A UML DRIVEN ASIC DESIGN METHODOLOGY AIDED BY AN AUTOMATED UML-SYSTEMC TRANSLATOR

NAVNEET ARVIND JAGANNATHAN

(B.Eng.(Computer Engineering), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OFMASTER OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgments

I would like to thank the following people for enabling me to see this endeavour through to fruition.

Dr. Tay Teng Tiow, my supervisor, from whose knowledge and experience I have gained immensely, for his constant guidance throughout the tenure of this project.

Dr. Ha Yajun, for taking interest in my work, and for his invaluable input at the outset of this project.

Cheah Yee Keng, an FYP student, who worked with me initially on the UML-SystemC translator until the allure of wealth (and the event of his graduation) drew him to a lifetime of drudgery.

Sun Xiaoxin, fellow Master's student, for his many useful contributions to this project, and for an 'Ubuntu' linux CD (which I have yet to try out) he passed to me once, in a moment of evangelical zeal.

Zhu Xiaoping, who has been my vital link to the DSA lab ever since I converted to part-time status. He has helped me with administrative issues, delivered messages to my supervisor, passed on CDs, thesis drafts etc. If I were him, I would ask to be paid.

King Hock, the lab technician, for ensuring that I had computers that worked.

Lastly, the **two noisy girls** in my lab, (I'm not sure what their names are), whose animated discussions of their private lives tremendously livened up my dull evenings at the DSA lab.

Contents

Acknowledgments **ii**

Summary **vi**

1 Introduction **2**

1.1 SystemC 4

1.1.1 The SystemC Platform 4

1.1.2 Review of the SystemC language 5

1.1.3 SystemC Language Architecture 7

1.2 UML 8

1.2.1 Use case diagrams 9

1.2.2 Class diagrams 10

1.2.3 Package and Object diagrams 11

1.2.4 Sequence diagrams 13

1.2.5 Collaboration diagrams 14

1.2.6 Statechart diagrams 15

1.2.7 Activity diagrams 16

1.2.8 Component and deployment diagrams 17

1.3 A Survey of Existing and Emerging Approaches 18

1.3.1 Pure HDL based coding 18

1.3.2 SystemC . . . 19

1.3.3 MATLAB . . . 21

1.3.4 SystemVerilog (www.systemverilog.org) . . . 23

1.3.5 UML for hardware design . . . 24

1.3.6 Model-driven architecture . . . 26

1.4 Thesis contributions . . . 27

2 UML Driven design methodology **30**

2.1 Modelling styles . . . 32

2.2 UML constructs employed . . . 32

2.2.1 Class Diagrams . . . 34

2.2.2 Statecharts . . . 36

2.2.3 Activity Diagrams . . . 36

3 The UML-SystemC Translator **40**

3.1 Class Parsing and code generation . . . 41

3.2 Statechart parsing and code generation . . . 43

3.2.1 Basic translation Schema . . . 43

3.2.2 Compound Transitions . . . 44

3.2.3 Hierarchy . . . 45

3.2.4 History States . . . 46

3.2.5 Concurrency . . . 47

4 Case Studies **51**

4.1 Experimental Tools . . . 51

4.1.1 ArgoUML . . . 51

4.1.2 Rational Rose Enterprise Edition 2002 . . . 52

4.1.3 The UML-SystemC Translator . . . 53

4.1.4 Cocentric SystemC Compiler 53

4.2 Design Implementation 54

4.2.1 JPEG Encoder 55

4.2.2 MAC Controller 61

4.2.3 FIR Filter 65

4.2.4 FFT 69

4.2.5 FIFO 71

4.2.6 VP3 Video Encoder 75

4.3 A Comparison with HDL Design Flow 164

5 Conclusion **169**

Summary

Embedded system design is faced with a challenge of increasing complexity. This means that an increasingly large effort is being expended in the conceptualisation stage of projects as minor errors in the design stage have the potential of snowballing into irreparable structural flaws in the later stages of a design. The focus of this thesis is on the hardware side of embedded systems.

Current design methodolgoies employ Hardware Description Languages such as VHDL and Verilog to describe designs. There has been a recent shift towards using *UML* (Unified Modelling language - a high-level modelling language widely used in the software world to design software systems) for high level system specification for hardware.

This thesis discusses a novel approach to hardware design that falls within the category of UML Model-driven hardware design. In our approach, we have made an attempt to move further ahead from mere design specification using UML, towards a more comprehensive and formalized Model Driven approach through the use of a platform independent *synthesizable* language in the form of UML with an automated route to a *SystemC* (an increasingly popular VHDL/Verilog -like C++ based hardware description language) implementation.

We have made an attempt to formalize UML through the use of stereotypes, allow designers to express designs in high-level terms such as statecharts and activity diagrams and allow direct translation of these descriptions to synthesisable SystemC code, through the use of an automated UML-SystemC translator, thus providing a direct path from UML Model to netlist for the designer.

The work of this thesis covers three main areas, namely, the definition of UML syntax for describing hardware, building of a UML-SystemC translator to act as a UML model-compiler to produce synthesisable SystemC code, and creating designs based on

the syntax , using the tool created. Three main ways of representing designs were defined.We use class diagrams for static structures, and use statecharts and activity diagrams for dynamic structures. A mixture of these elements was employed depending on the nature of the design.

List of Tables

2.1 RTL and Behavioural Modelling styles 33

4.1 Designs Created 55

4.2 JPEG Encoder synthesis settings 60

4.3 Results obtained from synthesis of JPEG Encoder 61

4.4 MAC synthesis settings 65

4.5 Results obtained from synthesis of MAC Controller 65

4.6 FIR synthesis settings 67

4.7 Results obtained from synthesis of FIR Filter (translated and original Cocentric) modules 68

4.8 FFT synthesis settings 71

4.9 Results obtained from synthesis of FFT Module through the translator compared with the original 71

4.10 FIFO synthesis settings 74

4.11 Results obtained from synthesis of FIFO through translator compared with original 74

4.12 VP3 Encoder synthesis settings 164

4.13 Results obtained from synthesis of VP3Encoder modules 165

List of Figures

1.1 SystemC 2.0 Language Architecture 8
1.2 UML Use Case Diagram 10
1.3 UML Class Diagram 11
1.4 UML Package Diagram 12
1.5 UML Object Diagram 12
1.6 UML Sequence Diagram 13
1.7 UML Collaboration Diagram 14
1.8 UML Statechart Diagram 15
1.9 UML Activity Diagram 16
1.10 UML Component Diagram 17
1.11 Typical HDL Flow 19
1.12 Typical SystemC Flow 22

2.1 UML Based Design Flow 30
2.2 FIR Filter Block Design 35
2.3 FIR Filter UML Class Diagram 35
2.4 Serial to Parallel converter statechart 37
2.5 FFT Activity Diagram 39

3.1 UML-SystemC Translation Flow 41

3.2 FIR UML Diagram 42

3.3 FIR SystemC Code produced by Translator 42

3.4 FIR Statechart diagram Diagram 47

3.5 FIR Statechart SystemC Code produced by Translator 48

3.6 Statechart illustrating concurrency 49

3.7 Statechart with AND-states : SystemC Code produced by Translator . . 50

4.1 Use case diagram for the JPEG encoder 56

4.2 A Section of the JPEG encoder design 57

4.3 The Run length encoder of the JPEG encoder 58

4.4 Statechart of the rle1 unit 59

4.5 Statechart of the rzs unit 59

4.6 Simulation waveform of JPEG encoder 60

4.7 Use Case Diagram of MAC Controller 62

4.8 Class Diagram of Receiver Block 62

4.9 Statechart of the Transmitter Block 64

4.10 Use Case Diagram of FIR filter 66

4.11 Class Diagram of the FIR Filter 66

4.12 Statechart of the FIR's FSM unit 67

4.13 Activity Diagram of the FIR's Data path unit 68

4.14 Use case diagram for the FFT module 70

4.15 Class Diagram of the FFT unit 70

4.16 Activitiy Diagram of the FFT unit 70

4.17 Use case diagram of FIFO 72

4.18 Class Diagram of the FIFO 73

4.19 Activity Diagram of the FIFO 73

4.20 Use Case Diagram of the VP3 Encoder 75
4.21 Class Diagram of the VP3 Encoder Top 79
4.22 Main Activity Diagram of the VP3 Encoder Top 80
4.23 Class Diagram of the PickIntra module 82
4.24 Activity Diagram of the PickIntra module 83
4.25 Class Diagram of the PickModes module 84
4.26 Activity Diagram of the PickModes module 85
4.27 Class Diagram of the UpdateFrame module 88
4.28 Activity Diagram of the UpdateFrame module 89
4.29 Class Diagram of the GetMBIntraError module 91
4.30 Activity Diagram of the GetMBIntraError module 92
4.31 Class Diagram of the GetMBInterError module 94
4.32 Activity Diagram of the GetMBInterError module 95
4.33 Class Diagram of the GetMBMVInterError module 96
4.34 Activity Diagram of the GetMBMVInterError module 97
4.35 Class Diagram of the GetMBMVExhaustiveSearch module 99
4.36 Activity Diagram of the GetMBMVExhaustiveSearch module 99
4.37 Class Diagram of the GetFOURMVExhaustiveSearch module 101
4.38 Main Activity Diagram of the GetFOURMVExhaustiveSearch module . 102
4.39 A Sub Activity Diagram of the GetFOURMVExhaustiveSearch module 103
4.40 Class Diagram of the QuadCodeDisplayFragments module 105
4.41 Activity Diagram of the QuadCodeDisplayFragments module 106
4.42 Class Diagram of the GetIntraError module 108
4.43 Activity Diagram of the GetIntraError module 109
4.44 Class Diagram of the GetInterErr module 110
4.45 Activity Diagram of the GetInterErr module 112

4.46 Class Diagram of the GetSumAbsDiffs module 114

4.47 Activity Diagram of the GetSumAbsDiffs module 115

4.48 Class Diagram of the GetNextSumAbsDiffs module 116

4.49 Activity Diagram of the GetNextSumAbsDiffs module 117

4.50 Class Diagram of the GetHalfPixelSumAbsDiffs module 119

4.51 Activity Diagram of the GetHalfPixelSumAbsDiffs module 120

4.52 Class Diagram of the QuadCodeComponent module 121

4.53 Activity Diagram of the QuadCodeComponent module 122

4.54 Class Diagram of the DPCMTokenizeBlock module 123

4.55 Activity Diagram of the DPCMTokenizeBlock module 124

4.56 Activity Diagram of the TokenizeDCTBlock module of the DPCMTokenizeBlock module 125

4.57 Class Diagram of the PackCodedVideo module 127

4.58 Activity Diagram of the PackCodedVideo module 128

4.59 Class Diagram of the TransformQuantizeBlock module 130

4.60 Activity Diagram of the TransformQuantizeBlock module 131

4.61 Class Diagram of the ClearDownQFragData module 131

4.62 Activity Diagram of the ClearDownQFragData module 132

4.63 Class Diagram of the EncodeDcTokenList module 133

4.64 Activity Diagram of the EncodeDcTokenList module 134

4.65 Class Diagram of the EncodeAcTokenList module 135

4.66 Activity Diagram of the EncodeAcTokenList module 136

4.67 Class Diagram of the PackAndWriteDFArray module 138

4.68 Activity Diagram of the PackAndWriteDFArray module 139

4.69 Class Diagram of the PackModes module 140

4.70 Activity Diagram of the PackModes module 141

4.71 Class Diagram of the PackMotionVectors module 143

4.72 Activity Diagram of the PackMotionVectors module 144

4.73 Class Diagram of the PackToken module 145

4.74 Activity Diagram of the PackToken module 146

4.75 Class Diagram of the SUB8 module 147

4.76 Activity Diagram of the SUB8 module 148

4.77 Class Diagram of the SUB8_128 module 149

4.78 Activity Diagram of the SUB8_128 module 150

4.79 Class Diagram of the fdct_short module 151

4.80 Activity Diagram of the fdct_short module 152

4.81 Class Diagram of the quantize module 153

4.82 Activity Diagram of the quantize module 154

4.83 Class Diagram of the MotionBlockDifference module 155

4.84 Activity Diagram of the MotionBlockDifference module 156

4.85 Class Diagram of the SUB8AV2 module 157

4.86 Activity Diagram of the SUB8AV2 module 158

4.87 Class Diagram of the RegulateQ module 160

4.88 Activity Diagram of the RegulateQ module 161

4.89 Class Diagram of the UpRegulateDataStream module 162

4.90 Activity Diagram of the UpRegulateDataStream module 163

4.91 VHDL Design Flow Example using the Synopsys VHDL compiler 166

Chapter 1

Introduction

Embedded system design is faced with a challenge of increasing complexity. This means that an increasingly large effort is being spent in the conceptualisation stage of projects as minor errors in the design stage have the potential of snowballing into irreparable structural flaws in the later stages of a design. Complexity is introduced at many stages - in the overall system design, in hardware/software partitioning decisions, in software design, in ASIC design and in hardware / software integration.

The embedded system design process can be expressed as follows:

1. Requirements Specifications
2. Definition of System Capabilities
3. Definition of System modules
4. Hardware Software Partitioning
5. ASIC Design and synthesis
6. Software Design and implementation
7. Integration
8. Testing / Debugging

There is thus a need for the following:

- A platform independent, gender-neutral, formal and detailed high level representation of embedded systems.
- Enhanced inter-disciplinary communication - e.g. consideration of sequence diagrams showing complex software-hardware interactions to reduce misunderstandings and disagreements between the disciplines earlier and more directly than would be expected with conventional specification approaches.
- Direct executability of the model, and the adoption of a new style of system development where the complexity of the system development process is completely managed via models and not through code.

There have been codesign frameworks proposed before, for example, POLIS (see Balarin et al., 1997), which is driven by FSM descriptions of modules. The design methodology proposed in this thesis is part of an overall UML-model driven codesign framework being created in the DSA laboratory, NUS, where the system is analysed and modelled using UML constructs such as Use case diagrams, Class diagram, Statecharts, and Activity Diagrams. The classes are then partitioned into hardware and software depending on performance requirements before proceeding further downstream to C code and ASIC synthesis. The focus of this thesis is on ASIC design and implementation stage of this process. It describes an effort to fuse well established ideas from the digital hardware and software engineering worlds to achieve the above objectives by proposing a UML model-driven approach to ASIC design aided by a UML-SystemC translator that generates *synthesizable* SystemC (Synopsys, 2002b) code from static and dynamic platform independent UML representations of hardware designs. The ASIC design methodology at the highest level support RTL-type designs as well as behavioural designs depending on the application. It is expected that the methodology

will lead to higher productivity by providing a direct route to synthesis from abstract descriptions of the system such as statecharts and activity diagrams. In addition it facilitates easier exchange of platform independent models of the system in UML which is already a universally accepted standard in the software world. Several designs were created in UML using this methodology and translated to synthesizable SystemC using the UML-SystemC Translator.

1.1 SystemC

1.1.1 The SystemC Platform

SystemC is the standard design and verification language built on C++ that spans from concept to implementation in hardware and software and is supported by the OSCI (Open SystemC Initiative) - see www.systemc.org. The SystemC community consists of a large and growing number of system houses, semiconductor companies, IP providers and EDA tool vendors. Many companies function in more than one of these categories, with each category having compelling reasons for wanting to see SystemC emerge as a de facto standard. Firstly, it is expected that SystemC will bring order to the chaos that exists today in the evolving system-level design software arena, with the emergence of a de facto standard ensuring a stable market for the rapid development of C++ libraries, models and tools. Second, with high quality commercial tools available from a variety of vendors, systems houses and semiconductor companies will have an excellent alternative to creating and supporting their own proprietary libraries and tools. In other words, SystemC will allow these companies to take advantage of the latest innovations in C-based design without diverting resources to support their own tools, focusing instead on their core competencies and competitive strengths. Finally, these companies will be able to leverage SystemC to share models inside and outside their

organizations. IP providers will benefit from the SystemC de facto standard because it will allow them to provide a single set of models for each of their cores to any systems house or semiconductor company they do business with. Today, in the absence of any standard, IP providers are compelled to provide customized versions of their models to meet the different C-based design environments of their customers. With SystemC, they can do the work of creating and validating a SystemC model once, and then their library development work is done. EDA vendors will benefit from the SystemC de facto standard because it will create a large and stable market for them to compete in. EDA companies have historically developed and marketed tools based on their own proprietary design languages. This approach, however, limits innovation, fosters small and fragmented niche markets, lengthens time to market, requires inefficient learning curves on designers and imposes substantial risks on customers, given the number of EDA tool vendors that go out of business each year. SystemC changes all of that for every vendor competing in the EDA space, allowing each to innovate and create tools in a format that will soon achieve widespread acceptance.

1.1.2 Review of the SystemC language

The language is comprehensively described in the SystemC Language Reference manual (see Aboulhamid et al., 2003). SystemC provides a set of modeling constructs that are similar to those used for RTL and behavioral modeling within an HDL such as Verilog or VHDL. Similar to HDLs, users can construct structural designs in SystemC using modules, ports and signals. Modules can be instantiated within other modules, enabling structural design hierarchies to be built. Ports and signals enable communication of data between modules, and all ports and signals are declared by the user to have a specific data type. Commonly used data types include single bits, bit vectors, characters, integers, floating point numbers, vectors of integers, etc. SystemC also includes support for four-

state logic signals (i.e. signals that model 0, 1, X, and Z). An important data type that is found in SystemC but not in HDLs is the fixed-point numeric type. Fixed-point numbers are frequently used in DSP applications that target both hardware and software implementations since floating point operations usually consume too many hardware resources. An example fixed point operation might be to add two signed numbers that have three bits of integer precision and four bits of fractional precision and assign the result to a similar fixed point number. Often users wish to specify rounding and overflow modes (e.g. saturate or wrap on overflow) when using fixed point numbers. It is easy and natural to model fixed-point numbers in SystemC, but this is very difficult to do in software. In VHDL, concurrent behaviors are modeled using processes. In Verilog, concurrent behaviors are modeled using "always" blocks and continuous assignments. In SystemC, concurrent behaviors are also modeled using processes. A process can be thought of as an independent thread of control which resumes execution when some set of events occur or some signals change, and then suspends execution after performing some action. In SystemC, there is a limited ability for specifying the condition under which a process resumes execution: the process can only be sensitive to changes of values of particular signals, and the set of signals to which the process is sensitive must be pre-specified before simulation starts. Since processes execute concurrently and may suspend and resume execution at user-specified points, SystemC process instances generally require their own independent execution stack. (An equivalent situation in the software world arises in multi-threaded applications-each thread requires its own execution stack.) Certain processes in SystemC that suspend at restricted points in their execution do not actually require an independent execution stack-these processes types are termed "SC_METHODs". Optimizing SystemC designs to take advantage of SC_METHODs provides dramatic simulation performance improvements when the number of process instances in a design is large. Hardware signals have several properties

that complicate the task of modelling them in software. First, users often want to simulate hardware signals and registers as being initialized to "X" when simulation starts. This is useful for detecting reset problems in designs via X propagation techniques in simulation. In SystemC , this feature is provided within the sc_logic and sc_lv data types. Second, hardware signals sometimes have multiple drivers. In this case, a function is needed to compute a resolved value based on each of the driving values. This function must automatically be called when any of the driving values changes. For example, when a signal is driven with a 1 and a Z, the resolved value should be 1, but when driven with a 1 and a 0, the resolved value should be X. In SystemC, resolved logic signals are provided to handle this modeling. Third, hardware signals do not immediately change their output value when they are assigned a new value, either in simulation or in the real world. There is always some delay (perhaps very small) until the new value assigned to a signal is made available to other processes in the design. This delay is crucial to proper modeling of hardware, since it allows, for example, two registers to swap values on a clock edge. In comparison, two software variables cannot swap values without the introduction of a third temporary variable.

1.1.3 SystemC Language Architecture

Figure 1.1 summarizes the SystemC 2.0 language architecture. There are several important concepts to understand from this diagram.

1. All of SystemC builds on C++.

2. Upper layers within the diagram are cleanly built on top of the lower layers

3. The SystemC core language provides only a minimal set of modeling constructs for structural description, concurrency, communication, and synchronization.

4. Data types are separate from the core language and user-defined data types are

fully supported.

5. Commonly used communication mechanisms such as signals and fifos can be built on top of the core language.

6. Commonly used models of computation (MOCs) can also be built on top of the core language.

7. If desired, lower layers within the diagram can be used without needing the upper layers.

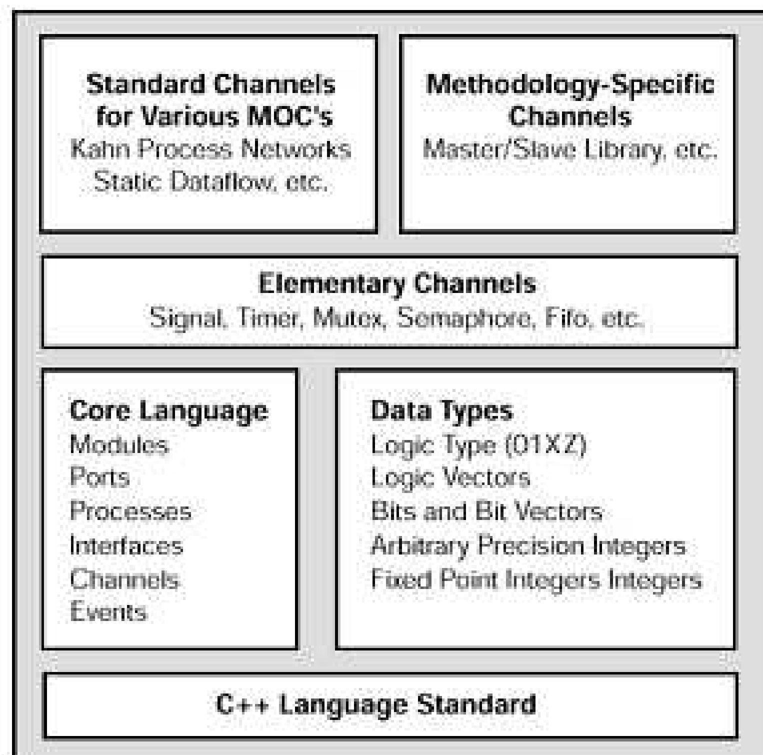


Figure 1.1: SystemC 2.0 Language Architecture

1.2 UML

The Unified Modelling Language (UML) is the industry-standard language for specifying, visualizing, constructing, and documenting the artefacts of software systems (see OMG, 2003). It simplifies the complex process of software design, making a "blueprint"

for construction. Modelling is an essential part of large software projects, and helpful to medium and even small projects as well. Using a model, those responsible for a software development project's success can assure themselves that business functionality is complete and correct, end-user needs are met, and program design supports requirements for scalability, robustness, security, extendibility, and other characteristics, before implementation in code renders changes difficult and expensive to make. Using any one of the large number of UML-based tools on the market, one can analyze an application's requirements and design a solution that meets them, representing the results using UML's standard diagram types. As hardware-software systems become more and more ubiquitous and the demand for an effective universal co-design framework increases, UML presents itself as an excellent candidate. It also lends itself well to hardware design which, at high-levels, can be represented using very similar constructs to software. At the centre of the UML are its nine kinds of modelling diagrams.

1.2.1 Use case diagrams

Use case diagrams describe what a system does from the standpoint of an external observer. The emphasis is on what a system does rather than how. Use case diagrams are closely connected to scenarios. A scenario is an example of what happens when someone interacts with the system. Here is a scenario for a video encoder. "A user changes the configuration of the video encoder. The system suspends its current operation and changes its internal settings. ". A use case is a summary of scenarios for a single task or goal. An actor is who or what initiates the events involved in that task. Actors are simply roles that people or objects play. Actors are stick figures. Use cases are ovals. Communications are lines that link actors to use cases. A use case diagram is a collection of actors, use cases, and their communications. A single use case can have multiple actors. This diagram is used in higher levels of the design flow and aids in

defining the modules that the system must contain.

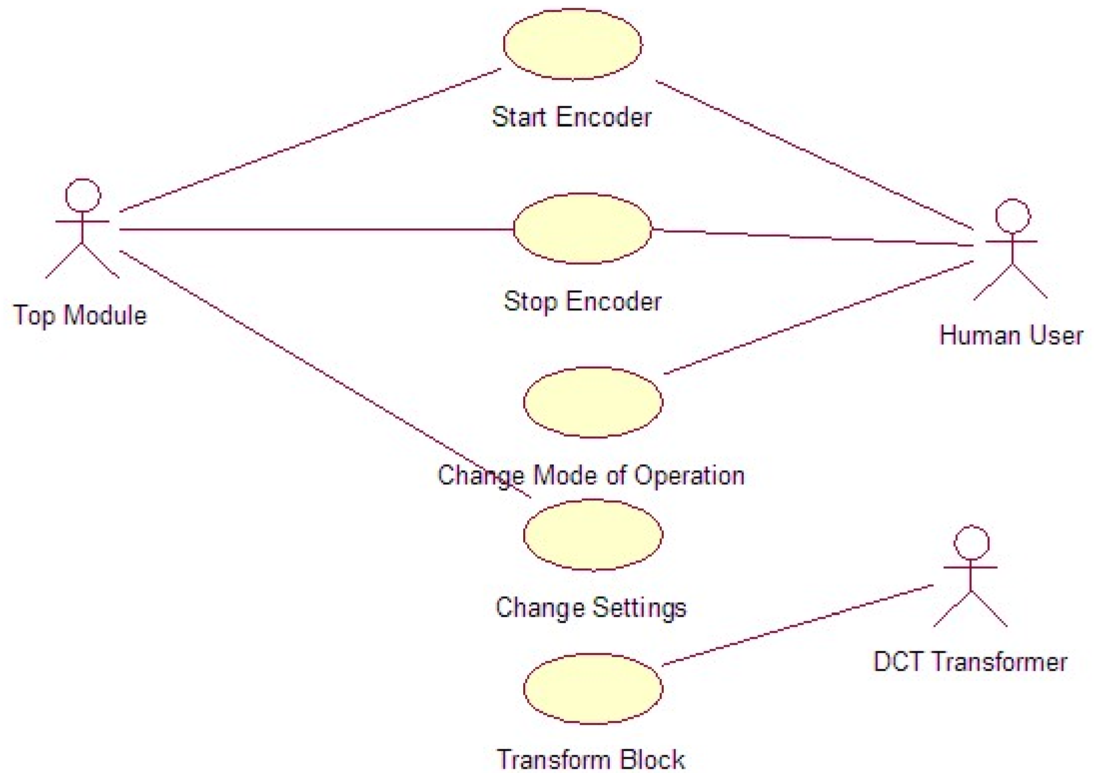


Figure 1.2: UML Use Case Diagram

1.2.2 Class diagrams

A Class diagram provides an overview of a system by displaying its classes and the relationships among them. Class diagrams are static – they display what interacts but not what happens when they interact. The class diagram in Figure 1.3 models an FIR filter. The central class is the FIR RTL. Associated with it is the fir_fsm which is the state machine and the fir_data.

The UML class notation is a rectangle divided into three parts: class name, attributes, and operations. Names of abstract classes are usually in italics. Relationships between classes are the connecting links. A class diagram has three kinds of relation-

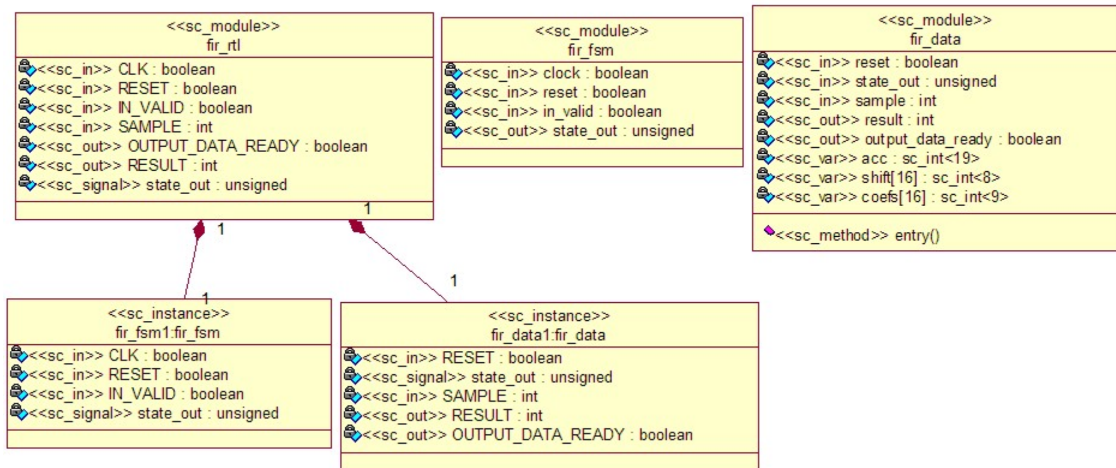


Figure 1.3: UML Class Diagram

ships.

- association
- aggregation
- generalization

A navigability arrow on an association shows in which direction the association can be traversed or queried. The multiplicity of an association end is the number of possible instances of the class associated with a single instance of the other end. The class diagram notation is appropriate for description of static structures in a system. In a software application this would mean classes and their associations. In a hardware application this would potentially mean high level modules/entities of the system as seen in Figure 1.3

1.2.3 Package and Object diagrams

To simplify complex class diagrams, classes are grouped into packages. A package is a collection of logically related UML elements. Figure 1.4 is a business model in which the classes are grouped into packages in a typical application of UML.

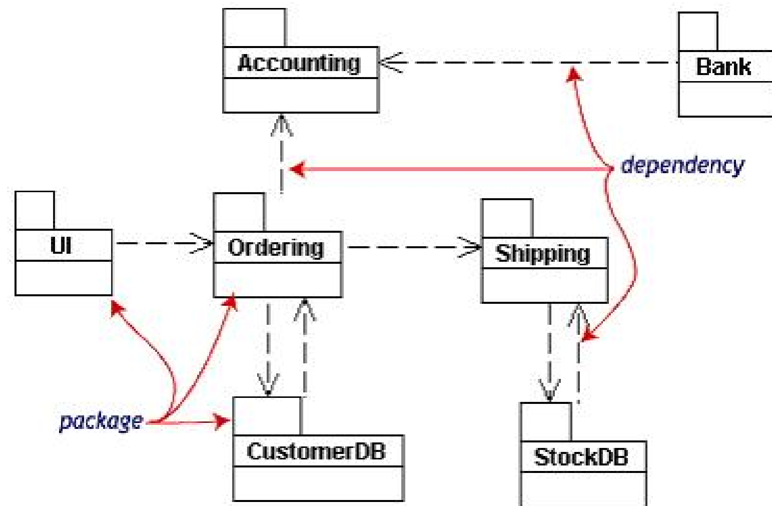


Figure 1.4: UML Package Diagram

Object diagrams show instances instead of classes. They are useful for explaining small pieces with complicated relationships, especially recursive relationships.

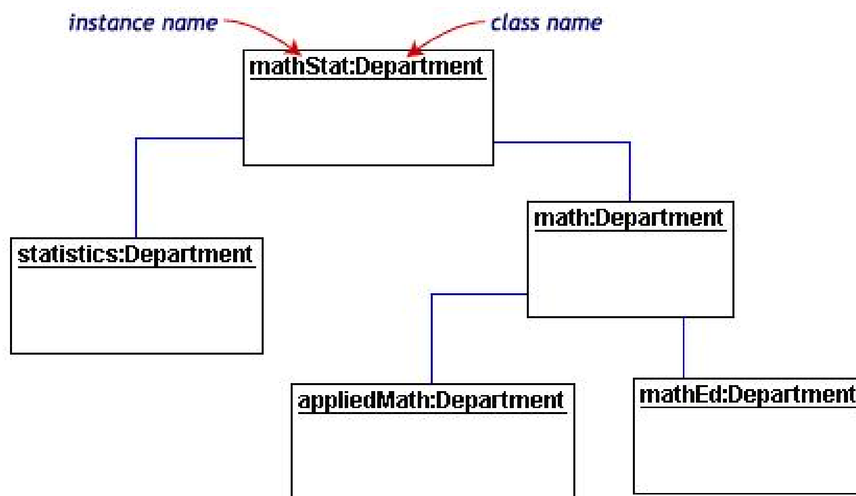


Figure 1.5: UML Object Diagram

Each rectangle in the object diagram corresponds to a single instance. Instance names are underlined in UML diagrams. Class or instance names may be omitted from object diagrams as long as the diagram meaning is still clear. Packages can be used to express libraries of software or hardware components.

1.2.4 Sequence diagrams

Class and object diagrams are static model views. Interaction diagrams are dynamic. They describe how objects collaborate. A sequence diagram is an interaction diagram that details how operations are carried out – what messages are sent and when. Sequence diagrams are organized according to time. The time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence. Figure 1.6 shows a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is a Reservation window.

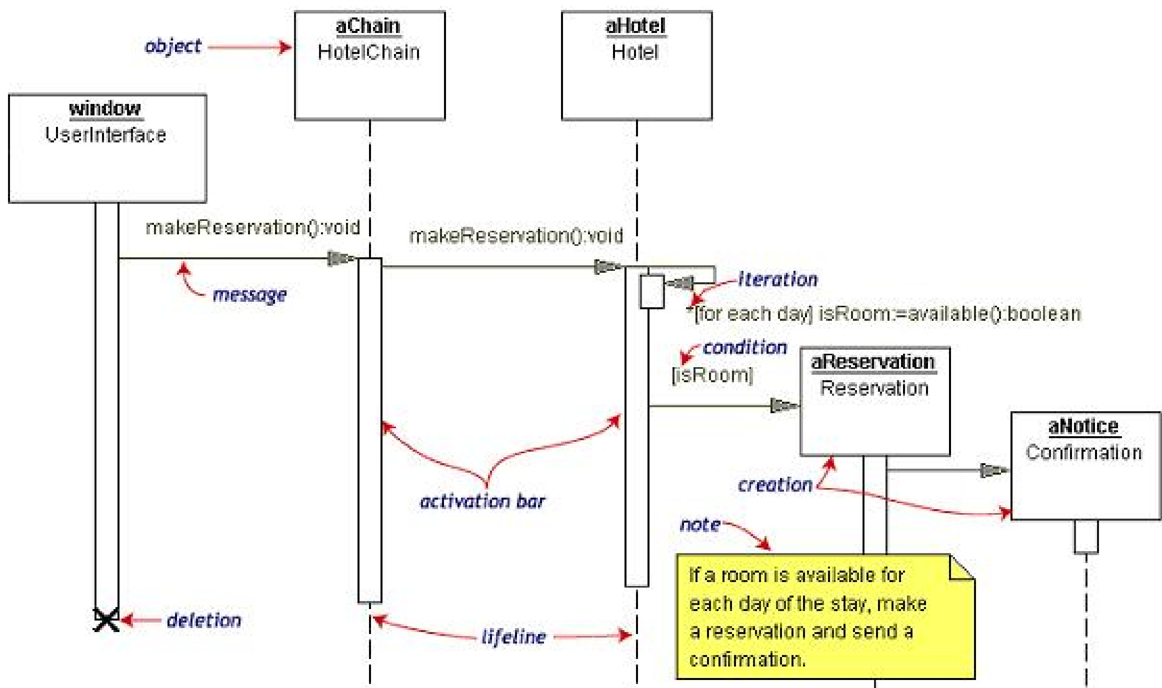


Figure 1.6: UML Sequence Diagram

Each vertical dotted line is a lifeline, representing the time that an object exists. Each arrow is a message call. An arrow goes from the sender to the top of the activation bar of the message on the receiver's lifeline. The activation bar represents the duration of execution of the message. The diagram has a clarifying note, which is text inside a dog-eared rectangle. Notes can be put into any kind of UML diagram. Sequence

diagrams once again help to flesh out the functionality of a class by exhaustively defining various scenarios and therefore all the messages that could possibly pass between any two instances of a class. This helps the designer to identify what sort of messages a particular class should handle. This is done upstream.

