Matisse: a concurrent and object-oriented system specification language

Julio Leao da Silva Jr.[1] *Chantal Ykman-Couvreur*[1] *Gjalt de Jong*[2]
{silva,ykman}@imec.be *jongg@sh.bel.alcatel.be*
[1]*IMEC, Kapeldreef 75, B-3001 Leuven, Belgium*
[2]*Alcatel Telecom, F.Wellesplein 1, B-2018 Antwerpen, Belgium*

Abstract

We present **Matisse**, a concurrent object-oriented system specification language, well-suited for protocol processing applications used in telecom networks. An industrial application used in ATM networks is introduced. From this case study, we derive the requirements that must be supported by **Matisse**. **Matisse** is the entry point for the methodology presented in [6], that bridges the gap between system specification and synthesis tools commercially available. In contrast to the system specification languages currently used in industry, **Matisse** is implementation-independent and permits the exploration of different embedded hardware/software realizations.

Keywords

System specification, hardware/software codesign, protocol processing applications, system synthesis and object-oriented languages.

1 INTRODUCTION

Modern telecom systems are rapidly increasing in design complexity. Telecom network applications include protocol processing systems for broadband networks [9], wireless infrastructures, and interactive video-on-demand servers. Currently, protocol processing applications are partitioned in hardware and software components that are designed separately, which often introduces specification and implementation mismatches that are only detected at the final design stages. System integration and test phases can take nearly 50% of the complete design cycle for typical protocol processing applications. Software components are usually specified using SDL [15]. C or C++ code is then generated and compiled to machine instructions for the target processor. Run-time support is added for managing concurrency and interprocess communication [18]. Hardware components are specified at the register transfer level, using VHDL or Verilog, where detailed clock cycles, and specific archi-

VLSI: Integrated Systems on Silicon R. Reis & L. Claesen (Eds.)

tectural decisions are already fixed. This level of specification is often hard to read, modify, and reuse. Small changes at the system level often require very substantial changes in the components specification.

There are several approaches focusing on system design for embedded hardware/software. The hierarchical FSM model is a powerful formalism for reactive control behaviors, but it does not support well abstract data structures and object-oriented features. Heterogeneous design environments, like Ptolemy [3] and CoWare [1, 2], aim to provide an open environment to integrate different models of computation. Most system-level research and CAD innovations today are focussed on Digital Signal Processing (DSP) applications (e.g. [3, 10, 11]). Commercial tools include SPW from Alta/Cadence, COSSAP from Synopsys and DSP Station from Mentor.

In contrast to DSP applications, protocol processing applications are different in nature. In particular, protocol processing applications require manipulation of complex data structures that are often dynamically created and destroyed at run time, as opposed to the signal flow present in DSP applications. DSP models are not well-suited for control-dominated data processing behaviors found in protocol processing applications that heavily rely on tight interactions between control-flow algorithms and stored data structures. Due to many differences in nature between these application domains, system models should be domain-specific.

Distributed programming languages have been proposed for programming general-purpose multiprocessor systems or distributed networks of workstations [4, 5, 12, 16]. While their underlying models are related to our **Matisse** model, their implementation targets are different: they rely on elaborate run-time environments and are intended for pure software implementations. In contrast, our implementation target is intended for optimized embedded single-chip hardware/software realizations.

Protocol processing systems are extremely complex and they must be modeled, debugged and simulated at a high level of abstraction before proceeding to implementation. In this paper, we present **Matisse**, a system specification language that supports high level specification of protocol processing applications. The remainder of this paper is organized as follows. In Section 2, we present an actual application example in order to derive the requirements that must be supported by our system specification language. In Section 3, we describe the **Matisse** language in a detailed way. In Section 4, we overview how **Matisse** fits in a hardware/software codesign flow and how **Matisse** can be used for design exploration. Finally in Section 5, conclusions are presented.

2 SYSTEM SPECIFICATION

This section presents an actual protocol processing application used in telecom networks. This case study is used to derive requirements to be supported by the **Matisse** language.

2.1 Case study

ATM [14] is a fast packet-switching transfer mode that supports high-speed integrated services by splitting all communication messages into equal 53-byte cells, called ATM cells. These cells can carry any kind of information, be it computer data, video, or voice. By using small cells to transfer data, the technology enables networks to support a wide variety of traffic, ranging from high to low bandwidths, and from bursty to steady bit rates.

