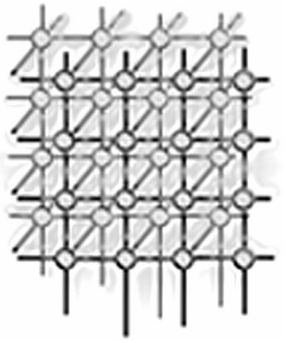
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



Formal System-level Design Space Exploration

Daniel Knorreck and Ludovic Apvrille and Renaud Pacalet*,†

System-on-Chip Laboratory (LabSoC)
Institut Telecom, Telecom ParisTech, LTCI CNRS
2229, routes des Crêtes, B.P. 193
F-06904 Sophia-Antipolis Cedex

SUMMARY

DIPLODOCUS is a UML profile intended for the modeling and the formal verification of real-time and embedded applications commonly executed on complex Systems-on-Chip. DIPLODOCUS implements the Y-Chart Approach, i.e., application tasks and architectural elements (e.g., CPUs, bus, memories) are first described independently and are related in a subsequent mapping stage in which tasks are mapped onto architectural elements. DIPLODOCUS endows both application models and mapping models with a formal semantics, thereby paving the way for formal proofs both before and after mapping. More concretely, application, architecture and mapping models can be edited in TTool - an open-source toolkit - using UML diagrams. Then, pre or post mapping UML models may be automatically transformed into a LOTOS-based representation. This specification is in turn amenable to model-checking techniques to evaluate properties of the system, e.g., safety, schedulability, and performance properties. A smart card system serves as case study to exemplify formal verification capabilities of DIPLODOCUS.

KEY WORDS: Design Space Exploration, Systems-on-Chip, UML, Formal Specification, Model Checking, TTool

*Correspondence to: System-on-Chip Laboratory (LabSoC)
Institut Telecom, Telecom ParisTech, LTCI CNRS
2229, routes des Crêtes, B.P. 193
F-06904 Sophia-Antipolis Cedex
†E-mail: {daniel.knorreck, ludovic.apvrille, renaud.pacalet}@telecom-paristech.fr

Received 10 December 2010
Revised 10 January 2011

1. Introduction

A System-on-Chip (SoC) is a set of functions distributed over hardware computation elements (CPUs, hardware accelerators) interconnected with complex communication elements (e.g., Network-on-Chip). The high complexity of applications executed on SoC - the smart card example provided in this paper is one prominent example of complex application - along with shortened time-to-market have pushed to their limits usual SoC designs methodologies. The analysis of systems at low abstraction levels yields a high degree of accuracy but comes with the downside of being demanding and slow. Indeed, traditional simulation techniques operating at register transfer level (RTL), instruction or transaction level are not appropriate for early design stages for two reasons: Only a very limited number of implementation alternatives can be examined due to the high modeling effort and extensive simulation runtime. Also, the lack of specification early in the design flow may prohibit the construction of detailed models even if the effort was acceptable. Thus, abstractions are the key to success and furthermore make the models amenable to formal methods by reducing the state space. This article elaborates on these formal techniques which are exhaustive by definition and thus provide the highest possible degree of confidence.

Design Space Exploration (DSE) is a major step in SoC design: it consists in selecting a software / hardware architecture complying to a set of functional and non-functional constraints (performance, power consumption, etc.). At Design Space Exploration stage, the complexity may already be non-manageable, and so, we suggest to perform that stage on very abstract models to pave the way for fast performance estimations as soon as possible in the SoC design flow, i.e., at system-level. DIPLODOCUS [†] is the environment we propose for addressing Design Space Exploration. As it relies on the Y-Chart approach, architecture and application are represented in an orthogonal fashion. DIPLODOCUS explicitly takes into account the hardware platform on which application tasks are executed. All system elements (application, architecture, mapping) can be efficiently represented in an abstract way using non-deterministic operators, complexity operators, and abstract hardware components. While modeling [1] and simulation [2] capabilities of DIPLODOCUS, as well as the toolkit supporting DIPLODOCUS [3] - TTool [4] - were already described in previous publications, the semantic support, one of the main strengths of DIPLODOCUS, has not been addressed so far. More precisely, for formal verification purpose, we have provided a formal semantics to all DIPLODOCUS diagrams (applications, hardware architectures, mapping of applications onto hardware architectures). Formal analysis is based on a process algebra named LOTOS [5] and on UPPAAL [6]. The paper is more particularly focused on abstractions applied to tasks and hardware platforms, on how formal analysis is leveraged by these abstractions, and on the associated toolkit (TTool) in which all underlying formal techniques are totally masked to the designer (press-button approach).

The paper is organized as follows. Section 2 reviews related contributions. Section 3 recalls the DIPLODOCUS environment. Section 4 focuses on the formal semantics in DIPLODOCUS,

[†]DIPLODOCUS stands for design space exploration based on formal description techniques, UML and SystemC

and more precisely on the abstractions offered by DIPLODOCUS to allow formal analysis with limited combinatory explosion. Section 5 presents the implemented support toolkit. Section 6 illustrates our approach with a smart card system. Finally, section 7 concludes the article.

2. Related Work

Design Space Exploration (DSE) of Systems-on-Chip is the process of analyzing various functionally equivalent implementation alternatives to select an optimal solution [7]. The most suitable design is commonly chosen based on metrics such as *functionality*, *performance*, *cost*, *power*, *reliability*, and *flexibility*. At system-level, DSE is challenging because the system design space is extremely large and so usual simulation-based analysis techniques fail to efficiently observe the above mentioned metrics. Contributions on DSE environments such as [8–15] generally rely on a high-level language to describe application functions and architectures. For example, [13–15] rely on UML or MARTE diagrams. Functions are sometimes described with only their cost [16]. Unfortunately, in many of these environments, architecture and application concerns are not independent [10], making the study of alternative solutions more complex. Second, they propose a way to map functions onto hardware execution nodes. Lastly, they introduce simulation techniques to simulate the system built from the mapping of functions over hardware nodes. The level of abstraction is commonly rather low to the detriment of simulation performance. For example, [8] relies on an Instruction Set Simulator which executes the real code of the application. In [12], hardware components are considered at micro-architecture level, hence leading to long simulation times. Otherwise, other environments offer formal exploration, but generally limited to sub-elements of the platform [17], or suffering the limitations described just below.