1.2.5 Collaboration diagrams

Collaboration diagrams are also interaction diagrams. They convey the same information as sequence diagrams, but they focus on object roles instead of the times that messages are sent as opposed to a sequence diagram, where object roles are the vertices and messages are the connecting links.

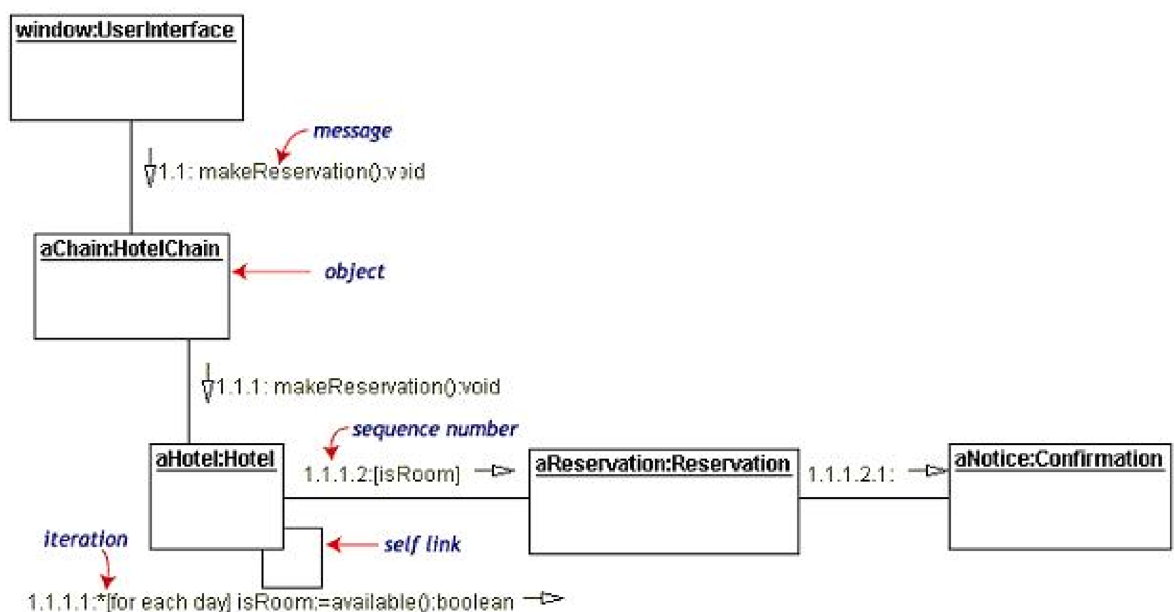


Figure 1.7: UML Collaboration Diagram

The object-role rectangles are labelled with either class or object names (or both). Class names are preceded by colons. Each message in a collaboration diagram has a sequence number. The top-level message is numbered 1. Messages at the same level (sent during the same call) have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.

1.2.6 Statechart diagrams

Objects have behaviors and state. The state of an object depends on its current activity or condition. A statechart diagram shows the possible states of the object and the transitions that cause a change in state. With reference to Figure 1.8, "logging in" can be factored into four non-overlapping states: Getting SSN, Getting PIN, Validating, and Rejecting. From each state comes a complete set of transitions that determine the subsequent state.

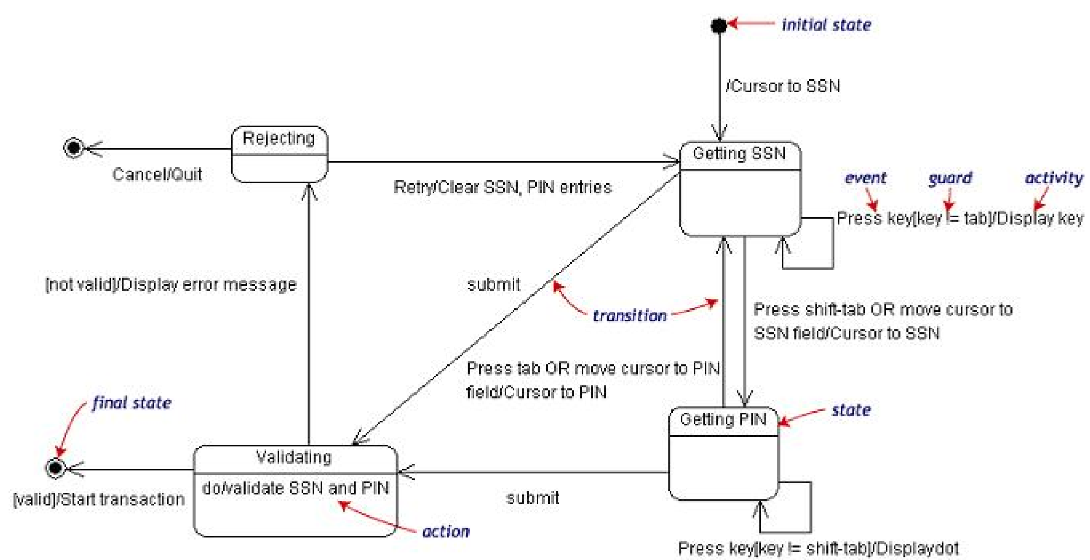


Figure 1.8: UML Statechart Diagram

States are rounded rectangles. Transitions are arrows from one state to another. Events or conditions that trigger transitions are written beside the arrows. The initial state (black circle) is a dummy to start the action. Final states are also dummy states that terminate the action. The action that occurs as a result of an event or condition is expressed in the form /action. While in its Validating state, the object does not wait for an outside event to trigger a transition. Instead, it performs an activity. The result of that activity determines its subsequent state. UML Statecharts are based on the statechart notation originally proposed by David Harel (see Harel, 1987). These are very useful in describing dynamic behaviour of both hardware and software modules in

response to various messages.

1.2.7 Activity diagrams

An activity diagram is essentially a fancy flowchart. While a statechart diagram focuses attention on an object undergoing a process (or on a process as an object), an activity diagram focuses on the flow of activities involved in a single process. The activity diagram shows the how those activities depend on one another.

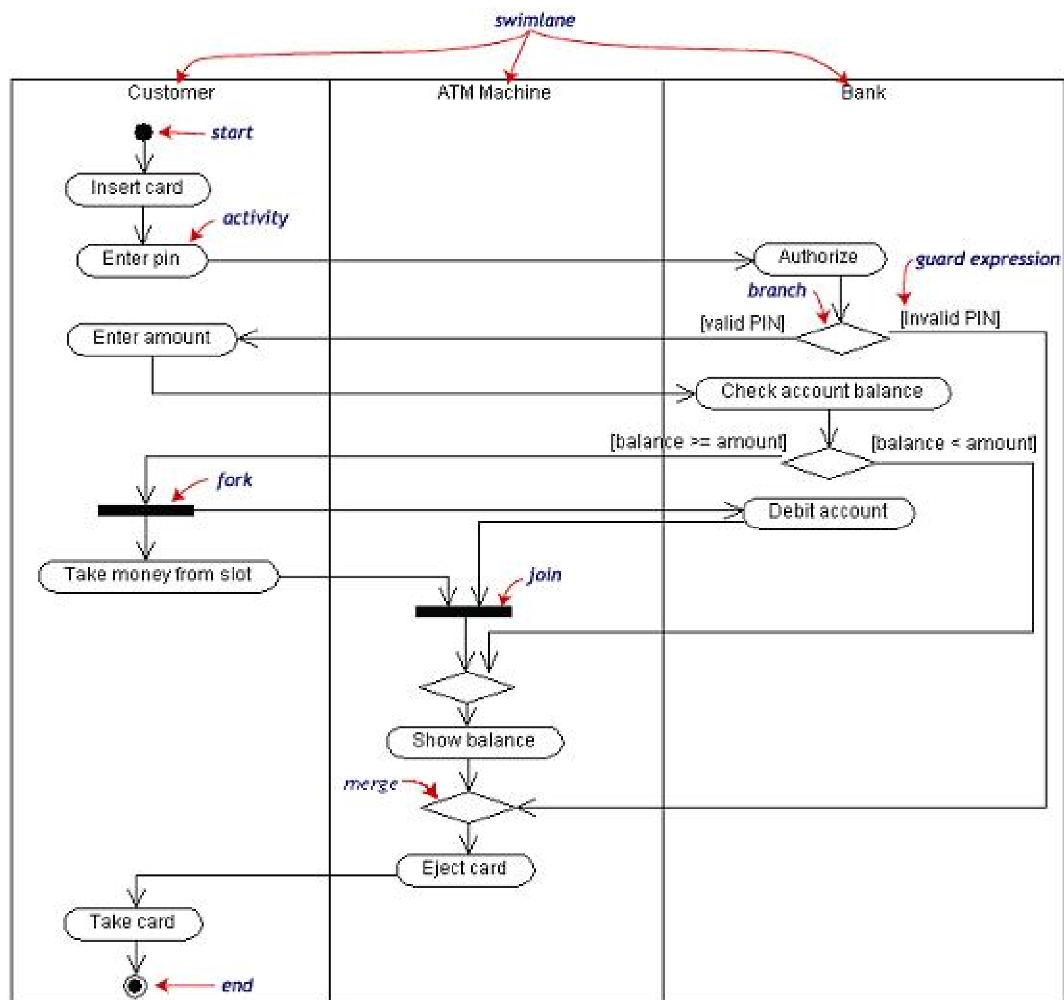


Figure 1.9: UML Activity Diagram

Activity diagrams can be divided into object swim lanes that determine which object is responsible for which activity. A single transition comes out of each activity, connecting it to the next activity. A transition may branch into two or more mutually exclusive transitions. Guard expressions (inside [ ]) label the transitions coming out of

a branch. A branch and its subsequent merge marking the end of the branch appear in the diagram as hollow diamonds. A transition may fork into two or more parallel activities. The fork and the subsequent join of the threads coming out of the fork appear in the diagram as solid bars. Activity diagrams are particularly useful in applications which cannot be described in a statechart form (eg. Video processing applications) and demand behavioural or algorithmic descriptions.

1.2.8 Component and deployment diagrams

A component is a code module. Component diagrams are physical analogs of class diagram. Deployment diagrams show the physical configurations of software and hardware. The following deployment diagram shows the relationships among software and hardware components involved in real estate transactions.

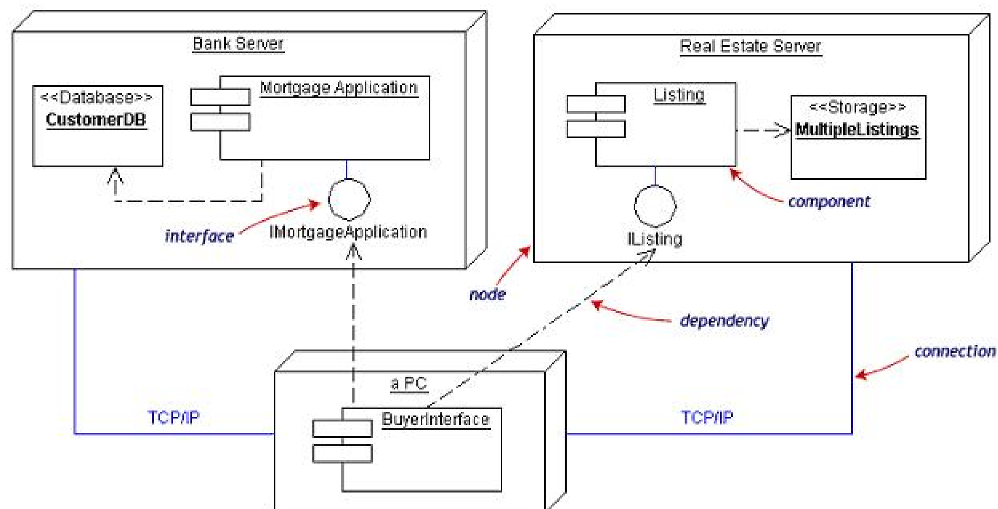


Figure 1.10: UML Component Diagram

The physical hardware is made up of nodes. Each component belongs on a node. Components are shown as rectangles with two tabs at the upper left.

1.3 A Survey of Existing and Emerging Approaches

1.3.1 Pure HDL based coding

In this approach, Hardware description languages (HDLs) are used to describe the architecture and behaviour of digital electronic systems. This is the most widely prevalent approach to designing digital systems. Hardware description languages have grown in importance, as digital systems have become more complex, to overcome the tedium of detailed design at the gate and flip-flop level. The move from transistors and solder, to netlists, to HDLs is analogous to the move, in the software world, from machine code to assembly language to high-level programming languages.

An HDL allows a digital system to be designed and debugged at a higher level before conversion to gate and flip-flop level. Top-down HDL design is most useful in large projects, where several designers are working concurrently. In such scenarios, HDLs allow structured development where once the architecture of the design is decided and major modules defined, work can proceed independently. The most popular hardware description languages are VHDL and Verilog, with SystemC now emerging as a popular alternative choice and expected to become the de facto.

HDLs support a mixed-level description where structural or netlist constructs can be mixed with behavioural or algorithmic descriptions. With this mixed-level capability, it allows the designer to describe the system at a higher algorithmic level and then incrementally refine the design to lower levels of abstraction. HDLs have the additional advantage of excellent support in the form of very advanced compilers to perform synthesis from high-level descriptions of the design.

An HDL based design methodology has several fundamental advantages over the older gate-level methodology.

1. It allows design verification via simulation of the HDL code. This allows the

designer to test out a variety of architectures and designs before narrowing down to the final design.

2. Due to the availability of compilers, high level designs can be translated down to gate level accurately thus improving productivity and reducing circuit design time. Optimization can also be performed using these tools.

3. HDLs are technology independent descriptions and are much easier to read and understand than complex netlist or schematic descriptions of circuits.

Shown in Figure 1.11 is a typical HDL design flow employed in designing and synthesising digital circuits.

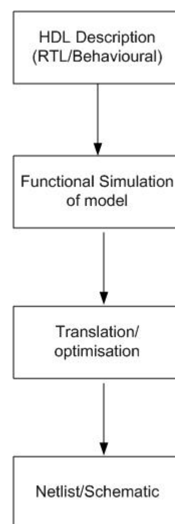


Figure 1.11: Typical HDL Flow

1.3.2 SystemC

SystemC is an increasingly popular choice for system designers. It was created to address the system-level design challenges of tool interoperability, team communication, and component modules creation and distribution. For a detailed description of the language and the platform, please refer to section 1.1.

SystemC is based on C++, the most popular language with system designers and

software engineers. The SystemC classes add the necessary constructs to C++ for modelling systems and hardware at various levels of abstraction-from abstract untimed models to cycle-accurate RTL models. (Synopsys, 2002a)

Untimed Functional Level

Algorithms are most easily verified using level of abstraction. At this level models communicate with each other using FIFO channels accessed using blocked read and write operations. This level of abstraction helps designers in two ways. Firstly, models are easy to develop (simple communication and implicit synchronization), and secondly simulation speed is high because unnecessary details are not modelled at this level.

Timed Functional Level

At this level additional delay annotation are added. This level of abstraction is used to analyse the effects of latency on system behaviour and system architecture in the early stages of the design process. Thus this level of abstraction will determine whether the system generates data on time or will miss deadlines. These models are also used for early Hardware-Software partitioning analysis. By evaluating the impact of mapping different modules to hardware(with one delay figure) and software (with another delay figure) an exhaustive tradeoff analysis can be performed.

Bus Cycle Accurate Level

In this level, also known as the 'Transaction Level', the exact implementation of the communication channels is not modelled, only the behaviour is modelled and expressed in terms of transactions. This level yield important design factors such as optimisation of bus utilisation and loading, cache scheme effectiveness, and confirmation of interaction between hardware and software.

Behavioural Hardware Model

This is also known as the Pin - accurate level, and requires the pin-level interface to be defined for every model. These can be thought of as refined Functional models. These models can be fully sufficient for verification purposes. However, for the purposes of synthesis they have to be coded in a subset of SystemC which is synthesisable. The synthesizable code can be written in either of two flavors, RTL style or behavioural style. RTL style is preferred when coding the exact register transfer and clocking details. Behaviour style is preferred when coding an algorithmic expression and allowing the compiler to propose a solution that performs the desired function.

Register Transfer Level

At the RTL, digital hardware is exactly described in terms of hardware registers, clocks, latches and combinational logic. RTL allows designers to exactly specify the internal structure of a silicon component in the detail required for logic synthesis. RTL code is written in a special RTL coding style (as opposed to the behavioral coding style).

As described earlier SystemC can be used to arrive at a gate-level netlist if coded using the synthesisable subset of the language. The design flow is shown in Figure 1.12.

1.3.3 MATLAB

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an ensy-to-use environment where problems and solutions are expressed in familiar mathematical notations. Technical professionals worldwide rely on MATLAB lo accelerate their research, reduce analysis and development time, minimize project costs, and produce effective solutions. It is widely used in

1. Math and computation

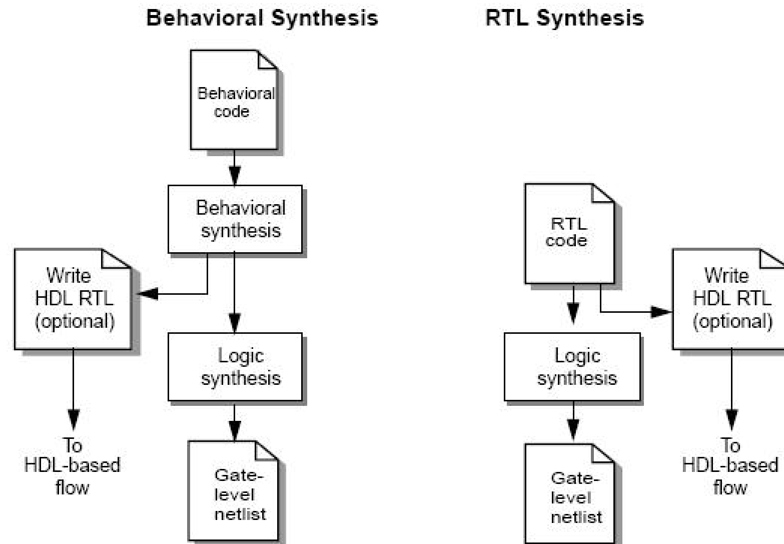


Figure 1.12: Typical SystemC Flow

2. Algorithm development

3. Data acquisition

4. Modeling, simulation, and prototyping

5. Data analysis, exploration, and visualization

6. Scientific and engineering graphics

7. Application development, including graphical user interface building

Although it is not strictly used in SoC design, MATLAB has found a use in accelerating research, reducing development time and producing effective solution. In a recent effort (Xi et al., 2003) the strengths of MATLAB such as stimuli generation, visualisation and result analysis have been applied to system design using SystemC. Two methods are used. One is to employ MATLAB as a computation engine. In this scheme, MATLAB is evoked, executed and terminated from within a SystemC testhench.The SystemC testbench invokes the MATLAB engine to execute MATLAB commands and functions, through which the needed stimuli are generated and fed to SystemC model under test. The output is then collected for analysis and visualization after execution. The other is

to use MATLAB as the test I/O for SytemC model. In this manner, the stimuli for the SystemC model are extracted from MATLAB and output results of the SystemC model are written to a MATLAB MAT file for comparison and analysis in MATLAB.

1.3.4 SystemVerilog (www.systemverilog.org)

SystemVerilog is the industry's first unified hardware description and verification language (HDVL) standard. SystemVerilog is a major extension of the established IEEE 1364TM Verilog language. It was developed originally to dramatically improve productivity in the design of large gate-count, IP-based, bus-intensive chips. SystemVerilog is targeted primarily at the chip implementation and verification flow, with powerful links to the system-level design flow. Its main extensions over Verilog include: Support of modeling and verification at the transaction level of abstraction. SystemVerilog's Direct Programming Interface (DPI) enables it to call C/C++/SystemC functions, and vice versa. SystemVerilog is thus the first Verilog-based language to enable efficient co-simulation with SystemC blocks , an invaluable link between system level design and chip implementation and verification.

1. A set of extensions to address advanced design requirements. Examples include modeling interfaces that greatly accelerate the development of bus-intensive designs; removal of restrictions on module port connections, allowing any data type on each side of the port; extended data types to allow C modeling; enhanced IP protection by nesting modules locally to their parent module and, consequently, not visible to other parts of the design.

2. A new mechanism to support assertion-based verification (ABV). In SystemVerilog, assertion information is built into the language, eliminating the need for the special modules, pragmas or PLI calls used in traditional Verilog. This approach helps to avoid recoding errors, increase test accuracy, simplify the testbench, and

enable test reuse. The full controllability and observability of internal circuit nodes afforded by ABV can significantly reduce design debug time.

3. New features to support hardware models and testbenches that utilize object-oriented techniques and testbenches that are eminently re-usable. For example, the combination of SystemVerilog's Interface method with object-oriented testbench creation techniques enables the easy implementation of a powerful constraint-driven verification methodology.

The application of SystemVerilog to real design problems has demonstrated a substantial productivity gain due to the reduced amount of code needed to get the same quality of results from synthesis. SystemVerilog features such as user defined data-types, multi-dimensional arrays and strong data type checking can effectively adopted by both Verilog as well as VHDL users (Fitzpatrick, 2004) thus addressing a number of existing issues with current HDLs.

1.3.5 UML for hardware design

During embedded systems development, the physical, logical, functional and dynamic requirements and capabilities of the system must be specified, and decisions made regarding which portions of the system will be implemented in software, hardware, or firmware, or whether they will be multitasking, multithreaded, multiprocessor systems with DSPs, digital logic, etc. Additionally, candidate commercial off the shelf components need to be defined in terms of their interfaces and capabilities, again without constraining possible solutions. The process of allocating the requirements to the domains and elements described above must be performed objectively, without assumptions or preconceptions.

Unfortunately, breaking-down a large problem may cause as many problems as the problem itself. Systems Engineering practitioners have recently been experimenting

with the use of the Unified Modelling Language (UML). Using UML Packages and Subsystems, a framework is created by the Systems Engineer, which is a dynamic evolutionary environment, providing traceability and consistency. Additionally, by making use of the UML extension mechanism, essential aspects of Systems Engineering issues such as concurrency, hardware architecture, and non-functional requirements can also be addressed. Specialised tools can then be used to access this environment to ensure a complete solution. It is now possible to analyze and design a system into logical units so that the allocation process can be successfully completed by creating a framework of system modes, use cases and concurrency to demonstrate the dynamic capabilities of the system.

Thus the object-oriented design process has been a hot topic in software development since it considerably improves productivity and product quality which are also issues in System on chip design. In (de Jong, 2002) a combination of UML and SDL is used in the 'formulation' phase and the 'formalization' phase of the design respectively. The approach furthers the earlier work done on the use of generic OO methodologies and structured analysis design in system specification. It combines the informal strengths of UML with the formal strengths of SDL to propose a comprehensive specification capturing methodology. In (Drosos et al., 2004), UML and its real-time extension is used for specification and co-development of the prototype of a HIPERLAN/2. The SLOOP system-level design process (Qiang et al., 2002, see), for example, makes use of UML as a specification language for the four models proposed in the methodology for the designers to use as a reference for implementation. In (Fornaciari et al., 2003) UML is used for system specification and as a basis for further downstream analysis.(Martin, 2002) is another endorsement of the use of UML as a specification language. Even in the industry there is an increasing trend towards enforcing systems engineering principles for embedded systems using UML.

1.3.6 Model-driven architecture

We are now at a stage in the development of embedded systems where the complexity of the software development and the software itself needs to be managed not via code but via models. This move to a more abstract representation of the system brings many new challenges to primarily the methodology and the style of system development. For an early work on using high level descriptions to model hardware for PLAs refer to (Drusinsky and Harel, 1989).

The Object Management Group's (OMG) Model Driven Architecture (MDA) is an initiative towards fully model based software development. However, while the MDA is aimed at large-scale, enterprise applications, it is also going to be used with embedded development and need to take into consideration the constraints that are not apparent or irrelevant larger systems.

In most existing approaches UML is used as an informal specification tool. This is analogous to the software industry's movement from high-level languages to a model-driven architecture. This methodology is expected to offer significant advantages over previous methods.

1. It allows design verification of the model via simulation of the code generated by the translator from the UML model. This allows the designer to test out a variety of architectures and designs before narrowing down to the final model.

2. UML models can be translated down to gate level accurately thus improving productivity and reducing circuit design time.

3. UML models are technology independent descriptions and are much easier to read and understand. This is becoming increasingly important as design complexity increases and HDL code sizes increase to unwieldy lengths.

1.4 Thesis contributions

The design approaches surveyed so far can be classified into three main categories,

1. High level design approaches such as those involving UML that the designer undertakes to form a top-level picture of the design before starting to code the individual modules in a lower level language.

2. System-level design such as SystemC and SystemVerilog that increase the expressiveness of HDLs so as to provide easier verification through increased levels of abstraction. A subset of these languages also provides a path to synthesis.

3. HDLs, the traditional approach, which provides a direct path to synthesis but do not enable higher level visualisations of the design.

The motivation of this thesis, therefore, is to provide an approach to top-level design that posseses the strength of UML for high-level specification, but with the added advantage of being able to automatically translate these formal specifications into a synthesisable form, rather than using them as a loose overview and guideline for design, thereby, enabling the use of UML as a 'language' in itself.

This approach discussed in this thesis is therefore a novel approach to hardware design that falls within the category of model-driven hardware design. In our approach, we have made an attempt to move further from mere design specification using UML, towards a more comprehensive and formalized model driven approach through the use of a platform independent *synthesizable* language in the form of UML with an automated route to a SystemC implementation.

We have made an attempt to *formalize* UML through the use of stereotypes, allow designers to express designs in strict high-level terms such as statecharts and activity diagrams and allow direct translation of these descriptions synthesisable SystemC code,

through the use of an automated UML-SystemC translator, thus providing a direct path from UML Model to netlist for the designer.

The contributions of this endeavour are hence threefold

1. Definition of an exhaustive UML syntax to describe hardware designs, complete with stereotypes to describe top level hardware modules, instances, signals, variables, and dynamic elements such as states, activities, and actions

2. Implementation of a model compiler to parse this high level description and translate this description into a *synthesisable* SystemC description

3. Employment of this high-level design methodology in creating actual designs

The advantages of this approach are :

1. The methodology, by requiring the designer to focus on high-level static and dynamic structures, and by helping the designer to perceive the system in terms of functional abstractions, is expected to result in more coherent and logical designs (which are synthesisable).

2. It is expected to minimise the build-and-fix loop which is the bane of structured design and help the designer document his design clearly and stick to it.

3. Elements such as statecharts and activity diagrams have always been useful ways of looking at systems. However they are rarely used in hardware design as there is no simple way to 'code a statechart', as it were, using HDLs (there are ways to code 'state machines', but they are not as expressive as statecharts). By providing a path from statechart/activity diagram to synthesis, this methodology enables the use of these structures for system design.

4. It is also expected to vastly improve the productivity of the designer by melding the erstwhile planning/design and translation/implementation phases into one step

through the use of a model compiler.

In the rest of the thesis we discuss in greater depth this UML-SystemC based design methodology. In Section 2, we discuss the various UML structures used to design hardware, the design of the UML-SystemC translator and the translation process. In section 3 we describe in detail the workings of the translator. In section 4 we draw up a summary of the designs created using this methodology with a discussion of its relative merits and demerits with respect to other methodologies and finally in Section 5 we conclude with a discussion of our experiences.

Chapter 2

UML Driven design methodology

The ASIC design methodology discussed in this thesis is part of a broader UML-based embedded system design framework. The design flow is as shown in Figure 2.1.

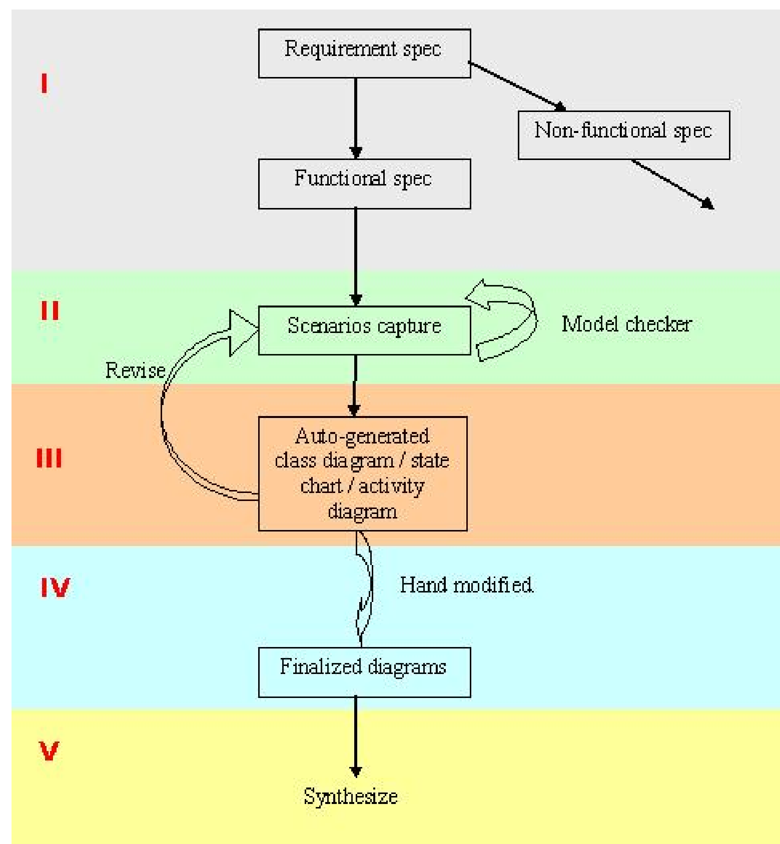


Figure 2.1: UML Based Design Flow

The flow of the design process is as follow:

1. The entire design splits into two parts from user requirements: functional requirements and non-functional requirements. This workflow is primarily concerned with the functional requirements part. There are of course lots of ongoing research in the non-functional (mainly constraints) requirements. [Region I]

2. The functional requirements are then modeled by use case modeling, which serves as a separation of functionalities of the system. [Region II]

3. Details of each use case are then captured by the scenarios involved, which are represented by sequence diagrams. This is our current research area. We aim to work out a modeling technique to completely and yet efficiently capture all possible scenarios. Embedded system designs involve concurrencies. However, there is no mature notation to model concurrencies in UML yet. So, our research also involves techniques for concurrency modeling that is convenient to use and unambiguous. [Region II]

4. Following this phase, is an automated class diagram and state-chart/activity diagram generation phase from scenarios specification. There has been some ongoing research on this area, but most of them work for simple sequence diagrams and requires quite some input from the designer before the generation. We are carefully examining their methods, building upon them and finally aim to come out with a technique that is well integrated with the scenarios modeling part. [Region III]

5. Auto-generated state-charts and activity diagrams are usually just an approximation of the target system. So, the designer will need to hand modify the generated state-charts and activity diagrams. This requires the auto-generated chart being easily understandable and well structured. This requires the syntax to be well defined and formalized. [Region IV]

6. Finally, the complete UML model is ready to be synthesized. It is passed through

the UML-SystemC translator which produces synthesizable SystemC code. Following this the code is synthesised using a SystemC synthesis tool. [Region V]

7. A similar parallel flow also exists for the embedded software with a translator generating C code from statechart descriptions serving as the tool.

8. Debugging information is provided at each stage starting from the UML descriptions to ensure syntactic correctness. Work has also been done to analyse errors in the C code and trace the error back upstream.

Research on capturing functional requirements through use cases and generating sequence diagrams, statecharts and activity diagrams is being carried out by Sun Xiaoxin. C Debugging support was added by Arun Thampi. Within this broad design framework being developed at the DSA lab, the focus of this thesis is the UML class diagram/statechart and activity diagram syntax definition, and the design and implementation details of the UML-SystemC translator tool.

2.1 Modelling styles

There are two main modelling styles employed in the design of digital systems, namely, the RTL modelling style and the Behavioural modelling style. The differences between the two can be summarized as follows:

2.2 UML constructs employed

The core principle of separating data path and control logic was adhered to. In keeping with this, two diagrams provided by UML were selected, namely 'Class diagrams' for representation of static (data path) structures, and 'Statecharts' (Harel, 1987, see) and 'Activity Diagrams' for representation of dynamic (control logic) structures.

RTL	Behavioural
An RTL model describes registers in the design and the combinational system between the registers.	A behavioural model is a high-level algorithmic description of the model.
The functionality of the system is specified in terms of a statechart and a datapath.	The functionality is specified in terms of when the inputs are read, the operations performed on the data and when the outputs are written. Due to its algorithmic nature activity diagrams are the best candidate to express these models.
The model is cycle accurate and allows the designer to specify at exactly which clock cycle each operation is performed.	The clock cycle for each operation is allocated by the compiler during behavioural synthesis. The designer can only define certain constraints.
These descriptions are less intuitive	These descriptions are smaller, easier to capture complex algorithms and easy to write and understand.
The design is rigid and doesn't allow late specification changes.	This form allows architecture exploration without modifying the overlying description and allows late changes with ease.
This method is used in the following instances: • It is easier to conceive of the design as an FSM and datapath. • The design is high-performance and hence complete control is needed over the architecture • The design uses complex memory. • The design is asynchronous.	This method is used in the following instances: • It is easier to conceive the design as an algorithm. Eg: FFT, DSP etc. • The design has a complex control flow.
Some applications where this style could be used include FIFOs, and microprocessors.	Some applications where this style is used include cellular phones, video processing applications and DSP applications.

Table 2.1: RTL and Behavioural Modelling styles

2.2.1 Class Diagrams

Class diagrams are widely used to model static software structures in object oriented systems. They contain elements to represent 'classes', 'objects', 'associations' between classes, 'composition' and 'aggregation'. These elements when appropriately defined and used can be rendered synthesizable and directly realized in hardware via SystemC. The basic building block in representing a SystemC static structure is the SC_MODULE. A SystemC module is a container in which processes and other modules are instantiated. A typical module can have: (Synopsys, 2002b)

- Single or multiple RTL processes to specify combinational or sequential logic. These processes are represented as 'operations' within a UML class. Appropriate custom stereotypes have been defined to correlate these elements to synthesizable SystemC constructs. For example, an RTL process is assigned the <<sc_method>> custom stereotype.
- Multiple RTL modules to specify hierarchy. A hierarchy can be completely described if a language provides facilities for module definition and module instantiation as is done in HDLs such as VHDL/Verilog/SystemC. In our methodology, corresponding UML structures/stereotypes facilitate this.
- One or more member functions that are called from within an instantiated process or module. These are merely defined as 'operations' within the UML class, with an added <<sc_method>> stereotype if the method is a process.

Appendix A describes the UML syntax employed in our methodology with the corresponding SystemC construct that it is mapped to. Table 1, below, lists down all the custom stereotypes used in various hardware representations and the specific UML element that these stereotypes bind to.

