# On the use of C++ for system-on-chip design

Diederik Verkest, Johan Cockx, Freddy Potargent
IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
Diederik.Verkest@imec.be

Gjalt de Jong
Alcatel, Francis Wellesplein 1,
B-2018 Antwerp, Belgium

Hugo De Man
Katholieke Universiteit Leuven, Belgium
IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

## Abstract

*To model complex embedded systems at a high level of abstraction, existing languages usually stress the specification of functionality. Our experience with industrial strength designs, however, shows that structure and timing must be considered from the beginning, and refined concurrently with functionality. We present a C++ library for the modeling of concurrency and timing in an executable system model. Together with the built-in extensibility of C++, this library is the foundation for a rich set of primitives supporting different modeling paradigms in a single language framework. This allows designers to start with a conceptual system model capturing functionality, structure and timing as desired, and gradually refine it to a fully detailed, implementable model using the most appropriate modeling paradigm for each phase and part of the design.*

## 1. Introduction

Embedded telecommunication (e.g. ATM) and multimedia (e.g. MPEG-4) systems are becoming more and more complex. Several factors complicate the specification and design of these systems.

**Lack of abstract executable models.** System specification and initial design is traditionally based on natural language documents and informal block diagrams, possibly supplemented by point tools (MATLAB[6], BONeS[7], SES/Workbench[8], ...) for a more detailed exploration of specific design aspects. An executable model for the complete system is typically only constructed at the register-transfer level. This approach is not feasible for a one hundred million transistor chip to be designed in a few months. There is a need for executable system models at a much higher level of abstraction.

**Heterogeneity** both at conceptual and implementation level. At the conceptual level, different modeling paradigms (dataflow, synchronous, reactive) are appropriate for different parts of the system. At the implementation level, systems consist of general purpose and application specific processors, on-chip busses and memories, and reconfigurable and dedicated hardware, and have a very substantial software content. A mix of formalisms is required to efficiently model such a heterogenuous system at different levels of abstraction.

**Dynamic behavior.** In recent designs, control flow is more and more data dependent (compare MPEG-2 to MPEG-4, think of ATM networks). Dynamic memory management and dynamic processes are needed to implement such functionality. This makes timing (or scheduling) and resource issues harder to analyze but at the same time more important. Existing abstract languages tend to make abstraction of these issues.

**Reuse of IP blocks.** Intensive reuse is the key to design complex systems in a short time. This however introduces architectural and timing constraints that have impact at the system level and are difficult to capture in an abstract, functional system model.

**Design variations.** The vast majority of new designs are variations or combinations of existing designs. When a complete system is integrated on a chip, new versions can no longer be created by replacing some components; instead, a more abstract model of the system needs to be modified. Documentation of design decisions and trade-offs for the original design is essential to guide such modifications.

**Changing requirements**, and unstable standards. It is usually difficult or even impossible to completely specify the required functionality before starting the design effort. To gain a better understanding of the problems involved in the specification of complex systems at a high level of abstraction, we have studied the *requirement specification*, *feasibility study* and *design specification* of an industrial,

ATM related design. From these natural language documents, we have tried to create an executable specification capturing the system functionality while making abstraction of implementation issues such as architecture and timing. *This did not work.* The available documents do not provide sufficient information to reconstruct the full functionality of the design. Instead, they contain ample detail on structural and timing issues.

Although the approach to system design used in these specifications is not compatible with a classical top-down approach, we believe that the – very experienced – system designers writing these specifications did a good job: they considered and elaborated the key aspects of the design first, making abstraction of less important aspects. For many parts of the system, structure and timing issues are more important than detailed functionality. For the whole system, functionality, structure and timing are all part of the initial specification and are refined concurrently.

We believe that system design should be based on an executable model in which functionality, structure and timing can be refined concurrently from concept to implementation. This executable model should support different modeling styles and be easily extensible. We have used C++ and added a library called TIPSY implementing concurrency and a model of time. C++ provides the flexibility and extensibility needed to support different modeling styles, and the TIPSY library provides the glue to join these different modeling styles in one model.

