UML-driven TTA-based protocol processor design

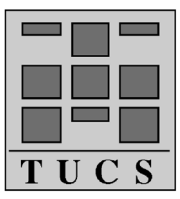
Lilius, Johan; Truscan, Dragos

Publicerad: 01/01/2003

Document Version
Förlagets PDF, även kallad Registrerad version

UML-driven TTA-based protocol processor design

Johan Lilius
Dragos Truscan



Turku Centre for Computer Science
TUCS Technical Report No 518
April 2003
ISBN 952-12-1147-4
ISSN 1239-1891

Abstract

A protocol processor is a programmable processor specifically designed to deal efficiently with protocols. The challenge in designing protocol processors is finding an architecture that is a good compromise between a general purpose processor and a custom, protocol-specific processor. We describe a methodology in which we design both a processor and a program for a network processing application. Our design flow is based on object oriented techniques and targets a TTA processor architecture. TTA is a special processor architecture whose instruction set can easily be tailored. It consists of a number of functional units that are implemented in hardware. One of the tasks in our methods is to identify such functional units. We explain the design flow steps with an IPv6 router implementation.

Keywords: protocol processor, internet protocol version 6, IPv6 routing, UML, design flow, transport triggered architecture

TUCS Laboratory
Embedded Systems Laboratory

Contents

1 Introduction

In the past several years addressing the increasing complexity and performance demands of network processing applications has become an important challenge for hardware designers. On the one hand there is a need for high performance of the applications (that was achieved by using dedicated hardware like ASICs), on the other hand a faster time-to-market where one would like to take advantage of the programmability of general purpose processors. The network (protocol) processors provide a solution to the previous aspects by offering good programmability and high speed hardware.

Reaching these goals simultaneously is very difficult to achieve with traditional approaches like general purpose processors or Application-Specific Integrated Circuits (ASICs). General application processors offer a good level of programmability and a set of resources targeting a wide range of applications, but provide reduced computation speed and also require large area and power consumption. At the other extreme, ASICs are tailored for specific applications in order to provide optimal performance through dedicated hardware, but they offer less programmability and flexibility in design.

In order to meet demanded performance and level of programmability of network processors, programable processors have become more widely used. Programable processors are a class of ICs based on the system on chip (SoC) technology that, by being programmed, perform communication specific tasks more efficiently than general purpose processors.

In this paper we address the problem of designing network applications using programable processors. This comes to suggesting an architecture for the processor starting from the application that it has to implement. The traditional approaches are to adapt the software application to fit the hardware platform, or to design an architecture to serve the software application's requirements. One type of programable processors are the Transport Triggered Architecture (TTA) processors that provide scalable and modular architecture along with a flexible configuration. The main task of the designer is to identify the processor configuration to offer support for a particular protocol processing application with respect to the physical constraints as speed, area or power consumption. We are currently looking at this problem as a hardware/software codesign problem. Our approach is based on the observation that for a given family of protocols (e.g. IP) a number of fundamental operations can be identified. It is these operation that should be provided in the protocol processor architecture as atomic operations.

The main contributions of this report are:

- proposes an UML high-level design flow for protocol processors
- reuses hardware components by applying domain knowledge

- suggests a configuration of the hardware platform by detecting the functionality it has to implement

In the following sections we will give a short description of the TTA architecture, then the design flow steps and their importance in gathering the necessary information for configuring the processor. For exemplification we use an IPv6 router implementation case-study. We also suggest some heuristics in discovering the required functional capabilities of the TTA architecture. Finally, we present an IPv6 Router configuration of our TTA-based protocol processing architecture.

2 TTA architecture

Transport Triggered Architecture (TTA) [8] is a novel processor architecture that brings a number of useful features into the area of embedded systems. The architecture of a TTA processor consists of a set of functional units (FUs) connected by an interconnection network. The interconnection network is represented by one or many busses to which the functional units are connected by input or output sockets. The FUs can be of different types, each of them implementing its function independently. There can be more than one functional unit of the same type in a TTA processor. A TTA can be configured with a variable number of buses and functional units (connected to one or many busses). The functional units are connected to buses by sockets, addressable through logical addresses. The processor is programmed using one type of operation only, the (*move operation*), that specifies data transports on the interconnection network. An operation of the processor occurs as a side-effect of the transports between functional units. Each transport has a source, a destination and data to carry from one unit to another.

TTA architecture provides features like modularity, scalability of performance, flexibility and control of the processor cycle time which are important concepts in the area of embedded systems design. Modularity of the architecture provides a good support for the automation of design. Each FU implements a piece a functionality, and the final configuration can be assembled by connecting different combinations of the functional units. The functional units are completely independent of the others and in the same time of the interconnection network. This structure provides good design flexibility, each FU can be designed separately from the other FUs or from the interconnection network. Moreover, performance can be scaled up by adding extra FUs, buses, or by increasing data transports and storage. When adding FUs, one does not have to change the instruction format, as long as the existing FUs are addressable by the source and destination address length. Since all the operations of the processor are side-effects of data transports on the buses, the processor cycle time depends on the availability of the results

of the functional units. In order to optimize the processor for data throughput, we can add registers to interface the functional units with buses and in the same time to scale down the processing time of each functional unit. Since the main emphasis of a TTA architecture is moving data, it provides an important platform for protocol processing which represent data-intensive applications.

2.1 TACO simulation framework

TACO (Tools for Application-specific hardware/software CO-design) is an integrated design framework for fast prototyping, simulation, estimation and synthesis of programable protocol processors using a TTA-based architecture [21]. The entire architecture of the TACO processors is simulated using a SystemC [17] model, their physical parameters (like area and power use) are estimated in a Matlab model and the processor configurations are synthesized using a VHDL model. The SystemC model and the VHDL model are co-developed so that module characteristics in both models are the same. After deciding the FU types needed by the processor we explore the design space for the target application at system-level. With SystemC simulations and Matlab estimations of different architectural configurations we find one or more candidates that comply with the performance, power and area requirements of the target application. The most feasible one(s) of these can then be synthesized into hardware. More implementation details on TACO processors can be found in [22].

2.2 TACO processor configuration

The development flow in TACO consists of 2 parts: identification of the functional units (FUs), type and number, needed for a specific application, and then the design of the control structures for a given protocol with respect to the required functionality of the application. Identification of functional units also consists of two steps:

- qualitative configuration - where the types of necessary hardware resources are discovered by analyzing the functionality of the system
- quantitative identification - where the number of resources of each type is suggested by analyzing the performance requirements with respect to the physical constraints.

Following we will address mainly the first part of the development flow, namely discovery of functional unit types needed to implement a given application. Identification of new functional units can be done either by selecting from already

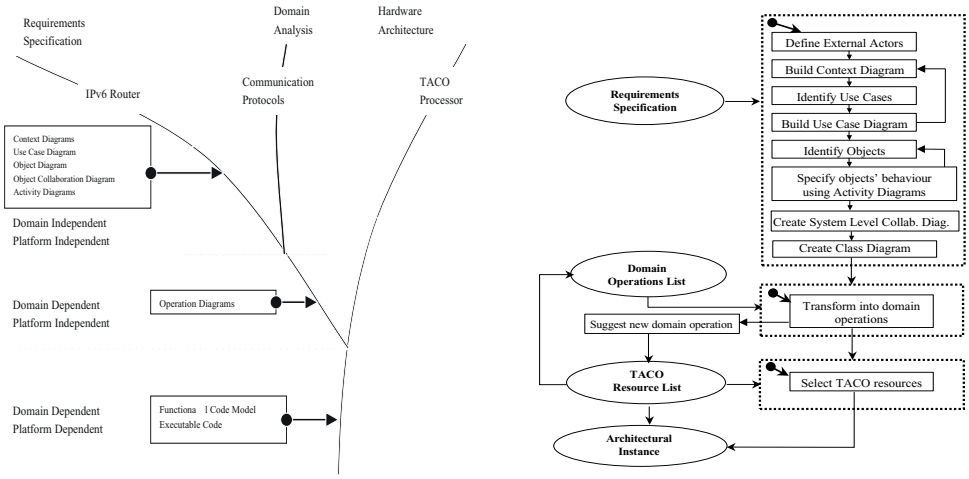


Figure 1: The design flow used in IPv6 Router implementation

created functional units, or by suggesting and building new ones. Each newly created functional unit is accompanied by speed, area and power consumption estimations, given by the Matlab estimation model of TACO. The availability of physical estimations during the design phase allows the designer to perform early design space exploration in order to identify the most suitable configuration according to the application's requirements and constraints.

3 Design Flow

In this paper we experiment a protocol processing design methodology for developing protocol processing applications for TTA processors, and in particular for TACO processors. Since we are talking about configurable processor architectures we have to come up with some heuristics for learning important aspects of the application that reflect in the hardware configuration of the platform. The information is then used in developing the architecture of the processor by suggesting types of resources needed and finally their number.

The Unified Modelling Language (UML) [6] plays a fundamental role in the approach. We use UML as the description language and exploit our work on the semantics of behavioral diagrams of UML [14] to obtain a well-defined executable semantics for our model. Moreover recent work on code-generation [4] can be extended to generate TACO micro-code directly from the final UML specifications.

For a better readability of the specification, we need a common language to de-

scribe both application specification and the hardware resources of the processor. UML proves to be a useful analysis and specification tool because of its graphical representation. Usually there is a big difference between how software and hardware parts of one systems are described. Classes, methods and associations are used in software, while signals, modules, and buses are used in hardware. UML has been initially created for software specification and construction, but it provides a flexible extension mechanism by using stereotypes. Stereotypes are extensibility mechanisms that allow defining conceptual types of objects to extend the UML representation. Through stereotypes one can add or modify notations to represent hardware concepts and to use UML for describing hardware platforms. By using a common way of representation for both hardware and software, the design flow is more easy to understand by both hardware and software engineers, facilitating a uniform design process for the entire system.

Although UML is not used on a large scale in embedded systems design, several UML-based design methodologies have been proposed in literature [2, 3, 12, 19]. In [2] the application analysis is performed and the design constraints captured in a Use Case representation. Then by using Activity Diagrams and Object Constraint Language (OCL), the design is transformed into a formal model that can be partitioned onto hardware and software, according to a fixed set of hardware resources and object-oriented techniques, respectively. [12] proposes a different approach from the traditional UML one, in which the design analysis is more driven by the object identification, rather than the identification of classes. In there, the behavioral aspect of the design have been represented using State Charts.