SymTA/S [18] [19] and Real Time Calculus (RTC) rely on formal methods such as the real-time scheduling theory and deterministic queuing systems to determine characteristics of distributed systems. In SymTA/S the behavior of the environment is modeled by means of standard event arrival patterns including periodic and sporadic events with jitters or bursts. RTC imposes less restrictions by allowing deterministic event streams to be modeled with the aid of arrival curves denoting lower and upper bounds for event occurrences. Event streams are propagated among resources of distributed systems in a way that each resource may be analyzed separately with classical algorithms. However, the applicability of scheduling theories requires the task model to be simplistic and thus it merely reflects best case and worst case execution times. Control flow within tasks cannot be considered at all. For that reason it may be tedious if not impossible to model tasks exhibiting a data dependent or irregular behavior.

The work of Hendriks and Verhoef [20] relies on timed automata to analyze timeliness properties of embedded systems. The UPPAAL model checker is used to evaluate the automata which must be created manually. There is no automated translation routine from a high level language (UML,...) and thus the creation of the automata turns out to be quite error prone, and not reusable.

Viehl et al. [21] provide means for formal and simulation based evaluation of UML/SysML models for performance analysis of SoC. UML Sequence diagrams constitute the starting point for the functional description. They are subsequently transformed into so-called communication

dependency graphs (CDGs) which thus capture the control flow, synchronization dependencies and timing information. CDGs are in turn amenable to static analysis in order to determine key performance parameters like best case response times, worst case response times and I/O data rates. A drawback of this approach is that data flow independence has to be kept, thus preventing case distinctions and loops with variable bounds to be part of the application model.
Marculescu et al. [22] present a framework for computation and communication refinement for multiprocessor Soc Design. Stochastic automata networks represent the application behavior and the authors claim that this formalism allows for fast analytical performance evaluations. When it comes to mapping an application on an architecture, transitions and states have to be added to the application model. Hence, application and architecture matters and not strictly handled in an orthogonal fashion. Due to a lack of data abstraction, the modeling of memory elements can quickly lead to state space explosion problem.
The PUMA [23] framework is a unified approach to software modeling. It provides an interface between high level input models (such as UML diagrams) and performance oriented models. For that purpose, input models are first translated into an intermediate format called CSM so as to filter out irrelevant information for performance evaluations. In a second step, CSM can be converted to Petri Nets, Markov models, etc., and the resulting performance figures and design advice is fed back to the initial model. However, this framework concentrates on the modeling of software and thus does not yield a mapping where functionality is associated to software or hardware elements.
DIPLODOCUS [1] offers a very clear separation between applications and architectures, and includes a high level of abstraction. Indeed, DIPLODOCUS is focused on control rather than on data, i.e., only abstract *samples* of data can be manipulated in the profile. Samples are untyped and carry no value: only their size is a relevant attribute. This high level of abstraction greatly reduces simulation times and makes formal proof techniques usable. In this paper, we apply these formal proof techniques to safety, performance and schedulability analysis purpose using the LOTOS process algebra. While LOTOS has already been successfully experimented for property proofs on hardware [24], we propose its use to more generic platforms (SoCs).

3. The DIPLODOCUS UML Profile

A UML profile customizes the UML language [25] for a given domain of systems. It may extend the UML meta-model, according to *semantic variation points*, and may provide a methodology. The DIPLODOCUS UML profile targets the modeling and Design Space Exploration of Systems-on-Chip at a high level of abstraction [1]. The DIPLODOCUS methodology, depicted in Figure 1, complies with the Y-Chart approach and comprises three main steps, that are further reviewed in next subsections:

1. **Applications** are first described as a network of abstract communicating tasks using a UML class diagram. The latter represents the static view of the application. Each task behavior is expressed in terms of one UML activity diagram.
2. Targeted **architectures** are modeled independently from applications as a set of interconnected generic hardware nodes modeled in UML: execution nodes (e.g. CPUs,

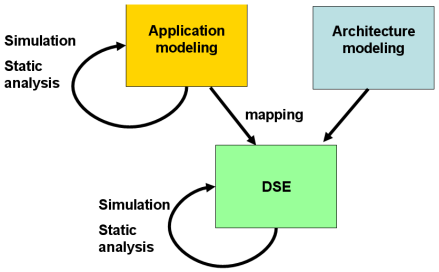


Figure 1. Methodology for Design Space Exploration

hardware accelerators), communication nodes (e.g., buses, bridges), and storage nodes (e.g., memories). These nodes are parametrized to offer specific behavior.

3. A **mapping** phase defines how application tasks are bound to execution nodes and also how abstract communications between tasks are assigned to communication and storage nodes.

3.1. Application modeling

At first, the application is modeled using UML class and activity diagrams. Tasks are modeled as classes interconnected with *channels*, *events*, or *requests* to communicate. Data abstraction is a key point: *channels* do not convey values, but only a number of samples (data abstraction). Also, *events* are used for synchronization purpose, and *requests* are used to spawn the execution of tasks.

As stated before, functions are modeled as a set of abstract tasks described with UML class diagrams. Task behavior is modeled using UML activity diagrams which are built upon the following operators: control flow and variable manipulation operators (loops, tests, assignments, etc.), communication operators (reading/writing abstract data samples in channels, sending/receiving events and requests), and computational cost operators (refer to 4.2.1 and 4.4.1 for detailed semantics and abstractions respectively). Operators are abstraction-focused, i.e., they have been defined for encouraging designers of performing the following data and functional abstractions.

- **Data abstraction**: Only the amount of data exchanged between functional entities is modeled. Data dependent decisions are abstracted and expressed in terms of non-deterministic operators, such as non-deterministic choices, complexity and time operators.
- **Functional abstraction**: Algorithms are described using abstract cost operators. The complexity of computations is thus taken into account without actually having to perform computations.