Our approach helps to solve the system design problems listed above. When structure and timing aspects are essential, a model including these aspects is more abstract than a model containing only irrelevant functional details. The flexibility and extensibility of C++ help to handle heterogeneity. Structural and timing aspects of reused blocks can be taken into account from the beginning. Executable models provide a good basis for documenting design decisions and trade-offs. Concurrent refinement of functionality, structure and timing corresponds to the spiral model[9] of software development, which is very effective at handling changing requirements and is compatible with concurrent engineering for a reduced design cycle.

## 2. Concurrent design

In this section, we explain our concurrent design approach in terms of a three-dimensional view of abstraction. We show that a pure functional model is not necessarily more abstract than a performance model stressing structure and timing. Next, we motivate our approach by showing how it helps to solve two key problems facing today's system designer: the feasibility study, and IP reuse.

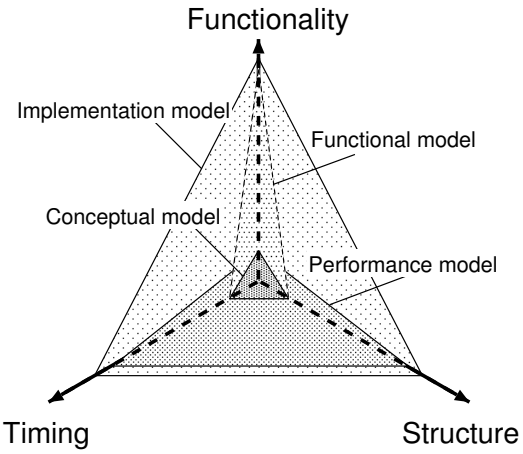


**Figure 1.** *A three-dimensional view of abstraction.*

### 2.1. A 3-dimensional view of abstraction

Traditionally, a system model is considered to be abstract if it stresses functionality and makes abstraction of implementation structure and timing. We believe that this top-down approach is not appropriate for complex embedded systems; structure and timing are key issues for these systems and must be specified and designed at the same time as, or sometimes even before, the functionality.

In general, the level of abstraction of a system model is independent of which aspects of the system are modeled (figure 1; for the meaning of terms such as "abstraction", "functionality", "structure", and "timing", we follow the RASSP taxonomy[1], which has also been adopted and extended by the VSI[4] initiative). Functionality, structure and timing are orthogonal aspects of a system model and can each be described very abstractly or very detailed. A pure functional model is very detailed in terms of functionality, but abstract in terms of structure and timing. A model that makes abstraction of functionality but details the structure and timing of a system is called a performance model; it is typically used to identify performance bottlenecks and resolve resource usage issues. A model combining functionality and timing is called a behavioral model. An initial, conceptual system model is typically abstract in all three dimensions. The final implementation model details all three.

The importance of structure and timing in an industrial system design flow (figure 2) is confirmed by the current popularity of performance modeling tools such as BONeS[7] and SES/Workbench[8] with many industrial system designers. A limitation of performance models is that it is usually hard to refine functionality within them. Performance models do not provide a smooth path to implementation. Academic approaches typically stress the creation of a pure functional model, from which an implemen-

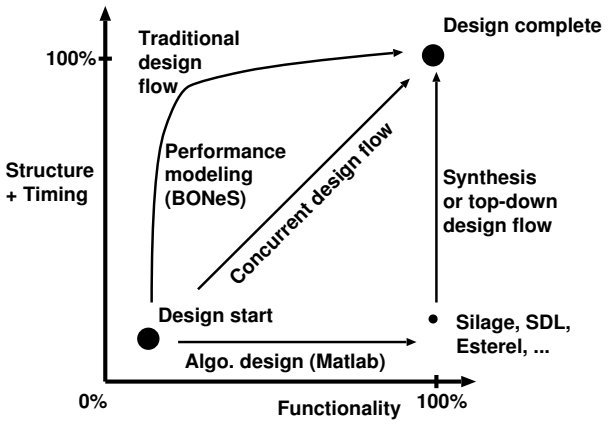


**Figure 2.** *Comparison of academic versus industrial system design flows.*

tation can be synthesized. We believe the best approach is concurrent refinement of all three aspects.