Stereotype	Applied To	Meaning
sc_module	classes	hardware entity
sc_instance	classes	instance of an entity
sc_signal	attributes	signal
sc_var	attributes	variable
sc_method	operations	process

Shown in Figure 2.2 is a block diagram of a Finite Impulse Response encoder core (supplied with the Synopsys Cocentric Compiler) and in Figure 2.3, the UML representation of the static structure of this module

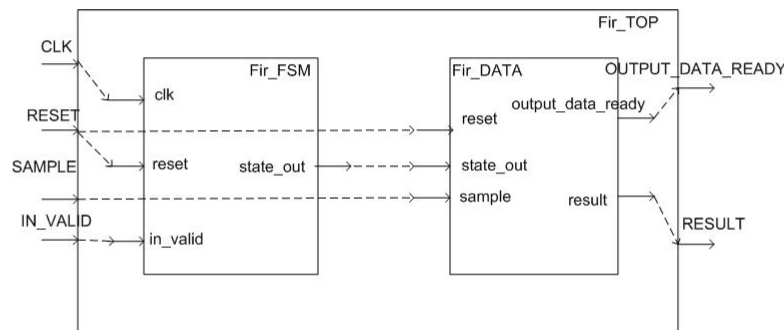


Figure 2.2: FIR Filter Block Design

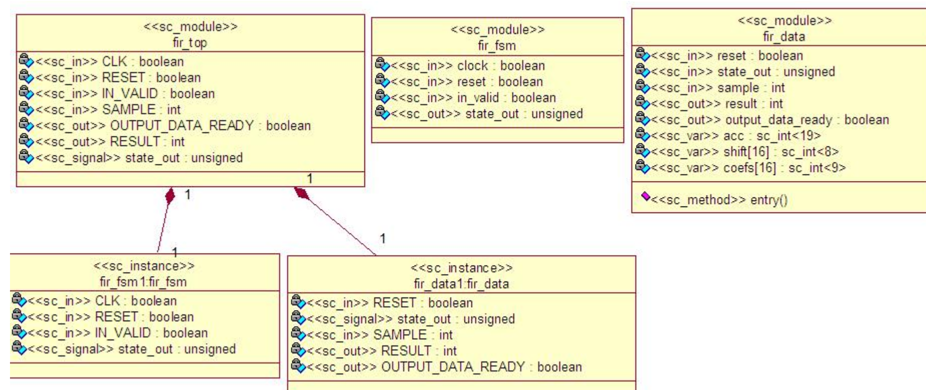


Figure 2.3: FIR Filter UML Class Diagram

As can be seen, every module template is described as a UML class with an <<sc_module>> template. Inputs are assigned an <<sc_in>> stereotype, outputs are assigned an <<sc_out>> stereotype. Variables and signals are assigned <<sc_var>> and <<sc_signal>>

stereotypes. When a module in instantiated within another module, it is assigned a composite association with the higher module as can be seen with fir_data1 and fir_fsm1, instances of the modules fir_fsm and fir_data instantiated within the fir_top module. The mapping of ports is done by order. Hence, for example, the four ports of fir_fsm1 are mapped to the I/O ports and signals listed in the <<sc_instance>> in that order.

2.2.2 Statecharts

Statecharts are used for dynamic descriptions of RTL designs. A Statechart diagram can be used to describe the behaviour of instances of a model element such as an object. Specifically, it describes possible sequences of states and actions through which the element instances can proceed during its lifetime as a result of reacting to discrete events (for example, signals, and operation invocations).(OMG, 2003, see) The semantics and notation described used in UML statecharts are substantially similar to David Harel's statecharts(Harel, 1987), with modifications made to make them object oriented. Statechart notation implements aspects of both Moore and Mealy machines, traditional state machine models thus making them apt for representation of FSM control logic in hardware. Please refer to (Drusinsky and Harel, 1989) for a convincing case for the use of statecharts in hardware design and an example of the early use of statecharts in designing hardware. In our methodology we use UML statecharts to capture the behaviour of the system.

Shown in Figure 2.4 is a statechart of a simple Serial to Parallel converter found at the receiver interface of a MAC controller.

2.2.3 Activity Diagrams

Activity Diagrams are used for dynamic descriptions of behavioural designs. An activity graph is a variation of a state machine in which the states represent the performance of

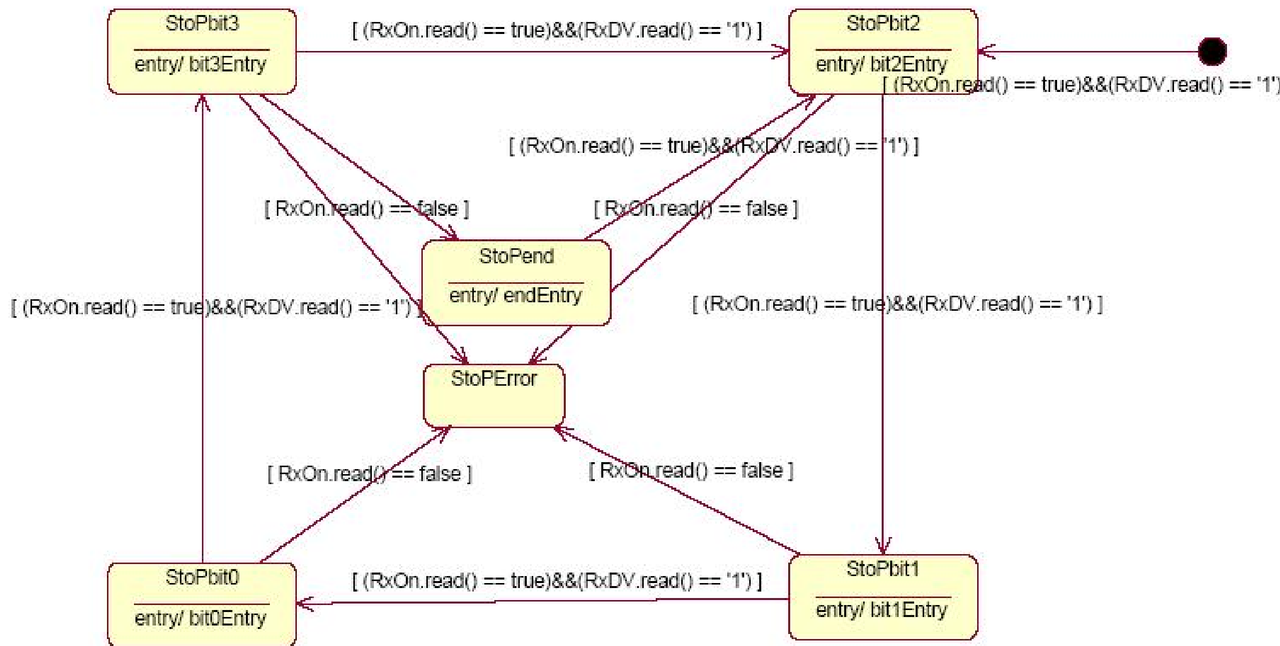


Figure 2.4: Serial to Parallel converter statechart

actions or subactivities. It represents a state machine of a computation itself. (UML). Activity Diagrams are used in situations where all or most of the events represent the completion of internally-generated actions (that is procedural flow of control)

Activity diagrams are widely used to capture procedural flow in business situations, and are increasingly being used to describe software. The use of activity diagrams to represent hardware is likely to increase in importance with the increasing use of hardware to implement complex algorithms that cannot be captured using statecharts. The syntax used in describing the algorithm are shown below:

Stereotype	Applied To	Meaning
for	decisions	for loop
if	decisions	if control structure
while	decisions	while loop
task	activities	activity between empty clock cycles
run	activities	activity continuing from past to next activity
start	activities	activity starting with empty clock cycle
end	activities	activity terminating in empty clock cycle
function_decl	activities	function declaration

The activity diagram can capture the functionality of the system to varying degrees of detail. The designer first starts with a broad picture of the system depicted primarily through 'high-level' tasks that hide details. One can then progressively refine this design to lower levels of detail. Most UML design tools like Rational Rose ©offer the designer the option to write out text attached to each symbol. Therefore, when describing low level tasks, the designer can make a choice as to whether he wishes to write out the details in code, or describe these details by further detailing out the UML diagrams. A compromise is struck between comprehensibility of the activity diagram and detail.

Either way, the synthesisabilty of the activity diagram through the UML-SystemC translator offers the designer a smooth transition from the algorithm to final implementation. It also provides a comprehensible UML description of a complex algorithm by hiding minor coding details behind a broad algorithmic picture. Shown in Figure 2.5, is the activity diagram describing a Fast Fourier Transform module supplied with the Synopsys Cocentric compiler, at a fine level of detail.

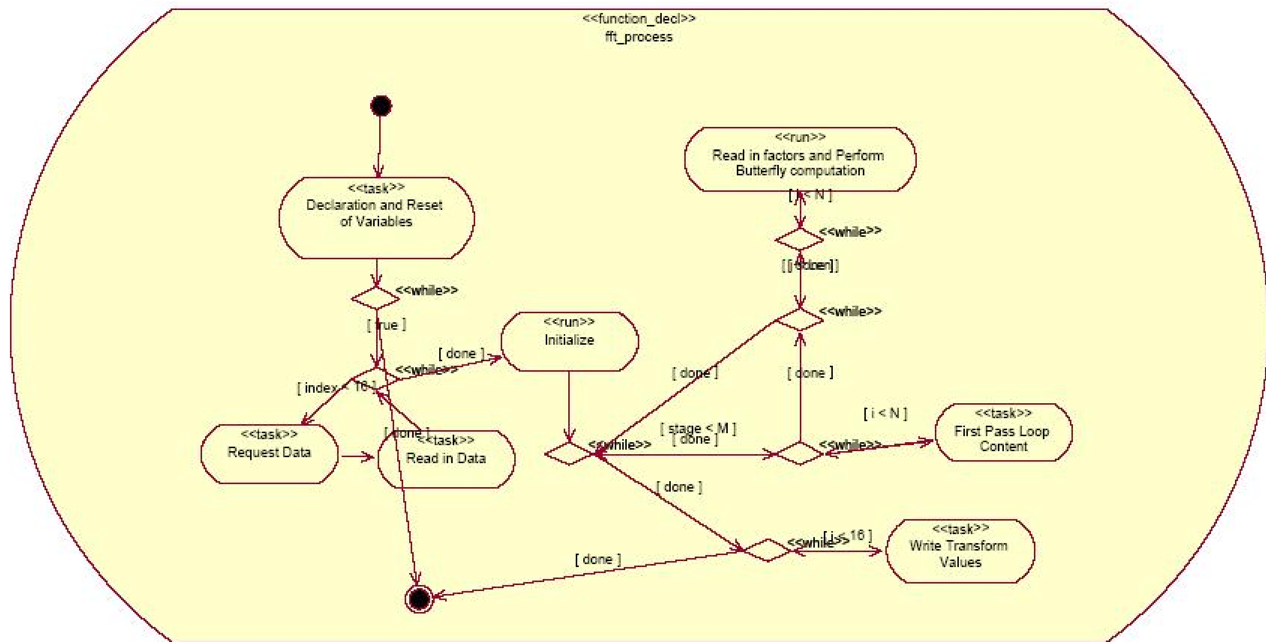


Figure 2.5: FFT Activity Diagram

Chapter 3

The UML-SystemC Translator

The UML-SystemC translator is the key enabler of this design methodology. The software was implemented in Java and intended to fulfil the following objectives:

- Provide a direct translation from the UML to SystemC strictly following the syntax mapping rules detailed in Tables 1, 2 and 3.
- Behave as a 'UML compiler' of sorts by identifying errors and possible hazards, such as incomplete module hierarchies, use of undeclared signals/variables/data types, incorrect port mapping, contradictory state transitions, isolated states, invalid states etc. thus giving the UML model an exhaustive, formal and consistent shape that would translate into error free SystemC code.
- Providing a faster path from conceptualisation to synthesis.
- Generate efficient SystemC code

The software can be divided into two sections, one engaged in parsing the class diagrams and the other in parsing the Statecharts and Activity Diagrams. Shown in Figure 3.1 is the flow of the translation.

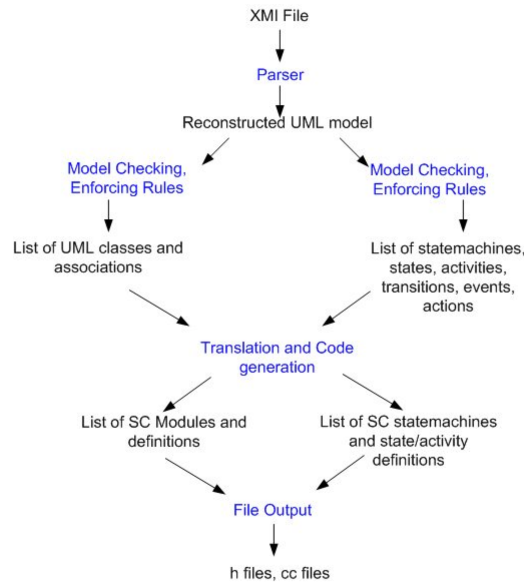


Figure 3.1: UML-SystemC Translation Flow

3.1 Class Parsing and code generation

The following is a summary of the steps involved in parsing the class diagram to generate the modules and module hierarchy.

- The XMI file is built into a 'Document' which is fed as input into the classparser
- The classparser builds a UML model from the data extracted.
- All the data types, modules, instances, ports, signals, variables etc are then extracted from this UML model and verified. Instances are mapped to the appropriate signals.
- The module hierarchy is constructed.
- In the code generation phase, the code generator analyses the static structure and generates SystemC SC_MODULEs and instances with their respective attributes and operations.

Given in Figure 3.2 is a class diagram and in Figure 3.3, the code generated.

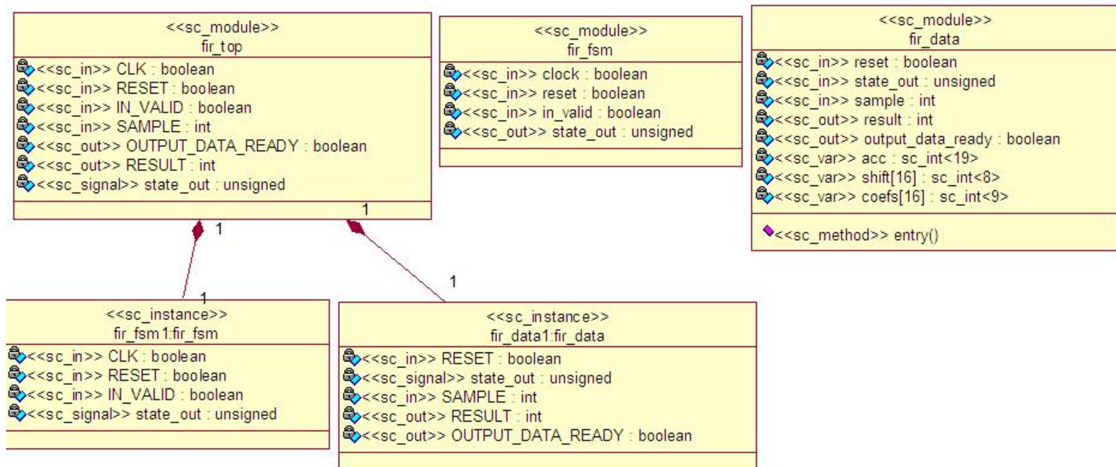


Figure 3.2: FIR UML Diagram

UML to SystemC Translator

File Tool Help

VHDL

UML
fir_rose_xmi10.xmi
SystemC
main.cpp
fir_data.h
fir_data.cpp
fir_fsm.h
fir_fsm.cpp
fir_top.h
fir_top.cpp

fir_rose_xmi10.xmi | main.cpp | fir_data.h | fir_data.cpp | fir_fsm.h | fir_fsm.cpp | fir_top.h | fir_top.cpp

```
SC_MODULE (fir_top) {
    // Module port declarations
    sc_in<bool> CLK;
    sc_in<bool> RESET;
    sc_in<bool> IN_VALID;
    sc_in<int> SAMPLE;
    sc_out<bool> OUTPUT_DATA_READY;
    sc_out<int> RESULT;

    // Internal signal variable declarations
    sc_signal<unsigned> state_out;

    // Data variable declarations

    // Member function declarations

    // Process declarations

    // Sub-module declarations
    fir_fsm *fir_fsm1;
    fir_data *fir_data1;

    // Module constructor
    SC_CTOR (fir_top) {

        // Initialise variables

        // Sub-module instantiations and port bindings

        fir_fsm1 = new fir_fsm("fir_fsm1:fir_fsm");
        fir_fsm1->clock(CLK);
        fir_fsm1->reset(RESET);
        fir_fsm1->in_valid(IN_VALID);
        fir_fsm1->state_out(state_out);

        fir_data1 = new fir_data("fir_data1:fir_data");
        fir_data1->reset(RESET);
        fir_data1->state_out(state_out);
        fir_data1->sample(SAMPLE);
        fir_data1->result(RESULT);
        fir_data1->output_data_ready(OUTPUT_DATA_READY);
```

Translation successful

Start 7:43 PM

Figure 3.3: FIR SystemC Code produced by Translator

3.2 Statechart parsing and code generation

This proceeds in a similar way. Various elements, namely states, transitions and actions are identified in the XMI file and extracted into a UML model. Each statechart is associated with a particular class/sc_module. In the code generation phase, firstly, all the composite states in the statechart are identified. Note that each composite state corresponds to one Mealy state machine.

SystemC state machines are implemented in the following ways: (Synopsys, 2002b)

- An SC_METHOD process for updating the state vector and a single common SC_METHOD process for computing both the output and the next state logic
- An SC_METHOD process for updating the state vector, an SC_METHOD process for computing the output logic, and a separate SC_METHOD process for computing the next state logic.
- A Moore machine with a single process for computing and updating the next-state vector and outputs

In our methodology, the Statechart is analysed, all state machines are identified, and each state machine is represented using the second method. State changes are assumed to be triggered by signal changes. 'Actions' can be specified at the entry into or exit out of a state and also along specific state transitions. These are translated into functions. Appendix A describes in detail the UML syntax used and the corresponding SystemC construct that it is mapped to.

3.2.1 Basic translation Schema

The conventional state machine model consists of a set of states, events (interpreted as signals), transition functions and output functions. Transitions between states are labelled in as 'Event[Condition]/Action'. 'Event' is a the Boolean event occurring on a

signal. '[Condition]' is a Boolean combination of events and is carried into the SystemC code verbatim. An additional semantic has been defined in our method, namely the use of the "!" operator before the event to signify "if (Event.read() == false)".

An action is a function call. Once again, the translator supports an additional semantic here, namely the use of a ";" between function calls to call more than one function. So "Event[condition]/action1();action2()" will translate into two function invocations (action1(); and action2();) in SystemC. Statements associated with the function can also be listed out if the UML tool permits. (In our case, ArgoUML allowed for this.) Finally, associated with every state, there is one entry action, one exit action, and potentially several actions associated with internal transitions. The rules for translation of these are similar to other actions.

Initially, the system starts from an Initial state, which is a pseudostate and does not translate into an actual state in the translation in our implementation.

3.2.2 Compound Transitions

By definition, a transition connects exactly two vertices in the state machine graph. However, since some of these vertices may be pseudostates-which are transient in nature-there is a need for describing chains of transitions that may be executed in the context of a single run-to-completion step (OMG, 2003). Such a transition is called a compound transition.

Two or more transitions emanating from the same junction point represent a static branch point. The semantics of static branches is that all the outgoing guards are evaluated before any transition is taken. In our notation, the choice point is interpreted to be a dynamic choice point. In this case , the guards of the outgoing transitions are evaluated at the time the choice point has been reached. The value of these guards may be a function of some calculations performed in the actions of the incoming transition.

Conditions as described earlier are Boolean expressions. There is no exact UML semantic to express these conditions. The assumption is that the users will input a condition that can be evaluated successfully by a C++ compiler.

3.2.3 Hierarchy

Statecharts contain three kinds of states, namely, basic states, and composite states which can be further sub-classified as 'OR-states', i.e exclusive states and 'AND-states', i.e concurrent states. All states mentioned in the previous section were basic states. Basic states are those that have no sub (child) states, i.e. they form the leaf nodes of the tree. OR-states have child states. In our method, an AND state is assumed to comprise of two or more OR states executing concurrently. The following figure, illustrates a statechart with OR-state. In this figure, state TOP is an OR-state consisting of three sub-state P1, P2, and P3 which are called basic states.

The algorithm employed to translate a hierarchical statechart is as follows:

Firstly all composite states (AND as well as OR) are identified by running through the tree recursively. Each composite state corresponds to one state machine. The enumerable states of each state machine are defined as those states that are one level below the composite state that the state machine is associated with. The OR states that execute concurrently are clubbed together into a concurrent state.

The algorithm then proceeds something like this:

```
for all statemachines identified
    for all enumerable states of this statemachine
        while (state.getSuperState() != NULL)
            Append_Next_State_statement(state);
            State = state.getSuperState();
```

By this, each state is made aware of events leading out of itself as well as all of its superstates in the hierarchy. This is an implicit flattening of all states to basic states.

Append_Next_state_statement(state) is carried out as follows:

```
for all transitions leading out of state
    FindActiveState(s)AfterTransition(transition); //In case of AND states usu. many
for all active states
    for all state machines in system
        if (active state lies within state machine) //at any level below
            Find_and_Append_next_state_of_state_machine;
        else if (active state == state machine compo. state)
            Set_next_state_of_state_machine_to_init/hist_state;
        else if (active state is superstate of state machine) //any lvl above
            Find_and_Append_next_state_of_state_machine;
        else
            Set_state_machine_next_state_to_idle;
```

In this function, there are other issues like contradictory transitions and history states that are tackled. There are other subtle minor issues that are also dealt with but not detailed in the above pseudocode.

3.2.4 History States

Statecharts feature two kinds of history connectors (pseudostates): shallow history and deep history. Any form of history introduces an additional element of complexity to the translation effort. It is dealt with in the following way. Each state has a hasHistory attribute. Each translated SystemC statemachine has a history_state variable associated with it that stores the last state that this particular state machine was in. Thus at the

exit of every state, it history_state variable is re assigned.

Also when history states are introduced into a statechart, a few additional details have to be worked into the previous algorithm to find the next_state that each statemachine must move into.

Every composite state is assigned a state machine and a history state. Every time the system enters a state the history state variable of that state machine is set to this state. With reference to the following figures, let state "Wait" be the active state in consideration. When triggered by the in_valid event the state machine "Top" now moves into the "Active" state and the state machine "Active" moves into its history state as seen in the translated code shown in Figure 3.5 illustrating the "next_state" behaviour of the top state machine from Figure 3.4.

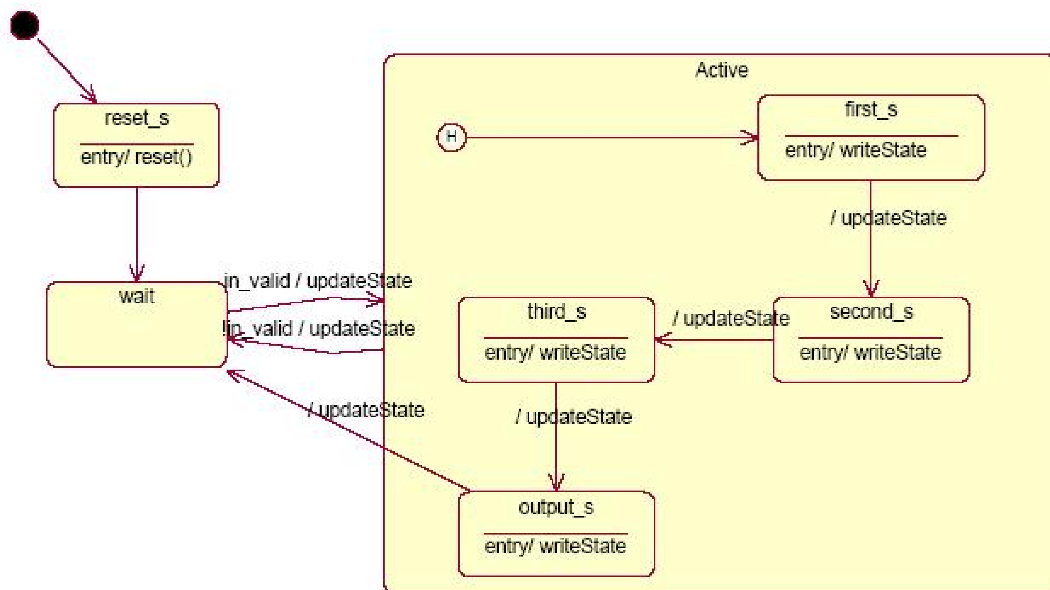


Figure 3.4: FIR Statechart diagram Diagram

3.2.5 Concurrency

Concurrency in statecharts is represented in the form of AND-states. All concurrent states in the AND-state are assumed to be composite states with independent statemachines which are simultaneously triggered when the AND-state is entered.

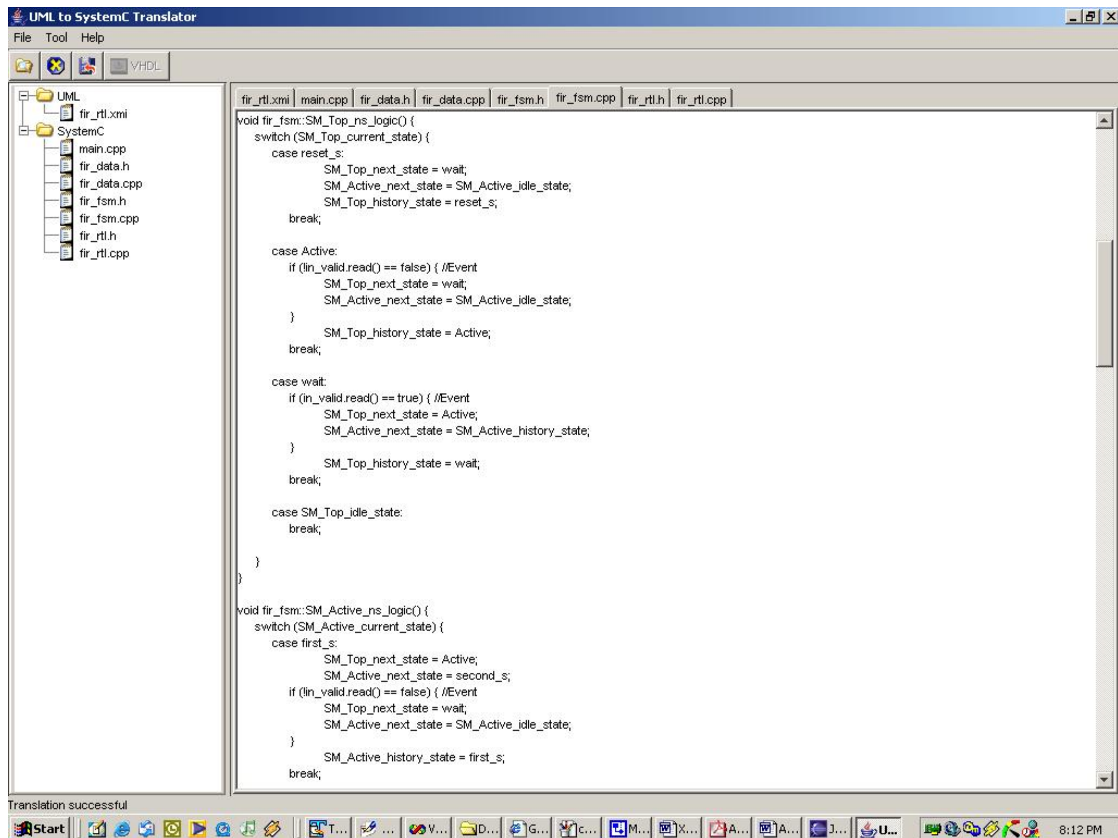


Figure 3.5: FIR Statechart SystemC Code produced by Translator

Users are constrained to introduce concurrency into statechart through the use of two pseudostates, namely fork and join. This is because, 1. Using fork and join pseudostates can define all the semantics of concurrency in a statechart. 2. In typical UML drawing tools (Visio, ArgoUML and Rational Rose), users have to user fork and joint notation to express the concurrency. There is simply no AND-state equivalent item.

Thus, concurrency actually implies dealing with fork and join pseudostates. A fork state has only one incoming transition and several outgoing transition; a join state has more than one incoming transition and only one outgoing transition.

In the following figure, when the fork state is reached, it means that the four states that the fork state branches to are the 'target' or 'active' states. All the related state machines are then triggered and operate orthogonally and concurrently as desired. When a join state is detected, the next_state algorithm ensures that the previous state machines move into idle states because the target state of the join state is the new active state

and has no connection with the earlier concurrent states. Given in Figure 3.6, is an example, and the resulting code in Figure 3.7.

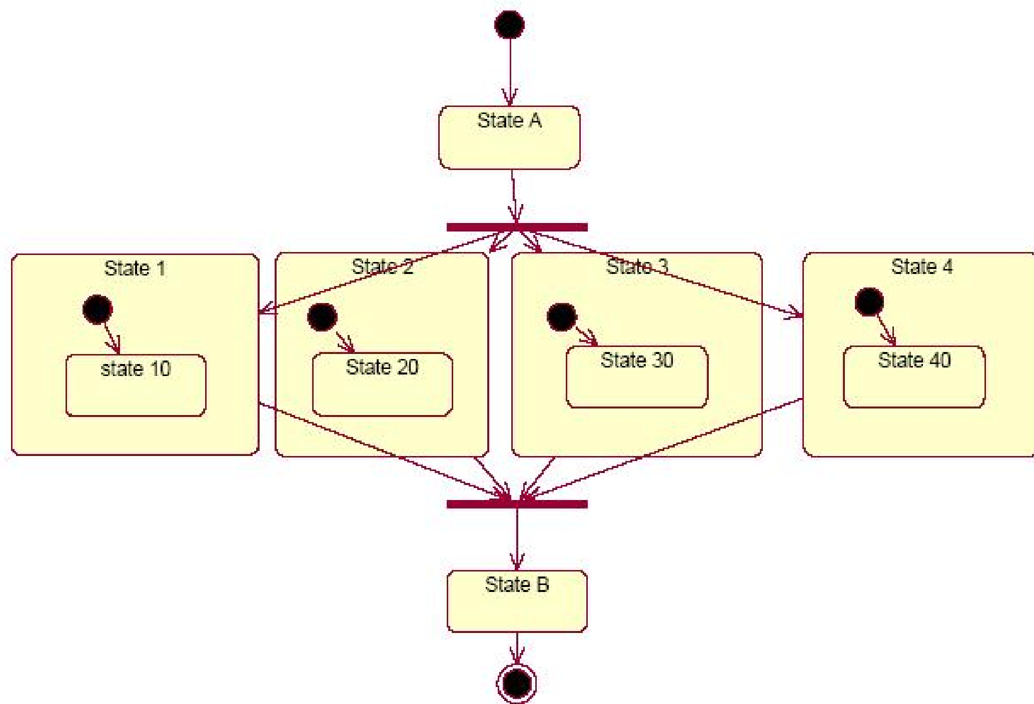


Figure 3.6: Statechart illustrating concurrency

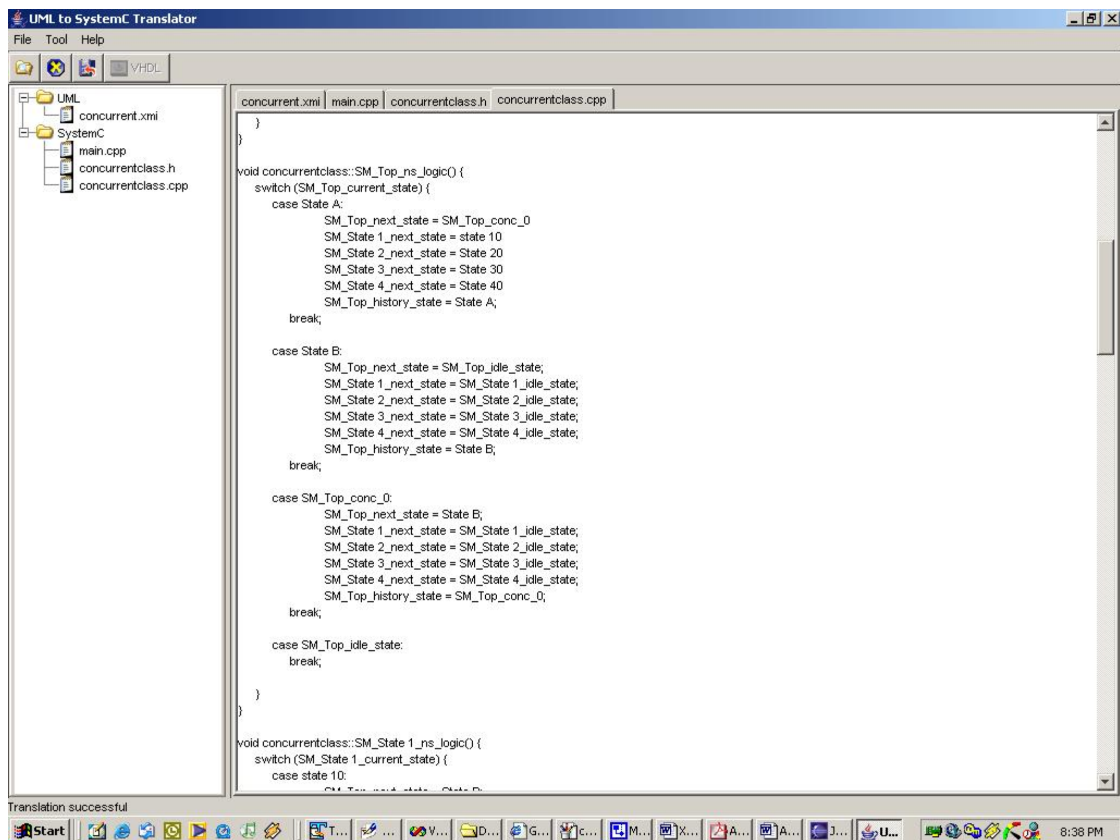


Figure 3.7: Statechart with AND-states : SystemC Code produced by Translator

Chapter 4

Case Studies

The UML driven methodology described thus far was employed in creating a number of different designs in UML and synthesising them. Starting from the requirement specifications and an understanding of the static structure and states/algorithms that would be needed to implement the system, a preliminary class diagram and statechart / activity diagram were first created. These was progressively expanded in detail until a point was reached where the description was detailed enough to capture the structure of the entire system and yet sparse enough to avoid unnecessary cluttering through inclusion of very low-level details. This section illustrates the design process in action starting from the initial stages to final synthesis results.

4.1 Experimental Tools

Various software tools were employed in the entire flow. Brief descriptions of these tools follow:

4.1.1 ArgoUML

ArgoUML is a powerful yet easy-to-use interactive, graphical software design environment that supports the design, development and documentation of object-oriented soft-

ware applications (see www.argouml.org for more details).

ArgoUML is widely used by software designers & architects, software developers, business analysts, systems analysts and other professionals involved in the analysis, design and development of software applications. Its main features include:

- Support for the following open standards: XMI, SVG and PGML
- 100% Platform independent due to the exclusive use of Java
- Open Source, which allows extending or customizeing.
- Cognitive features like: reflection-in-action, opportunistic design, comprehension and problem solving

4.1.2 Rational Rose Enterprise Edition 2002

The Rational Rose product family is designed to provide the software developer with a complete set of visual modeling tools for development of robust, efficient solutions to real business needs in the client/server, distributed enterprise, and real-time systems environments.(see Quatrani, 2002) Rational Rose products share a common universal standard, making modeling accessible to nonprogrammers wanting to model business processes as well as to programmers modeling applications logic.

The tool provides the capability to

- Identify and design business objects, and then map them to software components
- Partition services across a three-tiered service model
- Design how components will be distributed across a network
- Generate Visual Basic code frameworks directly from your model
- Use reverse engineering to create models from existing components and applications

4.1.3 The UML-SystemC Translator

This tool was described in detail in the previous section. It is essentially a UML Model Compiler. It is a Java based platform independent tool that accepts XMI 1.0 descriptions of UML designs (Both the UML tools listed above have the capability to export to XMI format) and translates it to synthesisable SystemC code. The tool also has a user friendly GUI and is compatible with XMI outputs generated by ArgoUML and Rational Rose.It can broadly be split into a parser and a code generator.