Three communication and synchronization primitives have been defined:

- **Channels** are characterized by a point-to-point unidirectional communication between two tasks. Channel types are:
  - Blocking Read/Blocking Write (BR-BW)
  - Blocking Read/Non Blocking Write (BR-NBW)
  - Non Blocking Read/Non Blocking Write (NBR-NBW)
- **Events** are characterized by a point-to-point unidirectional asynchronous communication between two tasks. Events are stored in an intermediate FIFO between the sender and the receiver. This FIFO may be finite or infinite. In case of an infinite FIFO, incoming events are never lost. Indeed, when adding an event to a finite FIFO, the incoming event may be discarded or the oldest event may be dropped if the FIFO is full. Thus, a single element FIFO may be used to model hardware interrupts. In tasks, events can be sent (*notify*), received (*wait*) and tested for their presence (*notified*).
- **Requests** are characterized by a multipoint-to-point unidirectional asynchronous communication between tasks. A unique infinite FIFO between senders and the receiver is used to store requests. Consequently, a request cannot be lost.

3.2. Architecture modeling

A candidate architecture is modeled in terms of interconnected hardware components (or *nodes*) using UML components placed in UML deployment diagrams. The following nodes types are available in DIPLODOCUS:

- **Computation nodes**. Typically, an abstract **CPU** model merges both the functionality of the hardware component and its operating system. The behavior of a CPU model can be customized with parameters such as: data size, pipeline size, cache miss ratio and scheduling algorithm. For the time being, **DMAs** and **hardware accelerators** are represented by adequately parametrized CPU nodes. See 4.4.2 for comments on abstractions made on CPU nodes.
- **Communication nodes**. A communication node is either a **bus** or a **bridge**. The bus model has the following parameters: data size, latency and arbitration policy. A **Link** connects a hardware node - except for buses - to a bus. A link may be annotated with a priority which may be considered by the bus arbitration policy.
- **Storage nodes**. **Memories** are characterized by the following parameters: latency and data size.

3.3. Mapping

A mapping imposes additional constraints on the application model by associating the latter to shared hardware resources. The objective of the mapping stage is to determine whether an architecture is able to accommodate the load defined by the application whilst complying to constraints. Application tasks and channels are therefore distributed over hardware nodes. A UML deployment diagram is used for that purpose. A given task must be mapped onto exactly one execution node. Also, channels are mapped onto paths built upon links, communications

nodes, and storage nodes.

3.4. Simulation and formal verification

As previously stated, the DIPLODOCUS design flow is supposed to be carried out at a very early stage. Hence, the main DIPLODOCUS objective is to help designers to spot a suitable hardware architecture even if algorithmic details have not yet been stipulated thoroughly. To achieve this, DIPLODOCUS relies (i) on fast simulation and formal proof techniques, both at application and mapping level, and (ii) on application models clearly separated from architecture models. Due to the high abstraction level of both application and architecture models, simulation speed can be significantly increased with regards to simulations usually performed at lower abstraction level [2], and formal proofs can be achieved: this article focuses on formal analysis techniques that may be applied before and after mapping. While simulation and formal verification usually target functional properties at application level, performance properties are rather investigated after mapping, e.g., resource sharing, that is, the scheduling on CPUs (can the architecture execute tasks on time), bus load (can a bus handled all data transfers), and also properties related to power consumption and silicon area.

4. Formal semantics and abstractions

4.1. Formal support: LOTOS and UPPAAL

The semantics of DIPLODOCUS models is defined both in LOTOS and UPPAAL, but the latter is out of scope of this paper.
LOTOS [5] is an ISO-standardized Formal Description Technique for distributed system specification and design. A LOTOS specification, being itself a process, is structured into processes. A LOTOS process is a black box that communicates with its environment through gates using a multiway rendezvous offer. Values can be exchanged at synchronization time. That exchange can be mono- or bi-directional. LOTOS specifications may be formally verified with, for instance, the CADP toolkit [26] . CADP implements model-checking and reachability graph minimization techniques

4.2. Semantics at application level

4.2.1. Tasks operators

As described in previous section, an application is composed of a set of communicating tasks. Operators used to describe task behavior are of four types:

- **Communication operators**: **read** from a channel, **write** a sample to a channel, **notify** an event, **wait** for an event, know whether an event has been sent (**notified**), **request** a task.
- **Control operators**: usual control operators, such as **variable modifications**, **loops**, **tests**, random number.
- **Complexity operators**: operators to model a number of operations on integers (**EXECI**), floats (**EXECF**) or custom (**EXECC**).
- **Temporal operators**: operators to model deterministic and non-deterministic physical **delay**.

This set of operators makes it possible to describe the communication behavior of applications and algorithms whilst encouraging the modeler to abstract data, thanks to channels that merely account for the amount of transmitted data, and thanks to non deterministic versions of some operators (e.g., choice).
The LOTOS semantics of all task operators is further described in Table I, column "LOTOS Semantics before mapping".

4.2.2. Communications between tasks

While **channels are used to model data stream between tasks** - i.e., channels carry unvalued samples -, events and requests represent synchronization schemes.
The semantics of these channels (as explained in 3.1) is quite obvious to describe in LOTOS: since channels convey no value, but only a number of samples, BR-BW and BRNBW channels can easily be translated into a simple process (see Figure 2) sharing a natural value (which represents the number of elements in the FIFO) between two processes using two gates: one gate to add a sample (*wr__ch*), another one to remove a sample (*rd__ch*). The last channel type (NBR-NBW) is also translated into a similar LOTOS process apart from the fact that no counter is necessary - since its is always possible to read and write -, and so no guards (*[]* operator) are used before the actions on gates *wr__ch* and *rd__ch*.

```
process ChannelBRBW__ch[rd__ch, wr__ch](samples:nat) : exit := (
[samples < 8] -> (wr__ch; ChannelBRBW__ch[rd__ch, wr__ch](samples + 1))
[]
[samples > 0] -> (rd__ch; ChannelBRBW__ch[rd__ch, wr__ch](samples - 1)) )
```

Figure 2. Application-level LOTOS semantics for a BR-BW channel, containing at most 8 samples

Events are meant to model synchronization between tasks. They can carry up to three parameters, and support three semantics: *infinite FIFO*, *finite blocking FIFO*, and *non-blocking FIFO*. The two first semantics have been selected because they reflect common synchronization schemes of embedded systems. The last one (Non-blocking finite FIFO) is particularly useful to model signal exchanges between tasks: indeed, software and hardware