Our design-flow (Figure 1) consists in a number of refinement steps of the problem specification until the level of detail allows gathering necessary architectural requirements. We combine the functional specification of the application with domain-based knowledge in order to provide component reuse and fast identification of resources. Since we are talking about a TTA-based processor as implementation platform, our goal is to find the basic operations the processor should be able to perform. The basic operations are implemented by dedicated functional units or combination of them, and can be expressed as parameterizable functions provided by the platform.

The design flows relies on three sources of information:

- application requirements (domain and platform independent)
- domain knowledge (domain-dependent, platform independent)
- hardware platform information (domain-dependent, platform-dependent)

We extract the functional specification of the problem using UML diagrams starting from requirement specification and we decompose it into smaller pieces

of functionality that can be easier to analyze (Requirements Analysis). These pieces, the functional objects, collaborate together in achieving the overall system functionality. We concentrate our efforts in creating a run-time view of the system expressed by the collaboration among objects and their internal functionality (Object Analysis). Objects have internal behavior which we describe in general terms, independently of the application domain and physical platform architecture. The entire process is done into a domain and platform independent manner. More details and pictures of the UML diagrams produced during the design flow can be found at [1].

Then we narrow the analysis of the functionality by restricting it to a specific application domain (Domain Mapping), i.e network processing. At this point, the description is still platform-independent, allowing the application to be implemented on various platforms.

From the analysis of the functionality in the domain space, we can extract useful information about the functionality that the TTA-processor should be able to implement and suggest its architecture from a functional point of view, using a domain and platform dependent view. With the suggested architecture we simulate the application on different architectural configurations and create estimations for each model.

When a final architecture has been suggested, the hardware design process can be started (Platform Implementation). Some heuristics can be used in order to narrow the design space exploration effort (i.e. estimations of area, power consumption and speed).

3.1 Requirements Analysis Phase

Usually, when specifying an application project, an informal description of the problem is given along with a bunch of documents describing different aspects of the system. Most of the time, the problem statement is given in a written ambiguous language that leaves room for interpretations. In order to gather all necessary information from different sources into one less ambiguous specification, one has to perform some analysis steps. We start from the problem specification and produce a list of artifacts expressed in UML. Firstly, we consider the system as a whole and extract the interaction between the system and its environment, by identifying external actors and their communication with the system (Context Diagram). Secondly, we identify pieces of functionality (along with their description) the system has to provide (Use Cases list). Thirdly, we express them as services offered to the external environment and we decide what external actors are using each of them (Use Case Diagram).

3.1.1 Requirements Specification

Specifications are given in a written form, using plain English. Requirements are pointed out in list manner along with textual details for each of them.

"The scope of the application is to design a simplified IPv6 router that routes datagrams over Ethernet networks using the Routing Information Protocol (RIPng). A router is a network device that deals with switching data from one network to another in order to reach its final destination. Two main functionalities have to be supported: forwarding and routing. Forwarding is the process of determining on what interface of the router a datagram has to be send on its way to destination, while routing is the process of building and maintaining a table (routing table) that contains information about the topology of the network. The router builds up the Routing Table by listening for specific datagrams broadcasted by the adjacent routers, in order to find out information about the topology of the network. At regular intervals, the routing table information is broadcasted to the adjacent routers to inform them about changes in topology. The IPv6 router should be able to receive IPv6 datagrams from the connected networks, to check their validity for the right addressing and fields, to interrogate the routing table for the interface(s) they should be forwarded on, and to send the datagrams on the appropriate interface. Additionally, a router should build and maintain a routing table that contains information about network topology."

More information is gathered from existing protocol specifications. The IPv6 protocol is described in [9] and [16]. For building and maintaining the routing table we use the Routing Information Protocol Next Generation (RIPng) [15] which is an UDP-based protocol. UDP (Use Datagram Protocol) is a connectionless protocol specified in [18]. Besides, an IPv6 router must also implement the ICMPv6 protocol [7] which is an internal part of the IPv6 protocol. ICMPv6 is a protocol that provides support for troubleshooting through informational and routing messages.

The IPv6 datagrams are composed of a simplified fixed size IPv6 header and a number of optional extension headers that provide improved flexibility and support for options. Moreover, IPv6 offers now extensions to support authentication, data integrity and optional data confidentiality. For the sake of simplicity we assume that IPv6 datagrams may only have a Routing Extension Header, otherwise they are formed of IPv6 header and the upper layer payload.

In this report, the term *datagram* refers to a package of data transmitted over a connectionless network. *Connectionless* means that no data connection has been established between the source and destination.

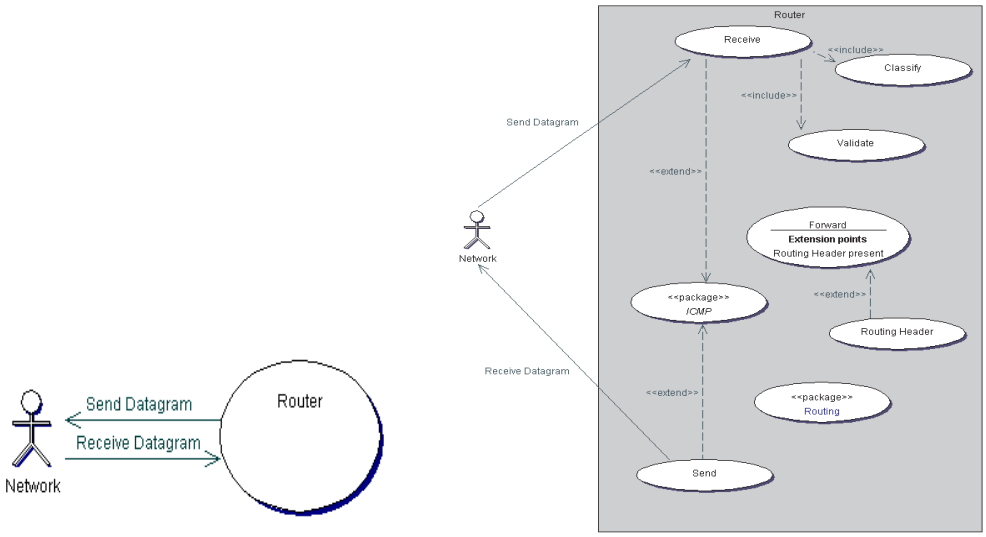


Figure 2: Context and Use Case Diagrams of the system

3.1.2 External Actors and Interactions with the system

The system has to interact with its external environment, to respond to external stimuli and to emit messages back to the environment. Based on the initial requirements we draw the context diagram (Figure 2, left) and define the boundary of the system along with its interface to the environment. We also extract the communication between the system and its environment.

The router is supposed to *receive* datagrams from the network, to *process* and to *send* them on the appropriate network. The *Network* is an actor interacting with the system. Although a router is connected to more than one network, we draw only one actor, because all connected networks have identical structure and behavior, with respect to the interaction with the router. Consequently, the interface with the network should provide means for *receiving* and *sending* of datagrams.

The above specification serves as input for the use case diagram, but it is part of an iterative process. After defining the use cases list and building the use case diagram some extra communication, not present in the Context Diagram, might be added. New details have to be updated in the Context Diagram, accordingly. After each iteration we can apply scenarios for the validation of the problem statement. At this level, scenarios are only involving the external actors and the system as a whole. This approach allows a better understanding of the messaging between the environment and the system. The iteration is stopped when no other modifications are added to the diagrams.

3.1.3 Capturing requirements into Use Cases

In order to capture the application requirements, we extract the functionality of the system and describe it using use cases. Several methods for identifying use cases have been given in literature [3, 11]. In [3] the authors consider that internal autonomous activities should be also considered as use cases (although they are not services provided to the external world), while in [11] they are not taken into account. We consider that is a good choice to specify both internal and external activities of the system as use cases, because both of them represent functionality that the system has to implement.

From the requirements specifications we identify 7 main use cases that represent services provided by the system to the external world. Use cases are accompanied by a short textual description that describes their functionality. The Use Case Diagram is given in Figure 2, right.

- Receive - the router receives a datagram from one of the connected networks.

- Validation - upon receiving the router verifies the correctness and right addressing of the datagrams. If an error is found, the IMCPv6 subprotocol is invoked to return an error message to the sender. Consequently, the datagram is discarded.

- Classification - a router has to classify received datagrams according to their scope and type. An incoming datagram can be either addressed to the routing process, to the forwarding one, or to the ICMPv6 protocol of the router. Otherwise the datagram should be discarded.

- Forward - decides on what interface a datagram addressed to the forwarding process has to be sent. In order to decide the next interface, the router interrogates its Routing Table for the next route. If no route is found, the ICMPv6 subprotocol is invoked to send an ICMPv6 Error Message to the originator of the datagram. A particular case of the forwarding process is when the datagrams bears a routing header. This requires special processing. We represent this use case as a subcase of Forward, linked by an ”extend” relationship.

- Routing - the router creates and maintains the routing table (RT) by gathering information from adjacent routers. Three main functionalities have to be implemented. They are specified as sub-use cases of the Routing use case:

  - Update - the router receives Response datagrams from adjacent routers in order to update its Routing Table information. Upon receiving, the datagram is checked for validity and correctness of the fields. If the datagram is not correct an ICMPv6 Error message is sent back to the originator.
  - Respond - the information stored in the Routing Table is send to adjacent routers to inform about changes in topological information. The information is sent periodically as multicast datagrams to all adjacent routers or as a unicast datagram as result of a special request from a single router.
  - Maintain - whenever the Routing Table of the router is empty a Request message is sent to all adjacent routers (as a multicast datagram) on each interface to request topological information. In addition, the router has to manage timers associated with each route in the routing table and to remove routes that have expired.

- ICMP - takes care of the ICMPv6 informational and error messages received or generated by the router. We also identify a number of sub-use cases:
  - Treat Info - creates response messages for the ICMPv6 informational messages received by the router.
  - Treat Error - if during the processing of a datagram an error is encountered or a forwarding route for a datagram is not found, an ICMP Error Message is sent back to the originator of the datagram.

- Send - the sends a datagram to the network. Depending on the datagram scope and type, it sends it to one or many interfaces. This use case also provides support for sending ICMPv6 messages.