4.1.4 Cocentric SystemC Compiler

Once the designs are created in UML and passed through the translator, we obtain synthesisable SystemC files of the modules. Synopsys' Cocentric compiler was chosen to take over the process from here on. CoCentric^{tm} SystemC Compiler synthesizes hardware from SystemC source code. It is the ideal tool for design teams needing a fast and high quality path from a system level hardware description coded in SystemC to gates or a synthesizable Verilog or VHDL RTL description. The designer then continues adding more detail until the HDL code can be synthesized into a gate-level netlist for IC implementation. For more information refer to www.synopsys.com.

The important settings in the compiler are

1. The Technology Library which specifies which ASIC technology will be used to synthesize the design

2. The Synthetic Library, which is a technology independent library of synthetic components such as adders and multipliers. Each synthetic component from this library is implemented using the primitive gates from the technology library.

3. The Map Effort which specifies the tradeoff between the speed of synthesis and the efficiency. Low effort produces the fastest synthesis.

4. The IO-mode option specifies how the compiler must handle IO operations while performing the synthesis. There are two modes, cycle fixed mode where there is no difference in the I/O behaviour of the behavioural description and synthesized design, and superstate fixed mode where the logical relationships of read and write operations are preserved but the compiler is allowed to add clock cycles to lengthen the time between I/O operations.

5. The hlo_resource_allocation variable which determines what factors the compiler takes into account when allocating operators.

4.2 Design Implementation

The list of designs created using the UML driven methodolgy is shown in Table 4.1. As can be seen a total of six designs were chosen to illustrate the design methodology. Three designs were RTL designs and three were Behavioural. (For a discussion on the differences between the two modelling styles please refer to section 3.1)

Of the RTL designs, the three designs chosen were a JPEG Encoder, a Medium Access Controller, and a Finite Impulse Response filter. The JPEG encoder was chosen for the intricacy of its static structure. The MAC was chosen to illustrate the design of control structures using UML.

Of the Behavioural designs, a VP3 video Encoder, an FFT module and a simple FIFO were designed. The VP3 encoder is a complex design and pushes the limits of this approach. The algorithms for every module of this design were taken from the open-source vp3 source code created by On2 Technologies. These were then presented as activity diagrams to the translator. The FFT, another complex algorithm was also chosen to illustrate the expressiveness of activity diagrams.

In the next section, we look at each design for its expected functionality, how it was

RTL Designs	Behavioural Designs
JPEG Encoder	VP3 Encoder
MAC Controller	FFT
FIR	FIFO

Table 4.1: Designs Created

approached, tested and implemented and what tangible results were obtained.

4.2.1 JPEG Encoder

The JPEG Encoder is an excellent example of a structurally and functionally complex design. It is based on a design by Richard Herveille. To understand the functionality of this system, we start with a use case diagram in Figure 4.1. This diagram captures, in lay terms, all the actions that the external actor (in this case, a testbench) can perform on the system. This helps us to arrive at the external interface for the module. Once this has been decided, we proceed to the next level of granularity, namely how to convert the input data into compressed jpeg data.

The design can be conceived as consisting of three main blocks, namely, the DCT block, Quantization and Rounding Block and the Run Length Encoding. Each of these blocks is composed of sub-blocks which are further composed of sub-blocks and so on as shown in Figure 4.1. The interaction between the various modules is controlled by small statecharts within each block, which defines the states that the block can be in and how it makes transitions between those states.

Figure 4.2 shows a section of the design.

As can be seen, there are three top-level classes with stereotype "<<sc_module>>". These classes are templates for these modules. In the JPEG design, the DCT is effectively composed of 8 X 8 DCTU modules. Each DCTUB module contains 8 DCTU 'instances'. Note the by-order port mapping. U and V are indicators of the row and column of the pixel that particular module processes. Each DCTU contains a "dct_cos_table"

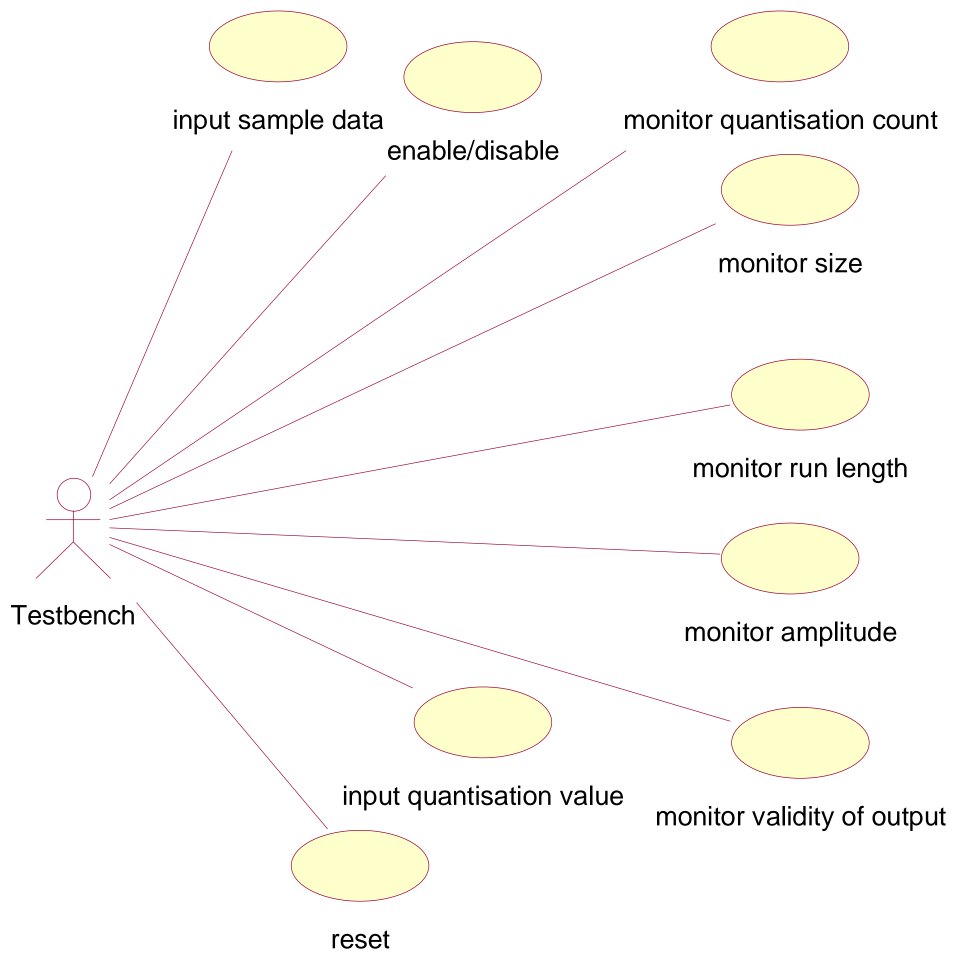


Figure 4.1: Use case diagram for the JPEG encoder

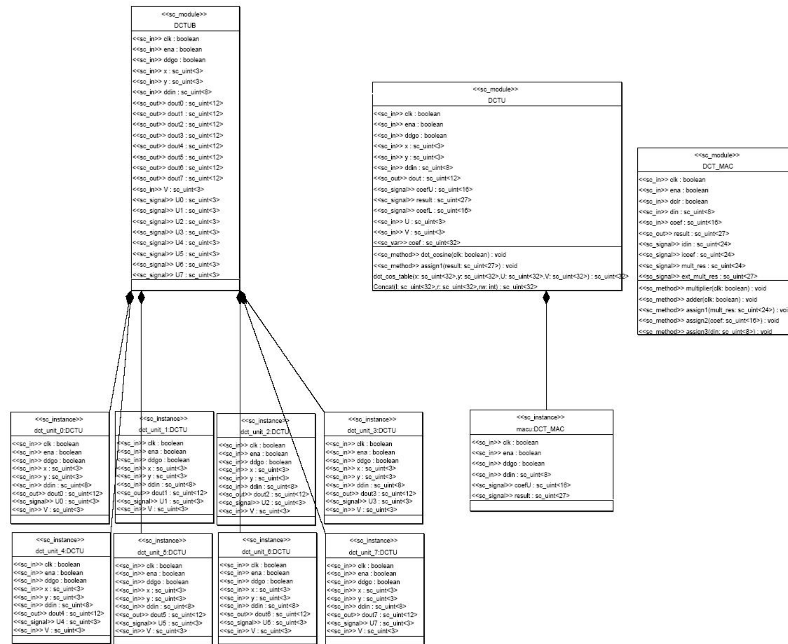


Figure 4.2: A Section of the JPEG encoder design

to select the coefficient to multiply the data. It supplies the coefficient and data to the DCT_MAC unit, which then performs the multiplication. As the tasks described above are mainly data flow type of tasks, there is no statechart required.The other section of the design requires a statechart for coordination, namely the run length encoder. Refer to Figure 4.3, Figure 4.4 and Figure 4.5 for the class diagram and the statecharts associated with this design.

The Run Length Encoder consists of the rle1 block, which maintains a count of blocks received and whether they are AC or DC coefficients. It takes the data input and assigns value to the size, run length and amplitude. The four rzs blocks are present to implement run lengths greater than 15. As can be seen in the statechart of rle1, it can exist in two states 'DC' and 'AC' to signify the type of coefficient being encoded. Since the first coefficient is DC note how the run length and size are reset. When the 'go' signal is received, it informs the next unit that a DC term will follow and enables the data. It then remains in AC state for the next 63 coefficients when it sends out the rlen, size and amplitude values until the count of 63 is reached where cnt_done is triggered

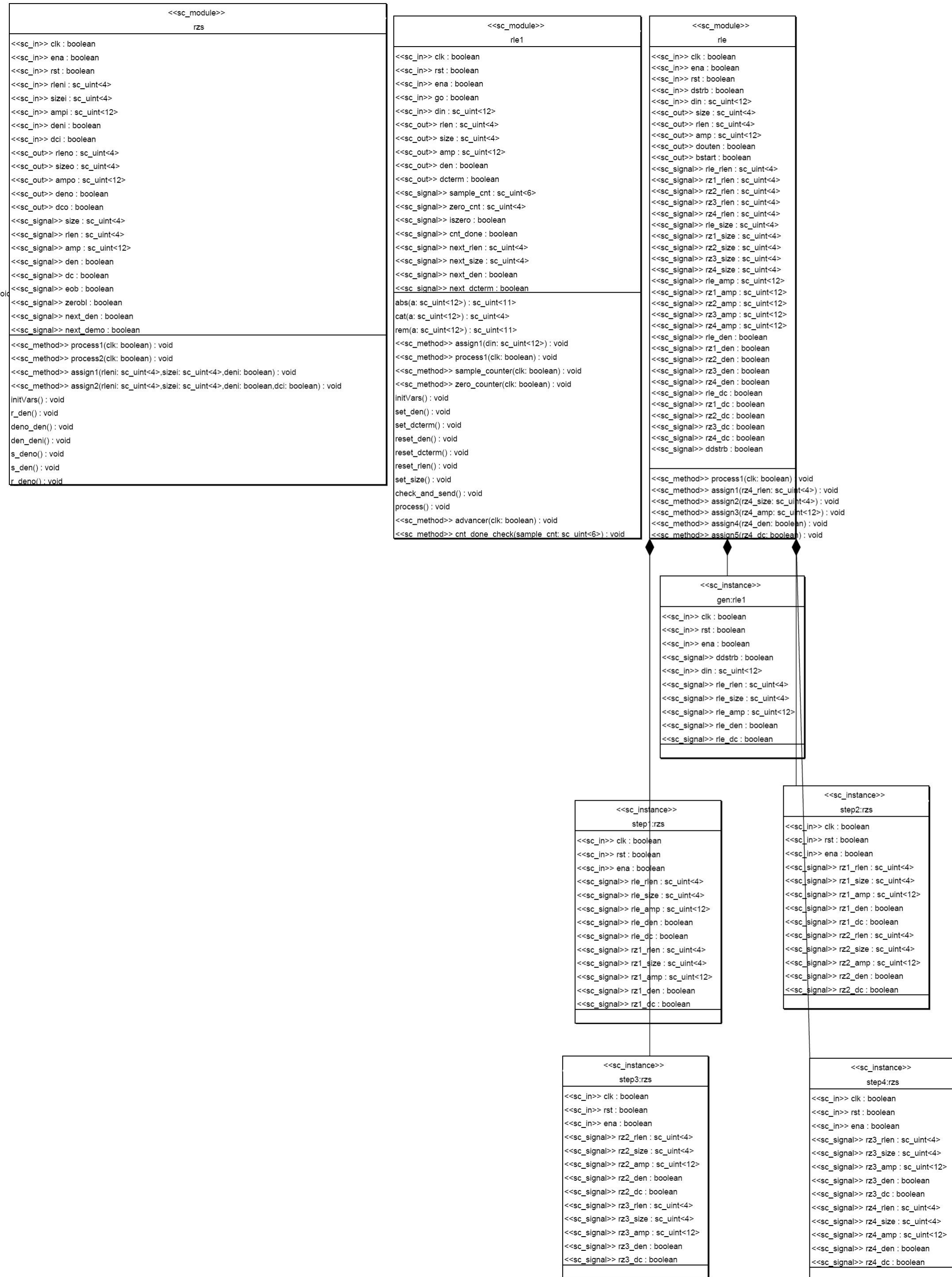


Figure 4.3: The Run length encoder of the JPEG encoder

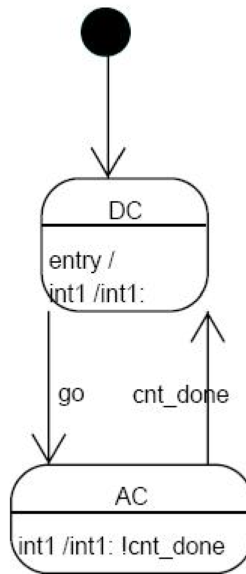


Figure 4.4: Statechart of the rle1 unit

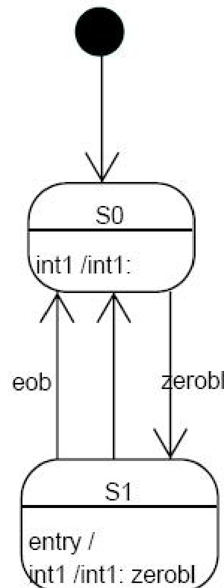


Figure 4.5: Statechart of the rzs unit

Setting	Value
Technology library	tc6a_cbcore.db
Synthetic library	dw01.sldb, dw02.sldb
Map Effort	low
Clock Period	10 ns

Table 4.2: JPEG Encoder synthesis settings

and the statechart goes back to the DC state. The rzs blocks have two states S0 and S1. When a run length of 15 and a size of 0 are received(symbolizing a run length longer than 16) the statechart goes to state S1 and stays in S1 as long as this is true until it finally processes the EOB block. The 4 rzs blocks handle upto 3 sequences of the special symbol as per the specification requirements.

The JPEG encoder UML design was run through the UML-SystemC translator. The lower level details were filled in and simulations were run. Shown in Figure 4.6 is a section of the simulation waveform. The output was monitored and verified to be correct.

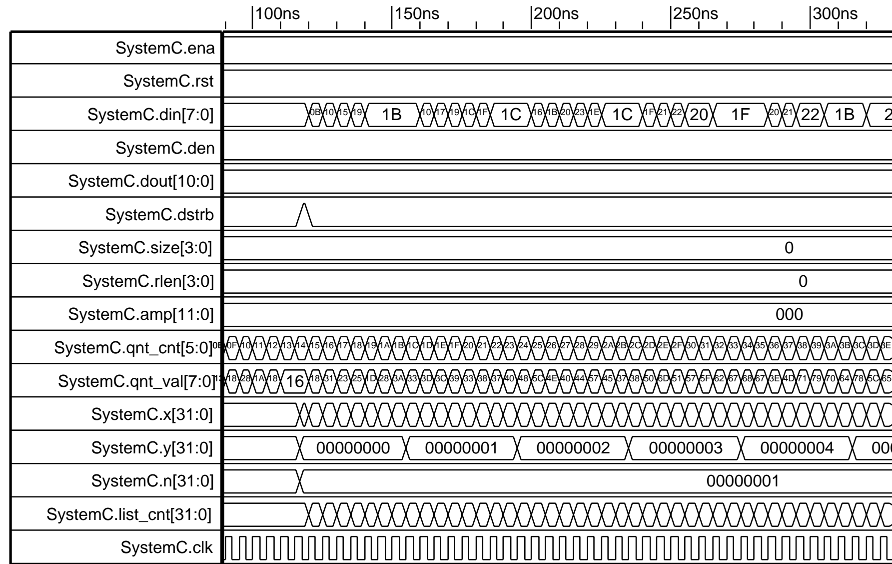


Figure 4.6: Simulation waveform of JPEG encoder

The design was later synthesized with the settings shown in Table 4.2

The results are shown in Table 4.3. The slack was met. The final gate count shown in the table includes flip flops and latches and is around 98 K. This is about 98% of the

Module	ports	nets	cells	refs	comb. area	non-comb. area	net inter-con area	total cell area	total area	gate count
DCT (UMLSC)	35	5743	5709	148	11366	458	62941	11825	74766	96000
QNR(UMLSC)	42	108	57	11	2648	4068	15855	6717	22572	1655
RLE(UMLSC)	38	127	6	3	278	220	11801	498	12300	819
Commercial JPEG2K_E core(Alma Technologies S.A.) (ASIC 0.18 process)	-	-	-	-	-	-	-	-	-	100K

Table 4.3: Results obtained from synthesis of JPEG Encoder

area occupied by the commercial design and is smaller than a commercially available JPEG core mainly because this core is minimal and has been designed in RTL style. The commercial core runs at 150 MHz.

4.2.2 MAC Controller

This design was based on the Xilinx “Ethernet Media Access Controller (EMAC) Specification” and a MAC Controller design from Infineon TC11lB Peripheral Units User Manual, which implements the IEEE 802.3 and operates at 100 Mbps or 10 Mbps. The use case diagram for this design is shown in Figure 4.7.

From this specification we arrive at a design with a transmit and a receive section. The transmit block consists of three further sub-blocks namely a Transmit FIFO, Core and Parallel to Serial Converter. The receive block consists of a Serial to Parallel Converter, a Core and a Receive FIFO. Shown in Figure 4.8 is a section of the UML design, namely the receiver. As can be seen the top level module is the RxBlock. This Block is Composed of three instances RxFIFO1, RxCore1,and StoP1 of modules RxFIFO, RxCore,and StoP respectively. Defined in the diagram also are the inputs, outputs and internal signals of the top level module and the by-order port-mapping for the lower level instances.

The Transmit and Receive blocks have been designed in contrasting styles to illus-

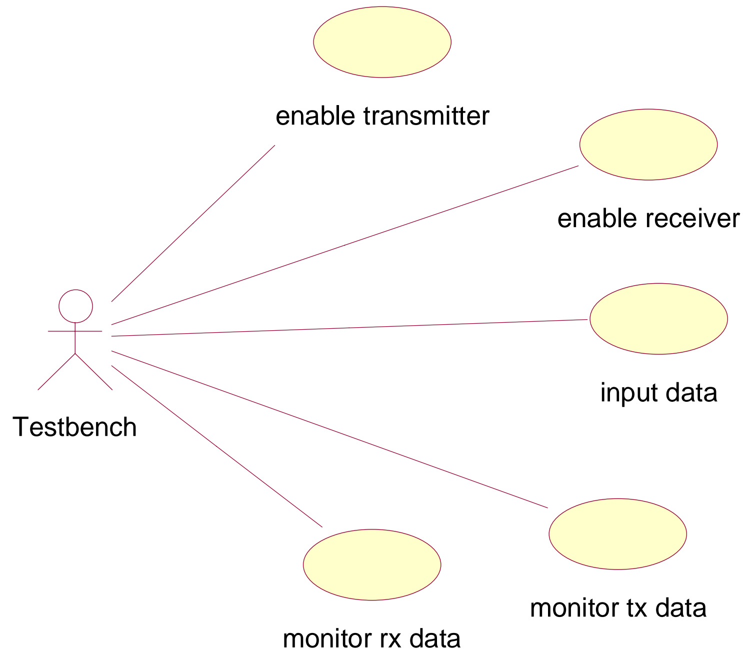


Figure 4.7: Use Case Diagram of MAC Controller

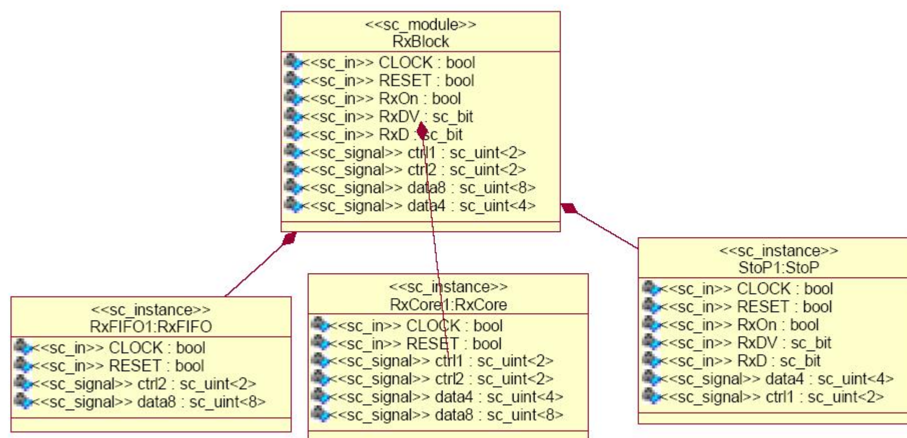


Figure 4.8: Class Diagram of Receiver Block

trate two different approaches. In the transmit block, the three low level instances are passive components that have functions and processes defined within them that activate upon certain events. There is a central state machine that runs in the top level module TxBlock which controls the passive datapath consisting of the TxFIFO, TxCore and PtoS. Shown in Figure 4.9 is this statechart. The signals triggered by this statechart activate functions within the desired low level module.

On the other hand, the receive block does not have a central state machine. The RxBlock is merely a passive top level module that is composed of active low level modules RxFIFO, RxCore and StoP, each of which has a small state machine running within it. Shown below is a statechart of the RxCore. As can be seen there are five main states namely the Pre, Header, Data, CRC and End states and one Error state. There is an activity that performs processing in each of these states, after which the data is forwarded to the next module which parallelizes the data. The design was translated to SystemC and simulated and verified to be correct. The waveform captures the transmission and successful receipt of an ethernet packet. The design was then synthesised using the settings in Table 4.4. The results are as shown in Table 4.5. An more comprehensive open source, Ethernet MAC controller performing similar activities including Frame Data Encapsulation and Decapsulation, Frame Transmission, and Frame Reception occupies about 28k gates and 2400 flip-flops. (The opencores design incorporates numerous other features such as a Wishbone interface etc and a Management module for a standard IEEE 802.3 media independent interface etc and is hence larger.) Our minimal MAC controller is therefore comparable to a bare bones design designed through other methods.

Figure 4.9: Statechart of the Transmitter Block

Setting	Value
Technology library Synthetic library Map Effort	tc6a_cbcore.db dw01.sldb, dw02.sldb low

Table 4.4: MAC synthesis settings

Module	ports	nets	cells	refs	comb. area	non-comb. area	net inter-con area	total cell area	total area	gate count
MAC Controller	16	16	2	2	12730	7303	79472	20034	99506	8k
Ethernet MAC (open-cores.org)	-	-	-	-	-	-	-	-	-	28k

Table 4.5: Results obtained from synthesis of MAC Controller

4.2.3 FIR Filter

The FIR filter design is an enhanced version on the Synopsys FIR design supplied as an example with the CoCentric compiler. The basic structure is shown in Figure 2.2. As is apparent it is a hierarchical RTL module which contains an statechart inside the fir_fsm module and a datapath inside the fir_data module.

Let us now see how one goes about designing this filter. We first start with the use case diagram as seen in Figure 4.10.

Given these requirements, we firstly define the static structure. It is very simple, comprising an FSM and a Datapath. The class diagram illustrating the static structure is shown in Figure 4.11:

Now for the dynamic sections. The module fir_fsm, containing the state machines can be expressed very well using statecharts. The datapath is passive and performs certain fixed activities when instructed, namely multiplying the input signal samples with the filter coefficients. It is therefore an excellent candidate for the use of activity diagrams. Activity diagrams where therefore used to flesh out the design of the datapath and illustrate the process flow in detail. Very Low level systemc details were filled in

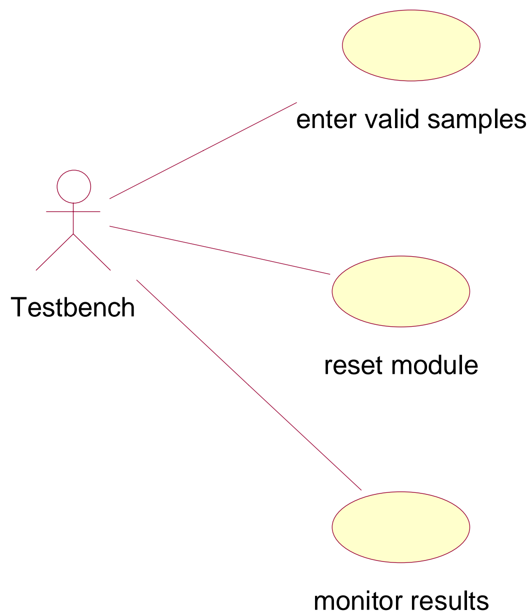


Figure 4.10: Use Case Diagram of FIR filter

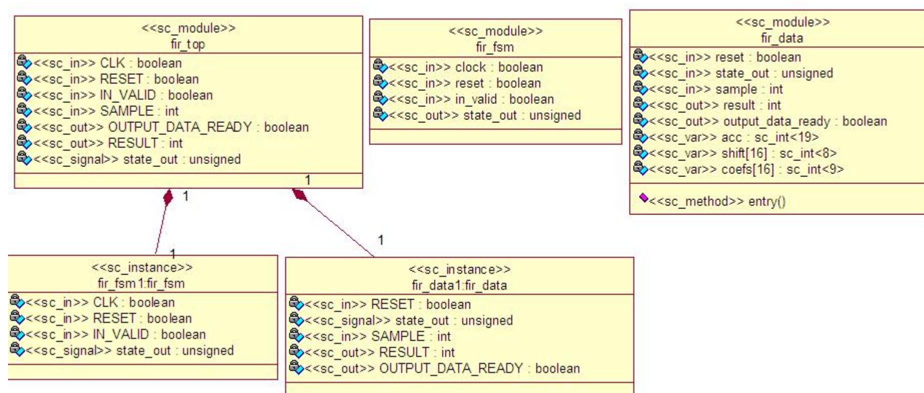


Figure 4.11: Class Diagram of the FIR Filter

Setting	Value
Technology library	tc6a_cbcore.db
Synthetic library	dw01.sldb, dw02.sldb
Map Effort	low

Table 4.6: FIR synthesis settings

later as the purpose of this design methodology is primarily to design the control system and data flow for very large and complex systems where low level details are of secondary importance. Shown in Figure 4.12 and Figure 4.13 is the statechart within the fir_fsm module and the activity diagram within the fir_data module.

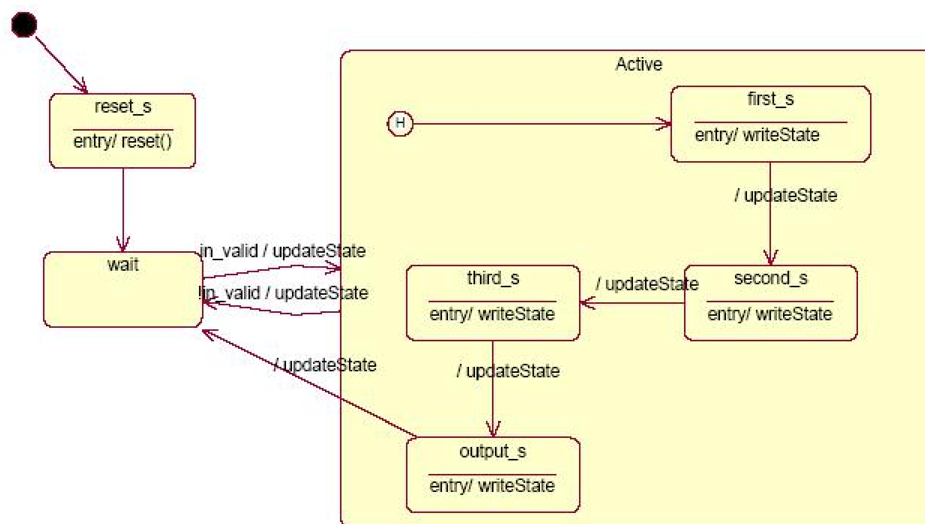


Figure 4.12: Statechart of the FIR's FSM unit

The design was translated to SystemC and simulated to verify that it satisfied the use cases. The design was later synthesised using the settings in Table 4.6. The results are as shown in Table 4.7:

The UML design was an enhanced version of the original design however still adhering to the initial use case specification. It provided the additional capability of returning to its original processing state after an invalid input, instead of resetting it completely. As can be seen the UML design is larger in size than the original design due to this enhancement.

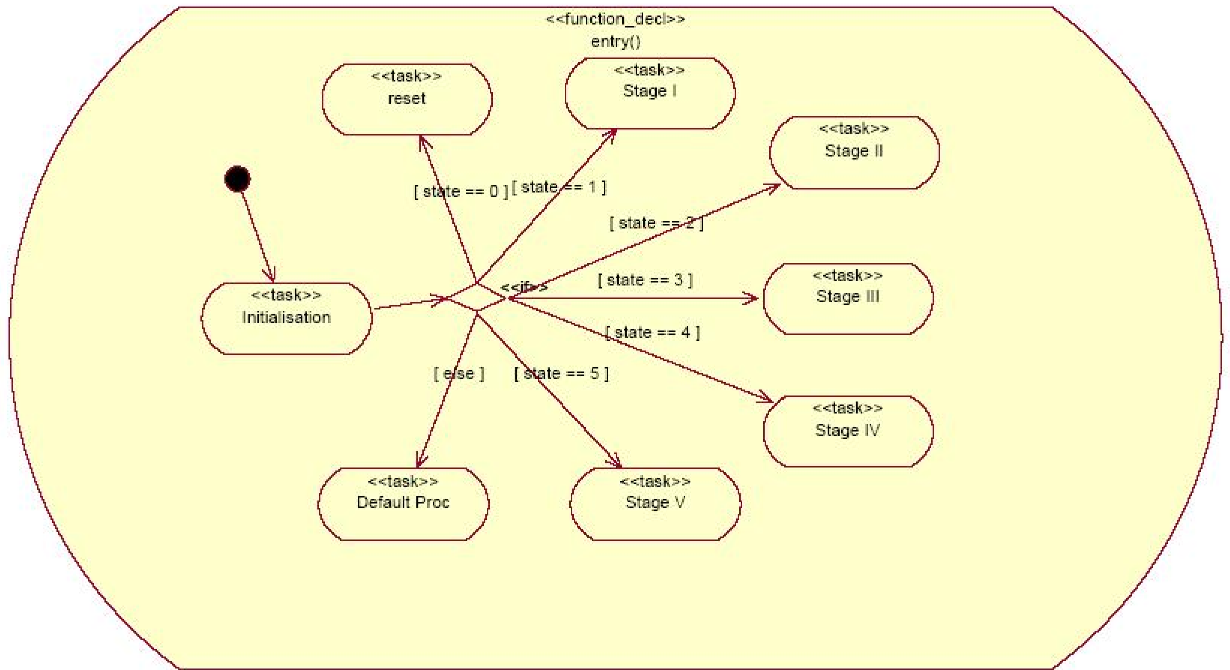


Figure 4.13: Activity Diagram of the FIR's Data path unit

Module	ports	nets	cells	refs	comb. area	non-comb. area	net inter-con area	total cell area	total area	gate count
FIR Filter(UMLSC)	68	100	2	2	5918	1625	26500	7544	34044	3.5k
FIR Filter(CCSC)	68	100	2	2	3015	984	14170	3999	18170	1.6k

Table 4.7: Results obtained from synthesis of FIR Filter (translated and original Cocentric) modules

4.2.4 FFT

The Fast Fourier Transform module is an excellent example of the kind of design that is more suited to behavioural design than RTL design. The design is based on the FFT module provided by Concentric.

The FFT is a complex algorithm that is more easily described as a flowchart rather than as a series of RTL level operations controlled by a state machine. The most suitable UML construct to express such a design is the Activity Diagram. A good way to represent the design is to group all operations that must be performed between any two "wait()"'s together into one <<task>>. The Cocentric behavioural compiler operates in two modes, 'IO' and 'Superstate'. In the IO mode, the latency between any two wait() statements is assumed to be one clock cycle. In the Superstate mode, the region between any two wait()s is assumed to be a 'Superstate'. This means that the compiler breaks the tasks found in between the wait()'s into lower levels of granularity so as to enable it to clock the design at higher speeds. In the first instance the designer has greater control whereas in the second instance the compiler does.

Shown in Figure 4.14, Figure 4.15 and Figure 4.16 are the use case diagram, class diagram and the activity diagram of the FFT module respectively.

As can be seen in the activity diagram there are three main kinds of activities namely <<task>> activities, <<run>> activties and <<end>> activities. Tasks refer to those activities which are constrained within clock cycles. Ŕunś are not constrained within clock cycles and éndsśignify the end of several runs. Hence one glance at the figure gives us an idea of which regions of the design are likely to produce long paths and where one must focus on to optimize the clock cycle latency. This is especially true if the design is running in the IO mode.