Type	Task operators	LOTOS Semantics before mapping	LOTOS Semantics after mapping
Channel	Write n samples to a channel	n Write operations in FIFO, i.e., n times action on gate $wr\_\_ch$, see Figure 2	n cycles, and a request on a bus.
	Read n samples from a channel	n read operations from FIFO, i.e., n times action on gate $rd\_\_ch$, see Figure 2	n cycles and a request on a bus.
Event	Notify an event	Adds an event to the corresponding FIFO, i.e., performs an action on gate $notify\_\_evt$, see Figure 3	Same as before mapping.
	Wait for an event	Tries to get an event from a FIFO, i.e., performs an action on gate $wait\_\_evt$, see Figure 3	Same as before mapping.
	Notified	Returns the number of event in a FIFO using action $notified\_\_evt$, see Figure 3	Same as before mapping.
Request	Send a request (operator is called "request")	FIFO management is similar to the one used for events	Same as before mapping.
Control	loop, variable modifications, tests	Direct translation in LOTOS with corresponding LOTOS operators	Direct translation. Operators are executed in 0-cycle.
Complexity	EXECx n,m i.e., between n and m integer instructions	No semantics before mapping, i.e., this operator is ignored	The task executes between $n * perf$ and $m * perf$ cycles with perf being a constant value depending on the hardware performance on which the task is mapped.
Temporal	Delay $d_{min}d_{max}$ unit	No semantics before mapping, i.e., this operator is ignored	The task is blocked for Between n and m cycles with $n = d_{min} * frequency$ and $m = d_{max} * frequency$.

Table I. Task operators

signals usually erase the previous one (e.g., Programmable Interrupt Controller, or UNIX signals). A separate LOTOS process accounts for each of the three semantics using the *Queue_nat* algebraic type. Figure 3 illustrates a non-blocking finite FIFO (the most complex case) for an event carrying only one natural parameter. Five cases have been taken into account:

1. The FIFO is not empty, and so, a *wait* action can be performed on the FIFO.
2. The FIFO is not full, and so, an event can be added to the FIFO (*notify*).
3. The FIFO is full, and so, an event can be added to the FIFO (*notify*) after the oldest one has been removed.
4. The FIFO is not empty, the *notified* action returns the value 1.
5. The FIFO is empty, the *notified* action returns the value 0.

Unlike channels and events which are one-to-one communications, **requests are many-to-one communications**. They rely on *n-to-one* infinite FIFO. The translation of requests is similar to the one of FIFO for events, apart from the fact that notification gates are instantiated n times, e.g., $notify\_i$ with $i \in 1 \ldots n$.

```
process Event__evt[notify__evt, wait__evt, notified__evt]
(fifo_1:Queue_nat, fifo_val_1:nat, nb:nat, maxs:nat) : exit :=
  [not (Empty (fifo_1))] -> wait__evt !First(fifo_1);p_0_Event__evt[notify__evt,
    wait__evt, notified__evt](Dequeue(fifo_1), fifo_val_1, nb-1, maxs)
  [] [nb<maxs] -> notify__evt ?fifo_val_1:nat;p_0_Event__evt[notify__evt,
    wait__evt, notified__evt](Enqueue(fifo_val_1, fifo_1), fifo_val_1, nb+1,
    maxs)
  [] [nb == maxs] -> notify__evt ?fifo_val_1:nat;p_0_Event__evt[notify__evt,
    wait__evt, notified__evt](Enqueue(fifo_val_1, Dequeue(fifo_1)),
    fifo_val_1, nb, maxs)
  [] [not (Empty (fifo_1))] -> notified__evt!1;p_0_Event__evt[notify__evt,
    wait__evt, notified__evt](fifo_1, fifo_val_1, nb, maxs)
  [] [Empty (fifo_1)] -> notified__evt!0;p_0_Event__evt[notify__evt, wait__evt,
    notified__evt](fifo_1, fifo_val_1, nb, maxs)
endproc
```

Figure 3. Application-level LOTOS semantics for a Non-blocking Finite FIFO

4.3. Semantics at mapping level

A mapping involves an application (i.e., a set of tasks and communications between those tasks), an architecture (i.e., a set of hardware nodes), a distribution of tasks onto hardware nodes (e.g., map the task *task1* onto the CPU *cpu1*), and a mapping of communication channels onto buses / memories. We have therefore defined a transformation function $tf()$ that takes as argument all above mentioned elements and generates a LOTOS specification (see Figure 4).

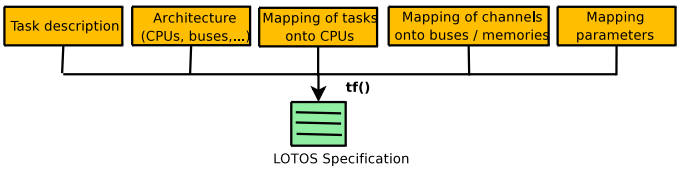


Figure 4. General approach

4.3.1. Mapping issues

The mapping phase sheds light on the question whether a system made of an application executed on a given architecture satisfies a set of constraints. More precisely, a mapping shall resolve contentions on shared resources (typically, a CPU, a bus, etc.) and therefore answer whether the computational and communication power offered by the architecture can execute the desired application, i.e., respect deadlines, etc. The LOTOS semantics is first defined with those issues in mind. As a consequence, the LOTOS specification of a mapping should take into account:

- The access control to shared resources, e.g., for tasks: access to CPUs, and for communication: access to buses. To that end, we explicitly account for scheduling policies of operating systems as well as for arbitration policies of buses.
- The time taken by tasks to execute operators, and the time taken by communications, e.g., bus and memory latencies.

4.3.2. The Mapping-to-LOTOS transformation

All task operators and hardware nodes parameters are taken into account by the Mapping-to-LOTOS transformation ($tf()$). However, the transformation does not yet incorporate the latest proposals on resource sharing in DIPLODOCUS, e.g., we do not yet support hierarchical scheduling and virtual nodes [27] but no technical limitation has been identified which could hamper their integration.
Basically, the LOTOS specification is built upon four functional blocks:

- The **Scheduling manager** schedules tasks on each CPU. $tf()$ transforms each task into a state machine modeled in LOTOS: preemption can occur when a task is blocked in a state, but never when a task performs a transition from one state to another.
- The **Communication manager** handles channel-based communication between tasks running on the same CPU, or on different CPUs. Events and requests are assumed not to consume communication resources. Indeed, the amount of data represented by those two synchronization features are assumed to be negligible with regards to channel-based communications. Similar assumptions were made for the simulation semantics [2] (which is less abstract and more tailored to simulation runtime issues).