The UML standard allows to define Use Case Packages, which may contain other use cases and the relationships between them. This helps us to decompose large pieces of functionality into smaller ones and to suggests the creation of subsystems of the main system. We create the Routing use case as a Use Case Package containing the Update, Respond and Maintain sub-use cases, and the ICMP Use case package containing Treat Info and Treat Error message sub-use cases.

3.1.4 Creating the Use Case Diagram

For constructing the Use Case Diagram, we first define the boundary of the system, the main actors interacting with the system and the list of external events.

This operation has as primary input the Context Diagram defined previously and the list of identified use cases.

Each use case represents one of the main functionalities of the system, as a service provided by the system to the external world. An use case can offer some subservices (subcases) and in the same time it can collaborate with other use cases to accomplish its task. A set of rules for constructing use cases and use case diagrams has been defined in [10]. The next step is to decide what actors in the Context Diagram use each use case according to the use case specification. Then we group the use cases into a Use Case Diagram (Figure 2, right) and we update the context diagram if necessary.

In the Context Diagram we have already identified the actors and the boundary of the system. By analyzing the use case list we associate the Send Datagram and Receive Datagram messages with the Send and Receive use cases, respectively. The other use cases (IMCP, Routing, Forward), although provide services to the external environment, they do not communicate directly with it, using the services provided by Send and Receive use cases.

3.2 Object Analysis Phase

After we have extracted the main pieces of functionality that the system has to accomplish, we start the process of identifying objects in the system and the way they collaborate in achieving the overall functionality of the system. First we identify the objects and then by applying use case scenarios we decide how they collaborate with the other objects in the system.

3.2.1 Identifying Objects

A couple of techniques for identifying objects have been given in literature. In [10] a class diagram is obtain from the requirements specification following and the objects in the system is obtained from class instances. Other approaches propose that the objects should be extracted from use case diagrams [12, 19] following a functional and structural decomposition. We adopt a slightly modified version of the methodology proposed in [12], where the objects are extracted directly from use cases. We consider this to be a useful approach, especially in the area of embedded systems, where we are more interested in identifying what objects implement a given piece of functionality, and less in the structural view and the object classification inside the system.

We split each use case into three objects: an interface, a data and a control object. We except the Use Case Packages from this transformation and only transform each of them into one single *subsystem* object. Subsystem objects can be

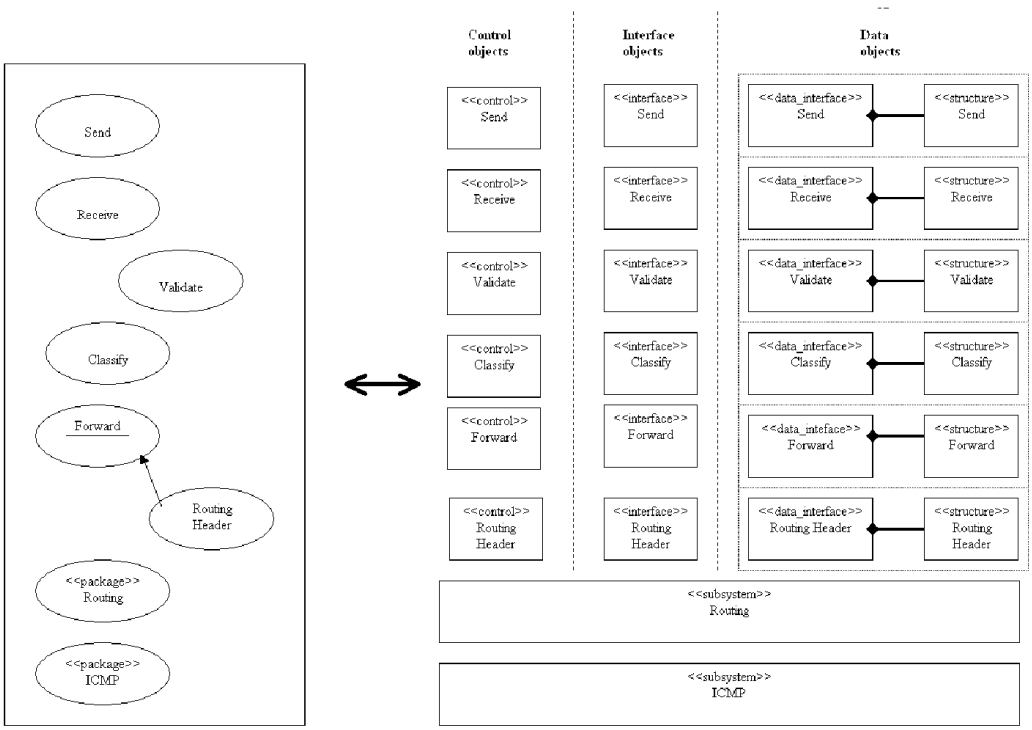


Figure 3: Splitting Use Cases into objects at system-level

---

seen as other systems (inside the main system) whose behavior is specified by sub-use cases, while external objects in the system are becoming their corresponding external actors. We apply the same algorithm recursively inside subsystems and split their use cases into 3 objects. For a better readability of the object diagram we make use of stereotyped objects to represent each type of object.

At this point we have obtained a set of objects containing three main types of functionality. *Interface objects* are used at the border of a system in order to intermediate the communication with external actors (objects). *Control objects* implement the algorithmic and the control behavior of the systems, while *data objects* take care of storing data and providing access to it. Although at this point in the design flow we are not addressing a specific implementation platform, special attention still has to be paid to how data objects are specified. We need to identify independently the interface (services) that data objects provide to the other environment and the way data is structured inside them. We express graphically this idea by splitting each data object into a data interface and a data structure object, linked by a compositions relationship.

Fist we start analyzing the entire set of objects we have created. To help us in our analysis we examine each use case and its scenarios. A scenario may involve more than one use case, from the point of view of functionality it specifies. At this level scenarios are extracted from functional requirements in a written form. By applying scenarios we are able to identify the interaction among objects and to decide which objects participate in achievement of system's functionality, and also to decide for each use case separately which object should be kept or disposed. Some of the objects could represent pieces of functionality not necessary for the system so they can be discarded.

Secondly, we analyze objects inside each vertical category (Figure 3, right), where all the objects have the same type. Some objects may have similar functionality, so they can be grouped together. In the same time, some objects could incorporate more complex behavior, and they can be split even further. Grouping objects does not changes the level of detail or the type of functionality, but offers support for reusability in the next phases of the design flow.

At this level of detail, the communication among objects is depicted as association between objects, no other details are given. The result of this process as applied to Figure 3 is shown in Figure 4.

For instance, **interface objects** are used at the border of the system to intermediate communication with external actors. Only objects directly communicating with the outer world/systems would need such interfaces, all the others are discarded. In the group of interface objects, we only keep the Send and Receive objects, that implement communication with the Network actor.

Control objects are also refactored. At this point we have 2 different data objects (Routing Table data and Forward data) that contain data with the same type and structure. Although the Forward data object is outside the Routing subsystem, we can move the object inside the subsystem and merge it with the Routing table data. The Forward control object is still placed outside the routing subsystem, but its data is situated inside the Routing subsystem. This suggest that part of or entire functionality of the Forward control object could be moved inside Routing subsystem. We split the Forward control object into two control objects, one placed inside and one placed outside the Routing Subsystem. The inside object is taking care of searching a Routing Table data for specific address, while the outside control object is used for the control flow synchronization in the system.

Similarly, we analyze **data structure objects** with respect to the type of data they contain and we notice that all the objects except the Routing subsystem store internet datagrams, so they can be grouped together into a single data structure object called Data Storage. Each network interface of the router has a local configuration, which contains network addresses and different parameters (MTU, costs, etc). The configuration is depicted by the Send and Receive data structure objects. Since the two objects are identical, we group them together and create a new data

structure object called LocalInfo. One particular case is the Forward use case, that is supposed to determine the next interface of the router by searching the Routing Table data for specific information. So, the Forward data object stores data that contains routing table entries. This means that we can join the Routing Table data object and the Forward data objects into a single object, called Routing Table data object.

Finally, we move inside each **subsystem object** and apply the same transformations. The subsystems also have the functionality expressed by use cases, and interact with external actors which are objects of the system (Figure 4). For instance, inside the Routing subsystem, we have 3 use cases Update, Respond and Maintain that create their own set of objects. As consequence, the Classify, Validate control objects and ICMP subsystem object use the services provided by the Routing subsystem, becoming its external actors.

At this step we are also able to learn the communication between objects and also to observe which objects participate in achievement of system's functionality. Communication is only depicted as association between objects, no other details are given.

3.2.2 Use Case Scenarios and Collaboration Diagrams

Each use case is represented by a one or many scenarios that together achieve its functionality. Scenarios can be applied at different levels of detail. Each scenario has it's own set of objects that collaborates for achieving the specified functionality, but objects from other use cases can participate as actors of the current scenario. Scenarios can be represented by Interaction Diagrams [6]. One particular type of Interaction Diagrams are the Collaboration Diagrams, which emphasize how the objects interact. Each scenario produces its own Collaboration Diagram (drawn from the initial specification) and involves a limited set of objects in the Object Diagram.

An object collaboration diagram specifies how the objects communicate by showing the information passed among them. This helps us out in determining what operations each object should provide to its neighbors. UML allows the description of the messages sent among the objects and their ordering in time. At this level we express the communication among objects as request of services (operations) from other objects. We annotate each message to show its precedence in time using a Dewey decimal numbering scheme. The messages have a name and can carry a list of parameters. Defining communication is an incremental process that ends up when all the scenarios of a given use case have been instantiated. Scenarios are applied also inside the subsystems recursively.