The design was translated to SystemC and verified. It was later synthesised using the settings in Table 4.8. The results are as shown in Table 4.9:

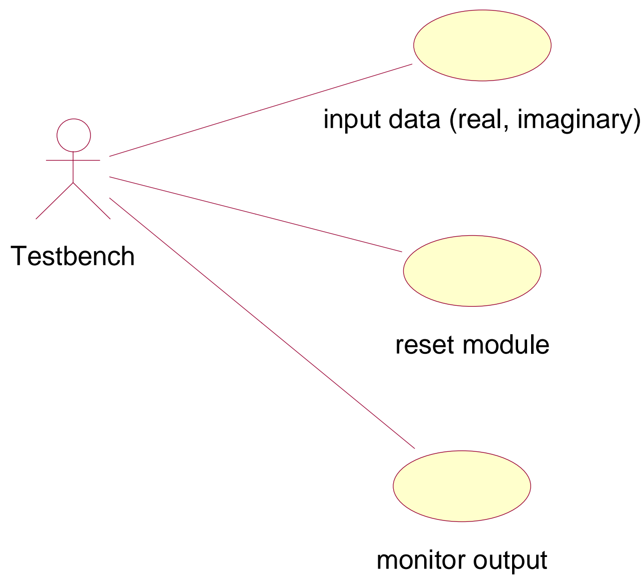


Figure 4.14: Use case diagram for the FFT module

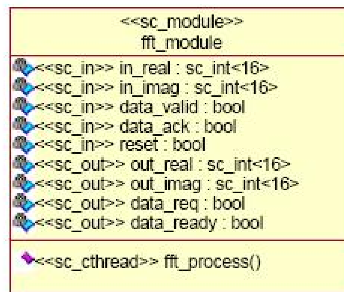


Figure 4.15: Class Diagram of the FFT unit

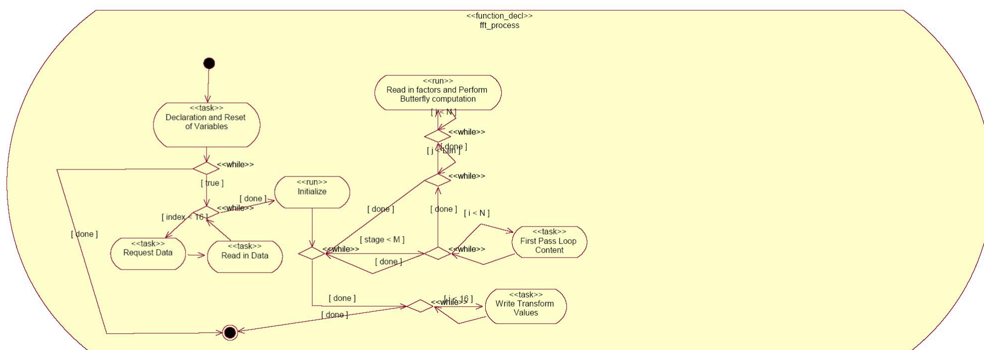


Figure 4.16: Activitiy Diagram of the FFT unit

Setting	Value
Technology library	tc6a_cbcore.db
Synthetic library	dw01.sldb, dw02.sldb
Map Effort	medium
I/O Mode	superstate
Clock period	25 ns

Table 4.8: FFT synthesis settings

Module	ports	nets	cells	refs	comb. area	non-comb. area	net inter-con area	total cell area	total area	gate count
FFT (UMLSC)	70	4606	4344	78	14353	5684	79964	20038	100003	7.7k
FFT (CCSC)	70	4606	4344	78	14353	5684	79964	20038	100003	7.7k

Table 4.9: Results obtained from synthesis of FFT Module through the translator compared with the original

The results obtained through the translator and the orignal design were identical mainly due to the small size of the design and the fact that the activity diagrams fed to the translator were faithful to the original design resulting in the overall control structure and hence the results being the same.

4.2.5 FIFO

The FIFO is another small design based on an example from the Synopsys Cocentric compiler examples to illustrate behavioural design using UML. Shown in Figure 4.17 is the use case diagram of this design.

Based on this we arrive at a FIFO that accepts a 32 bit integer value from the input and writes an integer to the output. The reset port clears all the data in the buffer. The size of the FIFO is specified by a macro BUFSIZE and the number of bits required to address it is specified by the LOGBUFSIZE macro. Since the design is behavioural there is no need to explicitly separate the control logic and the data path. The scheduling is handled by the lower level compiler. Shown in Figure 4.18 is the class diagram of the

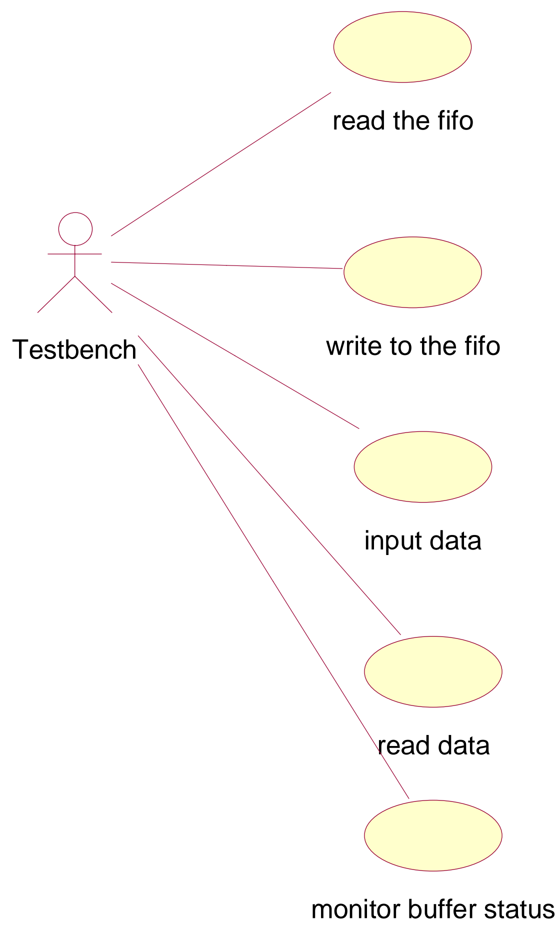


Figure 4.17: Use case diagram of FIFO

FIFO.

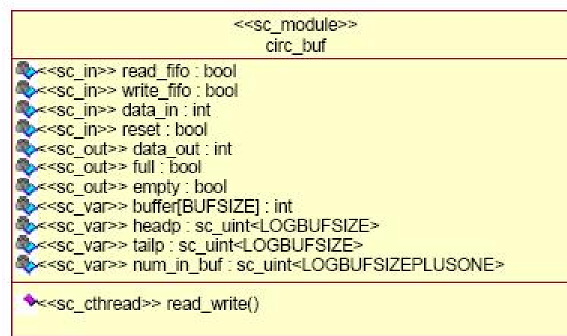


Figure 4.18: Class Diagram of the FIFO

A behavioural description usually contains an infinite loop within which the activity is described as can be seen in the activity diagram in Figure 4.19.

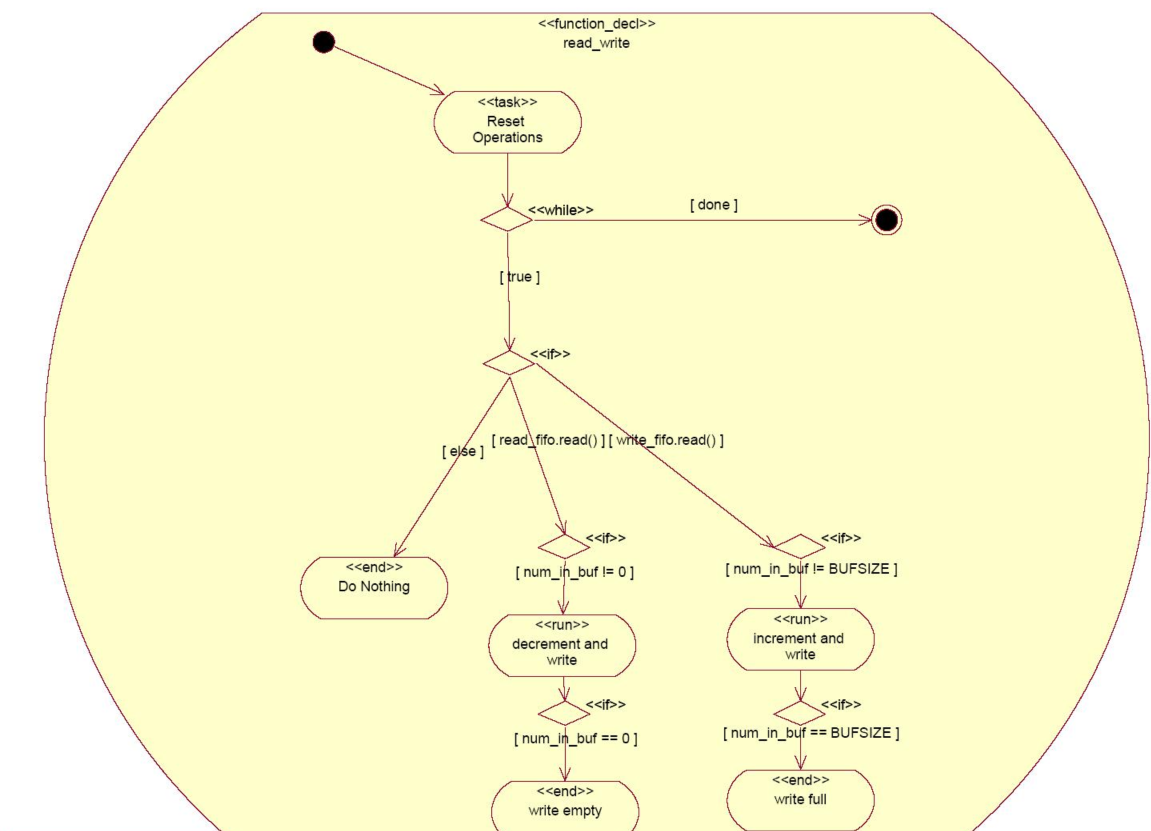


Figure 4.19: Activity Diagram of the FIFO

The UML design methodology offers the designer the flexibility to express the design in his preferred level of granularity. In this design, as compared to the previous FFT design, the activity diagram was fleshed out in greater detail, with more <<runs>> and

Setting	Value
Technology library	tc6a_cbcore.db
Synthetic library	dw01.sldb, dw02.sldb
Map Effort	medium
I/O Mode	superstate
Clock Period	10 ns

Table 4.10: FIFO synthesis settings

Module	ports	nets	cells	refs	comb. area	non-comb. area	net inter-con area	total cell area	total area	gate count
FIFO (UMLSC)	70	757	721	37	813	808	6770	1621	8392	721
FIFO (CCSC)	70	757	721	37	813	808	6770	1621	8392	721

Table 4.11: Results obtained from synthesis of FIFO through translator compared with original

<<ends>>. The difference between tasks, runs and ends can be summarized as follows - A <<task>> is constrained on both sides by <<wait>> statements, a <<run>> is not constrained whereas an <<endís the end of a path and contains a wait statement at its end. As can be seen, upon starting, the system is first reset. Following this there is an if-elseif structure. Each path contains some further processing depending on whether a "read" or "write" instruction was given to the FIFO and finally ends with a write to the "empty" or "full" outputs. One can therefore clearly see the three main paths in this design forking out after the reset and leading up from the if - elseif decoder all the way to the respective final <<end>> activities.

The design was first simulated and then translated, and synthesised using the settings in Table 4.10 and the results are as shown in Table 4.11

Once again the results were identical as the translator does not introduce any changes to activity diagram descriptions of control structures such as loops/branches from the original design.

4.2.6 VP3 Video Encoder

This design is based on the open source VP3 software video codec developed by On2 Technologies. Video encoding is best described algorithmically and hence naturally lends itself to description by activity diagrams. Being an application with a large number of modules and dynamic complexity it was adopted as an example to illustrate the design methodology advocated by this thesis. Shown in Figure 4.20 is the use case diagram for this design. Given this requirement, we start looking into lower level implementation details of this encoding process. We finally arrive at a list of the methods employed by VP3 described in the following sections. (The information was taken from the "VP3 Bitstream Format And Decoding Process" online document created by Mike Melanson.)

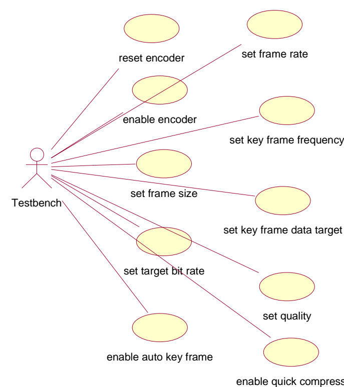


Figure 4.20: Use Case Diagram of the VP3 Encoder

DCT

The discrete cosine transform is performed on each block consisting of 8 X 8 = 64 pixels. The transform produces a list of 64 coefficients such that the first coefficient (called the DC coefficient) is the largest with subsequent coefficients getting smaller and smaller. The remaining 63 coefficients are called AC coefficients. This section of the design requires a large number of multiplications and hence optimisation of this module can result in large gains in latency and area.

Quantisation

This step is performed after the DCT. Quantisation essentially means reducing the magnitude of the number to be coded by dividing it by a pre-determined factor and discarding the remainder. The original number is, of course, recovered by multiplying the quantised data by the same factor. (There is, however, some loss of information in the process)

Run Length Encoding

In image and video encoding, one often encounters repetitive patterns of numbers. Instead of storing each one of these values, run length encoding stores only the repeating value, and the number of repetitions this value makes thus achieving further compression. In VP3 Coding RLE is used to record the number of zero value coefficients that occur before a non-zero coefficient is encountered.

Zigzag Ordering

After DCT and quantisation if performed on the data the samples are not in an optimal order for run length encoding. Zigzag ordering rearranges the samples to put more zeros between non-zero samples.

DPCM

Differential Pulse Code Modulation means encoding the difference between successive values instead of encoding the absolute values. This greatly reduces the magnitude of numbers to be coded.

Motion Compensation

In video encoding applications, instead of encoding the whole frame every time, commonalities between successive frames are exploited and portions of previous frames are copied into the current frame. This technique is combined with DCT and DPCM coding described earlier as well as fractional pixel motion.

Entropy Coding

This is also known as Huffman Coding. This is a process of reducing the number of bits used in coding the data by coding more frequently occurring symbols with fewer bits than symbols that are not likely to occur as frequently.

Variable Length Run Length Booleans

An initial Boolean bit is extracted from the bitstream. A variable length code is extracted from the bitstream and converted to a count. This count indicates that the next "count" elements are to be set to the boolean value. Afterwards the boolean value is toggled, the next VLC is extracted and converted to a count, and the process continues until all elements are set to either 0 or 1.

Overview

There are three kinds of frames in the VP3 coding process : Intra frames, and Inter frames. Inter frames are frames which use information from previous Intra frames.

The VP3 coding method encodes all three planes of the frame (Y,U,V) upside down from the bottom to the top. The first step is to break the video frame into a series of blocks called fragments. A Superblock encapsulates 16 fragments arranged in a 4X4 matrix. Each plane has its own set of superblocks. VP3 also has a notion of macroblocks. A macroblock encompasses 4 blocks from the Y plane arranged in a 2X2 matrix, 1 block

from the U plane and 1 block from the V plane. Hence a macroblock extends over all 3 planes.

To compress a golden frame, each fragment is tranformed using dct. Each sample is then quantised and the DC coefficient reduced via DPCM using a combination of DC coefficients from surrounding fragments are predictors. Then each fragment's DC coefficient is entropy coded into the output bitstream followed by each fragment's first AC coefficient, followed by each fragment's second AC coefficient, and so on.

Compressing an inter frame is more complicated. There are 8 coding modes available for coding an inter frame which the coder can choose between. Each of these modes involves coding a "fragment difference" from a different frame (either the previous one or the last golden frame) and from the same coordinate or from the same coordinate plus a motion vector, rather than a fragment. All these macroblock coding modes and motion vectors are then encoded in an interframe bitstream.

The following paragraphs explain each of these modules along with the their class diagram and main activity diagram describing the dynamic behaviour of the module:

VP3 Encoder

This is the top level module of the encoder and interfaces with the memory. This module contains important global structures for compression and playback. These structures contains variables related to configuration of the compressor, for holding statistics related to compression, frame statistics, regulation variables, block selection variables, various token counts and other variables used in the compression process. It also contains global motion vector storage variables used by functions in the motion compensation stage. The static structure of this module is as shown in Figure 4.21.

This module performs initialisations of libraries used, starts up the encoder, creates an overall control structure and then passes control to lower level modules to continue

Figure 4.21: Class Diagram of the VP3 Encoder Top

with the encoding process. The main function in this module is the EncodeFrameYUV function. The activity diagram of this function is shown in Figure 4.22.

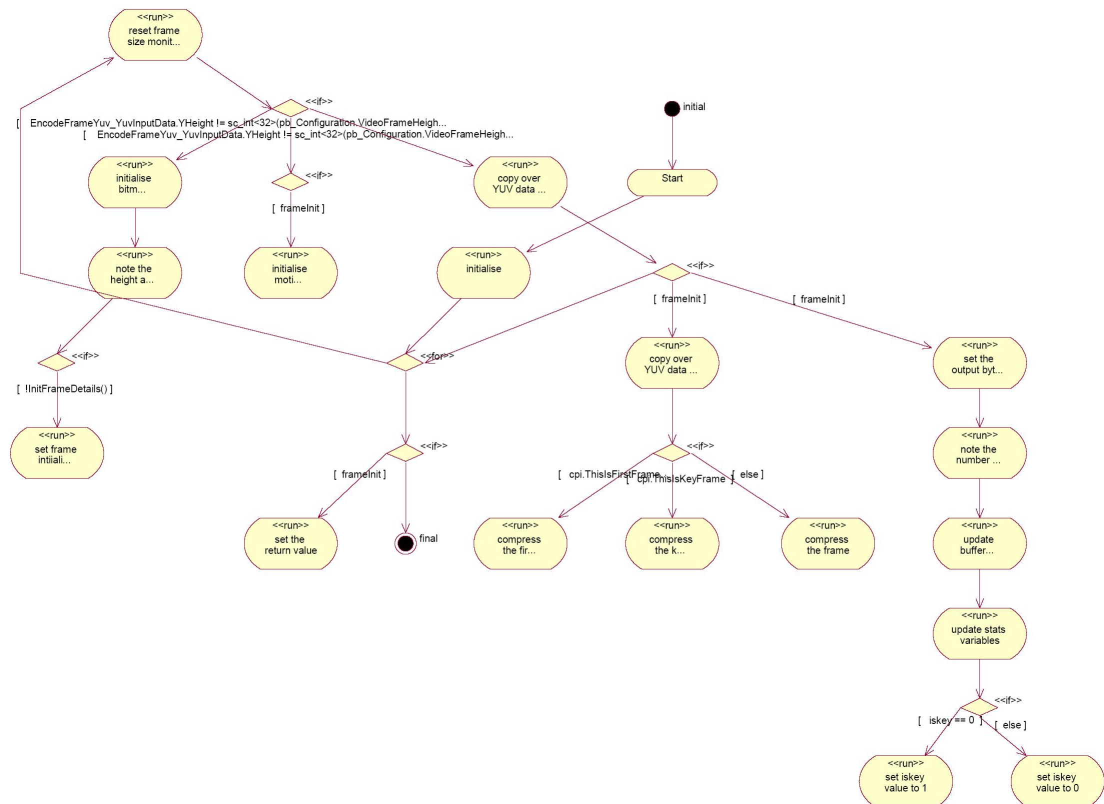


Figure 4.22: Main Activity Diagram of the VP3 Encoder Top

As can be seen, the first stage of this function consists of various initialisations followed by copying over frame data into the local buffers. Following this is the main section where the encoding is performed. If the frame is the first frame, the function CompressFirstFrame is called. If the frame is a key frame, the function CompressKeyFrame is called. Otherwise the function CompressFrame is called. Each of functions performs two important functions one, to decide the coding mode for all fragments and two, to update the frame. These functions are performed by lower level modules, “PickIntra”, “PickModes” and “UpdateFrame”.

PickIntra

PickIntra is one of the modules below VP3Encoder. This module picks INTRA coding for each macroblock of the current frame. The UML static structure for this module is shown in Figure 4.23.

As can be seen this module is a leaf module with no further sub modules. The activity diagram for the main function of this module is shown in Figure 4.24.

As seen above the function goes through each block in each superblock of the image and sets the MB coding mode to CODE_INTRA

PickModes

PickModes is another module one level below the top level module. The function of this module is to pick the coding mode for each macroblock of the image. This module is more comprehensive than PickIntra and is called when the frame has not been designated a Key frame. Hence this module has to make a choice from among 8 coding modes apart from also the CODE_INTRA mode. Shown in Figure 4.25 is the static structure of the PickModes module:

It contains a number of lower level modules which are used for computation of error scores to help the PickModes module decide which coding mode to use for a particular macroblock. Shown in Figure 4.26 is the activity diagram for the PickModes module.

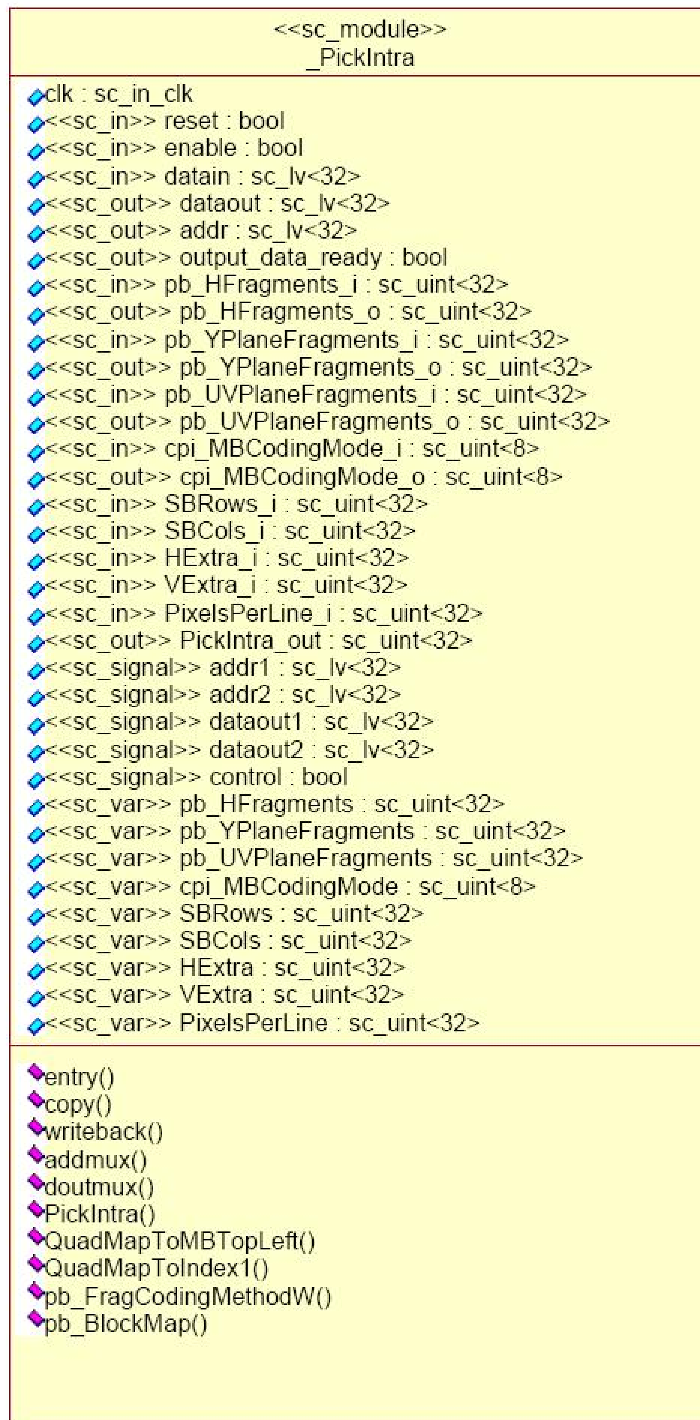


Figure 4.23: Class Diagram of the PickIntra module

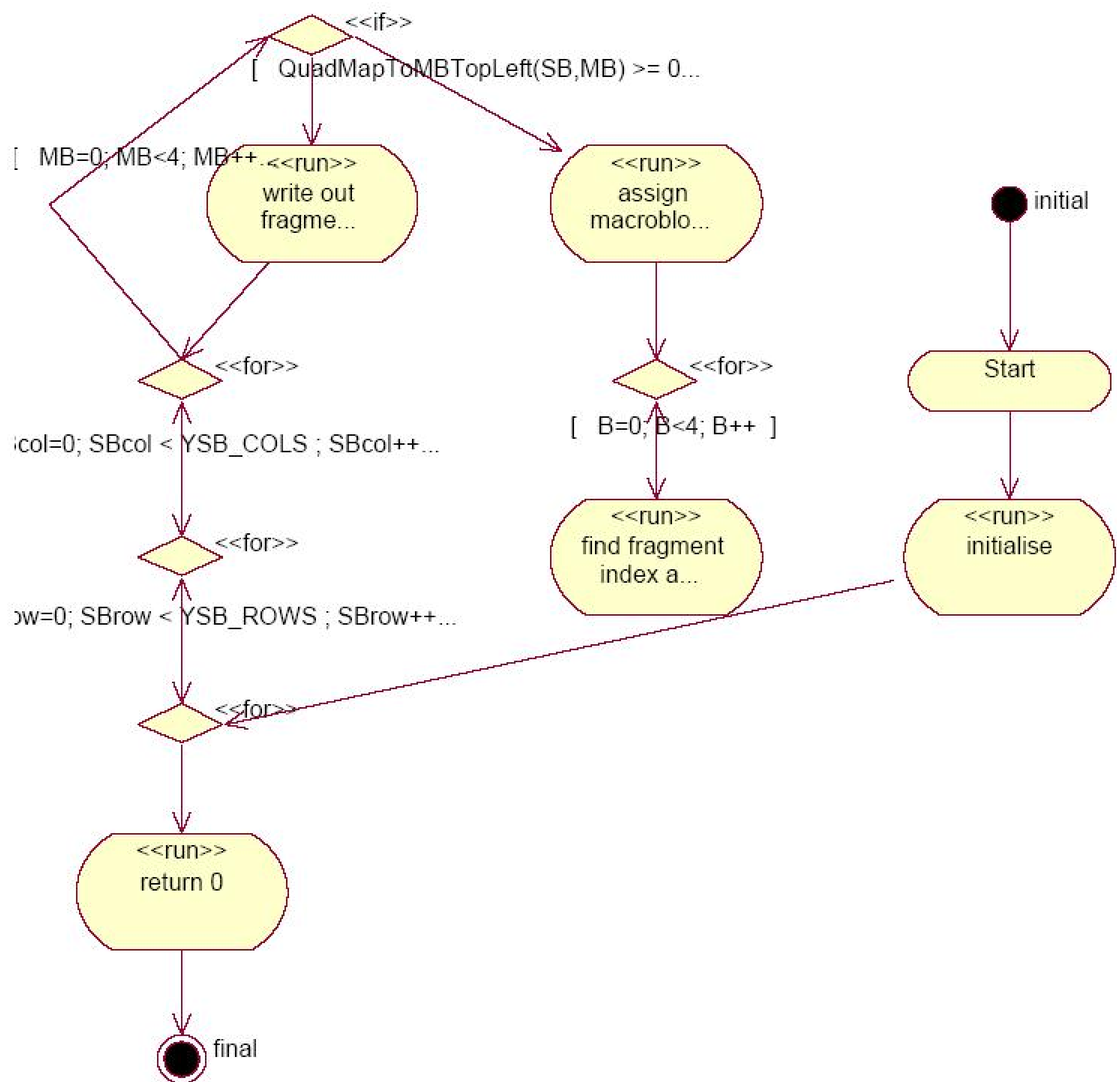


Figure 4.24: Activity Diagram of the PickIntra module

Figure 4.25: Class Diagram of the PickModes module

Figure 4.26: Activity Diagram of the PickModes module

The activity diagram is quite complex. It starts with initialisation of variables and temporary motion vectors , and error scores used later. Various thresholds are set for decision making. A iterative traversal is then performed through each macroblock of the frame. If the macroblock is coded we proceed further, otherwise we move to the next macroblock. Now, the lower level modules are used to find

- The intra coding error
- The golden frame error (difference from the last golden frame or I-frame encoded)
- The (0,0) error with the previous frame encoded
- The error with the previous frame, using the last motion vector used
- The error with the previous frame, using the prior last motion vector used

The best of these errors (the lowest) is remembered at this point. If the best error is above the required threshold, we must search for a new inter Motion Vector. A combination of simple and exhaustive searches is used in this calculation depending on the compression mode and the error computed in relation to the pre-decided threshold values. A new error value using the new Motion vector is then computed. If the best error is still above the threshold value, a search is performed for a new motion vector in the last golden frame encoded. If the error is still above the threshold, the 4 MV mode is investigated. In this mode, each of the four Y fragments gets its own motion vector and the U and V fragments share the same motion vector which is the average of the 4 Y fragment vectors. An error value is computed using this mode. For this to be considered the best error it must be above a certain threshold. After all the errors have been computed, the mode which produces the smallest error is chosen and the motion vectors (where relevant) and coding mode is set. If none of the methods produces a sufficiently large improvement (as decided by the fixed thresholds), the intra mode is used as the default coding mode.

UpdateFrame

This module is once again one level below the top level module and performs the function of writing the fragment data to the output file and updating the displayed frame. The static diagram is shown in Figure 4.27.