- The **Task execution manager** handles operators to execute in each task, that is transitions between various task states.
- The **Clock manager** handles clock cycles on hardware nodes, i.e., it activates necessary hardware nodes when a new cycle begins.

The main process of the LOTOS specification works as follows:

1. At first, an initialization phase is used to settle various data structures, for each CPU (e.g., all tasks of a CPU are put in "ready" state), and for the communication manager: data structures related to channels, queues related to events, and so on.
2. A main loop on clock cycles is started: The system waits for the next tick (*tick* bas been defined as a LOTOS action). Then, each CPU plus its operating system are considered one after another. Basically, a CPU is meant to interpret DIPLODOCUS application-level operators of the task selected by the scheduler. More precisely:
   (a) Depending on its clock rate, the CPU is activated or not by the Clock manager.
   (b) If it is activated, then a first test is performed to see whether one task is in *running* state, or not.
   (c) If one task is in *running* state, then, the running state is activated from its former state. The task executes until either (i) it blocks (for example, it tries to receive one given event, and that event is not available): in that case, the scheduler is called, or (ii) it can perform an instruction consuming cycles (e.g., writing a sample to a non-full channel).
   (d) When the scheduler is called, it first checks whether at least one task is runnable. If no task is runnable, the CPU goes *idle*. Otherwise, a scheduling algorithm - implemented in LOTOS - is called to select another task. Then, the state machine of that task may be called, and so on.
3. Once all CPUs have been selected, a communication manager resolves all inter-CPUs communication, i.e., all communications set-up by tasks in previous cycle (i.e., all *read*, *write*, *notify events*, etc.) are really performed only when all CPUs have terminated that cycle. This ensures (i) that a sample written on a CPU during a cycle may not be read by another CPU in the same cycle, and (ii) that the order of CPU evaluation has no impact on results.

The $tf()$ function may also generate debug information in the form of LOTOS actions performed at well-chosen points: actions to show scheduler data structures (e.g., list of runnable or blocked tasks), actions to monitor tasks states, actions to monitor the communication manager, etc.
Finally, $tf()$ has been defined with combinatory explosion in mind. Hence, $tf()$ tries to precompute possible synchronization between LOTOS processes: if possible, these synchronization are removed, and resulting processes put in sequence. Unfortunately, combinatory explosion may also be due to (i) non-deterministic elements: for example random, choice and temporal operators of tasks; (ii) Non-determinism in scheduling models: for example, in the round-robin scheduling policy, the possible indexes of tasks, in the tasks list. Abstraction is a key factor to reduce combinatory explosion. The main idea behind abstractions

is to remove all software and hardware-related concerns that have no or little impact on evaluated properties (e.g., load on CPUs and buses). The next subsection is dedicated to abstractions.

4.4. Abstractions

4.4.1. Task abstraction (see Table I, column "LOTOS Semantics after mapping")

- **Communication operators**. These operations are given a cost (in clock cycles), and are executed by the execution manager along with the communication manager, to make request on related buses. The cost in cycle depends of the hardware platform. For example, writing an 8-byte sample on a 32-bit processor takes two cycles. Also, the communication manager is involved for storing output samples, and for providing data to input operations. Note that these operations may be blocking, and so, the scheduling manager may also be involved.
- **Cost operators** are defined by the number of cycles depending on the hardware platform.
- **Other operators**: choice, loop, variable manipulation, etc. These operations are executed by the task execution manager. They take no cycle since there are used for modeling the control part of applications only, i.e., the execution cost in DIPLODOCUS is modeled only with *EXECx* operations, and definitely not with control operators.
- **Temporal operators**: They are defined by a number of time units.

4.4.2. CPU abstractions

- **Parameters of CPU**: Data size (used for communication in channels), size of default integer and floating point data (used for EXEC operations), cost for each EXECx instructions, pipeline size (used for calculating the penalty induced by miss branching), miss branching rate, data cache-miss ratio and penalty, time to enter/leave the idle mode, clock ratio.
- **The Operating System** is taken into account with scheduling algorithms (e.g., Preemptive priority-based, round-robin), switching time, synchronization management (events, requests) and communication delay (buffering for handling channels).

4.4.3. Communication abstractions (buses, memories)

Buses are meant to carry data samples with an arbitration policy between requests. The time a given transfer takes depends on the width of the bus. Bus arbitration is done on each cycle. Memory delays are modeled throughout bus latencies and cache-miss rates at CPU level, as proposed for the simulation semantics [2].

4.5. Formal verification

LOTOS specifications may be derived either from an application modeling, or from a mapping of applications onto a given architecture (Figure 1).
At application level, tasks have a maximum concurrency between themselves, concurrency which is reduced when the mapping phase occurs: buses and CPUs are shared resources. For example, two tasks mapped on the same CPU do not execute in parallel any more. An interesting property would be that formal traces obtained after mapping are a subset of formal traces obtained before mapping, that is, a mapping only constrains possible application-level traces, without violating application-level safety properties (e.g., absence of deadlock). One of our ongoing work is to prove that scheduling and arbitration policies we have defined for CPUs and buses, respectively, preserve safety properties proved at application level, that is a mapping is always *correct-by-construction* with regards to safety properties.

5. Toolkit

5.1. General overview

TTool [4] is an open-source toolkit initially developed for the TURTLE UML profile [28]. It now supports several other UML profiles such as the *CTTool* profile [29], the *DIPLODOCUS* profile [1] and the AVATAR profile [**?**] [30]. TTool includes diagramming facilities, code generators (LOTOS, etc.) and graph analysis tools.
TTool includes a Graphical User Interface for drawing DIPLODOCUS UML diagrams. From those diagrams, simulation or formal analysis (see Figure 5) may be performed. Underlying simulation and validation languages (e.g., LOTOS) are totally hidden to DIPLODOCUS users. From LOTOS specification, TTool relies on CADP [26] to generate a reachability graph than can be analyzed directly in TTool (in particular, to detect deadlock situations), to minimize it, and to compare it with other graphs (bisimulations). UPPAAL offers simulation and model-checking capabilities. Furthermore, TTool has very fast simulation capabilities [2]. Moreover, during simulations, UML models can be accordingly animated [31].