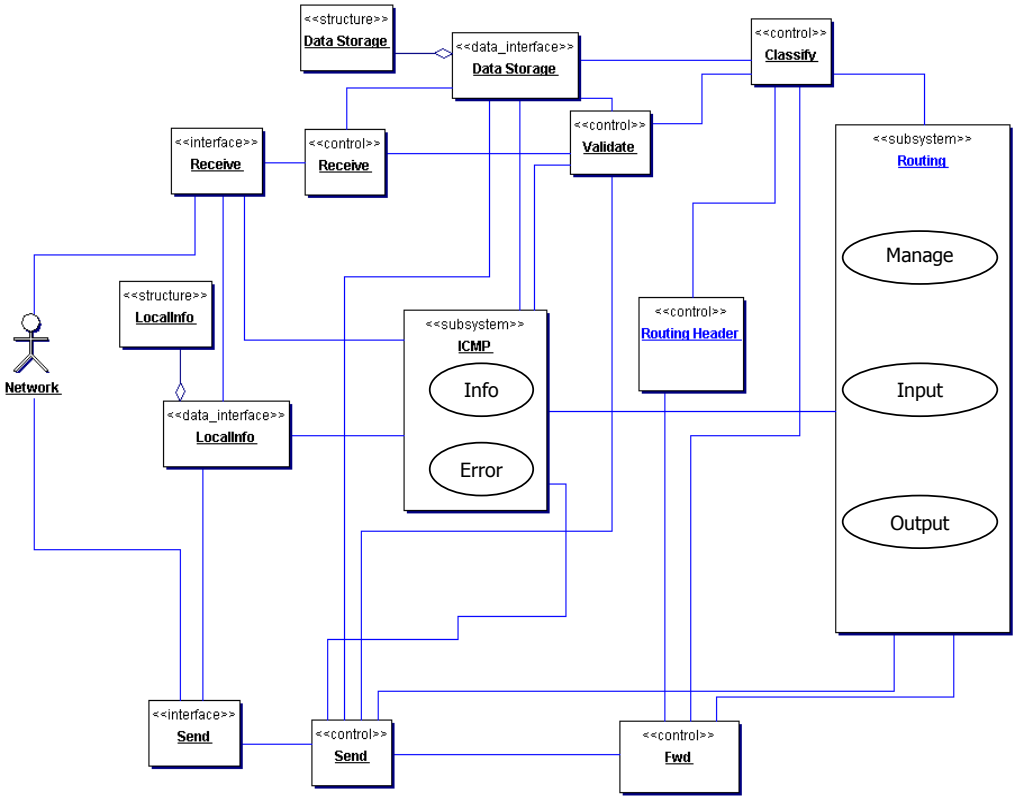


Figure 4: Final Object Diagram

3.2.3 Building System-Level Object Collaboration Diagram

Until now the communication was defined only from the point of view of the control flow among the objects. Scenarios represented by Collaboration Diagrams have been applied on different sets of objects and the communication between objects has been expressed as service requests.

UML specification defines the collaboration diagrams as corresponding to an instantiation of an use case by a scenario, but allows joining different collaboration diagrams originated from different scenarios into one collaboration diagram. We join all Collaboration Diagrams created earlier into a single one, that shows the object collaboration for the entire system. This is done by copying at least one instance of the objects along with their requests from each collaboration diagrams into the final diagram. In Figure 5 we present part of the system-level collaboration diagram obtained from grouping the Routing and Forward use cases.

In order to refine the communication inside the system-level object collaboration diagram we transform the requests messages into an event based communi-

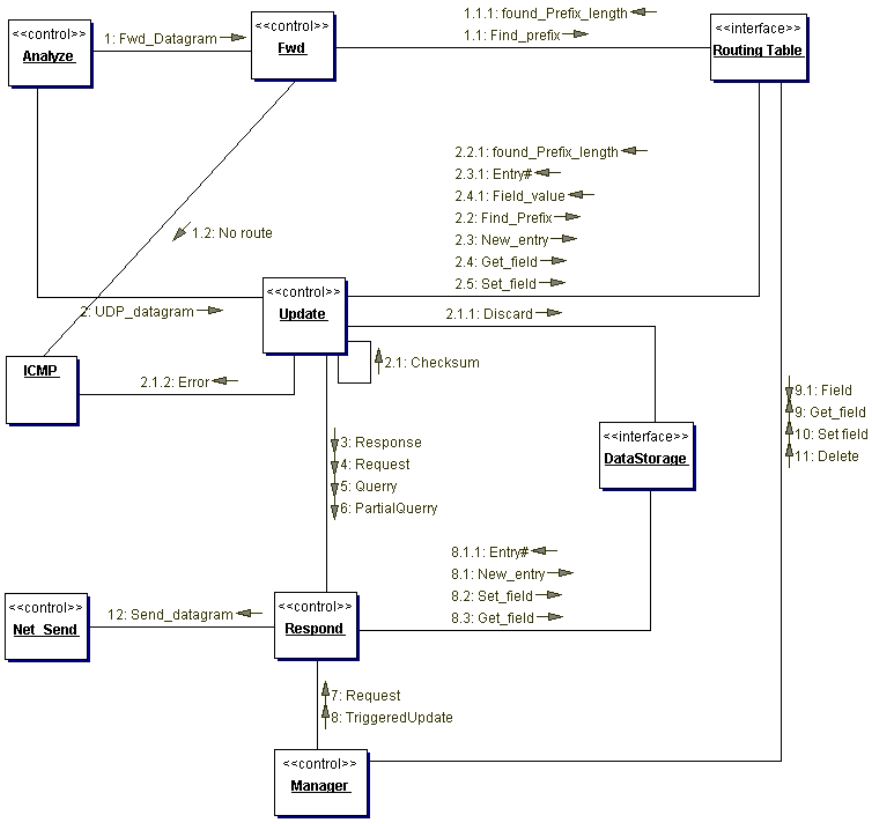


Figure 5: Joint collaboration diagram for Routing and Forward Use Cases

---

cation. Two of the event types that UML standard specifies are: signals and calls. Signals are events sent asynchronously from one object to another. Calls are usually synchronous events that represent dispatching of an object operation. Both types of events can have a name and can carry a list of parameters. We choose to have a call event whenever an object sends out a message, the control is passed to the receiver, the receiver changes its state and returns the control to the sender. If a message transfers the control from a sender object to a receiver object without returning the control, then we consider it to be a signal. Dividing events into signals and calls does not reduce the generality of the design, because a call event can be seen as two signal messages, one that invokes an operation of another object and one that returns control from the invoked object. This is possible because UML specifies input event queues for each object in the design, where the events are interpreted in a FIFO way.

Dividing events into signals and calls helps us to identify which are the objects

that provide the invoked operations and to identify the signature of the operation. The signature is given by the name of the operation, number and type of parameters, and the return value (if any) of the message.

3.2.4 Behavioral Analysis

In the previous subsection we have decomposed the system into control, data and interface objects. Control objects are the ones that coordinates the flow of control in the system, execute some algorithms or control the flow of information between parts of the system. The data structure objects, represents storage places that can be accessed in specific ways in order to read/write data. Objects are seen as functional modules of the system, that provide an interface to the environment and have internal behavior.

The internal behavior of the system resides mostly in the control objects which coordinate the control flow of the system. Since we are addressing the area of protocol processing, which require data-intensive and algorithmic control-flow, we choose the Activity Diagrams to implement the behavior of control objects. An activity diagram (Figure 6) is a set of actions implementing atomic computations that result in a change of the system's state. Each activity is triggered by a transition and executes until completion. UML does not prescribe a specific language for describing activities, allowing by this a great deal of flexibility for designer. Another important feature of Activity diagrams is the possibility of hierarchically decomposing the activities into sub-activities (sub-Activity Diagrams). This allows the designer to refine the initial Activity Diagram into a more detailed specification.

There exists two types of states inside a diagram: action states and activity states. Action states are executable atomic operations that cannot be further decomposed. Activity states are states that can be decomposed (expanded) into another activity diagram containing both action states and activity states, or action states only. More details on activity diagrams can be found in [6] and [10]. Additionally, activity diagrams provide mechanisms for sending and receiving events and also for synchronization of control flows (join and fork).

In our design, we build an activity diagram for each method of a class, in which all the activities are activity states in the beginning. Each activity is described in a natural language. Activity diagrams will have one or many input/output points represented either by the send/receive event states or by the start/stop states of the diagram. These points should correspond to the communication path already existent in the object diagram and represent an invocation of an operation of another object or sending a message to the environment. Activity states are then decomposed into other activity diagrams which contain a smaller parts of the system's functionality but at a higher level of detail. This process can take many iterations

until transforming all the activity states into action states.

During the refinement of the activity diagrams, since we describe a more and more detailed view of the system, we need to be able to refer to data involved in the system and some operations for it. The process of identifying data entities and their structures has as main input the requirement specifications and data structure objects in the system. The data should be described with respect to the functionality it provides, no concerns about the physical implementation being taken into account. We are interested in what a specific data field means and what size it has, but not in how it is physically represented and stored inside data objects. Because the description of the action states does not require a strict language, the operations applied on data can be defined without a strict syntax, but attention should be paid, so that their meaning does not overlap. In fact, a Data Dictionary of the system is strongly recommended. After many refinements of the activity states, the final version of each activity diagram will contain activity states only. At this point we have obtained an executable specification of the router, since all the action states are similar with the instructions of programming languages. Each action state represents an operation that is applied to one or many operands (data structure fields). Next to the operation we annotate the data size of the operands, since we are talking about generic operations that can be applied to different data sizes.

Figure 6 describes the activity diagram of the RH (Routing Header) object of Figure 4. We have used a new notation for the conditional activities in order to provide a better readability. A conditional execution is represented as test preceded by an activity state that evaluates a condition. During the refinement steps from initial activity diagram to the action states functional data types have been specified. They do not have any concrete type for the moment, they are only accompanied by the functional requirement of their size. We use variables for temporary storage of functional data values, but without any type or size constraints. The communication between objects is described by a message passing protocol where parameters can be passed to the adjacent objects through event states.

The level of detail in an activity diagram is given by the granularity of its action states. The way activity diagrams are defined helps in breaking complex computation in less complex and easy to handle parts. Due to the hierarchical representation of activities, one can expand or collapse the activities on different levels of detail and reuse the action states to implement many different activities.