One representative case study is an user transparent connectionless router called Alcatel Connectionless Transport Server (ACTS) [17] that provides the necessary functions for the direct provision and support of data communication between geographically distributed computers or between LANs over an ATM based broadband network. In its current implementation, the ACTS consists of several boards, each one consisting of several processors and co-processors (implemented as custom ASICs) and a programmable supervising microprocessor for executive control.

A concrete example of one of those ASICs, named Segment Protocol Processor (SPP), is used to demonstrate the requirements to be supported by the Matisse language. The algorithms, implementing the SPP functionality, make use of stored complex data structures, shown in Figure 1. The right-hand side of the figure shows a FIFO, where incoming user cells are buffered. Packet records are accessed through two levels of tables with the local (LID) and multiplexing (MID) identifiers. A packet record contains various fields, such as the number of cells received so far, the time the first cell was received and a pointer to a list of routing records.

These algorithms can be described as a set of tasks that cooperate with each other through the shared data structures shown in Figure 1. The tasks performed by the SPP are now briefly described:

1. Data In processing Process an incoming user cell. Look up the packet record, or allocate a new one if the cell is the first cell of a packet. Perform various checks and update various fields in the packet record. Store the user cell in the FIFO.

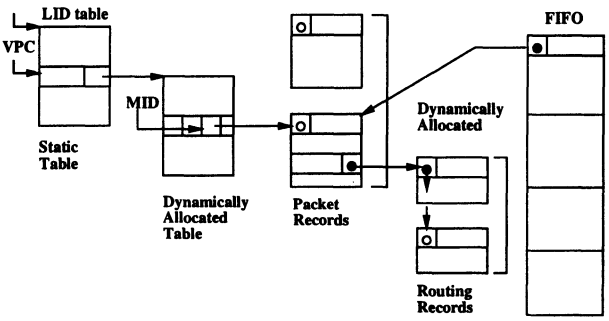

Figure 1 SPP stored data structures

2. Ingress Screening Request (ISR) generation Check the residency time of the cell in the FIFO. If the residency time is larger than a software defined threshold, generate an ingress screening request for the coprocessor. Ingress screening is needed for virtual private networks. It consists of checking whether the destination subscriber is a member of the closed group that the source subscriber is in. The processor which performs the ingress screening is also responsible for sending a routing request to a router in the network.
3. Routing Reply (RR) processing Input a routing reply, look up the packet record for which the routing information is meant, generate a routing record for the packet record, and update the state of the packet in the packet record.
4. Data Out processing Check the residency time of the cell in the FIFO. If the residency time is larger than a certain threshold, forward the cell on the network. If the cell is the last cell of a packet, deallocate the packet record.
5. Time Out processing This task is necessary to deal with situations where the last cell of a packet has been discarded somewhere in the network, so that the packet record is never deallocated. The task consists in inspecting the next packet record in the memory region allocated for packet records, checking the lifetime, and deallocating the record if the lifetime exceeds some threshold.
6. MID deallocation request generation Read the next packet in the packet record deallocation list, and generate a MID deallocation request for the packet, so that reserved bandwidth can be deallocated at the destination.

Others ASICs used in the ACTS, such as the Packet Handler Processor and the Preventive Congestion Control processor, may be described in a similar way, by means of a set of cooperative tasks operating on shared data structures.

2.2 Requirements

The requirements to specify protocol processing applications such as the case study previously described are now presented.

Protocol processing applications are data and memory intensive systems. They are conceptually seen as sets of concurrent tasks for accessing data. Therefore data have to be considered as stored objects from the beginning. Concurrency tends to be at the task level and is usually coarse to medium scale. However in today's design practice, due to a lack of a concurrent system specification language and an appropriate system design flow, conceptually concurrent tasks are implemented as interleaved consecutive tasks, without exploiting the nature of the concurrency itself.