5.2. Property analysis with TTool and CADP

1. At first, an application is modeled (e.g., a Smart Card functionalities: its main application, its TCP protocol stack, etc.). From that modeling a reachability graph is generated (let us call it rga), and model-checking techniques are used to prove a set P of properties on the application itself.
2. A hardware architecture is described in terms of CPUs, buses, etc..
3. From the mapping (tasks onto CPUs, etc.), a LOTOS specification is generated, and from that specification, CADP is used to obtain a reachability graph rg. The following verification features are supported:

    - Minimizing the reachability graph to *tick* actions. From that minimization, the longest path of ticks is calculated, therefore resulting in a performance information on the application (e.g., *Worst Case Execution Time*).

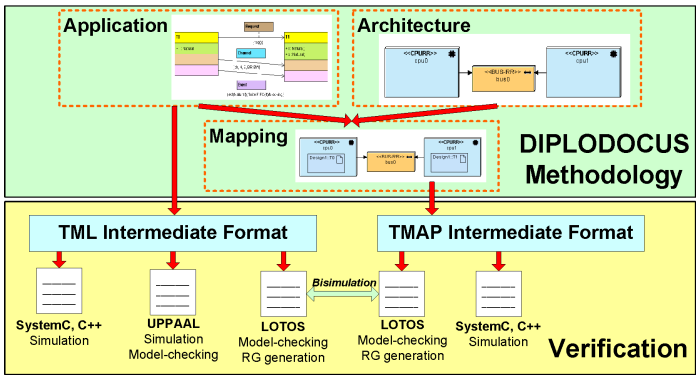


Figure 5. TTool for DIPLODOCUS: code generation capabilities

- Minimizing the reachability graph to *tick* and *transferOnBusX*. From that minimization, loads on buses can be deduced. Similar techniques can be used to compute CPU loads.
- Comparing rg and the reachability graph generated at application level, i.e., to rga. To do so, a toolkit integrated in TTool first modifies rg so as to make rg action names compatible with the one of rga, then CADP minimizes the resulting graphs: if it is proved that $rg \subset rga$, then safety properties proved at application level are preserved.
- Of course, all usual model-checking techniques can be directly applied to rga and rgb (e.g., using CADP).

6. Case Study

6.1. TCP/IP Protocol Implementation of a Smart Card

A smart card has the size of a credit card and is equipped with a microchip that securely stores data mainly used for identification purposes. The data may be periodically refreshed in order to maintain or enhance the functionality of the card. Smart cards are commonly used for telephone calling, electronic cash payments, establishing identity when logging on to some online account or when demanding public health services, paying small amounts of money (bus, parking, subway fees, etc.). Smart Cards comprise several hardware components like a microprocessor and different kinds of (non-)volatile memories (ROM, EEPROM, RAM Flash). Most smart card systems adhere to the standard ISO-7816 standard [32] which includes multiple parts defining for example physical characteristics, dimensions, involved protocols and

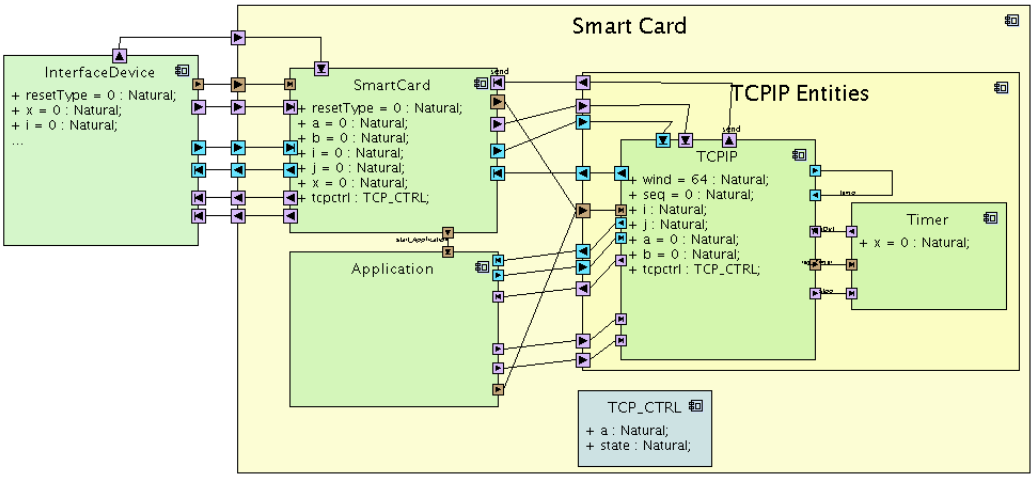


Figure 6. Component based Diagram of the Smart Card Application

other system properties. For the creation of our model, we mainly relied on the third part of the standard dealing with electronic signals and transmission protocols (ISO 7816-3).

6.2. Application modeling

The communication application has been decomposed into four DIPLODOCUS tasks corresponding to the main functional blocks. A task called *InterfaceDevice* represents the terminal the smart card communicates with, for instance the card reader at a cash desk. Another task (*SmartCard*) models the transmission protocol defined in ISO 7816-3. The *Application* task models a basic exemplary application which merely makes use of the basic TCP services: establishing a session, sending some application data and finally tearing down the connection. The *TCP* task accounts for the different phases of the TCP protocol like connection establishment, data transfer, connection termination. Last but not least, a *Timer* task may trigger time outs in the main *TCP* task.

6.3. Architecture and mapping

To demonstrate the applicability of our methodology, two candidate architectures are experimented with. A first basic mapping consists of one single CPU onto which all tasks are mapped. A second option is to map the *SmartCard*, *TCP* and *Application* tasks on one CPU named *MainCPU*, and to provide a dedicated Hardware Accelerator to the *Timer* and *InterfaceDevice* task respectively (see Figure 7). The three CPUs are connected via an on chip bus. For space reasons, the mapping of channels, events and requests is not shown on the figure.