3.2.5 Creating the class diagram

In the system-level collaboration diagram we have specified a set of objects that communicate at service-level by events, sent from one source to one destination. Our goal is to identify a set of classes from which these objects were instantiated

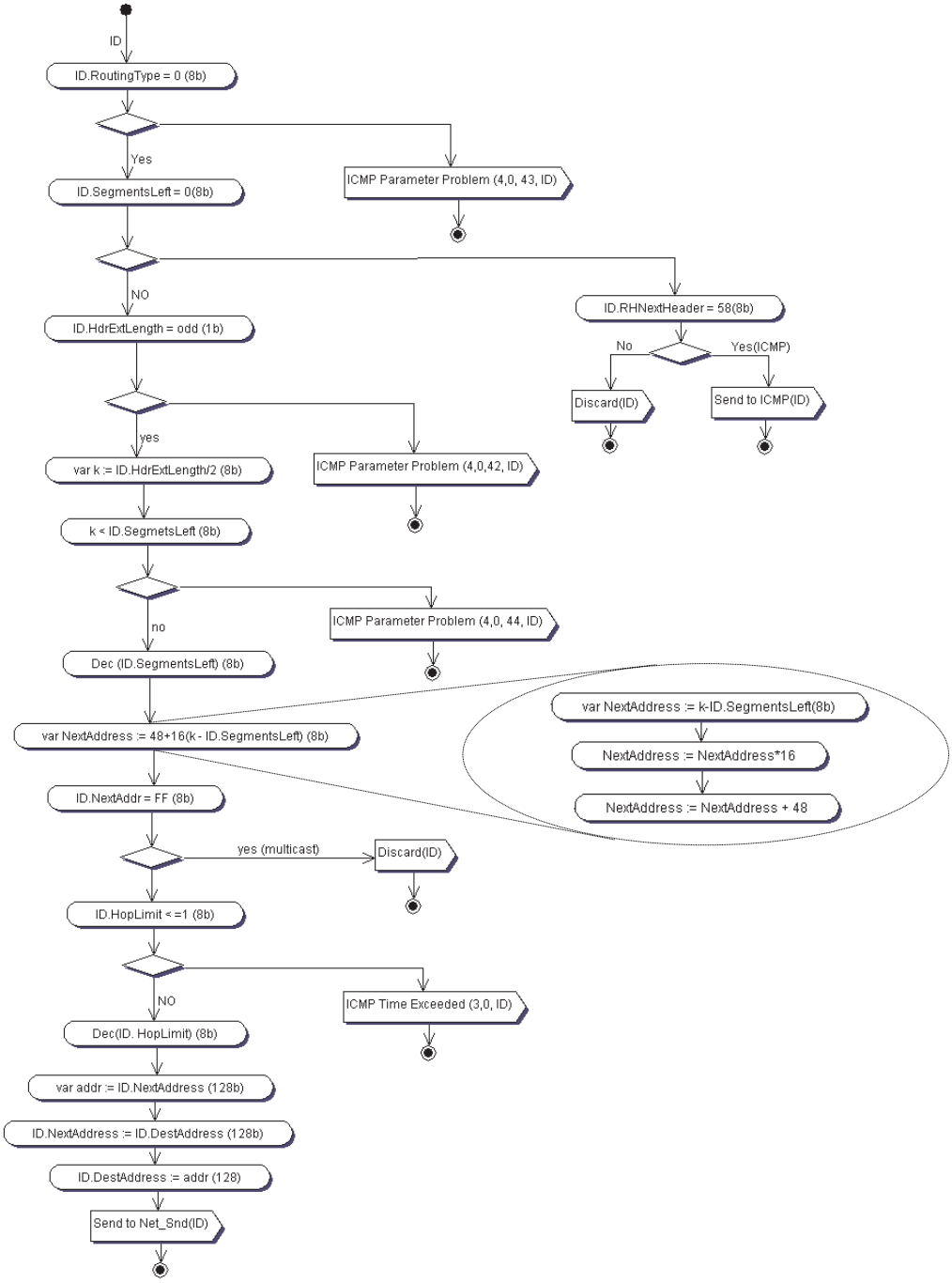


Figure 6: Activity Diagram of the RH Object

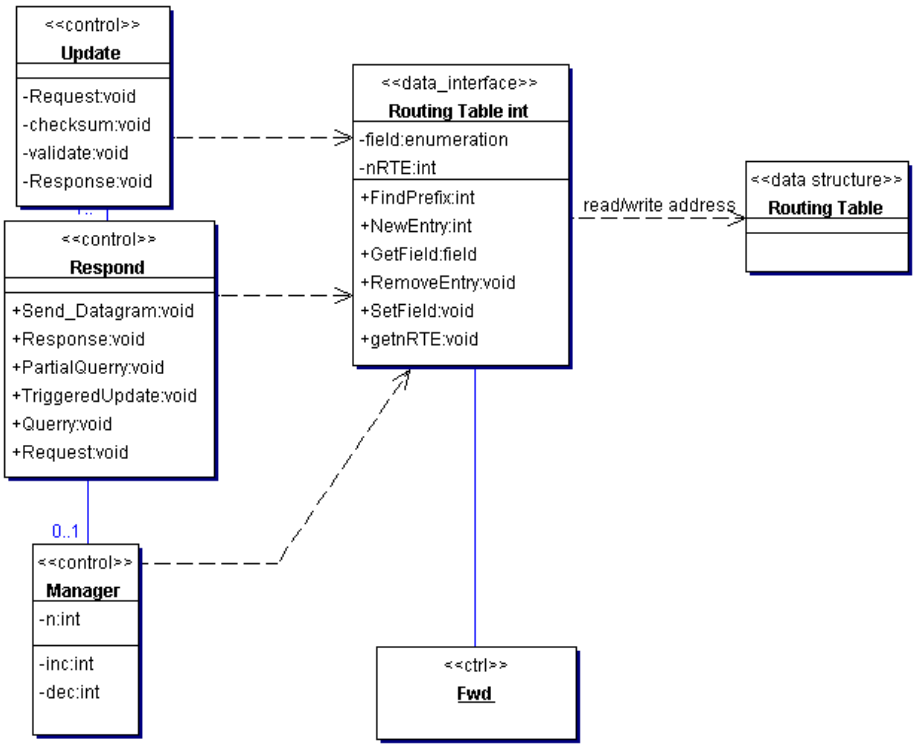


Figure 7: Class Diagram of the Routing Table Subsystem

---

and to specify the functionality they have to provide to the environment through their operations.

Since objects represent instantiations of classes and call events represent dispatch of objects operations, we can identify the classes in the system and some of their methods. Each object in the collaboration diagram is transformed into a class and each call event into a method of the class. The method should be placed in the class that is the receiver of that call event.

Attributes represent states of classes and are usually visible to external objects through methods only. Inside a class attributes can be shared by many methods. One should note that there might be hidden methods and attributes of the classes that are not directly identifiable from the collaboration diagram, but they can be obtained by an individual analysis of each class' functionality.

In Figure 7 we show the identified operations of the Routing Table interface object and the way they are accessed by the neighboring classes. The Routing Table interface class provides a set of operations through which other classes can access data in the Routing Table data class. For the moment we are only interested

of the Routing Table data interface, and not of the way it is implemented. Interface classes contains no control (state machines) only methods for implementing the interface.

3.3 Domain Analysis phase

It has been generally admitted that there is a commonality between different applications in the same field, and several approaches have been focusing on reuse-oriented domain analysis (e.g, [5, 13]). By analyzing and modelling the domain knowledge a set of generic reusable components can be extracted by identifying general abstractions and similarities of a set of applications. For a given range of applications, i.e protocol processing, we claim that there is a common set of basic operations that need to be supported/implemented by a system.

Designers can obtain the specific set of basic operations through incremental domain analysis based on previous designs and experience. One can easily create a list of basic operations that should be available during the design of new systems. This list includes functional data types and a set of basic operations available for them. Data types will be later refined into implementation specific data (platform-dependent), while the domain operations will be mapped onto operations provided by the hardware platform. The domain operations are specified by the function they accomplish, the number of parameters they require and width of the generic data-types they use. A classification of the operations in the list can group them into arithmetical (addition, multiplication), logical (and, or, not), test (match, compare) or memory access (i.e. get, put, etc..) operations.

We use the Domain Operations list as an interface between the UML specification of the application and the TACO processing platform, to help in translating the specification into resources provided by the processor. On the one hand we try to express the UML specification (Activity Diagrams) using available domain operations, on the other hand the domain operations are supported by given FUs or combination of them. For this we start refining the Activity Diagrams to contain only action states expressed by operations from the domain list, with respect to the number of parameters and functionality each operation provides.

There might be cases when a direct mapping is not possible or when the designer may identify new operations that are useful in the targeted problem domain. As a consequence, new domain operations have to be created. These can be included and documented in the operations list. In the same time, adding hardware support for new domain operations implies in our case either modifying an existing TACO FU or creating a new one, and the operation updated in the domain operation list. For instance, from the process of mapping Activity states onto domain operations the necessity of implementing a new *shift* operation or a new *set* operation (of selective bits in a bit-string according to some mask) have arisen.

Consequently, two new functional units have been designed: Shifter Unit and Set Unit, respectively.

3.3.1 Identifying Domain Operations

The TACO architecture might have many types of resources created in previous application implementations, similarly to a component library. Each specified functional unit can perform one or many operations according to its specification. The set of operation that TACO implements can be seen as a list of domain operations, which is defined as the union of all operations identified in previous configurations (implementations) for a given application domain. In our case we refer to the domain of network (protocol) processing applications. Different lists are created for different domains of applications.

Moreover, there can be more than one TACO processor architecture type available, depending on the bus width. Since the FUs are interfaced to buses by input-output registers, it means that for a given bus width the input output registers should have the same size. TACO processor does not support more than one size of buses in the same time, so the configurations has to be done with respect to a particular architecture (bus size). Each architecture has its own list of resources (FUs, buses, sockets, interconnection controller) and creates a separate list of domain operations.

3.3.2 Mapping domain operations onto hardware platform

Since TACO processor resources are simulated in SystemC using object-oriented mechanisms, we represent available resources by building a class diagram of TACO (upper part of Figure 8). Here we have three types of resources: buses, sockets, and modules. Additionally a Socket Manager class manages the connection of sockets to the functional units and a NetControl class implements the communication on the buses. All these classes are inherited from the SC_Module which specifies the basic module description class in SystemC. Configuring TACO for a specific architecture means firstly to select the bus width. The sockets and registers will be automatically set to use the same size.