Although the target implementation of a protocol processing application is often a mixture of software and hardware processors, protocol processing applications are best-suited to be conceived at the top level from a software perspective and advantages such as fast simulation and earlier design valida-

tion can be obtained. Control constructs, such as *if-then-else*, *for* and *while loops*, are essential for capturing the algorithmic behavior of each task. Each of the SPP tasks may be described as a sequential program using the usual control constructs available in languages such as C++.

While an object-oriented model is not necessary, it has been proven successful in the software design community, and it also plays a central role in large scale hardware/software system design. Object-oriented languages Object-oriented languages support data abstraction, encapsulation, polymorphism, function overloading and inheritance, which are invaluable features in any large scale development. With these abstraction facilities, implementation decisions and low-level specification details can be hidden or easily updated, allowing easy and fast design exploration. For instance, shared data structures may be initially specified as abstract data types, that will be refined later in the design flow.

Concurrent object-oriented models are well-suited to model protocol processing applications, since they are intended to model concurrent computations to be executed on more than one processor. Objects can encapsulate tasks as well as (shared) data structures. Remote procedure calls can encapsulate interprocess communications. The system specification language must also: reflect the conceptual partitioning of the system, seen as a set of concurrent tasks for accessing data, be independent from the final implementation, be manipulatable to permit easy updating and efficient design exploration, and be easily retargetable to different embedded hardware/software realizations. This contrasts with current system specification practices, which are using VHDL for specifying the hardware processors, and C/C++ for specifying the software processors.

3 MATISSE LANGUAGE

From the previous requirements, we decided to follow an object-oriented approach for the system specification of protocol processing applications. Therefore, the **Matisse** language is extended from the widely used object-oriented programming language C++. We introduce minimal syntactic extensions to C++ to allow the description of concurrent tasks, communication and synchronization among them. Compatibility with C++ enables new users already familiar with C++ to be productive in a very short amount of time. Also, existing debugging and compiling tools can be easily adapted for early functional validation of the system specification. Finally, this enables us to leverage on the wide corpus of existing software compilation and runtime support tools for our software implementation path. This is important since software implementation represents a substantial part of many of our target applications.

There are many concurrent object-oriented programming languages extended from C++. All of them are intended for specifying systems consisting of concurrent programs, running on a network of workstations. Compilers

for those languages generate C++ programs with calls to an elaborate run time Operating System (OS) designed for software processors only. Matisse is intended for specifying systems at the chip level instead. These systems consist of concurrent processes, running on a mixture of embedded software and hardware processors, each one with its own *ultra-light* OS. To be efficiently implementable in both hardware and software processors, these OS should offer only minimum support for task scheduling, interprocessor communication and synchronization.

More precisely, the Matisse language uses some of the high-level abstractions existing in Compositional C++ (CC++) [5]. CC++ is a concurrent object-oriented language extended from C++ using only a few new keywords. Simplifications were brought, taking into account the requirements introduced in Section 2.2. Also systems specified with Matisse must be synthesized into a hardware/software codesign at the chip level.

CC++ allows the user to specify concurrency at all levels, from fine grain to task level concurrency. In protocol processing applications, the user needs to specify concurrency only at the task level. Thus, Matisse allows the specification of concurrency only at this level. In CC++, both thread and local virtual memory space concepts are separated. To model tasks only created at compile time, Matisse allows to create active objects at compile time. These objects encapsulate together a local virtual memory space and a default thread of control, that is initiated at the creation of the active object. Due to these restrictions, run time support can be majorly reduced.

Similarly to CC++, communication between tasks is abstracted, without explicit specification of communication channels and an RPC mechanism is used to implement it. In CC++, data may be remotely accessed directly. In Matisse, data inside an active object are remotely accessed only through a reference to the active object itself. Due to the simplified communication mechanism, instead of providing two synchronization mechanisms as in CC++, Matisse only needs one synchronization mechanism.

Now the different concepts in Matisse are explained in more detail. The concepts are illustrated using simplified code for the SPP.

3.1 Passive and active classes

In Matisse, two types of classes are distinguished: *active* and *passive*.