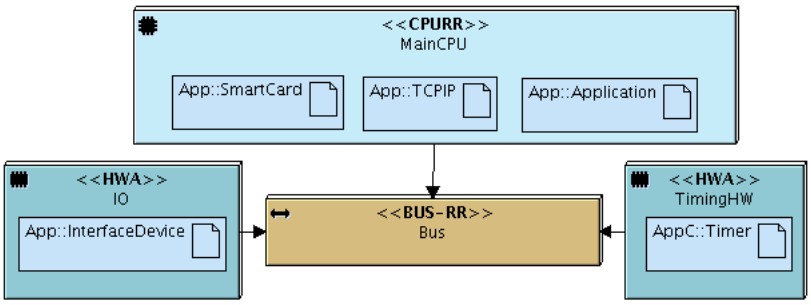


Figure 7. The Second Mapping for the Smart Card Application

Mapping	Min. Cycles	Max. Cycles
1	2474	3500
2	198	456

Shortest Paths | Longest Paths | General info. | Statistics | Deadlocks

Transition	Nb	
allCPUsTerminated<198>	1	(72, 87)
allCPUsTerminated<202>	1	(128, 87)
allCPUsTerminated<203>	2	(307, 87), (3
allCPUsTerminated<215>	5	(782, 87), (7
allCPUsTerminated<217>	2	(919, 87), (9
allCPUsTerminated<218>	1	(81, 87)
...		
allCPUsTerminated<425>	1	(1406, 87)
allCPUsTerminated<437>	4	(1490, 87), (
allCPUsTerminated<443>	1	(1519, 87)
allCPUsTerminated<445>	1	(1531, 87)
allCPUsTerminated<456>	1	(1540, 87)

Table II. Verification Results: Performance and Reachability Graph Statistics

In this second mapping, up to three tasks may execute concurrently and thus application level parallelism can be better exploited. Due to data and synchronization dependencies, further increasing the number of processing elements would not yield considerable performance improvements.

6.4. Property analysis

At first, performance measurements are carried out by transforming the UML model into its LOTOS equivalent. Subsequently, we rely on the CADP model checker to construct a reachability graph comprising all relevant system transitions. From TTool, the user is able to define state transitions of interest and to minimize the reachability graph accordingly. Depending on the respective verification objective, the user may select synchronization or

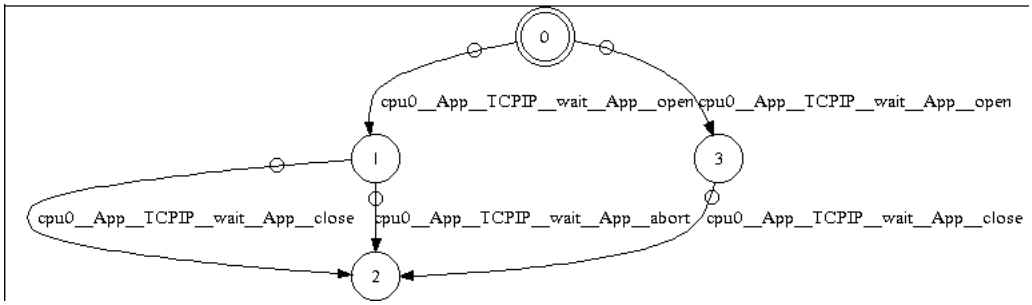


Figure 8. Minimized Reachability graph according to property

execution related transitions or add special transitions every x clock cycles. By counting the latter for every possible system execution, upper and lower bounds on the execution time can be obtained. An analysis of the first mapping revealed a minimum of 2474 cycles and a maximum of 3500 cycles, for the second mapping we obtained a minimum of 198 cycles and a maximum of 456 cycles (cf. Table II). TTool provides means for analyzing the reachability graph generated by CADP so that these results can be easily deduced (see statistics depicted in Table II). In addition to performance measurements, we are interested in proving the property that every established connection is correctly relinquished (hence either closed or aborted). To that end, we minimize the reachability graph to transitions corresponding to *open*, *close* and *abort* events generated by the TCP task. That way, we do not even have to express the property in temporal logics. The proof can be simply conducted by examining the reachability graph depicted in Figure 8.

7. Conclusions and future work

The paper presents an environment - named DIPLODOCUS - for formal functional and performance analysis of complex embedded and distributed systems. A system is described with communicating tasks, hardware architectures and a mapping of tasks and channels onto hardware architectures. A formal semantics is provided to tasks, communication between tasks and hardware architectures, making it possible to perform formal analysis before and after mapping. Moreover, DIPLODOCUS has been implemented in a UML-based and open-source toolkit named TTool. Formal analysis can be performed with absolutely no expertise in formal techniques. The DIPLODOCUS environment has been experimented within the scope of several case studies, including an MPEG2 application, a Smart Card Application and industrial case studies [27]. As stated in section 4, abstractions are a key factor of our models in order to alleviate combinatory explosion (and to greatly increase simulation speed [2]): Data and control flow abstraction at application level (see 3.1) and generic hardware components at architecture level (see 3.2).

Trading off accuracy against model complexity of hardware components will remain subject to our research. For example, instruction cache-misses and data cache-misses have been accounted for by static probabilities so far. Indeed, as algorithmic details are represented by symbolic instructions, the real code of the application is not available thus making state of the art cache models unsuited. Furthermore, the accuracy of bus and memory models shall be validated against a real embedded system. A fair comparison with a real implementation shall therefore reveal whether a set of parameters can be found to limit the inaccuracy to a reasonable percentage. To simplify the modeling of systems making extensive use of DMA engines, a specific UML stereotype could be introduced. This way, the designer would not have to model DMA transfers explicitly using a dedicated execution unit.
While being already operational, our environment will be enhanced with three main features. First, we will define a refinement process from the application modeling step to the after-mapping step, in order to preserve properties proved at application level. Second, we intend to assess and adapt post-mapping hardware abstractions - e.g., the ones used for memories and buses - by confronting mapping results with real implementations. Third, effort will be dedicated to finding adequate trade-offs between the two extreme cases of formal verification and conventional simulation. Simulation could cover several alternative system executions by exploiting indeterminsism inherent to the application model. In case the state space cannot be explored exhaustively, simulation could be guided by heuristics based on non-functional (CPU usage, bus loads, power consumption,...) or functional properties (for instance expressed in TEPE language [30]).

Acknowledgment

The authors would like to thank Chafic Jaber, who kindly granted us the permission to experiment with his Smart Card Model.