The functionality supported by TACO resides in its functional units behavior. Busses are only used to support data transports between functional units and implicitly to trigger FU operations. Each FU has a set of input and output registers that represent its interface with the interconnection network. We can consider each FU as a black box that receives an input, performs some computation and outputs some result. In addition, a FU implements one or many operations of the processor. For instance, the COUNTER FU was designed to perform addition,

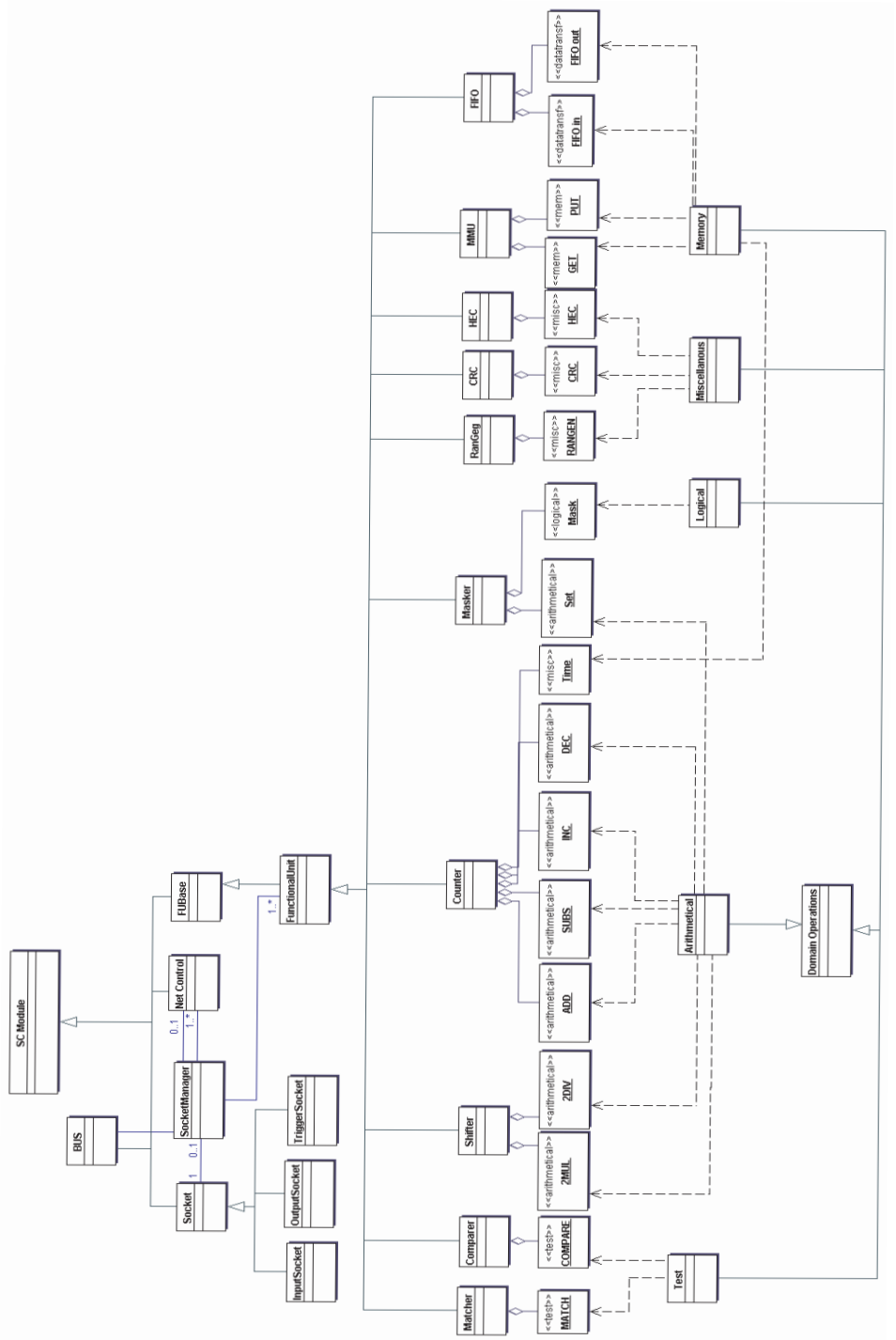


Figure 8: Class Diagram of the Routing Table Subsystem

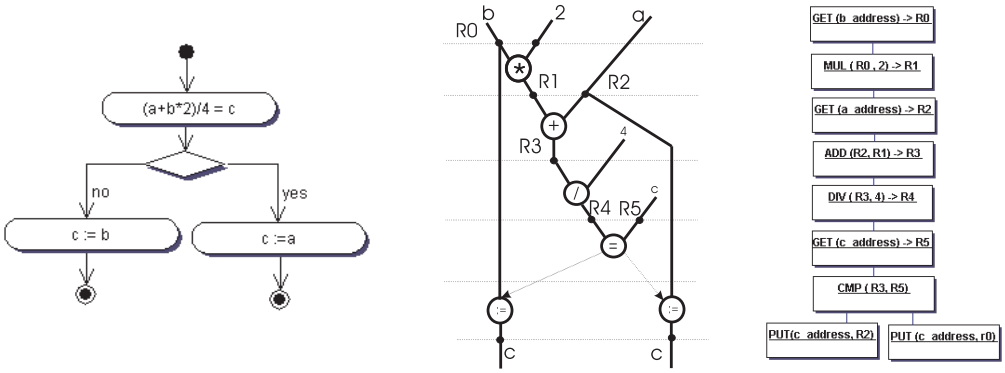


Figure 9: Mapping Activities onto Domain Operations

substraction, incrementation, decrementation, and timing (counting bus cycles) operations.

For executing a TACO operation, one or many bus transports are needed, because a functional unit usually has to be set up with input values, triggered and then waited for the result. This means that a domain operation is equivalent with a number of TACO *move* intructions between the FUs of the processor. One should notice that for a specific operation that a FU performs, every time the unit is triggered the same sequence of moves is used. Thus we can abstract the domain operation as a macro containing TACO bus transports having as parameters the source and destination of the involved FUs.

As consequence, every time a domain operation is invoked it can be automatically translated into TACO bus transports. In the domain operations list, each domain operation is assigned a TACO operation along with its list of generic parameters. One should note that a domain operation in the list can be implemented by one FUs or by a combination of FUs.

3.3.3 Mapping UML specification onto Domain Operations

Until now the design of the IPv6 router has been completely domain independent, no domain or platform related design decision being taken. The next step in our design flow is to try to match the states in the activity diagrams obtained in the previous phase, to the operations available in the operations list. This mapping operation is illustrated in Figure 9. Here a simple arithmetic expression is translated into a sequence of domain operations.

In order to make simpler the process of refining the UML specification using domain operations, we make a classification of domain operations based on their functionality (arithmetical, logical, memory, test, etc.). We create a hierarchy

of this operations and construct a class diagram in which stereotypes are used to classify each operation. The domain operations class diagram can be easily connected with the TACO SystemC class diagram (lower part of Figure 8) by associations, in order to obtain a more formalized model of the domain operations list.

The result of the domain mapping transformation is now domain dependent because the atomic operations in the activity diagrams are operations from a problem domain, i.e. protocol processing. The model is still architecture independent, since we have not decided on how the domain operations are implemented on a concrete TACO processor. As we have mentioned earlier in this report, one domain operation can be implementable by more than one FU. By selecting a given domain operation to be used in the application specification doesn't always imply that a certain FU will be used for implementation, but only that the domain operation at hand is "implementable".

3.4 Platform implementation

3.4.1 Qualitative configuration

Once the entire specification has been mapped onto domain operations, we browse through the application specification and select the functional unit types that implement required domain operations. We mention once again that we only identify FU types necessary to implement a given application, and not the number of instances of each FU type in the processor. When the entire application is mappable onto physical resources, we can configure the processor with one resource of each type and run the functional verification of the application.

3.4.2 Suggesting FU optimizations

Although we have refined the activity diagrams to contain only domain operations, they are still expressed in terms of activity states. Now, we look for eventual sequences of operations that require heavy computation, in order to optimize them.

By their definition, activity diagrams represent the control flow of a system. The states succeed in sequential order, given by the transitions among them. We can group successive states into *blocks of control*. A block of control starts when a *start* state is encountered or the precedent block ended, and ends up when a *stop* state is encountered or when the control flow splits. Control blocks and the transitions among them form a control graph in which the control blocks are represented by arcs, while the nodes are represented by the states that split the control flow.

Moreover, each activity diagram can be seen at run-time as a tree, where the nodes are branch states and arcs contains blocks of control. The leaves of the tree represent exit points of the diagrams. A flow of control starts in the root of the tree and exits on one or many leaves. On the way it passes through a number of decision (branch) states and the control blocks in-between them. In order to get a good performance of the control flow for each activity diagram we try to optimize the control blocks so that we have a similar (balanced) complexity for the control blocks on each level of the tree.

Since domain operations are implemented by already existent functional units, we have information available about how many processor cycles each domain operation takes and consequently the cost of each arc in the tree. For computation of a domain operations two timings have to be taken into account, based on the characteristics of the FU that implements them. One is the *setup time* of the FU's registers with input and the other is the *computation time* of the functional unit until the result of computation becomes available in the result register. In TACO processors, each bus transport is done in one bus cycle. If more then one bus is present, then the transports can be done in parallel, hence reducing the setup time of the FUs. The computation time is not dependent of the number of busses and cannot be scaled down by adding more busses to the processor. In the domain list we also annotate the number of cycles each operation needs to complete on a given FU (Figure 10). This allows us to estimate the number of cycles that different threads of control might take during the execution.

Grouping action states originating from different activities can suggest a new way of implementing the same functionality. For instance two successive divisions of the same operand by 2 can be grouped into one single operation of division by 4 or in a logical shifting of the operand. This might have a good effect in the final implementation because it reduces the number of cycles required by the application. We try to identify sequences of operations that:

1. are part of loops in the control flow, that would require long computational time at run-time

2. are identical (as functionality) in different activity diagrams or inside the same one

In the first case we analyze large sequences of successive states that take many processor cycles in the control flow of the system, especially when they are part of loops with large number of iterations. At run-time, loops will become long repeating sequences of operations. We identify these sequences and mark them as candidates for optimization. The identified sequences are in fact new activity diagrams (which contain only a sequential control flow) and become candidates for

W1 – 16 bit	W2 – 16 bit

Domain Operation	FU	SetUp Time	Computation Time
SHIFTR (W, 16, R1) *//shift W right to obtain W1*	SHIFTER	1c	1c
MASK (W, 0xFFFF 0000, 0, R2) *//set to 0 left part of W*	MASKER	2c	1c
ADD (R1, R2, R3) *//compute 2's complement sum*	COUNTER	1c	1c
SHIFTR (R3, 16, R4) *//get carry in R4*	SHIFTER	1c	1c
ADD (R3, R4, R5) *//compute 1's complement sum* *// by adding carry*	COUNTER	1c	1c
SUB (0xFFFF FFFF, R5, R6) *//complement the sum*	COUNTER	1c	1c
	Total time 1 bus: Total time 3 buses:	14 cycles 5 cycles	

Figure 10: Checksum optimization

---

further analysis. Identifying repeating sequences of operations require in general cases large algorithmic complexity, and we do not state that we can apply it for any problem.