A passive class is identical to a C++ class. For example, the packet record of the SPP application is a Matisse passive class declared as follows:

```
class packet_record {
  int     field1;
  boolean field2;
  packet_record* next;
};
```

Instances of a passive class are called *passive objects*, and are identical to C++ objects. Passive objects may be *created and destroyed either at compile time or at run time.*

An active class in the SPP application is the "Data In processing" task and it is declared as follows:

```
active class data_in {
    cell_record* cell;
    packet_record* packet;
public:
    data_in ();
    void body (packet_record_mngr *global pr,
               cell_fifo_mngr *global cf,
               input *global input) {
        cell = input->get();            // get a cell from the input
        switch (cell->type()) {
            case BOM:                   // cell is Begin Of Message
                packet = pr->alloc();   // create a new packet record
                //SOME BEHAVIOR
                pr->put(packet);        // store packet info
                cf->enqueue(cell);      // store cell in the fifo
            case COM:                   // cell is Continuation Of Message
                packet = pr->get();     // use an existing packet record
                // SOME BEHAVIOR
                pr->put(packet);        // store packet info
                cf->enqueue(cell);      // store cell in the fifo
        }
    };
};
```

Any instance of an active class is called an *active object.* An active class is identical to a passive class, except that each active object has its own local virtual memory space and may have its own default thread of control. This thread is then initiated at the creation of the active object. It is specified by a special public member function of the active class, called **body**.

In contrast to passive objects, active objects *may only be created at compile time*, to avoid creation of new threads at run time that may be difficult to implement on a hardware processor.

Active classes can inherit from base active classes too, and the usual C++ protection mechanisms apply. So private data elements and member functions of an active class can be used only by the member functions of it. Public data elements and member functions constitute the interface to the active objects of the active class.

3.2 Concurrency at the task level

In a typical Matisse program, the number of active objects is small, compared to the number of passive objects. The passive objects exist as data elements of active objects. Active objects are only created in the *main* function, which

yields the concurrent initiation of their bodies. Hence the *main* function provides the task-level concurrent structure of the Matisse program.

In a simplified SPP version, the main function first creates four active objects and then initiates their bodies that will run concurrently:

```
int main (int argc, char**argv) {
   packet_record_mngr* global pr;  // shared data
   cell_fifo_mngr* global cf;      // shared data
   data_in* global di;             // task
   data_out* global do;            // task

   pr = activenew packet_record_mngr();
   cf = activenew cell_fifo_mngr();
   di = activenew data_in(pr,cf);
   do = activenew data_out(pr,cf);
}
```

Currently we restrict the structure of the *main* function, which may only consist in creating a set of active objects. The keyword **activenew**, taken from UC++ [13], is used to specify the creation of an active object. The semantic of **activenew** is: create an active object, using the C++ **new** function, execute the constructor of each active object and then start all the **body** member functions of the active objects concurrently. Note that the 'bodies' are wrapped in an infinite loop.

3.3 Communication

Accessing data elements within an active object is regarded as local and hence cheap. A thread executing in an active object can access its data elements directly, by using C++ pointers. Active objects can be accessed by each other using **global** pointers. Except for their potentially higher cost of use, global pointers are used just like C++ pointers.

Inside a thread, computation can be executed in another active object via a Remote Procedure Call (RPC), as follows:

```
X *global gp;
result = gp->p(a,b,c);
```

where **gp** is a global pointer to an active object of the active class **X**, **p(a,b,c)** is a call to a member function **p()** defined in the active object referenced by **gp**, and **result** is a variable set to the value returned by **p(a,b,c)**.

An RPC proceeds in three stages: first the arguments of the function **p()** are packed into a message, communicated to the remote active object, unpacked and the calling thread suspends execution, then a new thread is created in the remote active object to execute the called function and at last upon termination of the remote function, the function return value is transferred back to the calling thread, which resumes execution.

Data can be communicated between active objects, by using global pointers, as follows:

```
int len1;
A *global gp;        // A is an active class with data element len2
len1 = gp->len2;     // reading data from another active object
gp->len2 = 5;        // writing data to another active object
```

Reading and writing global pointers must be used with caution, since it involves two communications: one to send the read or write request, and one to return a result or a completion signal.

3.4 Synchronization

Due to concurrent computations, several accesses to data elements or member functions in an active object can occur simultaneously. Matisse provides one method for controlling the order in which things happen, by using atomic functions.