As can be seen this module makes use one main lower level module called QuadCodeDisplayFragments which performs the important function of encoding, tokenising and packing the frames. Shown in Figure 4.28 is the activity diagram illustrating the workings of this module.

```
<<sc_module>>
UpdateFrame
clk : sc_in_clk
<<sc_in>> reset : bool
<<sc_in>> enable : bool
<<sc_in>> datain : sc_lv<32>
<<sc_out>> dataout : sc_lv<32>
<<sc_out>> addr : sc_lv<32>
<<sc_out>> dma_cw : sc_uint<3>
<<sc_out>> dma_saddr : sc_uint<32>
<<sc_out>> dma_daddr : sc_uint<32>
<<sc_out>> dma_blocks : sc_uint<32>
<<sc_in>> dma_done : bool
<<sc_out>> output_data_ready : bool
<<sc_in>> store8_i : sc_uint<8>
<<sc_out>> store8_o : sc_uint<8>
<<sc_in>> store16s_i : sc_int<16>
<<sc_out>> store16s_o : sc_int<16>
<<sc_in>> pb_Configuration_VideoFrameW...
<<sc_out>> pb_Configuration_VideoFrame...
<<sc_in>> pb_Configuration_YStride_i : sc...
<<sc_out>> pb_Configuration_YStride_o : s...
<<sc_in>> pb_Configuration_UVStride_i : s...
<<sc_out>> pb_Configuration_UVStride_o ...
<<sc_in>> pb_Configuration_HFragPixels_...
<<sc_out>> pb_Configuration_HFragPixels...
<<sc_in>> pb_Configuration_VFragPixels_i...
<<sc_out>> pb_Configuration_VFragPixels...
<<sc_in>> cpi_Configuration_TargetBandw...
<<sc_out>> cpi_Configuration_TargetBand...
<<sc_in>> cpi_Configuration_OutputFrame...
<<sc_out>> cpi_Configuration_OutputFram...
<<sc_in>> cpi_Configuration_ActiveMaxQ_...
<<sc_out>> cpi_Configuration_ActiveMaxQ...
<<sc_in>> pb_Vp3VersionNo_i : sc_uint<8>
<<sc_out>> pb_Vp3VersionNo_o : sc_uint<...
<<sc_in>> pb_CodingMode_i : sc_uint<8>
<<sc_out>> pb_CodingMode_o : sc_uint<8>
<<sc_in>> pb_FrameType_i : sc_uint<8>
<<sc_out>> pb_FrameType_o : sc_uint<8>
<<sc_in>> pb_KeyFrameType_i : sc_uint<8>
<<sc_out>> pb_KeyFrameType_o : sc_uint...
<<sc_in>> pb_FrameQIndex_i : sc_uint<32>
<<sc_out>> pb_FrameQIndex_o : sc_uint<...
<<sc_in>> pb_ThisFrameQualityValue_i : s...
<<sc_out>> pb_ThisFrameQualityValue_o ...
<<sc_in>> pb_CodedBlockIndex_i : sc_int<...
<<sc_out>> pb_CodedBlockIndex_o : sc_ir...
<<sc_in>> pb_VFragments_i : sc_uint<32>
<<sc_out>> pb_VFragments_o : sc_uint<32>
<<sc_in>> pb_HFragments_i : sc_uint<32>
<<sc_out>> pb_HFragments_o : sc_uint<32>
<<sc_in>> pb_YPlaneFragments_i : sc_uin...
<<sc_out>> pb_YPlaneFragments_o : sc_u...
<<sc_in>> pb_UVPlaneFragments_i : sc_ui...
<<sc_out>> pb_UVPlaneFragments_o : sc_...
<<sc_in>> pb_YSuperBlocks_i : sc_uint<32>
<<sc_out>> pb_YSuperBlocks_o : sc_uint<...
<<sc_in>> pb_UVSuperBlocks_i : sc_uint<...
<<sc_out>> pb_UVSuperBlocks_o : sc_uint...
<<sc_in>> pb_SuperBlocks_i : sc_uint<32>
<<sc_out>> pb_SuperBlocks_o : sc_uint<32>
<<sc_in>> pb_YSBRows_i : sc_uint<32>
<<sc_out>> pb_YSBRows_o : sc_uint<32>
<<sc_in>> pb_YSBCols_i : sc_uint<32>
<<sc_out>> pb_YSBCols_o : sc_uint<32>
<<sc_in>> pb_UVSBRows_i : sc_uint<32>
<<sc_out>> pb_UVSBRows_o : sc_uint<32>
<<sc_in>> pb_UVSBCols_i : sc_uint<32>
<<sc_out>> pb_UVSBCols_o : sc_uint<32>
<<sc_in>> pb_EOB_Run_i : sc_uint<32>
<<sc_out>> pb_EOB_Run_o : sc_uint<32>
<<sc_in>> pb_InvLastIntraDC_i : sc_int<16>
<<sc_out>> pb_InvLastIntraDC_o : sc_int<...
<<sc_in>> pb_InvLastInterDC_i : sc_int<16>
<<sc_out>> pb_InvLastInterDC_o : sc_int<...
<<sc_in>> pb_LastIntraDC_i : sc_int<16>
<<sc_out>> pb_LastIntraDC_o : sc_int<16>
<<sc_in>> pb_LastInterDC_i : sc_int<16>
<<sc_out>> pb_LastInterDC_o : sc_int<16>
<<sc_in>> pb_DataOutputInPtr_index_i : int
<<sc_out>> pb_DataOutputInPtr_index_o : int
<<sc_in>> cpi_LastKeyFrame_i : sc_uint<32>
<<sc_out>> cpi_LastKeyFrame_o : sc_uint...
<<sc_in>> cpi_DropFrameTriggerBytes_i : ...
<<sc_out>> cpi_DropFrameTriggerBytes_o...
<<sc_in>> cpi_DropFrameCandidate_i : bool
<<sc_out>> cpi_DropFrameCandidate_o : b...
<<sc_in>> cpi_KeyFrameCount_i : sc_int<6...
<<sc_out>> cpi_KeyFrameCount_o : sc_int...
<<sc_in>> cpi_TotKeyFrameBytes_i : sc_in...
<<sc_out>> cpi_TotKeyFrameBytes_o : sc_...
<<sc_in>> cpi_LastKeyFrameSize_i : sc_ui...
<<sc_out>> cpi_LastKeyFrameSize_o : sc_...
<<sc_in>> cpi_TotalByteCount_i : sc_int<64>
<<sc_out>> cpi_TotalByteCount_o : sc_int<...
<<sc_in>> cpi_CurrentFrame_i : sc_int<64>
<<sc_out>> cpi_CurrentFrame_o : sc_int<6...
<<sc_in>> cpi_CarryOver_i : sc_int<64>
<<sc_out>> cpi_CarryOver_o : sc_int<64>
<<sc_in>> cpi_ThisFrameSize_i : sc_uint<3...
<<sc_out>> cpi_ThisFrameSize_o : sc_uint...
<<sc_in>> cpi_BufferedOutputBytes_i : sc_...
<<sc_out>> cpi_BufferedOutputBytes_o : s...
<<sc_in>> cpi_FrameBitCount_i : sc_uint<3...
<<sc_out>> cpi_FrameBitCount_o : sc_uint...
<<sc_in>> cpi_ThisFrameTargetBytes_i : s...
<<sc_out>> cpi_ThisFrameTargetBytes_o : ...
<<sc_in>> cpi_BpbCorrectionFactor_i : sc_...
<<sc_out>> cpi_BpbCorrectionFactor_o : s...
<<sc_in>> cpi_frame_target_rate_i : sc_int<...
<<sc_out>> cpi_frame_target_rate_o : sc_i...
<<sc_in>> cpi_OptimisedTokenCount_i : sc...
<<sc_out>> cpi_OptimisedTokenCount_o : ...
<<sc_in>> cpi_RunHuffIndex_i : sc_uint<32>
<<sc_out>> cpi_RunHuffIndex_o : sc_uint<...
<<sc_in>> cpi_RunPlaneIndex_i : sc_uint<3...
<<sc_out>> cpi_RunPlaneIndex_o : sc_uint...
<<sc_in>> cpi_TotTokenCount_i : sc_uint<3...
<<sc_out>> cpi_TotTokenCount_o : sc_uin...
<<sc_in>> cpi_TokensToBeCoded_i : sc_ir...
<<sc_out>> cpi_TokensToBeCoded_o : sc_...
<<sc_in>> cpi_TokensCoded_i : sc_int<32>
<<sc_out>> cpi_TokensCoded_o : sc_int<3...
<<sc_in>> cpi_MvListCount_i : sc_uint<32>
<<sc_out>> cpi_MvListCount_o : sc_uint<32>
<<sc_in>> cpi_ModeListCount_i : sc_uint<3...
<<sc_out>> cpi_ModeListCount_o : sc_uint...
<<sc_in>> cpi_RunLength_i : sc_uint<32>
<<sc_out>> cpi_RunLength_o : sc_uint<32>
<<sc_in>> cpi_MaxBitTarget_i : sc_uint<32>
<<sc_out>> cpi_MaxBitTarget_o : sc_uint<3...
<<sc_in>> cpi_BitRateCapFactor_i : sc_int<...
<<sc_out>> cpi_BitRateCapFactor_o : sc_i...
<<sc_in>> cpi_MBCodingMode_i : sc_uint<...
<<sc_out>> cpi_MBCodingMode_o : sc_uir...
<<sc_in>> cpi_byte_bit_offset_i : sc_int<32>
<<sc_out>> cpi_byte_bit_offset_o : sc_int<3...
<<sc_in>> cpi_DataBlock_i : sc_uint<32>
<<sc_out>> cpi_DataBlock_o : sc_uint<32>
<<sc_in>> cpi_mybits_i : sc_uint<32>
<<sc_out>> cpi_mybits_o : sc_uint<32>
<<sc_in>> cpi_ByteBitsLeft_i : sc_uint<32>
<<sc_out>> cpi_ByteBitsLeft_o : sc_uint<32>
<<sc_in>> cpi_MVector_x_i : sc_int<32>
<<sc_out>> cpi_MVector_x_o : sc_int<32>
<<sc_in>> cpi_MVector_y_i : sc_int<32>
<<sc_out>> cpi_MVector_y_o : sc_int<32>
<<sc_signal>> addr1 : sc_lv<32>
<<sc_signal>> addr2 : sc_lv<32>
```

```
<<sc_module>>
QuadCodeDisplayFragments
clk : sc_in_clk
<<sc_in>> reset : bool
<<sc_in>> enable : bool
<<sc_in>> datain : sc_lv<32>
<<sc_out>> dataout : sc_lv<32>
<<sc_out>> addr : sc_lv<32>
<<sc_out>> dma_cw : sc_uint<3>
<<sc_out>> dma_saddr : sc_uint<32>
<<sc_out>> dma_daddr : sc_uint<32>
<<sc_out>> dma_blocks : sc_uint<32>
<<sc_in>> dma_done : bool
<<sc_out>> output_data_ready : bool
<<sc_in>> store8_i : sc_uint<8>
<<sc_out>> store8_o : sc_uint<8>
<<sc_in>> store16s_i : sc_int<16>
<<sc_out>> store16s_o : sc_int<16>
<<sc_in>> pb_Configuration_VideoFrameWidt...
<<sc_out>> pb_Configuration_VideoFrameWi...
<<sc_in>> pb_Configuration_YStride_i : sc_ui...
<<sc_out>> pb_Configuration_YStride_o : sc_...
<<sc_in>> pb_Configuration_UVStride_i : sc_u...
<<sc_out>> pb_Configuration_UVStride_o : sc...
<<sc_in>> pb_CodingMode_i : sc_uint<8>
<<sc_out>> pb_CodingMode_o : sc_uint<8>
<<sc_in>> pb_FrameType_i : sc_uint<8>
<<sc_out>> pb_FrameType_o : sc_uint<8>
<<sc_in>> pb_FrameQIndex_i : sc_uint<32>
<<sc_out>> pb_FrameQIndex_o : sc_uint<32>
<<sc_in>> pb_ThisFrameQualityValue_i : sc_u...
<<sc_out>> pb_ThisFrameQualityValue_o : sc...
<<sc_in>> pb_CodedBlockIndex_i : sc_int<32>
<<sc_out>> pb_CodedBlockIndex_o : sc_int<3...
<<sc_in>> pb_VFragments_i : sc_uint<32>
<<sc_out>> pb_VFragments_o : sc_uint<32>
<<sc_in>> pb_HFragments_i : sc_uint<32>
<<sc_out>> pb_HFragments_o : sc_uint<32>
<<sc_in>> pb_YPlaneFragments_i : sc_uint<32>
<<sc_out>> pb_YPlaneFragments_o : sc_uint...
<<sc_in>> pb_UVPlaneFragments_i : sc_uint<...
<<sc_out>> pb_UVPlaneFragments_o : sc_uir...
<<sc_in>> pb_YSuperBlocks_i : sc_uint<32>
<<sc_out>> pb_YSuperBlocks_o : sc_uint<32>
<<sc_in>> pb_UVSuperBlocks_i : sc_uint<32>
<<sc_out>> pb_UVSuperBlocks_o : sc_uint<3...
<<sc_in>> pb_SuperBlocks_i : sc_uint<32>
<<sc_out>> pb_SuperBlocks_o : sc_uint<32>
<<sc_in>> pb_YSBRows_i : sc_uint<32>
<<sc_out>> pb_YSBRows_o : sc_uint<32>
<<sc_in>> pb_YSBCols_i : sc_uint<32>
<<sc_out>> pb_YSBCols_o : sc_uint<32>
<<sc_in>> pb_UVSBRows_i : sc_uint<32>
<<sc_out>> pb_UVSBRows_o : sc_uint<32>
<<sc_in>> pb_UVSBCols_i : sc_uint<32>
<<sc_out>> pb_UVSBCols_o : sc_uint<32>
<<sc_in>> pb_DataOutputInPtr_index_i : int
<<sc_out>> pb_DataOutputInPtr_index_o : int
<<sc_in>> cpi_ThisFrameSize_i : sc_uint<32>
<<sc_out>> cpi_ThisFrameSize_o : sc_uint<32>
<<sc_in>> cpi_BufferedOutputBytes_i : sc_uin...
<<sc_out>> cpi_BufferedOutputBytes_o : sc_u...
<<sc_in>> cpi_FrameBitCount_i : sc_uint<32>
<<sc_out>> cpi_FrameBitCount_o : sc_uint<32>
<<sc_in>> cpi_ThisFrameTargetBytes_i : sc_u...
<<sc_out>> cpi_ThisFrameTargetBytes_o : sc...
<<sc_in>> cpi_OptimisedTokenCount_i : sc_ir...
<<sc_out>> cpi_OptimisedTokenCount_o : sc_...
<<sc_in>> cpi_RunHuffIndex_i : sc_uint<32>
<<sc_out>> cpi_RunHuffIndex_o : sc_uint<32>
<<sc_in>> cpi_RunPlaneIndex_i : sc_uint<32>
<<sc_out>> cpi_RunPlaneIndex_o : sc_uint<32>
<<sc_in>> cpi_TotTokenCount_i : sc_uint<32>
<<sc_out>> cpi_TotTokenCount_o : sc_uint<3...
<<sc_in>> cpi_TokensToBeCoded_i : sc_int<3...
<<sc_out>> cpi_TokensToBeCoded_o : sc_int...
<<sc_in>> cpi_TokensCoded_i : sc_int<32>
<<sc_out>> cpi_TokensCoded_o : sc_int<32>
<<sc_in>> cpi_MvListCount_i : sc_uint<32>
<<sc_out>> cpi_MvListCount_o : sc_uint<32>
<<sc_in>> cpi_ModeListCount_i : sc_uint<32>
<<sc_out>> cpi_ModeListCount_o : sc_uint<32>
<<sc_in>> cpi_RunLength_i : sc_uint<32>
<<sc_out>> cpi_RunLength_o : sc_uint<32>
<<sc_in>> cpi_MaxBitTarget_i : sc_uint<32>
<<sc_out>> cpi_MaxBitTarget_o : sc_uint<32>
<<sc_in>> cpi_BitRateCapFactor_i : sc_bigint...
<<sc_out>> cpi_BitRateCapFactor_o : sc_bigi...
<<sc_in>> cpi_MBCodingMode_i : sc_uint<8>
<<sc_out>> cpi_MBCodingMode_o : sc_uint<8>
<<sc_in>> cpi_byte_bit_offset_i : sc_int<32>
<<sc_out>> cpi_byte_bit_offset_o : sc_int<32>
<<sc_in>> cpi_DataBlock_i : sc_uint<32>
<<sc_out>> cpi_DataBlock_o : sc_uint<32>
<<sc_in>> cpi_mybits_i : sc_uint<32>
<<sc_out>> cpi_mybits_o : sc_uint<32>
<<sc_in>> cpi_ByteBitsLeft_i : sc_uint<32>
<<sc_out>> cpi_ByteBitsLeft_o : sc_uint<32>
<<sc_in>> cpi_MVector_x_i : sc_int<32>
<<sc_out>> cpi_MVector_x_o : sc_int<32>
<<sc_in>> cpi_MVector_y_i : sc_int<32>
<<sc_out>> cpi_MVector_y_o : sc_int<32>
<<sc_out>> QuadCodeDisplayFragments_out...
<<sc_signal>> addr1 : sc_lv<32>
<<sc_signal>> addr2 : sc_lv<32>
<<sc_signal>> dataout1 : sc_lv<32>
<<sc_signal>> dataout2 : sc_lv<32>
<<sc_signal>> control : bool
<<sc_signal>> store8 : sc_uint<8>
<<sc_signal>> store16s : sc_int<16>
<<sc_signal>> pb_Configuration_YStride : sc_...
<<sc_signal>> pb_Configuration_UVStride : s...
<<sc_signal>> pb_CodingMode : sc_uint<8>
<<sc_signal>> pb_FrameType : sc_uint<8>
<<sc_signal>> pb_CodedBlockIndex : sc_int<...
<<sc_signal>> pb_HFragments : sc_uint<32>
<<sc_signal>> pb_YPlaneFragments : sc_uint...
<<sc_signal>> pb_SuperBlocks : sc_uint<32>
<<sc_signal>> pb_DataOutputInPtr_index : int
<<sc_signal>> cpi_ThisFrameSize : sc_uint<3...
<<sc_signal>> cpi_BufferedOutputBytes : sc_u...
<<sc_signal>> cpi_FrameBitCount : sc_uint<32>
<<sc_signal>> cpi_ThisFrameTargetBytes : sc...
<<sc_signal>> cpi_OptimisedTokenCount : sc...
<<sc_signal>> cpi_RunHuffIndex : sc_uint<32>
<<sc_signal>> cpi_RunPlaneIndex : sc_uint<3...
<<sc_signal>> cpi_TotTokenCount : sc_uint<3...
<<sc_signal>> cpi_TokensToBeCoded : sc_in...
<<sc_signal>> cpi_TokensCoded : sc_int<32>
<<sc_signal>> cpi_MvListCount : sc_uint<32>
<<sc_signal>> cpi_ModeListCount : sc_uint<32>
```

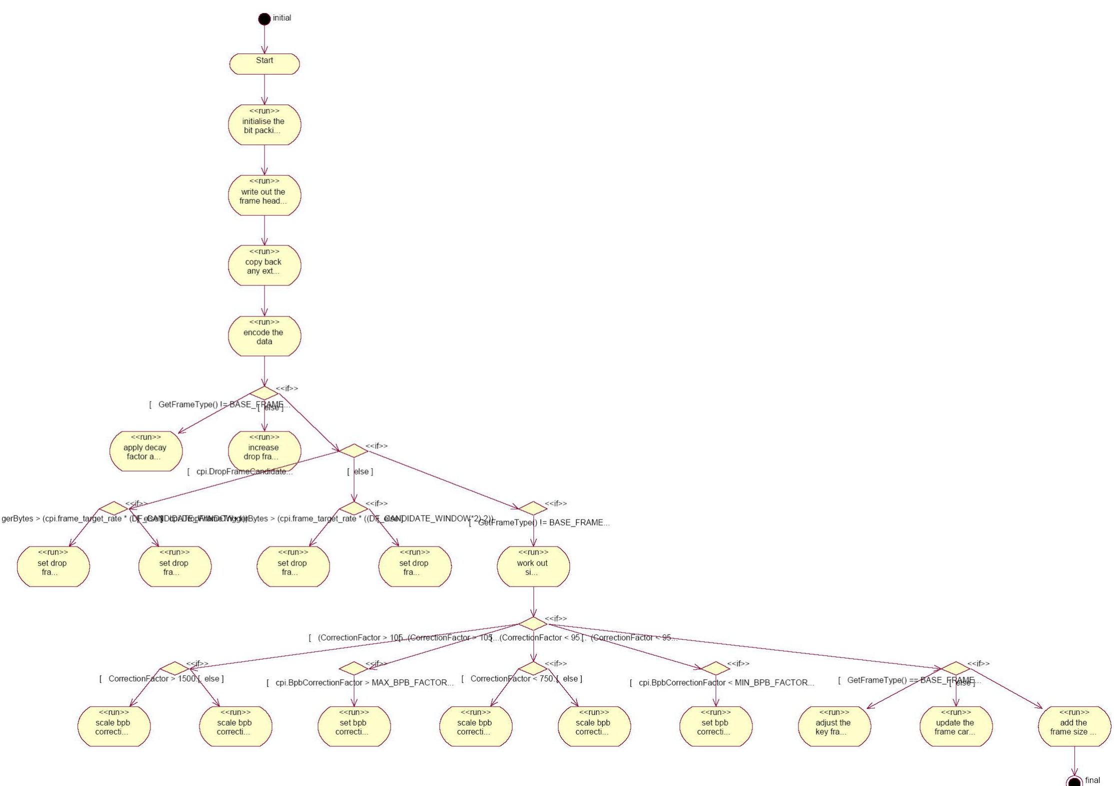


Figure 4.28: Activity Diagram of the UpdateFrame module

The first task is to initialise variables used in the process. Then the bit packing mechanism is initialised. Following this, any extra frags that are to be updated by the codec as part of the background cleanup task are copied back. Then the data is encoded with the help of the lower level module. Next, the "Drop Frame" trigger (measured in number of bytes) is adjusted by adding in the size of the frame just encoded to the figure. A Test for overshoot is performed which may require a dropped frame next time around. If we are already in a drop frame condition but the previous frame was not dropped then the threshold for continuing to allow dropped frames is reduced. The BpbCorrectionFactor variable is then updated according to whether or not we were close enough with our selection of DCT quantiser. Finally the carry over and the key frame context are adjusted.

GetMBIntraError

This module is one level lower the PickModes module described earlier which assigns a coding mode to each block by computing various error scores. The purpose of this module is to calculate a variance score for an intra macro block. Shown in Figure 4.29 is the static structure:

As seen, this module has one lower level module called GetIntraError which performs the function of finding the intra error for one fragment. Shown in Figure 4.30 is the activity diagram for the GetMBIntraError module.

The flow of logic is fairly simple as can be seen. The function incrementally adds together the intra errors for those blocks in the macro block that are coded (Y only) and returns an "IntraError" for the MacroBlock.

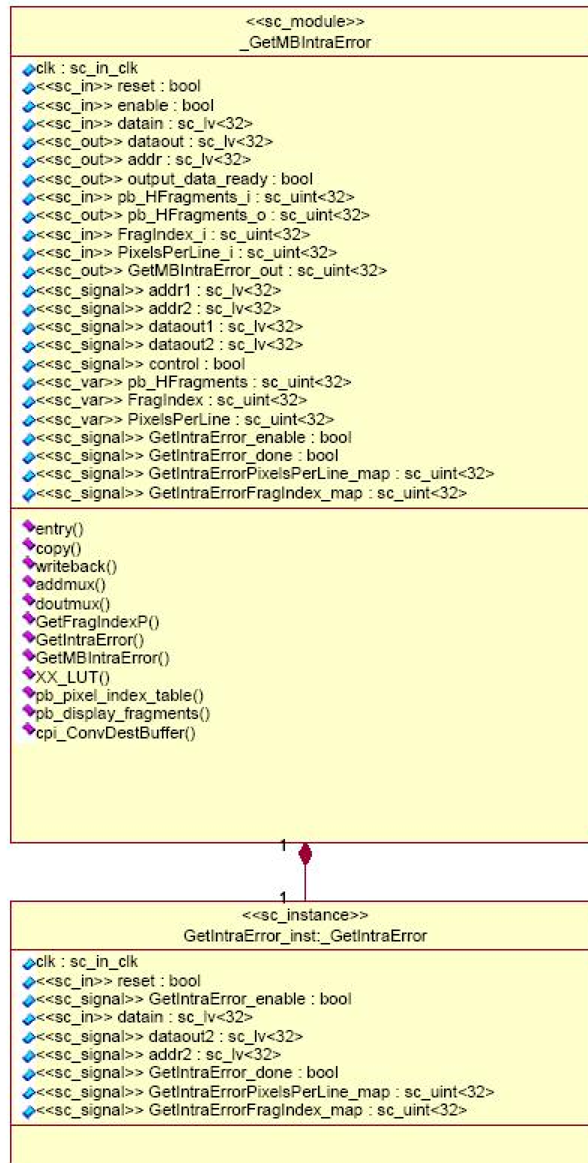


Figure 4.29: Class Diagram of the GetMBIntraError module

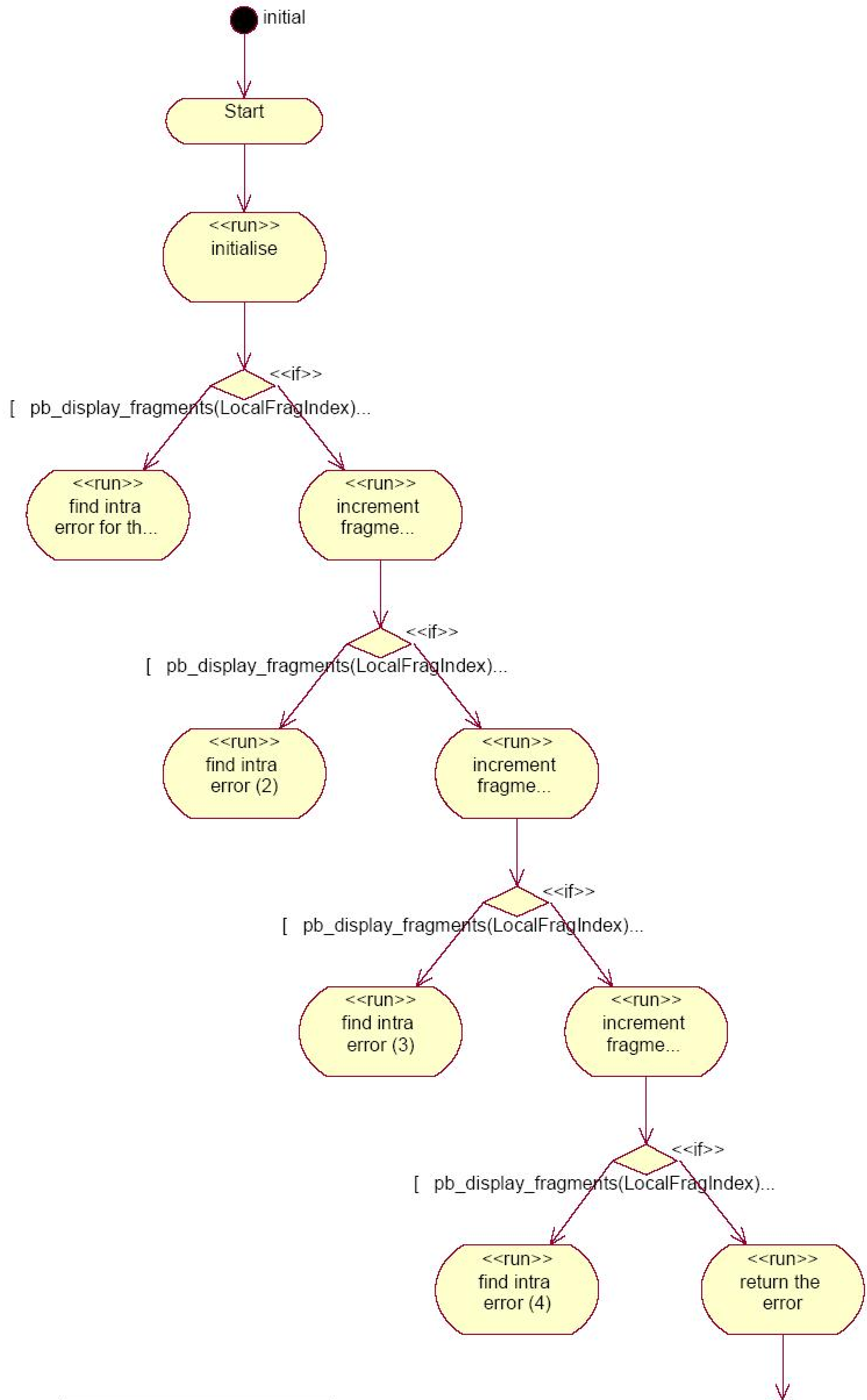


Figure 4.30: Activity Diagram of the GetMBIntraError module

GetMBInterError

This module is one level below the PickModes described earlier. The function of this module is to calculate a variance score for an inter MB with motion vectors.

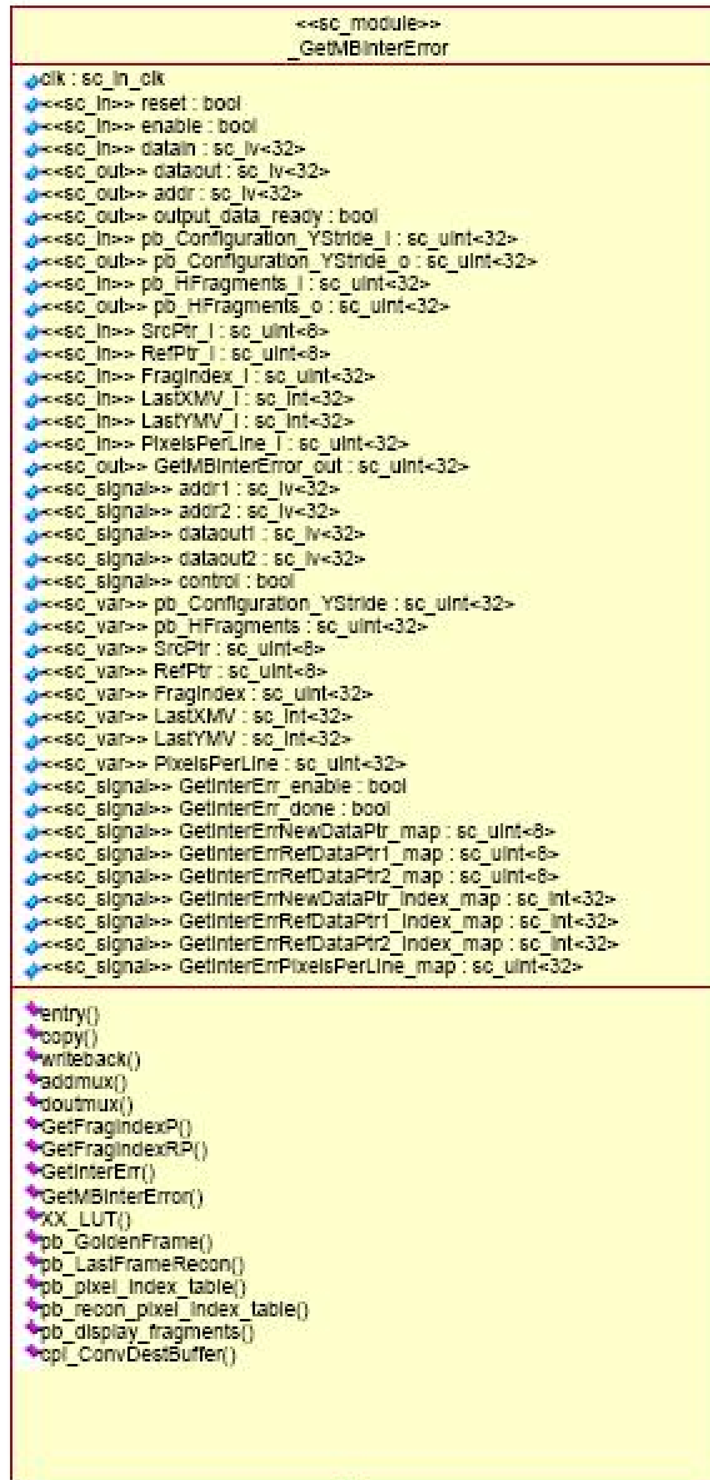
As in seen in the static structure of the module in Figure 4.31, it has one lower level module GetInterErr which is used to find the Inter Error for each block. The functioning of this module is illustrated in Figure 4.32 in the activity diagram:

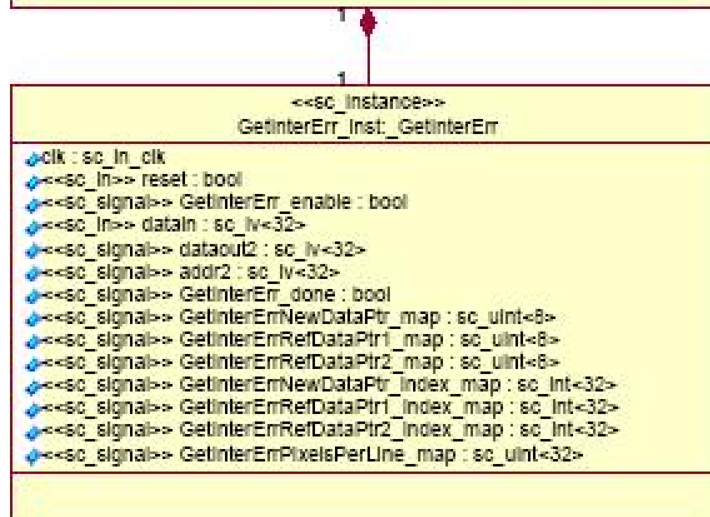
To start with, pixel offsets of the source buffer and the reference buffer (taking into account the default motion vector) are computed. Following this a second reference pointer is computed for odd values of X or Y. We then proceed block by block (Y only) , and send the source pointer, and two reference pointers for the lower level block to compute an "inter" error. The block errors are all added up and returned as an "inter" error for the macroblock.

GetMBMVInterError

This module is once again one level below the PickModes module and its purpose is to calculate a fresh Motion Vector using a heirachical search. As described earlier, this module is called when Intra and Inter Errors using the default motion vectors do not bring about a sufficiently large improvement in error scores. Refer to Figure 4.33 for the class diagram.

This module has a number of lower level modules namely the GetSumAbsDiffs, GetNextSumAbsDiffs, GetHalfPixelSumAbsDiffs, GetMBInterError used to calculate the Sum of Absolute Differences for a block in the current frame with one in a reference frame, Sum of Absolute Differences at half pixel accuracy (by averaging two pixels in the reference frame) for a block in the current frame with one in a reference frame, and to find the MB Inter Error as described earlier. The activity diagram is shown in Figure 4.34:


<<sc_module>>
_GetMBInterError
clk : sc_in_clk
<<sc_in>> reset : bool
<<sc_in>> enable : bool
<<sc_in>> datain : sc_lv<32>
<<sc_out>> dataout : sc_lv<32>
<<sc_out>> addr : sc_lv<32>
<<sc_out>> output_data_ready : bool
<<sc_in>> pb_Configuration_YStride_i : sc_uint<32>
<<sc_out>> pb_Configuration_YStride_o : sc_uint<32>
<<sc_in>> pb_HFragments_i : sc_uint<32>
<<sc_out>> pb_HFragments_o : sc_uint<32>
<<sc_in>> SrcPtr_i : sc_uint<8>
<<sc_in>> RefPtr_i : sc_uint<8>
<<sc_in>> FragIndex_i : sc_uint<32>
<<sc_in>> LastXMV_i : sc_int<32>
<<sc_in>> LastYMV_i : sc_int<32>
<<sc_in>> PixelsPerLine_i : sc_uint<32>
<<sc_out>> GetMBInterError_out : sc_uint<32>
<<sc_signal>> addr1 : sc_lv<32>
<<sc_signal>> addr2 : sc_lv<32>
<<sc_signal>> dataout1 : sc_lv<32>
<<sc_signal>> dataout2 : sc_lv<32>
<<sc_signal>> control : bool
<<sc_var>> pb_Configuration_YStride : sc_uint<32>
<<sc_var>> pb_HFragments : sc_uint<32>
<<sc_var>> SrcPtr : sc_uint<8>
<<sc_var>> RefPtr : sc_uint<8>
<<sc_var>> FragIndex : sc_uint<32>
<<sc_var>> LastXMV : sc_int<32>
<<sc_var>> LastYMV : sc_int<32>
<<sc_var>> PixelsPerLine : sc_uint<32>
<<sc_signal>> GetInterErr_enable : bool
<<sc_signal>> GetInterErr_done : bool
<<sc_signal>> GetInterErrNewDataPtr_map : sc_uint<8>
<<sc_signal>> GetInterErrRefDataPtr1_map : sc_uint<8>
<<sc_signal>> GetInterErrRefDataPtr2_map : sc_uint<8>
<<sc_signal>> GetInterErrNewDataPtr_index_map : sc_int<32>
<<sc_signal>> GetInterErrRefDataPtr1_index_map : sc_int<32>
<<sc_signal>> GetInterErrRefDataPtr2_index_map : sc_int<32>
<<sc_signal>> GetInterErrPixelsPerLine_map : sc_uint<32>
entry()
copy()
writeback()
addmux()
doutmux()
GetFragIndexP()
GetFragIndexRP()
GetInterErr()
GetMBInterError()
XX_LUT()
pb_GoldenFrame()
pb_LastFrameRecon()
pb_pixel_index_table()
pb_recon_pixel_index_table()
pb_display_fragments()
cpi_ConvDestBuffer()
1
1


<<sc_instance>>
GetInterErr_inst:_GetInterErr
clk : sc_in_clk
<<sc_in>> reset : bool
<<sc_signal>> GetInterErr_enable : bool
<<sc_in>> datain : sc_lv<32>
<<sc_signal>> dataout2 : sc_lv<32>
<<sc_signal>> addr2 : sc_lv<32>
<<sc_signal>> GetInterErr_done : bool
<<sc_signal>> GetInterErrNewDataPtr_map : sc_uint<8>
<<sc_signal>> GetInterErrRefDataPtr1_map : sc_uint<8>
<<sc_signal>> GetInterErrRefDataPtr2_map : sc_uint<8>
<<sc_signal>> GetInterErrNewDataPtr_index_map : sc_int<32>
<<sc_signal>> GetInterErrRefDataPtr1_index_map : sc_int<32>
<<sc_signal>> GetInterErrRefDataPtr2_index_map : sc_int<32>
<<sc_signal>> GetInterErrPixelsPerLine_map : sc_uint<32>