In the second case, during the analysis of all the activity diagrams in the system, one can notice sequences of states that repeat (without being a loop) in many activity diagrams or even in the same diagram. These sequences indicate a special functionality (pattern) that the platform should be able to handle as efficient as possible. Usually we identify this type of sequences inside different activity diagrams. We also mark found sequences as candidates for further analysis.

All the candidate sequences have common characteristics: they are composed of a sequence of states (no flow branching inside) and represent pieces of functionality frequently accessed by the application. One can notice that adjacent action states can originate from different initial activities. The definition of activity diagrams allows regrouping them into new activity states that can contain domain operations from more than one adjacent activities. Grouping is done with respect to the functionality the states implement and its highly dependent of designer's

experience and knowledge about the implementation platform. Newly formed activities suggest larger pieces of functionality that might need special attention. Regrouping the action states into activity states and collapsing them is similar with creating a new function in software or building a new specialized module in hardware.

For example, by analyzing the entire design we have identified the Checksum method of the Input Controller class and Find method of Routing Table Interface Class as candidates for optimization, belonging to the second category of optimizations. The IPv6 checksum calculation is computed as ”16-bit one’s complement of one’s complement sum” and requires repeated number of additions and 1’s complement operations. For instance, if we refer to TACO32 architecture, we would use 32-bit buses and registers. To compute the checksum of a 32-bit word, we have to split it in two 16-bit words, to compute one’s complement sum of the two numbers and complement the result. This will require 14 processor cycles (Figure 10) using one bus and 5 cycles using 3 buses. We have suggested and implemented a new functional unit that is able to compute the checksum of a 32 bit word in one cycle. Designing the Checksum FU and suggestion of its architecture was left entirely at hardware designer’s attitude. Moreover, the newly suggested functional unit was configured with 3 input and one output registers. This will allow computing 3 32-bit words in one cycle (when using 3 32-bit buses). There should be a trade off between the costs of designing and implementing the resource, and the increase in performance that it provides.

Another important optimization is searching of an entry inside the routing table. Although Find is a method of the Routing Table class, it is also specified as an activity diagram containing domain operations. Its functionality is to search the Routing Table entries for the right prefix and represent the most critical task of a router with great impact on performance. The sequence of operations that Find operation creates becomes also candidate for optimization.

3.4.3 Quantitative configuration - Domain space exploration

We have now identified, then modified or created all the necessary resources for the TACO processor to implement the requested functionality of the application. In order to obtain an optimally performing architecture with respect to some constraints we have to explore the design space by proposing different architectural configurations. This involves physical performance estimation and the approach is presented in [20]. As a result of this optimization stage new functional units may be proposed. These are incorporated into the TACO processor library, and the operations the FUs implement have to be included in the domain operations list.

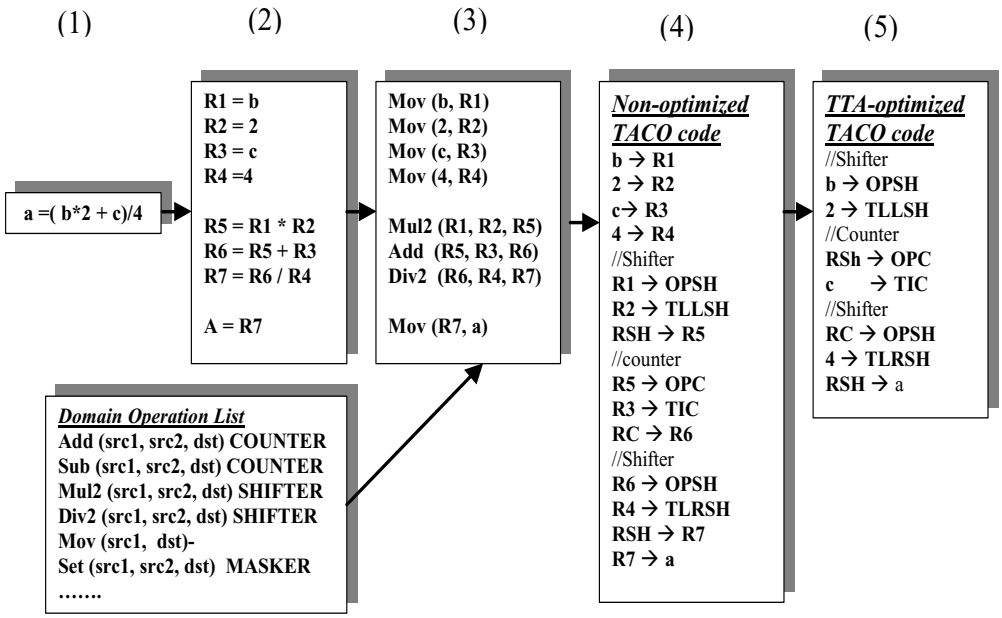


Figure 11: Mapping Process

Exploring all possible configurations in order to find the optimal solution requires huge computation effort. But analyzing the qualitative configuration some preliminary information can be extracted, for instance bottlenecks in processing and area and power consumption information. Doing design space exploration for the TACO processor means estimating physical performance of the architecture by running simulations of different configurations. The configurations are obtained by multiplying the number of resources of each type according to some heuristics. For each configuration, we transform the actions in Activity Diagrams into plain TACO code. The transformation process is straight forward due to the nature of activity diagrams. The activity states are in fact sets of TACO instructions (bus transports), and transitions between them can be modelled by relative or absolute JUMP operations (if needed).

3.4.4 Generating the application code

The application is implemented at this point by a sequence of domain operations, which are mappable onto parameterized macros of TACO instructions. In order to get TACO executable code of a given configuration we apply a set of transformations of the domain operations inside each activity diagram. A short example is presented in Figure 11.

1. we assign generic registers to the operands in the activity states. Generic registers are only used for temporary storing the values of operands, they do not represent physical resources.

2. we decompose the activity states into basic operations, implementable through domain operations.

3. each activity state is mapped onto a domain operation, and generic registers are assigned to domain operation parameters.

4. domain operations are expanded into macros of move operations between TACO functional units (sockets).

5. the resulted TACO assembler code is optimized with respect to TTA architecture. The optimizations include: moving operands from an output register to an input one can be done without additional temporary storage (bypassing), use of the same output register or general purpose register for multiple data transports (operand sharing), removing registers not in use anymore, etc. All these techniques help in speeding up the execution time and reducing the code size of the application, as well as decreasing the number of general registers needed for implementation.

In addition, general compiler optimizations can also be performed on the TTA optimized assembler code: sinking, loop unrolling, etc.

For implementing communication among the activity diagrams of the different functional objects in the specification, we transform each event into jump operations. For passing parameters we store their values in generic registers. TACO architecture specifies two types of jump instructions: relative and absolute jumps. More details in [22].

The entire design reduces now to bus scheduling and registry allocation problems, where we want to allocate generic registers to physical resources. For this, we have to schedule the moves instructions on the bus(es) and to allocate registers to the operands of the instructions. The scheduling and allocation policies have been widely discussed in the literature and we are not suggesting here any method. The compiler does the necessary allocation and scheduling, along with some final optimizations.

3.5 A TACO qualitative configuration for IPv6 Routing

Following we present the qualitative configuration of the IPv6 router developed during the design flow. The Router is configured with a number of functional units that accomplish different computational tasks, like i/o processing, arithmetical

and logical computations and units that provide data access (Figure 12). The implementation details of the TACO IPv6 router are given in [22].

The TACO processor is used as stand-alone processor that offers support for the IPv6 layer. The processor has to be be interfaced with the network interface through buffers. Each interface has an input buffer and an output buffer, and they can receive or send datagrams independently. When a new datagram is received on one of the input buffers, it is saved in the main memory and the processor starts processing it. When a datagram is to be sent, it is taken from the main memory and saved into the output buffer of the corresponding interface. Two new functional units have been designed for managing the input and output traffic from the Network.

The Preprocessing Unit (iPPU) scans the input buffers for new datagrams. If a datagram is pending it is stored in the main memory. A pointer to the memory address where the datagram was stored is saved in a queue, along with the interface identifier of the input buffer. The iPPU is connected to the buses in the processor's interconnection network and also provides a 1-bit signal connected to the Interconnection Network Controller to notify it if entries are pending in the queue. The PostProcessing Unit (oPPU) manages the output traffic of the router. The unit contains an internal queue in which pointers to memory addresses of the datagrams to be sent are stored along with the output interface identifer. The oPPU interrogates its internal queue and for each entry it moves the corresponding datagram from the data memory to the specified output buffer.

A number of arithmetical and logical unit support the processing of the datagrams. The Counter Unit performs arithmetical operations (increment, decrement, addition, substraction) and counting (upwards or downwards from a start value to a stop value). When the stop value has been reached a result signal directly connected to the Network Controller is enabled. For comparing operands with a given value a Comparer Unit has been designed. The result of a comparison is signaled to the Network Controller via a result signal. The Matcher and the Masker are bit-string manipulation FUs that process only parts of their input operands according to a given mask. The Matcher reports its result to the Interconnection Network Controller by means of a result bit signal directly connected between them. The Masker sets the bits of a register according to a given mask and a given value. In addition to logical shifting, a Shifter can be used for arithmetical multiplication by 2. The router also has a number of FU that provide fast access to data storages (Routing Table and Local Info) and a Memory Management Unit to read and write data in the memory.