The use of atomic functions is illustrated in the following example:

```
active class packet_record_mngr {
    packet_record *head, *tail;
public:
    packet_record_mngr ();                  // initialize head and tail
    atomic packet_record* alloc ();         // create a new packet record
    atomic packet_record* get ();           // get a packet record from the list
    atomic void put (packet_record*);       // put a packet record in the list
};
```

Whenever several threads are calling an atomic function, this atomic function is executed the required number of times in a sequential order. Also the execution of an atomic function never interleaves with the execution of another atomic function of the same active object. This concept of atomic function is based on the monitor concept introduced by Hoare in [8].

In order to avoid deadlocks, some rules for defining atomic functions must be followed, such as: the body of an **atomic** function must terminate in a finite time, implying that it may not do an RPC, that it may not call other atomic functions of its class, and that a **body** function running for ever may not be declared **atomic**. Member functions may be declared **atomic** in both active and passive classes.

Using atomic functions instead of atomic objects helps the user to specify critical sections that must be as short as possible. In an object-oriented approach, each object (either active or passive) is responsible for its own protection. In Matisse, this is still valid, but deciding which member functions have to be declared **atomic** is currently left to the user.

3.5 Shared Data structures

Each active object represents one local virtual memory space with a default thread executing in it. Passive objects are never shared between active objects. If a passive object needs to be shared by several active objects, the user has several options. He can, for example, specify typical active objects with no **body** function and whose data elements are those passive objects to be shared. In this way, the active object is a kind of memory manager for passive objects. For instance, in the `main` function of our example, introduced in Section 3.2, *pr* is such an active object. *pr* is an instance of the class *packet_record_mngr*, declared in Section 3.4, and it is used to manage *packet records.*

4 SYSTEM DESIGN FLOW

The **Matisse** language, described previously, is used as input to the system design flow [6] depicted in Figure 2. **Matisse** is also used for functional validation and it facilitates design exploration due to its high level of abstraction.

The system design flow starts from an initial concurrent object-oriented specification within the **Matisse** language and targets an heterogeneous implementation of software and hardware processors. The *Matisse program*, using abstract data types, as sets, collections of data, and association tables, specifies the system to be described. This program can be executed, allowing functional validation and debugging.

The **Matisse** program is internally represented as a network of communicating processing objects, managed by an ultra-light OS. This internal representation allows efficient system design exploration and it is still independent from the final HW/SW realization. *Refinement and optimization* consists in: refining abstract data types into efficient complex data structures [20], generating memory management of these complex data structures, which are dynamically allocated and deallocated by concurrent processes [7], optimizing memory ac-

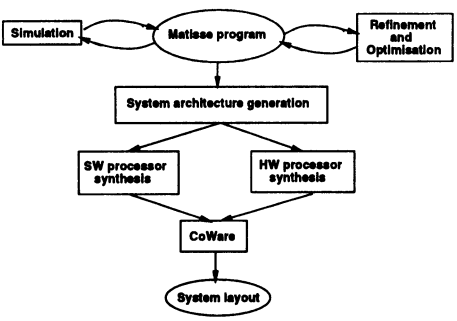

Figure 2 Matisse Design Flow

cess yielding ordering of concurrent threads [19], and exploration of different embedded HW/SW realizations, based on interprocess communication costs.

System architecture generation consists in allocating a number of hardware and software physical processors and mapping the internal representation of Matisse into the target architecture. Communications between active objects assigned to the same physical processors are refined into intraprocessor communications. Communications between active objects assigned to different physical processors are refined into interprocessor communications.

Software processor synthesis consists in generating the complete specification of each software processor, so that synthesis is made possible by using traditional software design tools for code generation. *Hardware processor synthesis* consists of memory synthesis, that generates a distributed shared memory architecture, and VHDL code generation, so that synthesis is made possible by using traditional hardware/behavioral synthesis tools. *CoWare* [1, 2] is used for interprocessor communication synthesis.

5 CONCLUSION

In this paper, we have addressed the system specification problem for protocol processing applications used in telecom networks. We introduced the SPP, an industrial example that demonstrates the main requirements to model such applications at the system level. We presented Matisse, a system specification language extended from C++. The concepts present in Matisse were shown sufficient to specify the SPP. Currently, we are evaluating and refining our design flow using this case study. At the moment, the Matisse compiler generates an abstract machine that can be executed using the CoWare environment.