REFERENCES

1. Apvrille L, *et al.*. A UML-based environment for system design space exploration. *13th IEEE International Conference on Electronics, Circuits and Systems (ICECS'2006)*, Nice, France, 2006.
2. Knorreck D, Apvrille L, Pacalet R. Fast simulation techniques for design space exploration. *Objects, Components, Models and Patterns*, *Lecture Notes in Business Information Processing*, vol. 33, Springer Berlin Heidelberg, 2009; 308–327, doi:10.1007/978-3-642-02571-6_18.
3. Apvrille L. TTool for DIPLODOCUS: An Environment for Design Space Exploration. *8th annual international conference on New Technologies of Distributed Systems (NOTERE'2008)*, Lyon, France, 2008.
4. LabSoc. The TURTLE Toolkit. *http://labsoc.comelec.enst.fr/turtle/ttool.html*.
5. ISO-LOTOS. A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. *Draft International Standard 8807, International Organization for Standardization - Information Processing Systems - Open Systems Interconnection*, Geneva, 1987.
6. Bengtsson J, Yi W. Timed automata: Semantics, algorithms and tools. *Lecture Notes on Concurrency and Petri Nets*, W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, 2004.
7. Muhammad W, *et al.*. Abstract application modeling for system design space exploration. *Euromicro Conference on Digital System Design (DSD'06)*, Dubrovnik, Croatia, 2006.
8. Balarin F, *et al.*. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*. 5 edn., KLUWER ACADEMIC PUBLISHERS, 2003.

9. Watanabe Y. Metropolis : An integrated environment for electronic system design. Cadence Berkeley labs, 2001.
10. Wolf PVD, *et al.*. A methodology for architecture exploration of heterogeneous signal processing systems. *1999 IEEE Workshop on Signal Processing Systems (SiPS99)*, 1999.
11. Chatelain A, *et al.*. High-level architectural co-simulation using Esterel and C. *Proc. of IEEE/ACM symposium on Hardware/software codesign*, 2001.
12. Assayad I, Yovine S. A framework for modelling and performance analysis of multiprocessor embedded systems: Models and benefits. *Proceedings of the 8th conference on Nouvelles Technologies de la Distribution (NOTERE'2007)*, Marrakech, Marocco, 2007.
13. Schattkowsky T, *et al.*. A model-based approach for executable specifications on recon figurable hardware. *Design, Automation and Test in Europe Conference and Exhibition, 2005. DATE'05*, 2005; 692–697.
14. Kukkala P, *et al.*. Performance Modeling and Reporting for the UML 2.0 Design of Embedded Systems. *Proc. of the 2005 International Symposium on System-on-Chip*, 2005; 50–53.
15. Vidal J, de Lamotte F, Gogniat G, Soulard P, Diguet JP. A co-design approach for embedded system modeling and code generation with uml and marte. *Design, Automation and Test in Europe Conference and Exhibition, 2009. DATE'09*, 2009; 226–231.
16. Ristau B, Limberg T, Fettweis G. A mapping framework for guided design space exploration of heterogeneous mp-socs. *Design, Automation and Test in Europe Conference and Exhibition, 2008. DATE'08* March 2008; :780–783doi:10.1109/DATE.2008.4484910.
17. Avnit K, Sowmya A. A formal approach to design space exploration of protocol converters. *Design, Automation and Test in Europe Conference and Exhibition, 2009. DATE'09*, 2009; 129–134.
18. Hamann A, Jersak M, Richter K, Ernst R. A framework for modular analysis and exploration of heterogeneous embedded systems. *Real-Time Syst.* 2006; **33**(1-3):101–137, doi:http://dx.doi.org/10.1007/s11241-006-6884-x.
19. Henia R, Hamann A, Jersak M, Racu R, Richter K, Ernst R. System level performance analysis - the symta/s approach. *Computers and Digital Techniques, IEE Proceedings* - Mar 2005; **152**(2):148–166, doi:10.1049/ip-cdt:20045088.
20. Hendriks M, Verhoef M. Timed automata based analysis of embedded system architectures. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006; 8 pp.–, doi: 10.1109/IPDPS.2006.1639422.
21. Viehl A, Schonwald T, Bringmann O, Rosenstiel W. Formal performance analysis and simulation of UML/sysml models for esl design. *Design, Automation and Test in Europe Conference and Exhibition, 2006. DATE'06* March 2006; **1**:1–6.
22. Marculescu R, Ogras UY, Zamora NH. Computation and communication refinement for multiprocessor soc design: A system-level perspective. *ACM Trans. Des. Autom. Electron. Syst.* 2006; **11**(3):564–592, doi:http://doi.acm.org/10.1145/1142980.1142983.
23. Woodside M, Petriu DC, Petriu DB, Shen H, Israr T, Merseguer J. Performance by unified model analysis (puma). *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, ACM: New York, NY, USA, 2005; 1–12, doi:http://doi.acm.org/10.1145/1071021.1071022.
24. Wodey P, Camarroque G, Baray F, Hersemeule R, Cousin JP. LOTOS code generation for model checking of stbus based soc: the stbus interconnection. *This paper appears in: Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings. First ACM and IEEE International Conference on* June 2003; :204–213.
25. OMG. UML 2.0 Superstructure Specification. *http://www.omg.org/docs/ptc/03-08-02.pdf*, Geneva, 2003.
26. Garavel H, Lang F, Mateescu R, Serwe W. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. *Computer Aided Verification (CAV'2007)*, vol. 4590, Berlin Germany, 2007; 158–163.
27. Jaber C, Kanstein A, Apvrille L, Baghdadi A, Moenner PL, Pacalet R. High-level system modeling for rapid hw/sw architecture exploration. *Proc. of the 20th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP'2009)*, 2009.
28. Apvrille L, *et al.*. TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit. *IEEE transactions on Software Engineering*, vol. 30, 2004; 473–487.
29. Ahumada S, *et al.*. Specifying Fractal and GCM components with UML. *XXVI International Conference of the Chilean Computer Science Society (SCCC'07)*, Iquique, Chile, 2007.
30. Knorreck D, Apvrille L, Saqui-Sannes Pd. TEPE: a sysml language for time-constrained property modeling and formal verification. *Proceedings of the third IEEE International workshop UML and Formal Methods - ULM&FM'2010*, IEEE, 2010.

31. Knorreck D, Apvrille L, Pacalet R. An interactive system level simulation environment for Systems on Chip. *ERTSS - Embedded Real Time Software and Systems*, 2010.
32. Iso 7816 smart card standard:. Http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816.aspx.