This approach to system design is similar to solving a jigsaw puzzle. Such a puzzle is not solved top-down; instead, it is solved "concurrently", by first placing key pieces such as border pieces and pieces with easily recognizable features at all levels. The holes are filled in later.

### 2.2. The feasibility problem

A crucial system design problem is the feasibility problem. Suppose that the high level requirements, both functional and non-functional, for a new chip design are known, and that a designer has to figure out whether the chip is feasible. This basically involves estimating the chip area, power consumption and pin count. Often, the only way to do this is to ask an experienced designer to define an initial architecture for the chip, reusing existing blocks where possible, and to design a rough register-transfer level implementation for the new blocks. The architecture and implementation are probably only defined on paper, with block diagrams and natural language, or possible even only in the head of an experienced chip designer, to spend as little effort as possible as long as the feasibility of the project is not established. Based on the gate-counts in this first rough design, technology information, and detailed information for reused blocks, the designer can estimate chip area, power consumption and pin count, and derive both technical feasibility and a cost estimate for economical feasibility.

For very complex chips, it is no longer possible for a designer to do the feasibility study on paper; an executable model is needed to support the process. In the context of a feasibility study, it is essential that an executable model captures only relevant details of the system; every additional modeling effort unnecessarily increases the effort spent on a possible infeasible design. But which details are relevant may differ strongly from design to design, and from block to block within a design; for some blocks, the detailed functionality may be essential, while for others, a very abstract functional model with detailed timing information may be required. For example, if the chip is going to implement a known standard on a new architecture, exploration of the architecture is essential and the functionality need not be modeled in detail. Thus, an executable system modeling environment needs to support the concurrent modeling of diverse system aspects at different levels of abstraction.

### 2.3. Reuse

The puzzle approach to system design provides a good framework for IP reuse. For reused blocks, a very detailed model specifying functionality, architecture and timing, typically at the register-transfer level or lower, is already available and fixed; a more abstract, functional model may or may not be available.

In a top-down approach, it is difficult to ensure that the original, functional model is refined in terms of architecture and timing in a way that is compatible with the pre-existing reuse block. If no abstract model of the block is available, it is also difficult to integrate a block model with structural and timing details in a functional system model.

With a concurrent design methodology, the relevant structural and timing characteristics of the reused block can be incorporated in the system model from the beginning on. Both the system model and the block can include abstract structural and timing information. If an abstract model of the reused block is not available, the detailed model can more easily be integrated in the system model.

## 3. Timed Concurrent C++

The modeling of functionality, structure and timing of a heterogeneous system at different levels of abstraction requires a mix of features that cannot be found in any existing language. Instead of creating a new language, we have taken an existing general purpose programming language (C++) and extended it with library constructs (TIPSY). These library constructs semantically extend the language with new primitives. When these extensions are used in a system model, their implementation is not considered as part of the system model, but as part of the modeling environment (figure 3). We have extended C++ with primitives for concurrency and timing, and called the new language Timed Concurrent C++.

In this section, we first motivate our choice for C++ as a base language. Then, we present the TIPSY library. We show that Timed Concurrent C++ can model functionality,

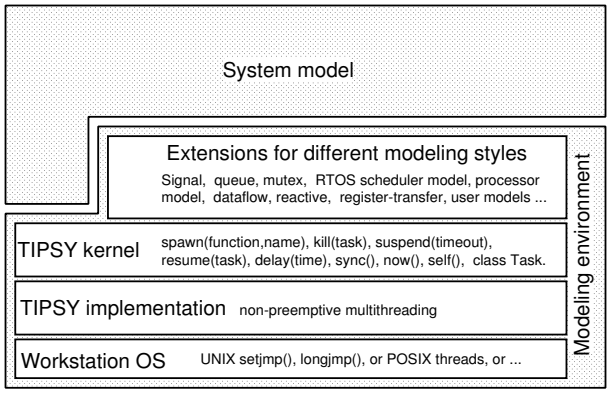


**Figure 3.** *The layered structure of the TIPSY library.*

structure and timing at several levels of abstraction. Finally, we sketch a system design flow based on C++ and TIPSY.