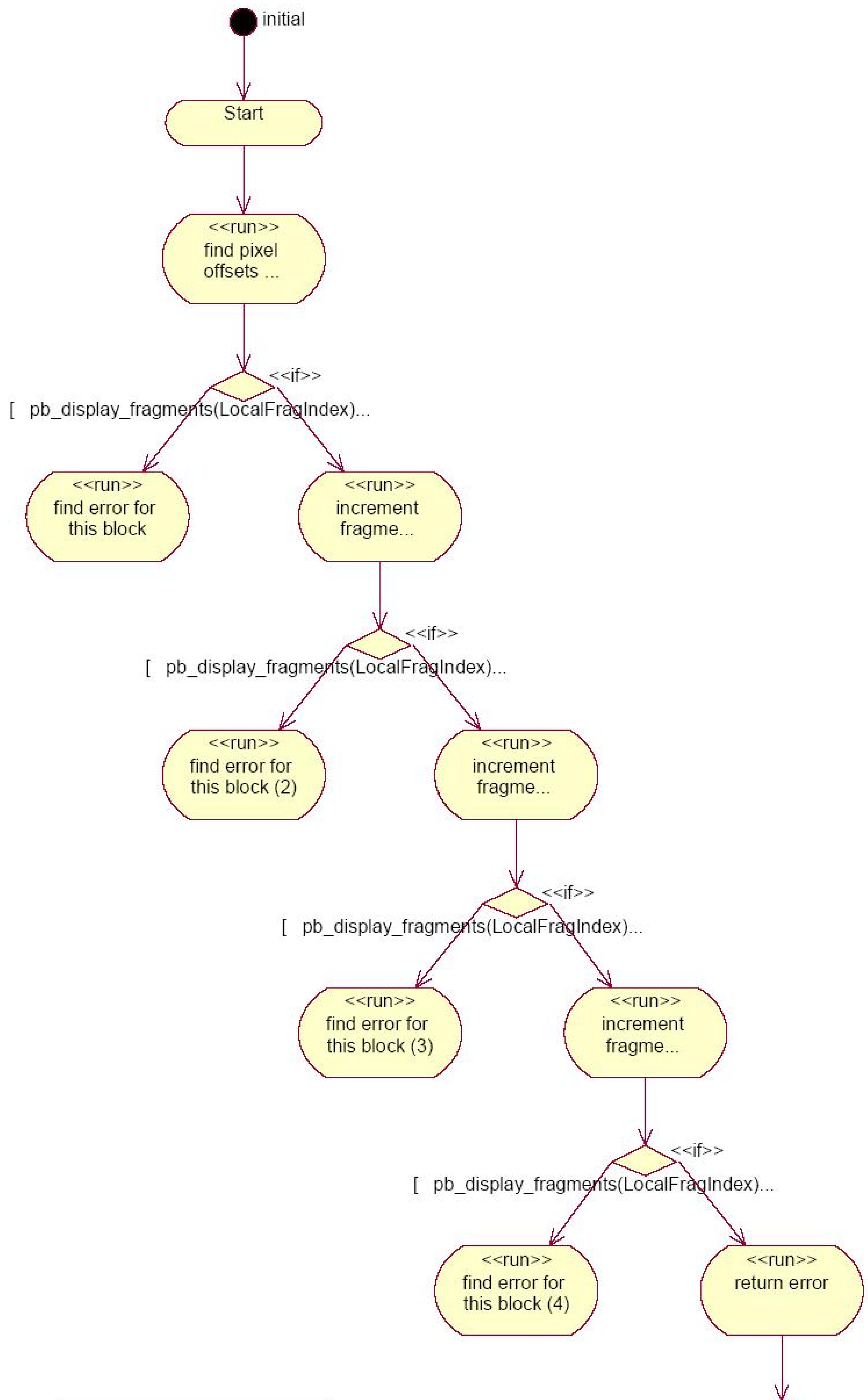


Figure 4.32: Activity Diagram of the GetMBInterError module

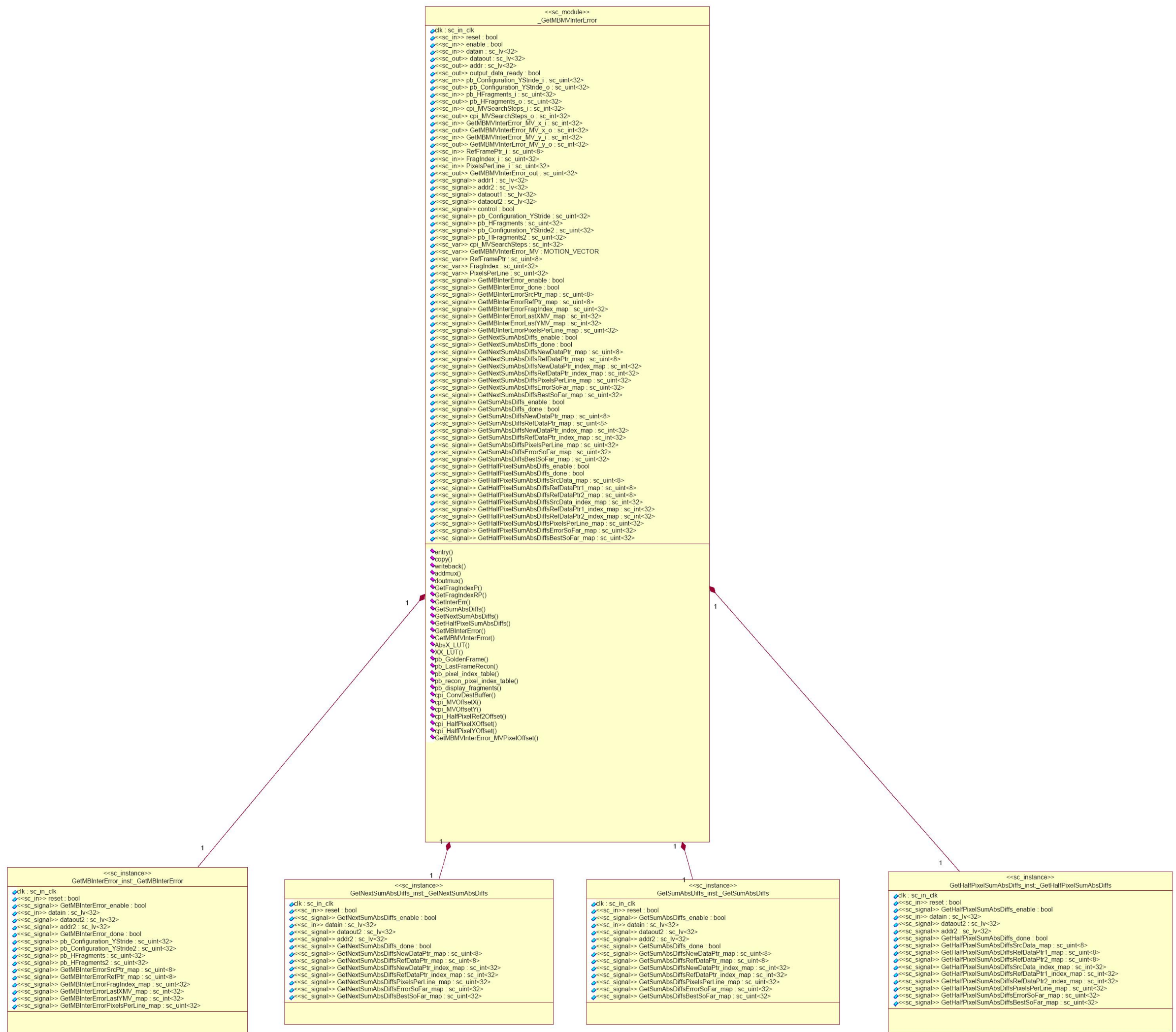


Figure 4.33: Class Diagram of the GetMBMVInterError module

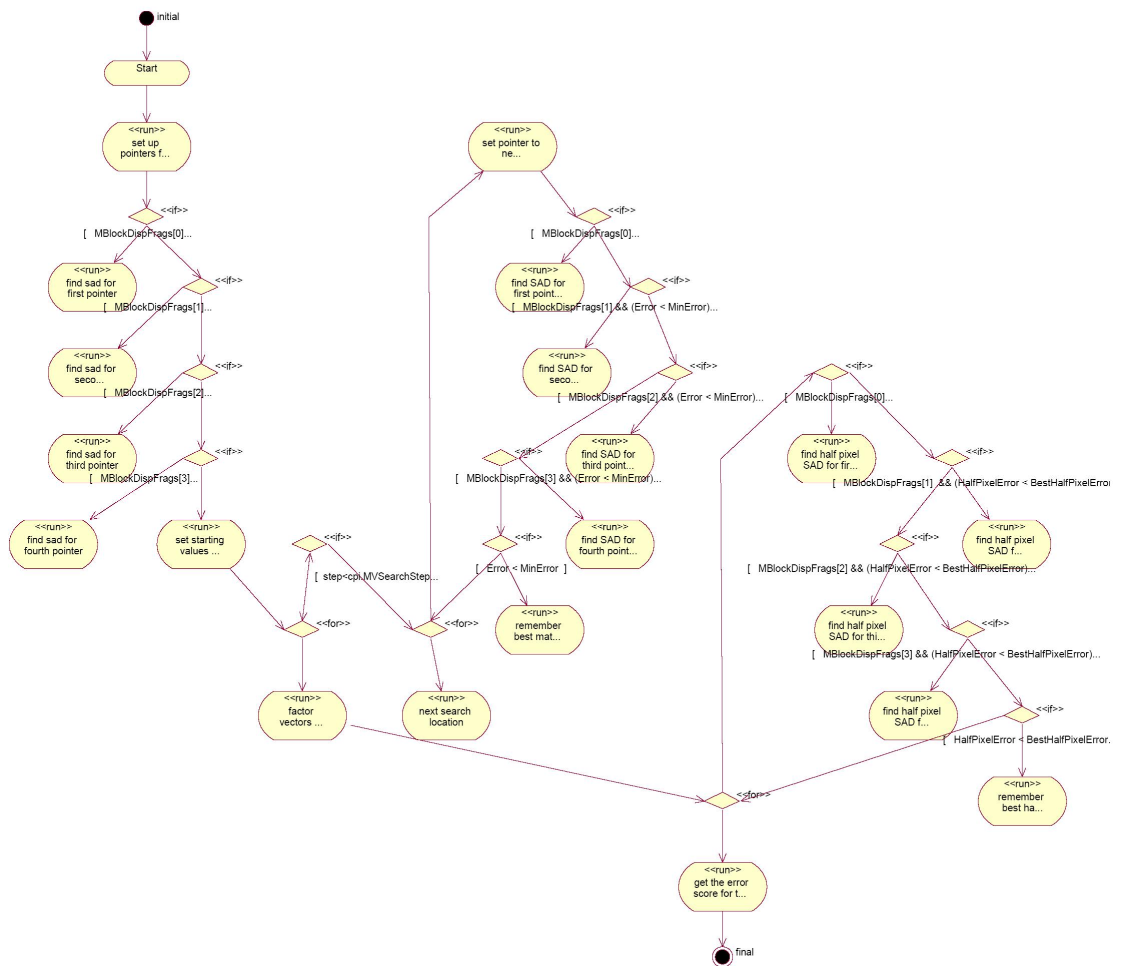


Figure 4.34: Activity Diagram of the GetMBMVInterError module

The first stage consists of initialisations. A note is then made of which of the fragments in the macroblock are coded. Pointers are then set up for the source blocks and the reference block. The first step is the find the sum of absolute differences between the source and reference blocks using a (0,0) motion vector.

Following this, we go through a pre-decided number of search steps to search for a new motion vector. At each search step, we set one pointer to the source pixel and the reference pointer, in turn, to the 8 surrounding pixels to the (0,0) reference pointer. For each of these 8 vectors, in each of the numerous search steps, an error score is computed. The best error and the corresponding motion vector is remembered.

Now the vectors are factored to do the half pixel pass. The reference pointer is set to the best reference pointer found through the last pass. We then get the half pixel error for each half pixel offset from this reference pointer location. Finally, we get the error score for the chosen 1/2 pixel offset as a variance and return the score of the best matching block.

GetMBMVExhaustiveSearch

This module is one level below the PickModes module and its purpose is to calculate a macro block Motion Vector using an exhaustive search as opposed to the hierarchical search adopted by the previous module. Refer to Figure 4.35 for the class diagram.

This module has a number of lower level modules namely the GetSumAbsDiffs, GetHalfPixelSumAbsDiffs, GetMBInterError used to calculate the Sum of Absolute Differences for a block in the current frame with one in a reference frame, Sum of Absolute Differences at half pixel accuracy (by averaging two pixels in the reference frame) for a block in the current frame with one in a reference frame, and to find the MB Inter Error as described earlier. The activity diagram is shown in Figure 4.36.

The first stage consists of initialisations. A note is then made of which of the

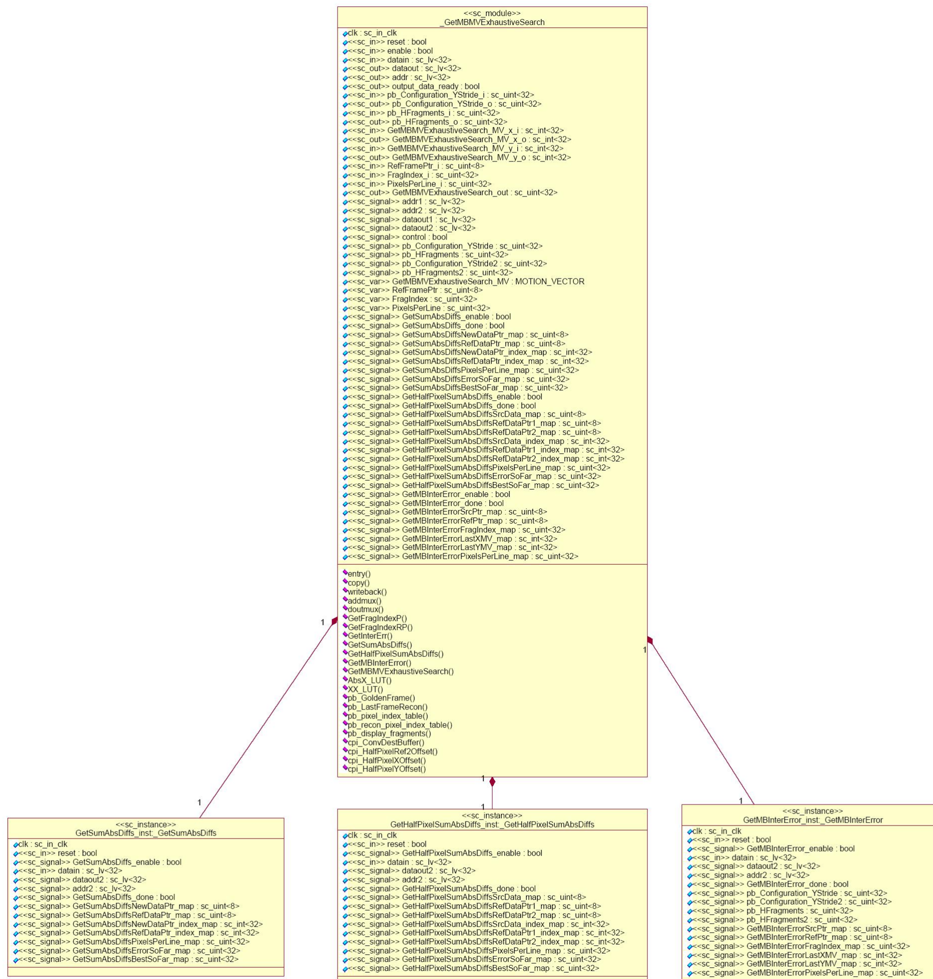


Figure 4.35: Class Diagram of the GetMBMVExhaustiveSearch module

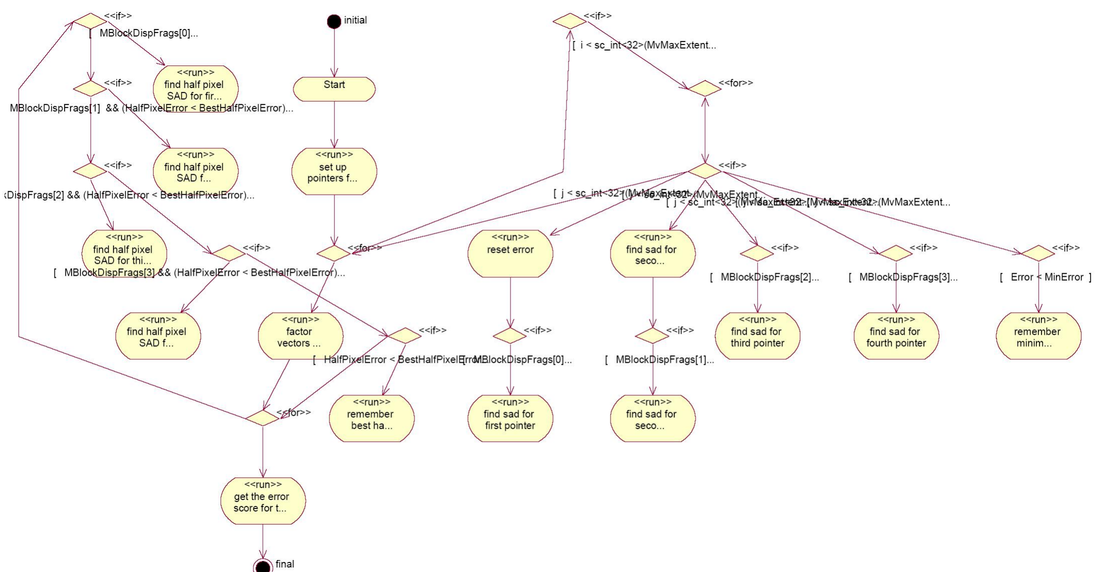


Figure 4.36: Activity Diagram of the GetMBMVExhaustiveSearch module

fragments in the macroblock are coded. Pointers are then set up for the source blocks and the reference block. The reference pointer is set back in the x and y directions by a pre-decided "maximum MV extent" which determines the maximum search distance in half pixel increments that we are allowed to undertake.

To start with, the whole range (MaxMVExtent*MaxMvExtent) of pixels is traversed by the reference pointer and the sum of absolute differences is found at each point. The best error and the corresponding motion vector is noted after this process.

The vectors are then factored to 1/2 pixel resolution. The reference pointer is set to the best pointer found earlier and the the half pixel error for each half pixel offset is obtained. Finally, the error score for the chosen 1/2 pixel offset as a variance is computed and we return the score of the best matching block.

GetFOURMVExhaustiveSearch

This module is one level below the PickModes module and its purpose is to calculate a Motion Vector for each Y block in a macro block using an exhaustive search as opposed to finding one for the entire macroblock as in the previous module. Refer to Figure 4.37 for the class diagram.

This module has a number of lower level modules namely the GetSumAbsDiffs, GetHalfPixelSumAbsDiffs, GetMBInterError used to calculate the Sum of Absolute Differences for a block in the current frame with one in a reference frame, Sum of Absolute Differences at half pixel accuracy (by averaging two pixels in the reference frame) for a block in the current frame with one in a reference frame, and to find the MB Inter Error as described earlier. The activity diagram is shown in Figure 4.38 and Figure 4.39:

The 4MV mode is only deemed to be valid if all four Y blocks are to be updated. Hence this is first verified. If this is found to be true, we first reset the error score and then get the error component from each coded block.

Figure 4.37: Class Diagram of the GetFOURMVExhaustiveSearch module

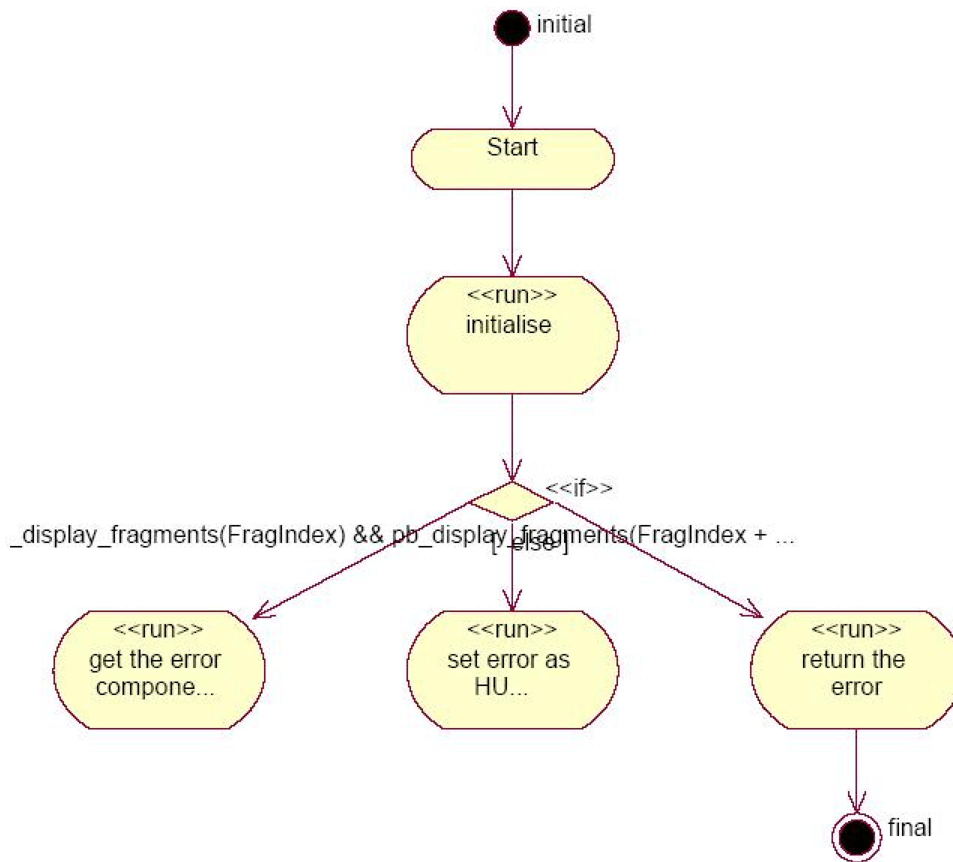


Figure 4.38: Main Activity Diagram of the GetFOURMVExhaustiveSearch module

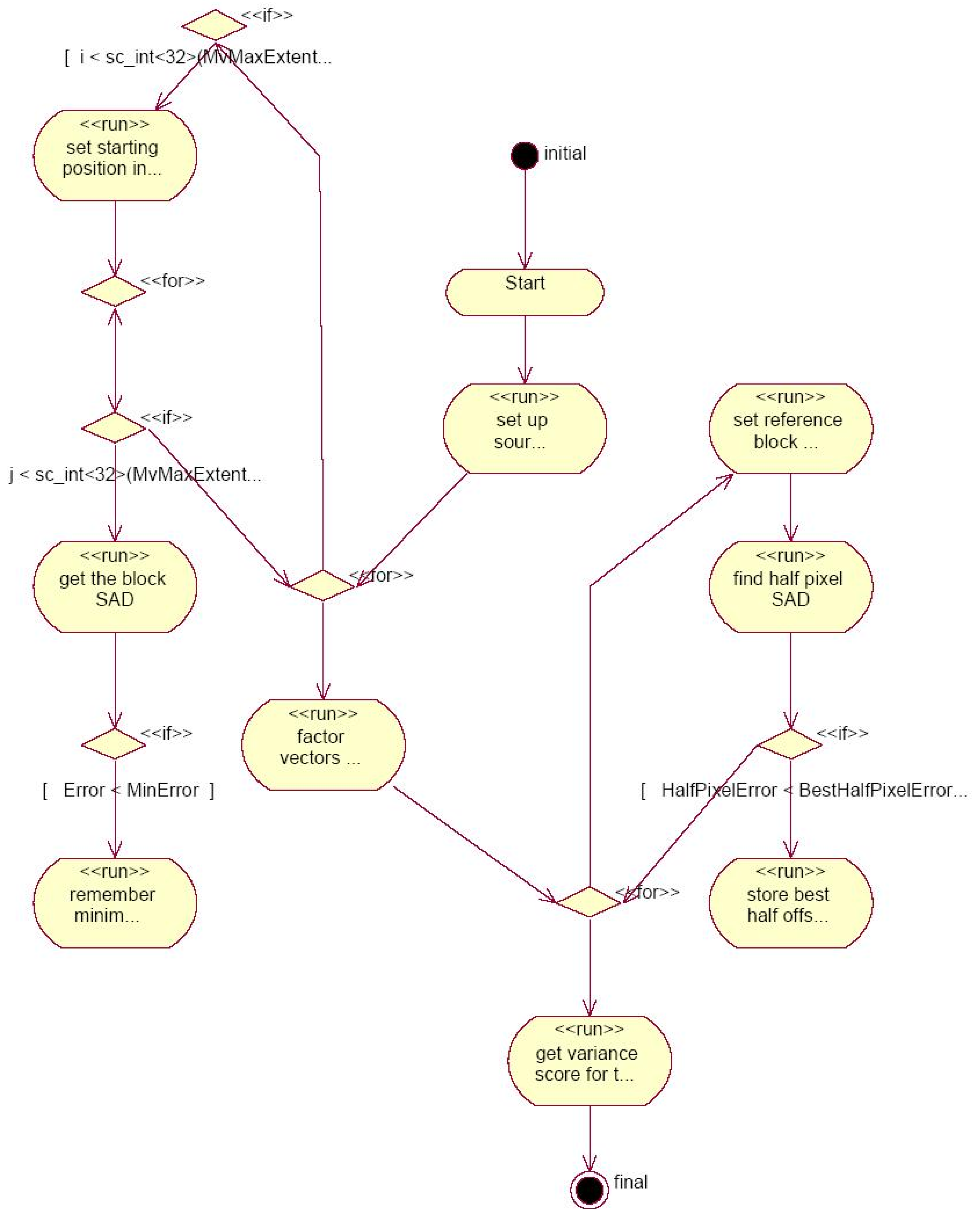


Figure 4.39: A Sub Activity Diagram of the GetFOURMVExhaustiveSearch module

The process of getting the error component for a particular block is as follows. Firstly initialisations are performed. Pointers are then set up for the source block and the reference block. The reference pointer is set back in the x and y directions by the pre-decided “maximum MV extent” which was described in the previous module.

To start with, the whole range (MaxMVExtent*MaxMvExtent) of pixels is traversed by the reference pointer and the sum of absolute differences is found at each point for this particular block. The best error and the corresponding motion vector is noted after this process.

The vectors are then factored to 1/2 pixel resolution. The reference pointer is set to the best pointer found earlier and the the half pixel error for each half pixel offset is obtained. Finally, the error score for the chosen 1/2 pixel offset as a variance is computed and we return the score of the best matching block.

QuadCodeDisplayFragments

This module is the only module below the UpdateFrame module described earlier. The purpose of this module is to code the frame using the quad tree method. Refer to Figure 4.40 for the class diagram.

This module has a number of lower level modules as can be seen, namely the QuadCodeComponent, DPCMTokeniseBlock, and PackCodedVideo blocks whose function it is to code the display_fragments array of a particular component (plane) of the frame, tokenise a DCT block, and create an output bitstream repectively.

Figure 4.40: Class Diagram of the QuadCodeDisplayFragments module

Figure 4.41: Activity Diagram of the QuadCodeDisplayFragments module

Figure 4.41 describes the dynamic behaviour of this module. To start with initialisations are performed. We then proceed to encode and tokenise the Y, U and V components with the help of the QuadCodeComponent module. The next few steps are repeated for each of the three planes (Y,U,V). First we initialize our array of last used DC Components. We then traverse the whole plane, fragment by fragment. We then proceed to perform prediction if fragment is coded and is on a non intra frame, or if all fragments are intra. The prediction is performed as follows. For each block, the original DC values of the surrounding blocks are found. We also decide which of these fragments can be used for prediction. Only fragments which are coded and which come from the same frame as the one we are predicting satisfy this condition. If no suitable predictor is found around the block we find the nearest one that is coded. If none match we fall back to the last one. On the other hand if a suitable predictor is found, we compute a predictedDC value based on the surrounding DC coefficient values and subtract this value computed from the quantised data.

We then pack the DC tokens and adjust the ones we couldn't predict. Following this we bit pack the video data using the packcodedvideo module, and end the bit packing run. Finally we measure the inter reconstruction error for all the blocks that were coded for use as part of the recovery monitoring process in subsequent frames.

GetIntraError

This simple module is contained within the GetMBIntraError module described earlier and performs the function of calculating a variance score for a block. Refer to Figure 4.42 for the class diagram.

This module is a leaf module and contains no further sub modules.

The activity diagram of the module is shown in Figure 4.43 and is very simple. The function finds the sum of each pixel location and the sum of squares of the difference of

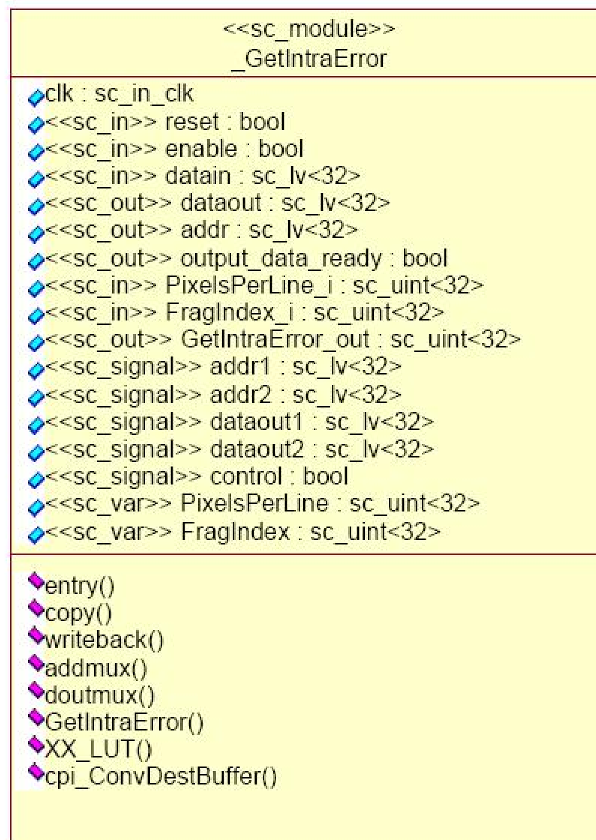


Figure 4.42: Class Diagram of the GetIntraError module

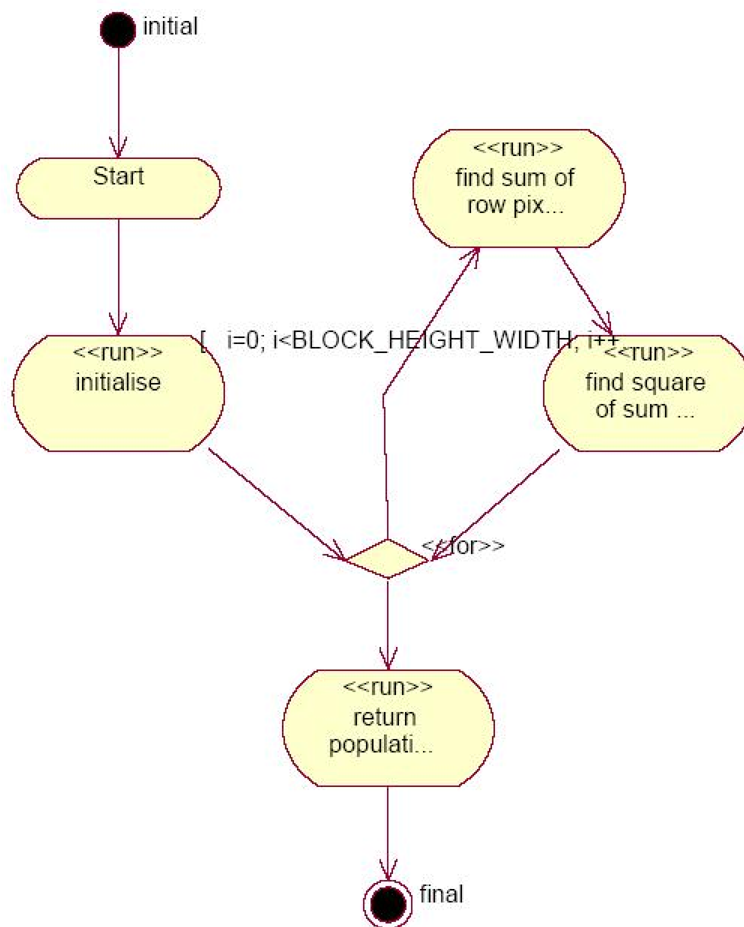


Figure 4.43: Activity Diagram of the GetIntraError module

each pixel with 255. Finally it computes a population variance as mismatch metric.

GetInterErr

This simple module is contained within the GetMBInterError module described earlier and performs the function of calculating a difference error score for two blocks. Refer to Figure 4.44 for the class diagram.

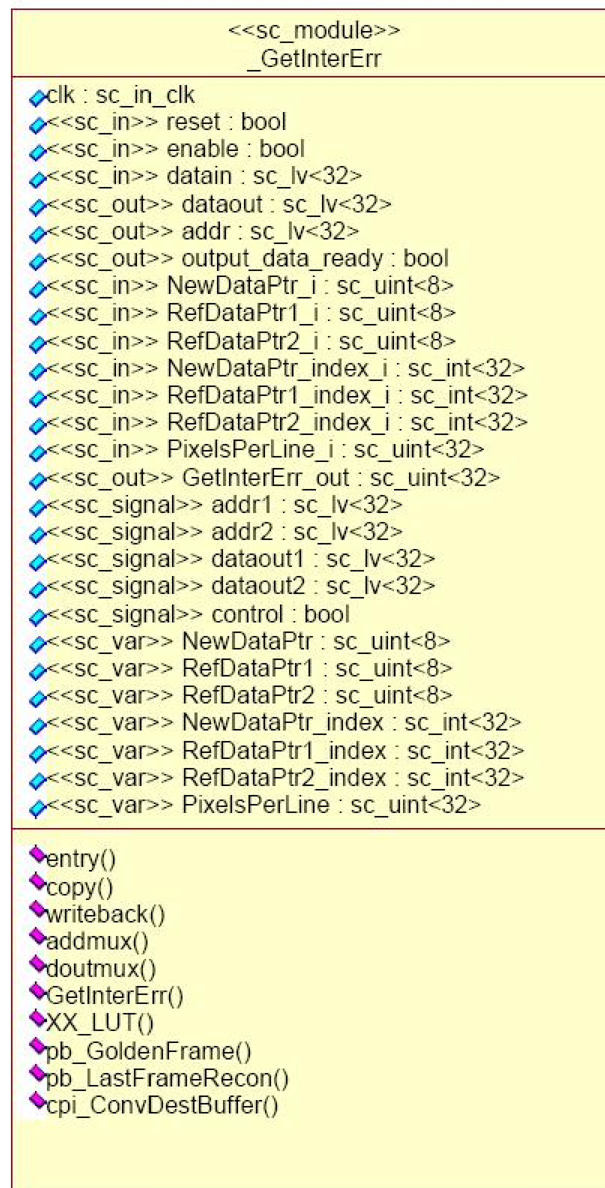


Figure 4.44: Class Diagram of the GetInterErr module

This module is a leaf module and contains no further sub modules.

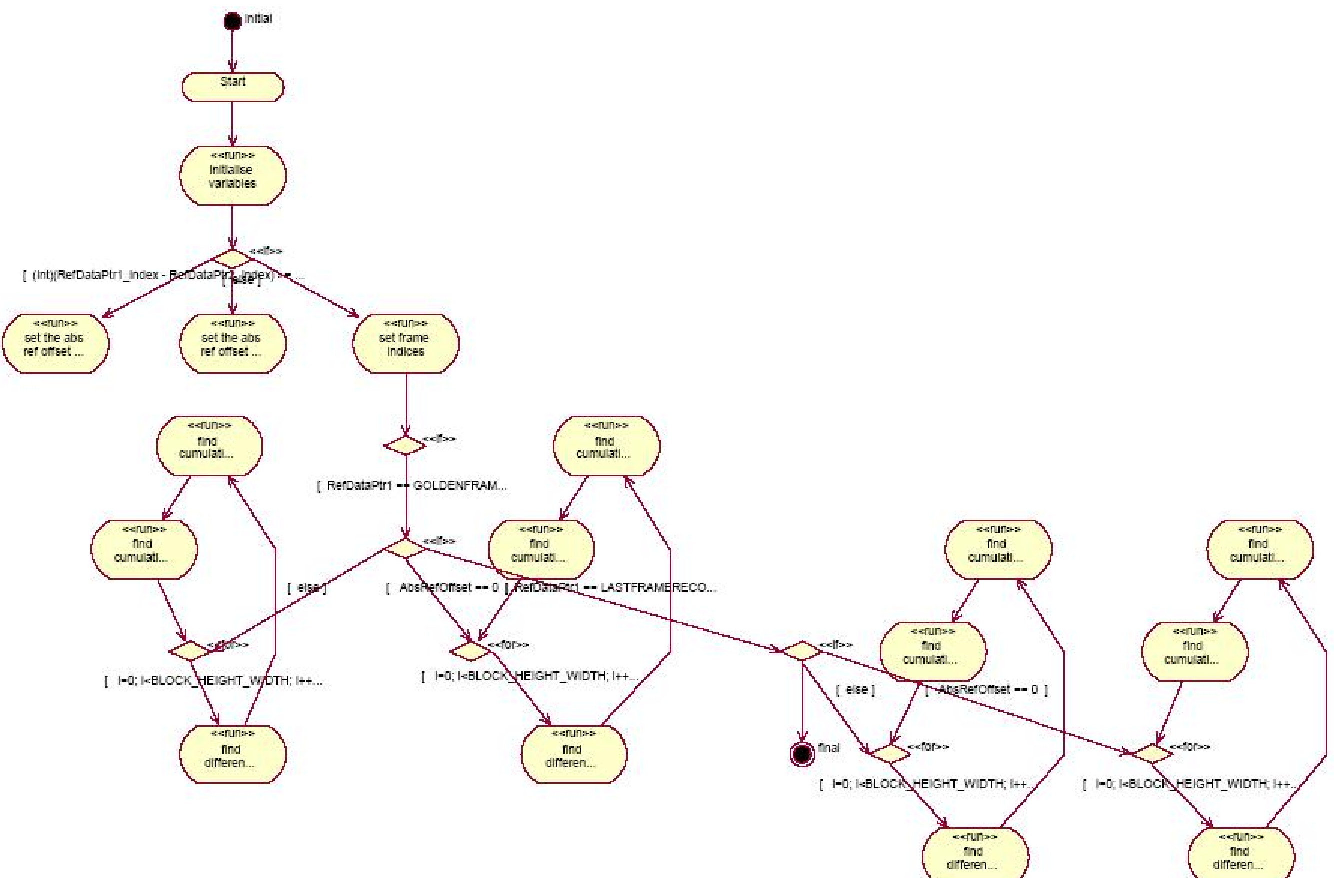


Figure 4.45: Activity Diagram of the GetInterErr module

The activity diagram of the module is shown in Figure 4.45. Firstly, a mode of interpolation is chosen based upon on the offset of the second reference pointer. (one reference / two reference interpolation) If it is a simple one reference interpolation, the function finds the sum of the difference of each pixel location between the two blocks, and the sum of the square of the difference of each pixel location between the two blocks minus 255. For a two reference interpolation, the function finds the sum of the difference of each pixel location in the source frame with the average of two pixel locations in the reference frame, and the sum of the square of the value found earlier minus 255. Finally it computes a population variance as mismatch metric.

GetSumAbsDiffs

This module is contained within a number of higher level modules described earlier and, as can be expected, performs the function of calculating the sum of the absolute differences. Refer to Figure 4.46 for the class diagram.

This module is a leaf module and contains no further sub modules.

The activity diagram of the module is shown in Figure 4.47. The functioning is very simple. The function goes to each pixel in the source and reference blocks and finds the sum of the absolute difference between the two pixels.

GetNextSumAbsDiffs

This module is contained within a number of higher level modules described earlier and, as can be expected, performs the function of calculating the sum of the absolute differences but with a minor difference namely that of having a breakout clause.Refer to Figure 4.48 for the class diagram.