Using Matisse and the proposed system design flow, the user is able to: write a system specification, independent from the final implementation, and easily retargetable to different embedded hardware/software realizations; validate functionally the specification and explore the design space at system level.

In the near future, we want to show the suitable applicability of our concurrent object-oriented approach on other actual telecom applications. We are also investigating on how to include timing constraints in the system specification and support them through the system design flow.

Acknowledgments This work is part of a joint collaboration between IMEC and Alcatel, partially supported by IWT under project "HASTEC". We also would like to thank B. Lin, K. Croes and E. Umans for numerous insightful discussions. The first author is supported by a Brazilian Governement Fellowship - CAPES.

REFERENCES

[1] I. Bolsens et al. Hardware-software codesign of digital telecommunication systems. *IEEE Proceedings*, April 1997.

[2] CoWare, 2385 Santa Ana Street, Palo Alto, CA 94303, USA and Kapeldreef 60, B-3001 Heverlee

[3] J. Buck et al. Ptolemy: A framework for simulating and prototyping heterogeneous systems. Technical report, University of California, Berkeley, August 1992.

[4] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, September 1993.

[5] K. M. Chandy and C. Kesselman. *CC++: A declarative concurrent object-oriented programming notation.* MIT Press, 1993.

[6] J. Leao da Silva Jr. et al. A system design methodology for telecommunication network applications. In *The Seventh Great Lakes Symposium on VLSI*, 1997.

[7] G. de Jong et al. Background memory mangement for dynamic data structure intensive processing systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 515–520, November 1995.

[8] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[9] B. Kumar and J. Ranade. *Broadband Communications, A Professional's Guide to ATM, Frame Relay, SMDS, SONET and B-ISDN.* McGraw-Hill Series on Computer Communications, 1994.

[10] R. Lauwereins et al. Grape-ii: A system-level prototyping environment for DSP applications. *IEEE Computer*, pages 35–43, February 1995.

[11] H. De Man et al. Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms. *IEEE Proceedings*, 72(2):319–335, February 1990.

[12] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, September 1993.

[13] T. O'Brien et al. UC++ v1.3: Language and compiler documentation. Technical report, London Parallel Applications Center, January 1995.

[14] M. De Prycker. *Asynchronous Transfer Mode, Solution for Broadband ISDN.* Ellis Horwood, 1991.

[15] R. Saracco and P.A.J. Tilanus. CCITT SDL: An overview of the language and its applications. *Computer networks and ISDN Systems, Special Issue on CCITT SDL*, 13(2):65–74, 1987.

[16] B. Selic et al. *Real-Time Object-Oriented Modeling.* Wiley, 1994.

[17] Y. Therasse, G. Petit, and M. Delvaux. VLSI architecture of a SDMS/ATM router. *Annales des Telecommunications*, 48(3-4), 1993.

[18] E. Verhulst. Virtuoso: Providing submicrosecond context switching on DSPs with a dedicated nano kernel. In *International Conference on Signal Processing Applications and Technology, Santa Clara*, September 1993.

[19] S. Wuytack et al. Flow graph balancing for minimizing the required memory bandwidth. In *Proceedings of the International Sympopsium on System Synthesis*, pages 127–132, 1996.

[20] S. Wuytack et al. Transforming set data types to power optimal data structures. *IEEE Transactions on Computer-Aided Design*, 15(6):619–628, June 1996.

BIOGRAPHY

Julio Leao da Silva Jr is a Ph.D. candidate at IMEC. He holds a Ms.C. in Computer Science from the Federal University of Rio Grande do Sul, Brazil. His main interests are in system level specification, hw/sw codesign.

Chantal Ykman-Couvreur is mathematician. She worked at PHILIPS Research Laboratory of Belgium, from 1979 until 1991. She joined IMEC in 1991 to develop techniques for specifications and synthesis of asynchronous circuits. She is currently working on hw/sw codesign of systems at the chip level.

After having been a researcher at IMEC, Belgium, Gjalt de Jong has joined staff research group of Alcatel Telecom. His research interests are in the field of formal specification and verification. He holds a Ph.D. and M.S.E.E. from Eindhoven University of Technology, the Netherlands.