### 3.1. Why C++?

C++ is best known as an object-oriented language, but is more accurately described as a multi-paradigm language[5]. C++ mechanisms such as classes, templates and operator overloading can be used to elegantly integrate features that are not available in basic C++. Classes and operator overloading can be used for example to define an integer with an arbitrary number of bits; such integers are commonly used in hardware descriptions. Templates can be used to code a set of generic algorithms; these algorithms can then be reused for different types in different designs. Operators can be overloaded to refine their semantics, for example by adding an execution time, without changing the code. This extensibility is crucial to support a mix of modeling paradigms in a single language framework and makes C++ well suited for executable system modeling. Using C++, we can provide a rich set of modeling primitives to system designers without creating a new syntax or compiler.

### 3.2. TIPSY library primitives

The TIPSY (TImed Parallel SYstem modeling) library implements primitives for concurrency and time. Extensions supporting different modeling styles are implemented on top of these primitives.

Concurrency is modeled using non-preemptive multithreading. Threads or tasks can dynamically spawn and kill other tasks, and can block (suspend) until resumed by another task or a timeout. For efficiency reasons, each task has a local copy of the current time; tasks synchronize their local times only when they communicate. Processing delay is modeled by calling the delay function; if the delay function is not called, all computations are zero-delay.

Inter-task communication is based on shared data, for the practical reason that it is the basic communication mechanism available in a multi-threading environment. Shared data allows an efficient implementation of other communication primitives (figure 3). It is normally not used directly in the system model. Before accessing shared data a task (or the communication primitive called by that task) must call the sync function to synchronize its local copy of the current time with that of the other tasks; this ensures that shared data accesses are executed in correct time order. The sync function is a simulation primitive only, and does not represent a mechanism that will be part of the final implementation.

The TIPSY primitives can be used either directly in the code modeling a system, or indirectly through extensions for different modeling styles (figure 3). A modeling style can refer to a model of computation (dataflow, reactive, ...) or implementation platform (a real-time operating system for software implementation, registers and logic gates for register-transfer level hardware, ...), or it can refer to a design style (a specific architecture template, a bus or communication protocol, ...). Extensions can be completely generic, or specific for a certain application domain, design group, or design. Extensions from different sources can be combined as they all use the same underlying model of time and concurrency.

The TIPSY primitives as well as some extensions (mutexes, signals, queues, timers) look very much like the functions provided by a Real-Time Operating System (RTOS), but there is a fundamental difference. An RTOS is used for implementation, and tries to fulfill timing requirements using fixed hardware. TIPSY, however, is used for simulation, and *obeys the specified timing behavior* (in simulation time, not real time) assuming that the necessary hardware is available. The presence of a delay function in TIPSY makes this difference concrete. An RTOS does not have a delay function; instead, execution delay follows implicitly from the time taken to execute the code on the available hardware. The fundamental reason why RTOS primitives and TIPSY primitives are similar is that both try to provide a rich set of primitives closely matching the designer's way of thinking.

### 3.3. Modeling of the 3 dimensions

Basic C++ only supports the modeling of functionality and to some extent also structure: classes can be used to represent design blocks, but the concurrent behavior of blocks is not supported. Concurrency is essential for system modeling. Hardware is intrinsically parallel and system software is usually implemented as a set of cooperating tasks under control of a real-time operating system. Even at the conceptual level, many systems are naturally decomposed

into a set of concurrently executing tasks. Finally, C++ does not provide a model of time. Timed Concurrent C++, however, can model functionality, structure, timing at different abstraction levels.

**Functionality** can be modeled in full detail, to the bit level if desired with an appropriate bittrue library; but functionality can also be abstracted and replaced by a statistical behavior and/or delays for performance modeling.

