1] X)
```

- After a writing action onto channel **CI1**, the corresponding reading is eventually reachable:

```
P4:[true*.write_CI1] mu X.(<true> true
                                and [not (read_CI1)] X)
```

The properties are verified at each level hence preserved from M^0 down-to M^3. Table 2.3 shows the time consumption for the properties verification at each level from the experiment (and for each test case). Furthermore, we compare the time required for the verification of these properties at each level of refinement with the time required by the refinement-based strategy. We observe that without using the strategy of checked refinement, the properties have to be verified at each level until Level-3. In the refinement-based strategy, once the validity of the properties has been established on M^0 and the refinement relation satisfied, the properties are guaranteed to be true in subsequent levels.

The Table 2.3 shows the benefits of the refinement-based strategy by comparing *verif time of any property at Level*-0 + *Refine time of any Level-i (for* $i > 0$*)* with *verif time of the property at Level-i*. When multiple properties have to be satisfied (which is generally the case), it is worth using our refinement strategies instead of a direct verification of properties on the Level-3 models. On small models (see left part of Table 2.3), the refinement verification is always much smaller than any single property verification time. For more complex models, the refinement time becomes similar to single property verification time, and this approach remains interesting when multiple behavioral properties have to be checked.

2.6 Conclusion

We presented a refinement-based methodology for design-space exploration of system on chip. 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 exemplified 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. We saw in the experimental section that with more complex systems, this application-dependent refinement verification becomes as costly as the verification of a single property (and remains interesting in case of the verification of multiple properties). A generic proof, based on the formal definitions of our transformations is under study and would simplify the overall verification process. Another perspective concerns the extension of the approach to task computation refinement. For instance, we shall add computation details as computation scheduling.

References

1. Abdi S, Gajski D (2006) Verification of system level model transformations. Int J Parallel Prog 34:29–59. doi:10.1007/s10766-005-0001-y
2. Abrial JR (1996) The B-book: assigning programs to meanings. Cambridge University Press, New York
3. Apvrille L, Muhammad W, Ameur-Boulifa R, Coudert S, Pacalet R (2006) A UML-based environment for system design space exploration. In: Proceedings of the 13th IEEE international conference on electronics, circuits and systems (ICECS) 2006, pp 1272–1275. doi:10.1109/ICECS.2006.379694
4. Arnold A (1994) Finite transition systems—semantics of communicating systems. Prentice Hall, New Jersey
5. Berthomieu B, Bodeveix J, Farail P, Filali M, Garavel H, Gaufillet P, Lang F, Vernadat F (2008) Fiacre: an intermediate language for model verification in the TOPCASED environment. In: Proceedings of the embedded real time software and systems (ERTS2) 2008, Toulouse, France
6. Colley JL (2010) Guarded atomic actions and refinement in a system-on-chip development flow: Bridging the specification gap with Event-B. PhD thesis, University of Southampton
7. Garavel H, Lang F, Mateescu R, Serwe W (2013) CADP 2011: A toolbox for the construction and analysis of distributed processes. Int J Softw Tools Technol Transfer (STTT) 15:89–107. doi:10.1007/s10009-012-0244-z
8. Kahn G (1974) The semantics of a simple language for parallel programming. In: Rosenfeld JL (ed) Information Processing '74: Proceedings of the IFIP Congress, North-Holland

9. Kempf T, Doerper M, Leupers R, Ascheid G, Meyr H, Kogel T, Vanthournout B (2005) A modular simulation framework for spatial and temporal task mapping onto multi-processor SOC platforms. In: Proceedings of DATE'05. Munich, Germany, pp 876–881
10. Lieverse P, van der Wolf P, Deprettere E (2001) A trace transformation technique for communication refinement. In: CODES'01: Proceedings of the ninth international symposium on Hardware/software codesign, ACM
11. Marculescu R, Ümit Y (2006) Computation and communication refinement for multiprocessor SoC design: A system-level perspective. ACM Trans Des Autom Electron Syst 11:564–592
12. Mateescu R, Thivolle D (2008) A model checking language for concurrent value-passing systems. In: Proceedings of the 15th international symposium on formal methods (FM), Springer, Berlin, pp 148–164. doi:10.1007/978-3-540-68237-0_12
13. Mokrani H (2014) Assistance au raffinement dans la conception de systèmes embarqués. PhD thesis, LTCI/Telecom-ParisTech
14. Mokrani H, Ameur-Boulifa R, Coudert S, Encrenaz E (2011) Approche pour l'intégration du raffinement formel dans le processus de conception des SoCs. Journal Européen des Systèmes automatisés, MSR'11 pp 221–236
15. Pimentel A, Erbas C, Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Comput 55(2):99–112
16. Pratt VR (1984) The pomset model of parallel processes: Unifying the temporal and the spatial. In: Proceedings of the seminar on concurrency
17. Vahid F, Givargis T (2002) Embedded system design—a unified hardware/software introduction. Wiley, New York
18. Zivkovoc V, Deprettere E, van der Wolf P, de Kock E (2002) Design space exploration of streaming multiprocessor architectures. In: Proceedings of SIPS'02, San Diego, CA

http://www.springer.com/978-3-319-06316-4

Languages, Design Methods, and Tools for Electronic System Design
Selected Contributions from FDL 2013
Louërat, M.-M.; Maehne, T. (Eds.)
2015, XXXIV, 305 p. 117 illus., 50 illus. in color.,
Hardcover
ISBN: 978-3-319-06316-4