One of the complex operations that an IPv6 router has to implement is the checksum calculation, because it involves heavy data transfers. From the analysis process we have concluded that a special functional unit is necessary to improve the performance of the application. It computes the checksum of a datagram by

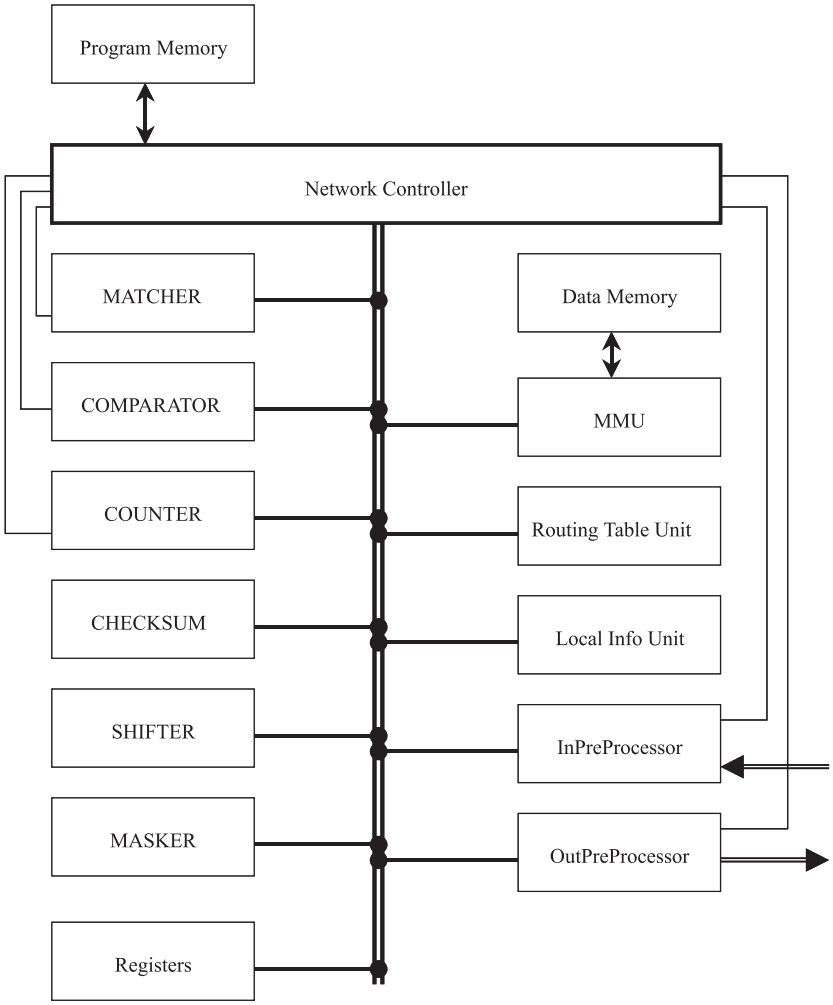


Figure 12: IPv6 Router Instance of TACO processor

successively feeding in small chunks of the datagram, thus reducing the number of data transports on the busses and the number of memory accesses.

An important design feature of a TTA processor is the modularity of the architecture. Each FU computes independently of the interconnection network and other units. So the performance of the processor is reflected by the number of transports on the busses and implicitly by the time in which each operand becomes available in the output registers of the functional units. In order to decrease the waiting time, FUs have to complete their computation in as few cycles as possible. In these sense a balance should exist between the amount of complexity and the response time of functional units. The functional units of TACO have been

designed so that they complete their computation in one cycle. One should note that one cycle is the same time a transport is done on the bus.

4 Traceability issues

Traceability is an important issue in designing complex systems. It helps to determine how a specific part of a system has been created and for what functionality has been defined. Traceability also helps in debugging and reverse engineer complex systems.

During the system specification, we have started from the requirements specifications, than we have determined use cases, identified the objects and then refined them into the final specification. During the identification of the objects, we have repeatedly split and grouped them into smaller or larger pieces of functionality. If at some point we have to trace back the functionality of an object or to add extra functionality, we need a mechanism that shows to what initial pieces of functionality each architectural element belongs. The UML standard defines the *invisible hyperlink* notation, as a suggested notational element for UML tools, but does not suggest how it should be implemented by tools. Hyperlinking facilitates browsing through the elements of a project (objects, classes, diagrams, etc) in order to determine how they relate to each other. The linkage between architectural elements has to be done manually by the designer at specification-time.

We use hyperlinks during different phases of design in order to have a history of each design step and decision. We start from the use case diagram. After each decomposition of an use case into three objects is done, we hyperlink the obtained objects to the initial use case. The Hyperlink is a unidirectional link, so in order to provide back-and-forth navigability through the design we draw also a hyperlink from the use case to each object it creates. During objects identification phase, objects can be joined or split according to their functionality. If two objects are grouped, the new defined object will contain all the hyperlinks from the initial objects. If an object is split, each newly created object will contain all the hyperlinks of the initial object. After identifying objects and defining the communication among them, a class diagram is created from the collaboration diagram. Each object represent an instantiation of a class. When we define classes for the objects we also create hyperlinks from object to class and from class to the corresponding object. Inside classes the functionality of the system is specified by methods expressed by activity diagrams. Activity diagrams are composed of activities that in the end are transformed into domain operations. Since the entire design is expressed hierarchically, hyperlinks can be used at each specification step to show the origins of different artifacts.

The design-flow we are using has a top-down structure, starting from a high-

level specification and ending with a detailed one. Each phase uses diagrams created in previous phases, and creates a new set of artifacts. Using hyperlinks, one can easily trace in what design step an artifact was created, from what diagrams was it derived and by what diagrams it is used in the following design steps.

5 Conclusions

We have presented an UML-based design methodology for protocol processing applications that has as result suggesting and configuring the architecture of TTA-based processors, in order to serve a given application. The application is developed by configuring the hardware and creating the software in the same time.

The approach allows an early design space exploration of TTA-processors configuration for a given application domain. Simulations and estimations of the configuration can be obtained at system-level before implementing the hardware components, by using the SystemC and Matlab model, respectively.

The methodology can be applied for different processor architecture types, by identifying their domain operation list. The fact that the application specification is completely independent of the platform helps in exploring many architectural choices. In the same time, by creating the domain operation list, the analyst does not have do have good knowledge about the hardware resources he/she is targeting the application on.

Reuse is also addresed in this methodology. IP-reuse is becoming more and more an important requirement for designing embedded systems. It reduces the cost of hardware design and implementation processes and shortens the overall development time of the system. From previous applications, TACO provides a set of ready to synthesize resources that can be simulated and estimated with respect to the application speed and hardware platform power consumption and silicon area used.

Future work should focus on giving a formalized description of the design flow, allowing functional verification of the specification in early stages. Due the similarity of the Operation Diagrams and TACO executable code, a method for suggesting an optimal architecture configuration for a given application should be designed. A automated tool might also prove to be useful in implementing automated transformations between different steps of the design flow.

References

[1] http://www.abo.fi/~dragos.truscan/ipv6/rd/diags.html.

[2] C. Arpnikanondt and V. K. Madisetti. Constraint-Based Codesign (CBC) of Embedded Systems: The UML Approach. Technical Report YES-TR-99-01, Georgia Tech, 1999.

[3] M. Awad, J. Kuusela, and J. Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach using OMT and Fusion*. Prentice Hall, 1996.

[4] D. Björklund, J. Lilius, and I. Porres. Towards efficient code synthesis from statecharts. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Workshop of the pUML-Group held together with the UML2001 conference*, Lecture Notes in Informatics, pages 29–41. GI, oct 2001.

[5] D. Bjørner. Software Systems Engineering, From Domain Analysis via Requirements Capture to Software Architecture. In *Proceddings of Asia Pacific Software Engineering Conference(APSEC'95)*, 1995.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley Longman, 1999.

[7] A. Conta and S. Deering. Internet Control Message Protocol (ICMPv6) for Internet Protocol Version 6(IPv6) Specification. *RFC 2463*, December 1998.

[8] H. Corporaal. *Microprocessor Architectures - from VLIW to TTA*. John Wiley and Sons Ltd., Chichester, West Sussex, England, 1998.

[9] S. Deering and R. Hinden. Internet Protocol, Version 6(IPv6) Specification. *RFC 2460*, December 1998.

[10] B. P. Douglass. *Doing Hard Time*. Addison-Wesley, 1999.

[11] B. P. Douglass. Ropes: Rapid object-oriented process for embedded systems. White-Paper, 1999.

[12] J. M. Fernandes, R. J. Machado, and H. D. Santos. Modelling Industrial Embedded Systems with UML. In *Proceedings of CODES 2000*, San Diego, CA USA, 2000.

[13] R. B. France and T. B. Horton. Applying domain analysis and modeling: an industrial experience. In *Proceedings of the the 17th international conference on software engineering on Symposium on software reusability*, pages 206–214. ACM Press, 1995.

[14] J. Lilius and I. P. Paltor. Formalising UML state machines for model checking. In *Proceedings of UML'99*, volume 1723 of *LNCS*, pages 430–445. Springer Verlag, 1999.

[15] G. Malkin and R. Minnear. RIPng for IPv6. *RFC 2080*, January 1997.

[16] M. A. Miller. *Implementing IPv6. Supporting the Next Generation Internet Protocols*. M&T Publishing Ltd, Second edition, 1999.

[17] Open SystemC Initiative. *http://www.systemc.org*.

[18] J. Postel. User Datagram Protocol. *RFC 768*, 28 August 1980.

[19] D. Rosenberg and K. Scott. *Use Case Driven Object Modelling with UML*. Addison-Wesley, 1999.

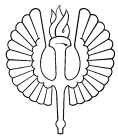
[20] S. Virtanen, J. Lilius, T. Nurmi, and T. Westerlund. TACO: Rapid Design Space Exploration for Protocol Processors. In *Proceedings of Electronic Design Processes (EDP'02)*, April 2002.

[21] S. Virtanen, J. Lilius, and T. Westerlund. A Processor Architecture for the TACO Protocol Processor Developement Framework. In *Proceedings of the 18th IEEE Norchip Conference*, November 2000.

[22] S. Virtanen, D. Truscan, and J. Lilius. TACO IPv6 Router - A Case Study in Protocol Processor Design. Technical Report 528, Turku Centre for Computer Science, April 2003.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi



University of Turku
- **Department of Information Technology**
- **Department of Mathematics**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**