**Structure** can be represented using C++ classes. A class can represent a block (or component) of the design with a well-defined interface. Class data members can be other blocks, thus representing hierarchy. The class constructor can spawn one or more TIPSY tasks representing the concurrent behavior of the block. Blocks can be instantiated more than once and each instantiation works independently. Blocks can represent the logical as well as the physical structure of the design.

**Timing behavior** of a task can be specified directly using the delay function, or indirectly by waiting for an event in another task using the suspend and resume function. The suspend and resume functions provide an abstract model of time, expressed for example as a (partial) ordering between events.

### 3.4. Design flow

Timed Concurrent C++ is the glue that enables a design flow based on concurrent refinement of functionality, structure, and timing. Such a design flow starts from a conceptual model of the design, and gradually refines it to an implementation model (figure 1).

The first, conceptual model of a system uses free-style C++ and whatever library support is available and appropriate to capture the original ideas or requirements as directly as possible. The rich set of primitives required for this approach are either directly available in TIPSY or implemented as extensions. The absence of restrictions allows the system designer to use his own judgement and concentrate on the key design issues, ignoring all irrelevant details. As long as the feasibility of the design is not established and some basic design choices have not been made, insisting on adherence to a formal approach may increase the cost of building an executable model so much that it is no longer cost-effective.

As design choices are made, the system model is gradually refined: by filling in functional or timing details; by expanding design blocks into an equivalent set of cooperating subblocks; by generally reorganizing the code so that it more closely matches the target architecture; for example, the code for conceptually distinct tasks that will be implemented in one hardware block can be merged in one C++ class or even one task; by restricting communication such that it uses only the communication primitives supported in the chosen implementation path for each part of the design; for example, RTOS primitives for software parts, signal-based protocols for hardware parts, fifo's and firing rules for a part that will be implemented with a dataflow compiler.

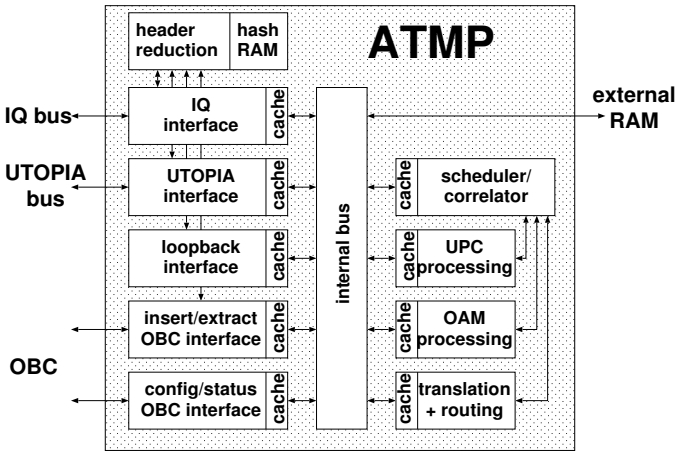


**Figure 4.** *ATMP architecture.*

The refinement process stops when the code can be translated directly, possibly with a syntax chance but without taking any further design decisions, to a description in terms of the implementation platform: for example software tasks communicating through RTOS calls, register-transfer level VHDL for logic synthesis.

## 4. Results

TIPSY has been used to model the ATMP (ATM Processor see Figure 4), a representative state-of-the-art industrial design in the telecommunication area, at different stages of the design process, starting from a conceptual model.

The ATMP is located at a local exchange and routes ATM cells from the core network through ADSL modems and standard telephone lines to private homes and vice versa. It performs header translation, OAM (Operations and Maintenance), UPC (Usage Parameter Control, also called policing or traffic contract verification), and supports QOS (Quality Of Service). The ATMP was originally specified and designed using a classical approach with natural language documents. Functionality, structure and timing were refined concurrently. We have studied these documents and created several TIPSY-based executable models representing different refinement steps in the design flow.

In the first, conceptual model, TIPSY tasks represent virtual connections in the ATMP. ATM is connection-based: cells travel through the network along virtual connections, which can share the same physical connections. Virtual connections can be created and destroyed dynamically; this corresponds to the dynamic spawning and killing of TIPSY

tasks.

The creation, configuration and destruction of virtual connections is controlled by software running on the OBC (On Board Controller, not shoen in Figure 4). In principle, a model of the algorithm governing the creation, configuration and destruction of connections could have been included in the conceptual model, but the original specification already assumed a separate software processor for this functionality. This design decision was respected in the TIPSY model by creating separate ATMP and OBC classes and implementing the different tasks as member functions of the corresponding class. There is no other difference between hardware and software tasks in the conceptual ATMP model; for example, tasks that are dynamically created and destroyed cannot be implemented directly in hardware, but are still used to describe functionality that will be implemented in hardware.

The code for the virtual connection tasks in this first, conceptual model is not functionally complete. The original specification simply referred to ATM standards for functional details. Ideally, these standards should be available in an executable form, so that they can simply be linked into the conceptual model. As a practical alternative, we have first elaborated only the most essential functionality and replaced the remainder by comments, delays and/or random choices.

This model proved useful in two ways. First, it is a clear form of documentation, and allowed a new project member to understand the ATMP design in a few weeks instead of months. Second, it allowed us to study the distribution of the load over the connections and find a good mapping of these dynamic connections to a static structure.

The conceptual model was gradually refined. To get rid of dynamic task creation and destruction, a static connection table was created. Each entry of that table stores the state of one virtual connection, and a fixed number of tasks processes the ATM cells one by one, accessing the state of the corresponding virtual connection in the table. The connection table, cell queues and other data structures were moved to an off-chip RAM and a shared bus and caches were added. The resulting model closely matches the ATMP architecture (figure 4), with one C++ class representing each block, and one or two TIPSY tasks per class. The top-level ATMP class simply instantiates these blocks as data members. Finally, the IQ interface block was refined to obtain a cycle-true model for the IQ bus access protocol.

TIPSY was also used for an ADSL modem design, to build a virtual prototype of the hardware and test the embedded software on it The virtual prototype includes timing information at a high level of abstraction, i.e. without going to a detailed register-transfer level implementation model.

Finally, TIPSY is used for system-level synthesis in the Matisse compiler[2] to model timing and concurrency.

## 5. Summary

Our approach to system design is based on the observation that abstraction has three independent dimensions: functionality, structure and timing. We believe that functionality, structure, and timing should be refined concurrently, using a mix of modeling styles to match the heterogeinity of the system to be designed. To create an executable model supporting our approach to system design, we use C++. Classes, templates and operator overloading make C++ a very extensible language, in which new features can be introduced without syntax changes. We created the TIPSY library to introduce concurrency and timing in a C++ system model. Different modeling styles can be combined because they all rely on the same primitives for time and concurrency. The TIPSY models are executable and avoid the ambiguity of natural language.

**Acknowledgements** This research was sponsored by Alcatel and IWT (MEDEA SMT A-403). Their support is gratefully acknowledged. We also thank Alcatel for making available the ATMP design documents and the ATMP designers for their invaluable time.

## References

[1] RASSP Taxonomy Working Group, "RASSP VHDL Modeling Terminology and Taxonomy", revision 2.3, June 23, 1998, `http://www.atl.external.lmco.com/rassp/taxon/rassp_taxon.html`

[2] J. L. da Silva Jr. et al, "Efficient System Exploration and Synthesis of Applications with Dynamic Data Storage and Intensive Data Transfer", *DAC'98*.

[3] Patrick Schaumont et al, "A Programming Environment for the Design of Complex High Speed ASICs", *DAC'98* 315-320.

[4] VSI initiative, `http://www.vsi.org/`

[5] "The *Real* Stroustrup Interview", *IEEE Computer* 110-114, June 1998.

[6] MATLAB, `http://www.mathworks.com/`

[7] BONeS `http://www.altagroup.com/alta/products/bonesdat.html`

[8] SES/Workbench, `http://www.ses.com/`

[9] Barry W. Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, April 1988, 61-72.