This module is a leaf module and contains no further sub modules.

The activity diagram of the module is shown in Figure 4.49. The functioning is very

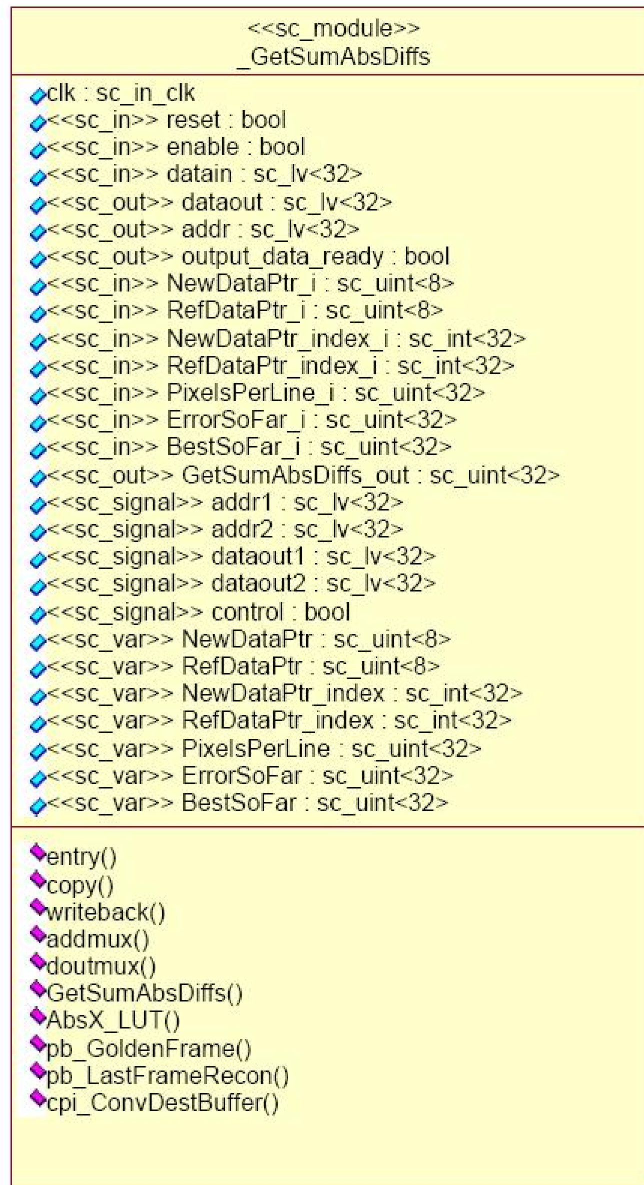


Figure 4.46: Class Diagram of the GetSumAbsDiffs module

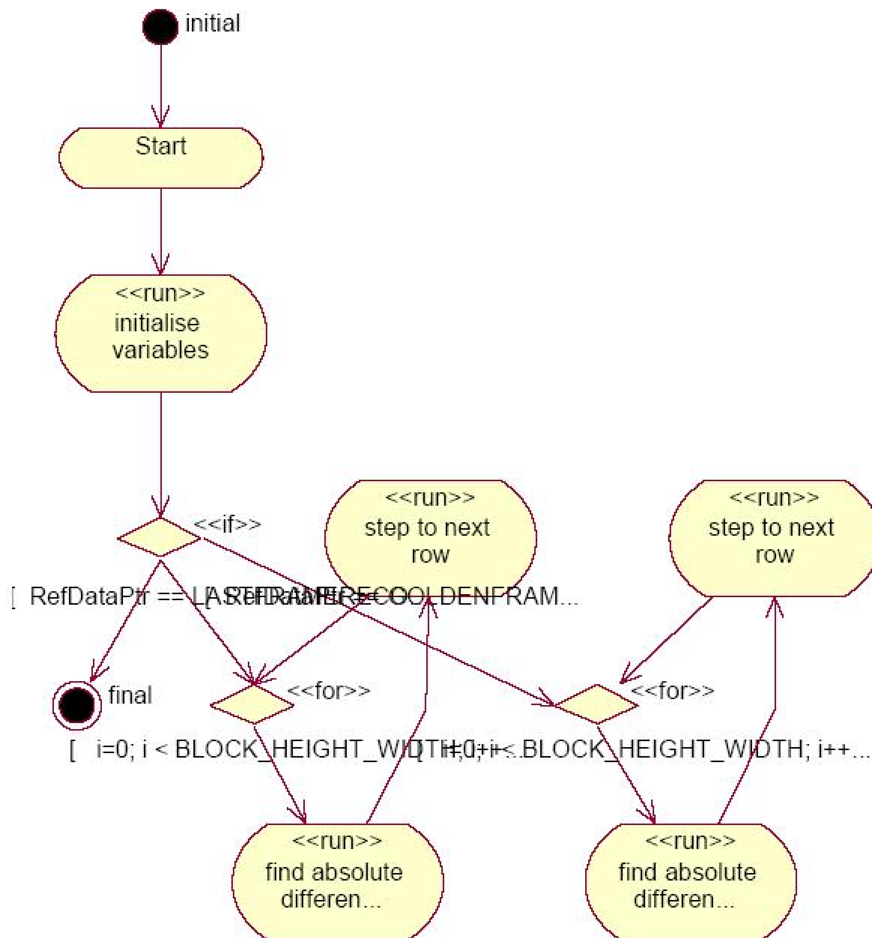


Figure 4.47: Activity Diagram of the GetSumAbsDiffs module

Figure 4.48: Class Diagram of the GetNextSumAbsDiffs module

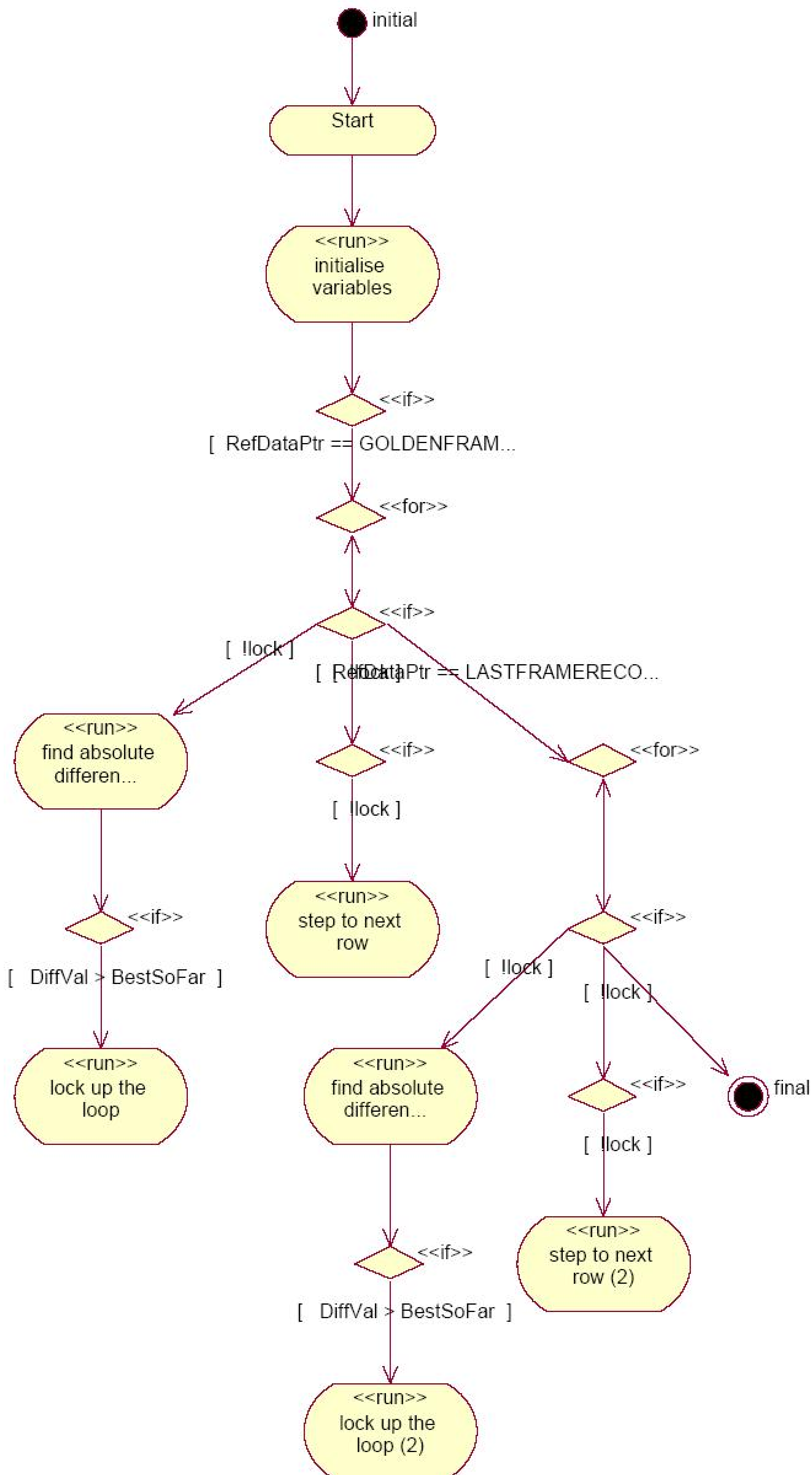


Figure 4.49: Activity Diagram of the GetNextSumAbsDiffs module

simple. The function goes to each pixel in the source and reference blocks and finds the sum of the absolute difference between the two pixels. The module stops accumulating error values when the error value exceeds the best error found so far.

GetHalfPixelSumAbsDiffs

This module is contained within a number of higher level modules described earlier and, as can be expected, performs the function of calculating the sum of the absolute differences against half pixel interpolated references. Refer to Figure 4.50 for the class diagram.

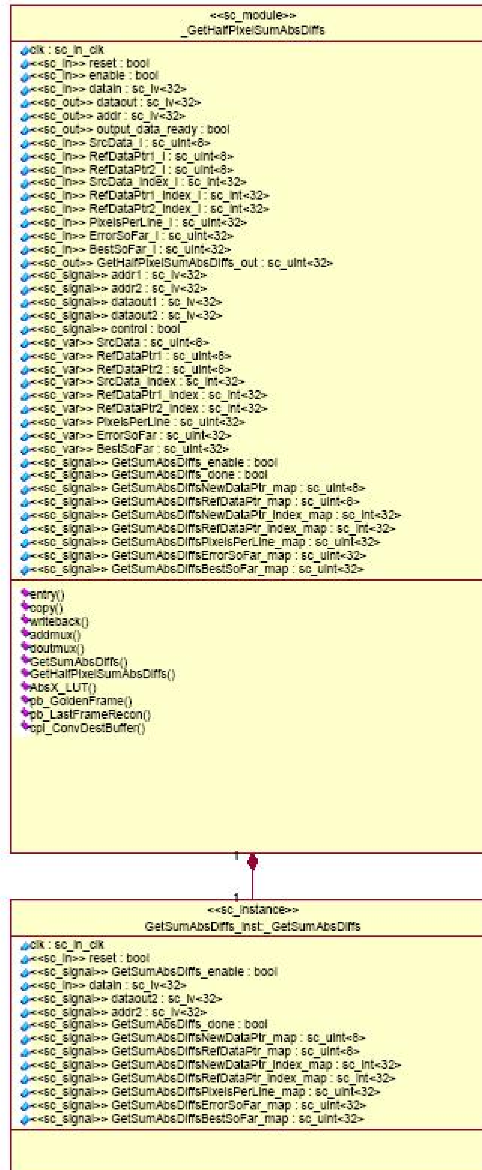
This module makes use of the GetSumAbsDiffs module described earlier.

The activity diagram of the module is shown in Figure 4.51. The reference offset, or the difference in the locations pointed to by the two reference pointers is calculated. If this offset is zero, we proceed to find a simple Sum of Absolute Differences making use of the GetSumAbsDiffs lower level module. If not, the function goes to each pixel in the source and reference blocks and finds the sum of the absolute difference between the source pixel and the average of two pixels pointed to by the two reference pointers. The module stops accumulating error values when the error value exceeds the best error found so far.

QuadCodeComponent

This module is one of the sub modules of the QuadCodeDisplayFragments module described earlier. It codes the display_fragments array as a quad-tree starting with 32x32 Super-Blocks, then 16x16 Macro-Blocks, and finally 8x8 Blocks. Refer to Figure 4.52 for the class diagram.

This node has one child , the 'TransformQuantizeBlock' module whose purpose it is to DCT transform and quantise the block. The activity diagram of this module is



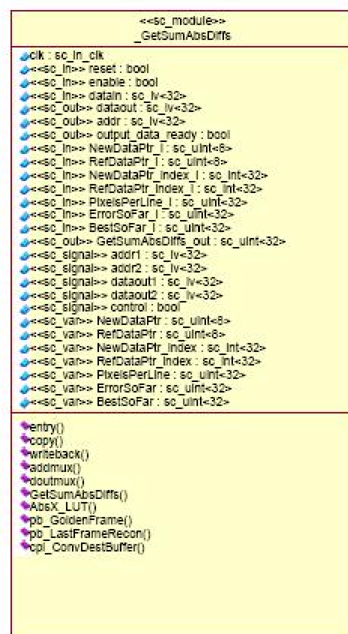


Figure 4.50: Class Diagram of the GetHalfPixelSumAbsDiffs module

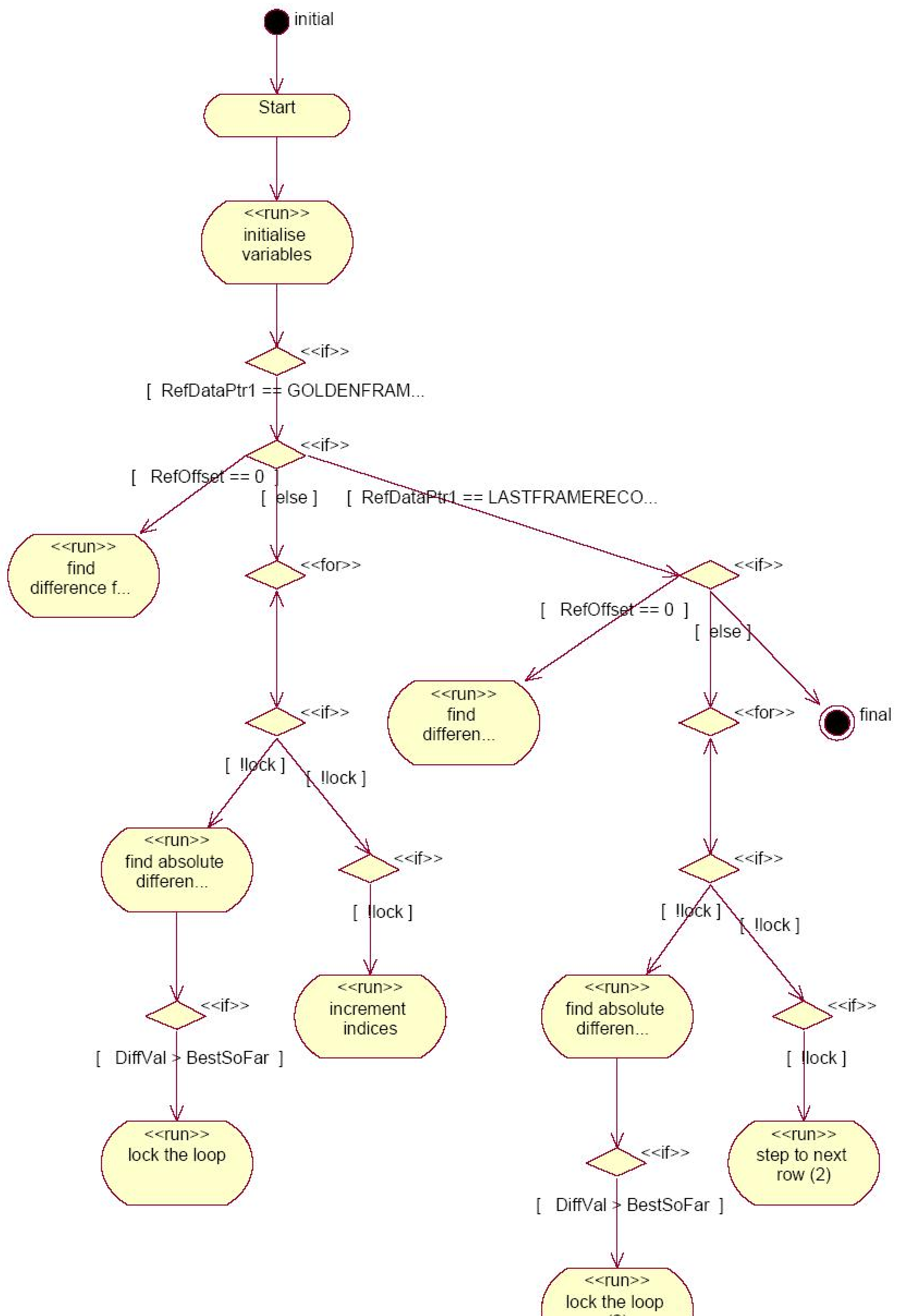


Figure 4.51: Activity Diagram of the GetHalfPixelSumAbsDiffs module

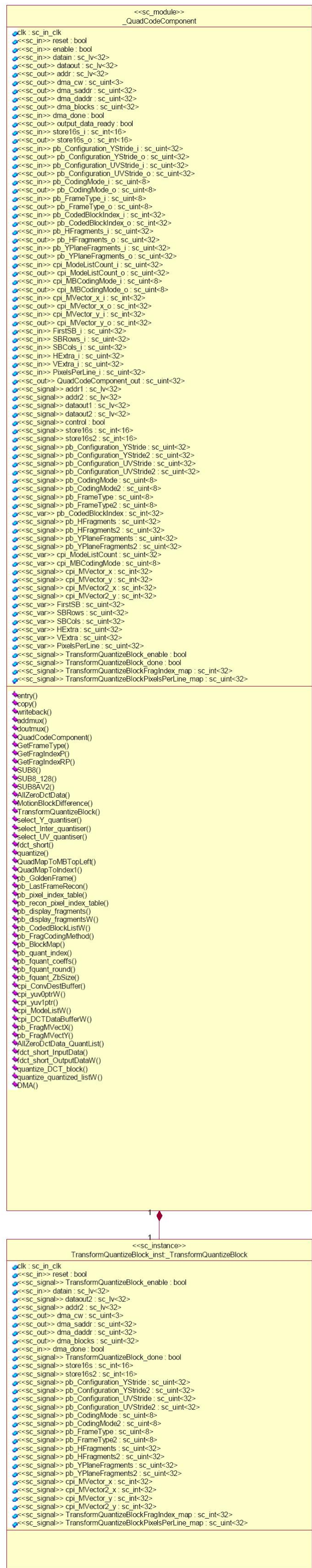


Figure 4.52: Class Diagram of the QuadCodeComponent module

shown in Figure 4.53.

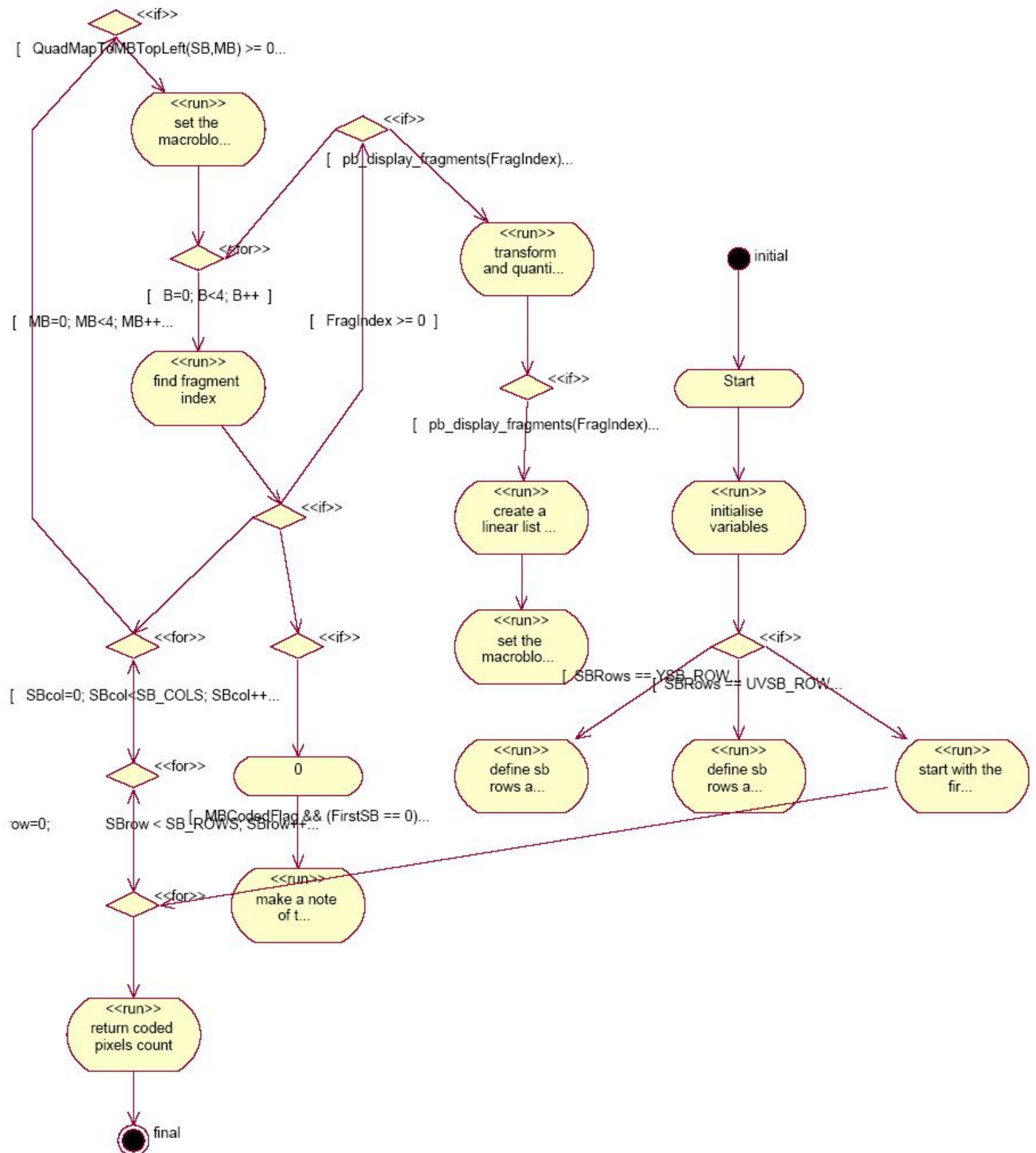


Figure 4.53: Activity Diagram of the QuadCodeComponent module

It starts with various initialisations. We then proceed to actually transform and quantize the image now that we've decided on the modes. We parse the image in quad-tree ordering. For each Superblock in a superblock row , for each superblock in a superblock column, and for each macroblock in that superblock, we go block by block,

checking for whether the block lies in the frame and whether it is coded. If it is we proceed to transform and quantise the block with the help of the lower level module. If the block has not got struck off (no MV and no data generated after DCT)we mark it and the associated MB as coded and create a linear list of coded block indices. Finally, if this macroblock is marked as coded and we are in the Y plane then the mode list needs to be updated. We therefore make a note of the selected mode in the mode list.

DPCMTokeniseBlock

This module is another one of the sub modules of the QuadCodeDisplayFragments module described earlier. It codes a DCT block under the assumption that motion vectors and modes are defined at the macroblock level. Refer to Figure 4.54 for the class diagram.

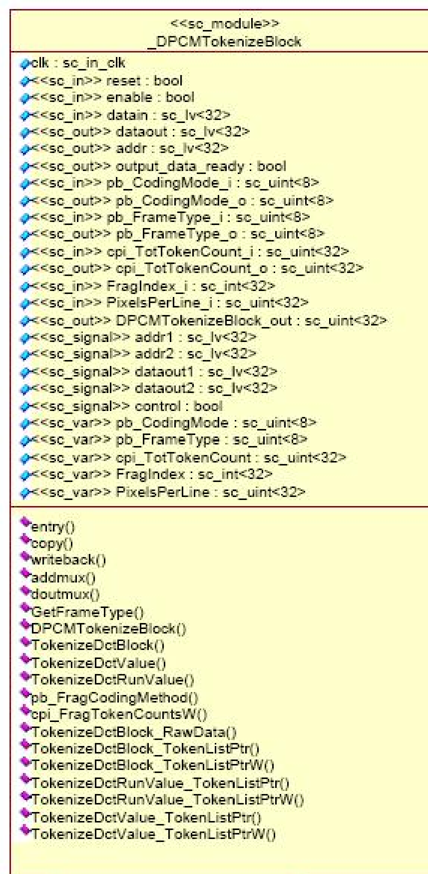


Figure 4.54: Class Diagram of the DPCMTokenizeBlock module

As seen above this module is a leaf node and has no further children.

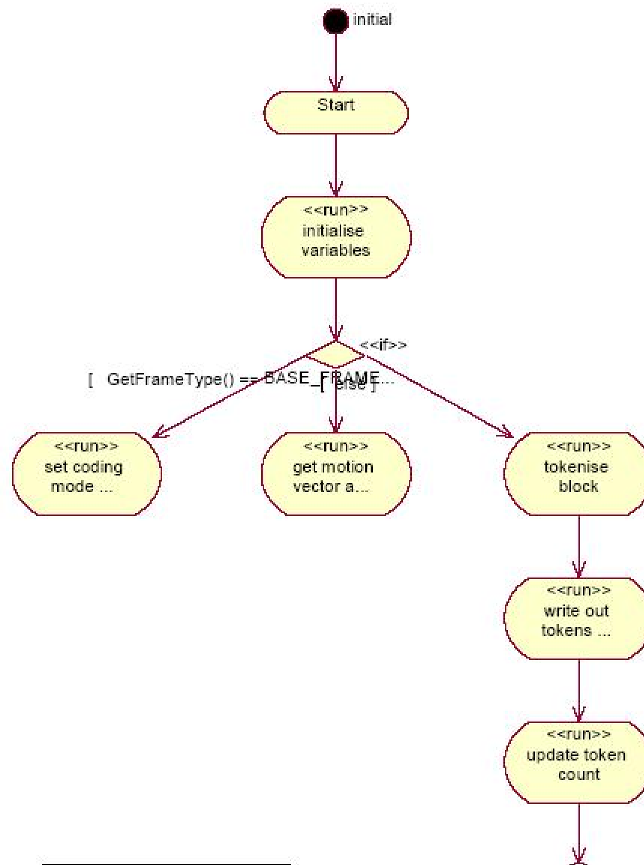


Figure 4.55: Activity Diagram of the DPCMTokenizeBlock module

The main activity diagram is shown in Figure 4.55. To start with, initialisations are performed. The coding mode for the block is then determined. Once done, we proceed to tokenise the DCT block. The activity diagram for this process is shown in Figure 4.56.

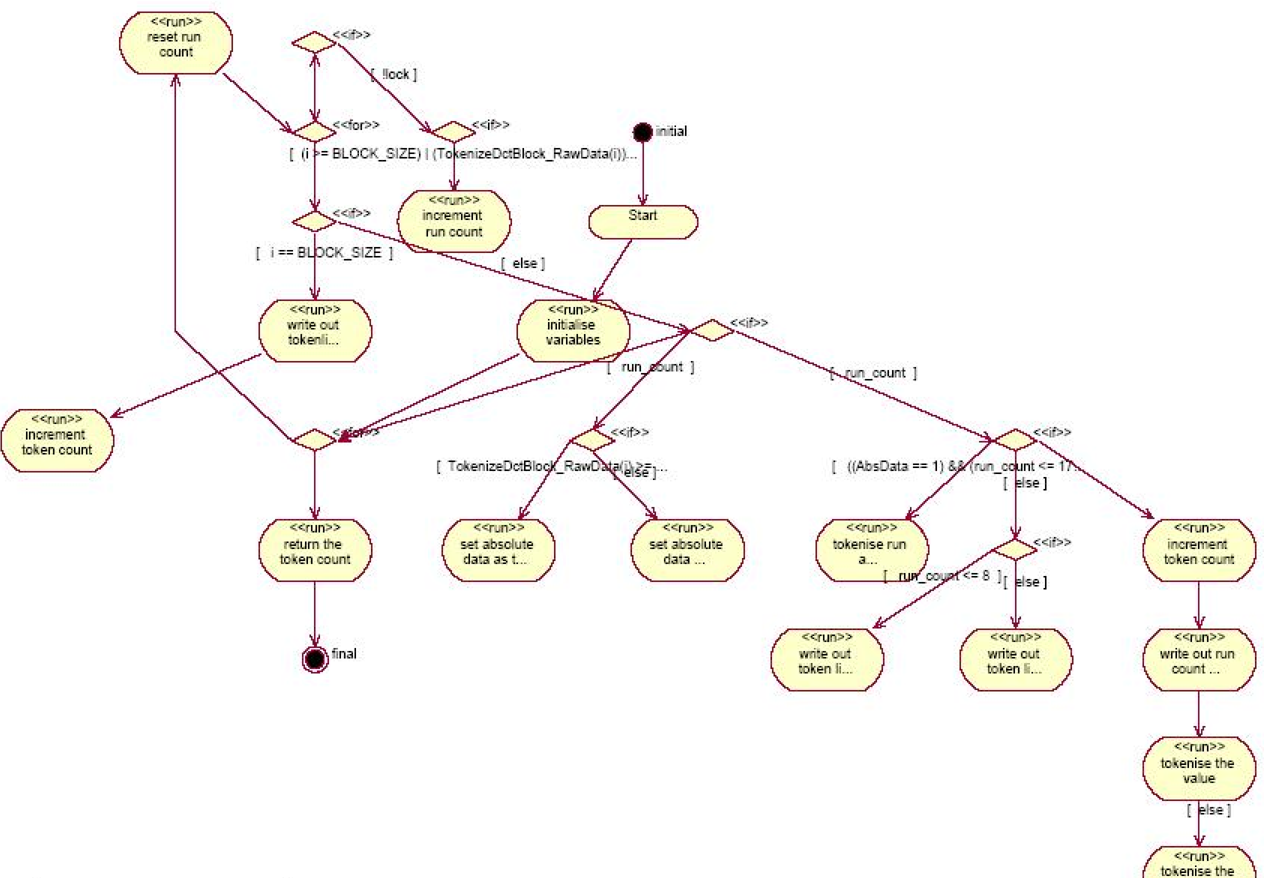


Figure 4.56: Activity Diagram of the TokenizeDCTBlock module of the DPCMTokenizeBlock module

The above activity diagram describes the tokenising process which encodes a DCT block into a stream of tokens. Most tokens are followed by an additional bits (up to 8 bits of additional data). As described in the overview of VP3, RLE is used to record the number of zero-value coefficients that occur before a non-zero coefficient. We first run through the block, searching for a zero run. A short zero run followed by a low data value is coded as a composite token. If there is a long non-EOB run or a run followed by a value token ¿ MAX_RUN_VAL then the run and token are coded separately. The values are then tokenized as a category value and a number of additional bits that define the position within the category.

PackCodedVideo

The PackCodedVideo module is another one of the sub modules of the QuadCodeDisplayFragments module described earlier. It takes the encoded token lists etc. and creates an output bitstream. Additional restrictions to control bitrate are also applied at this point. Refer to Figure 4.57 for the class diagram.

PackCodedVideo performs a number of functions which it deputes to a number of leaf modules shown in the UML static diagram. These are the ClearDownQFragData, EncodeAcTokenList, EncodeDcTokenList, PackAndWriteDFArray, and PackModes modules.

As seen in Figure 4.58, we first perform various initialisations and reset the count of second order optimised tokens. We then calculate the bit rate at which this frame should be capped and blank the various fragment data structures before we start. If the frame is not a key frame, we pack the quad tree fragment mapping. We then pack and code the mode list and motion vectors. Each token is then packed followed by any outstanding EOB tokens. We then output the optimised DC token list using the appropriate entropy tables as described in the huffman coding process described earlier.

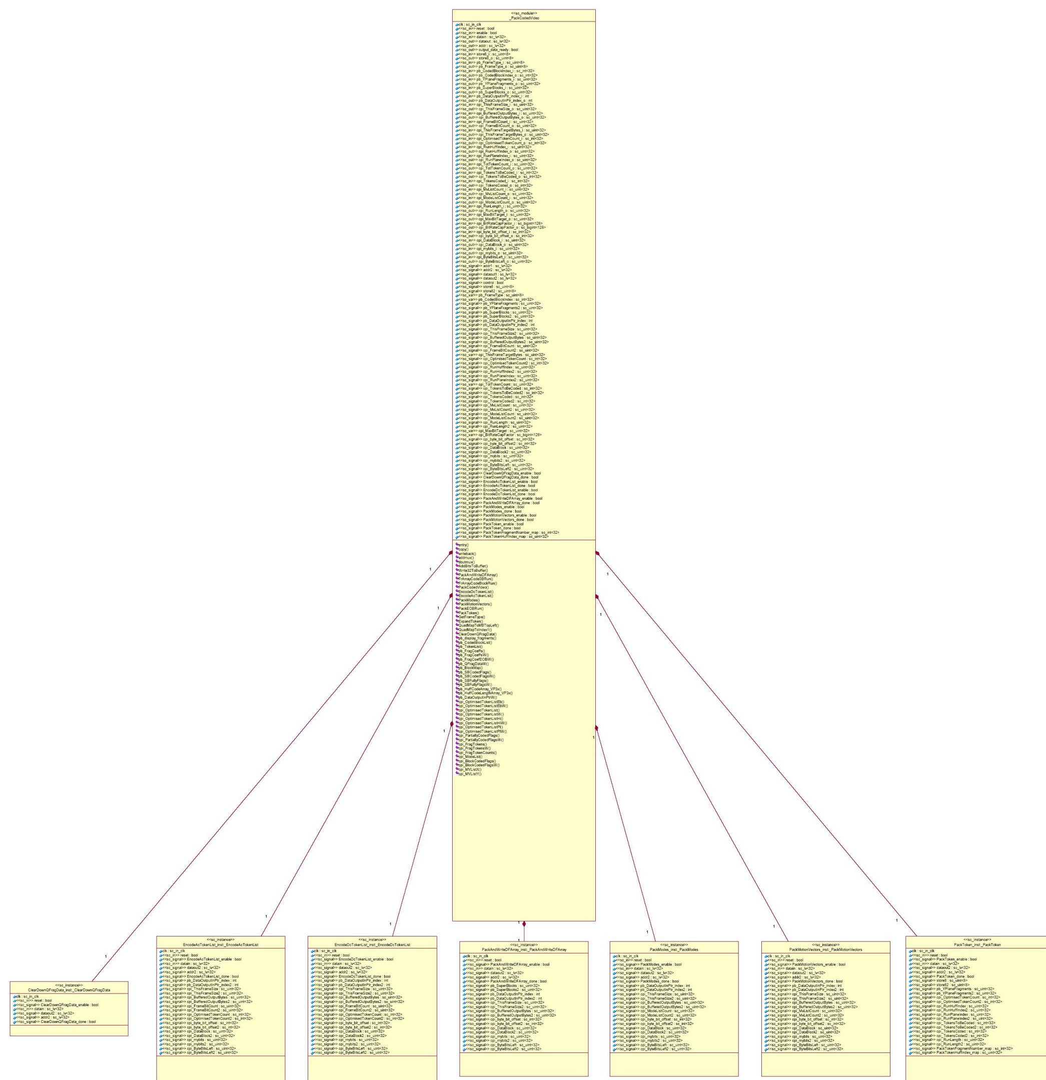


Figure 4.57: Class Diagram of the PackCodedVideo module

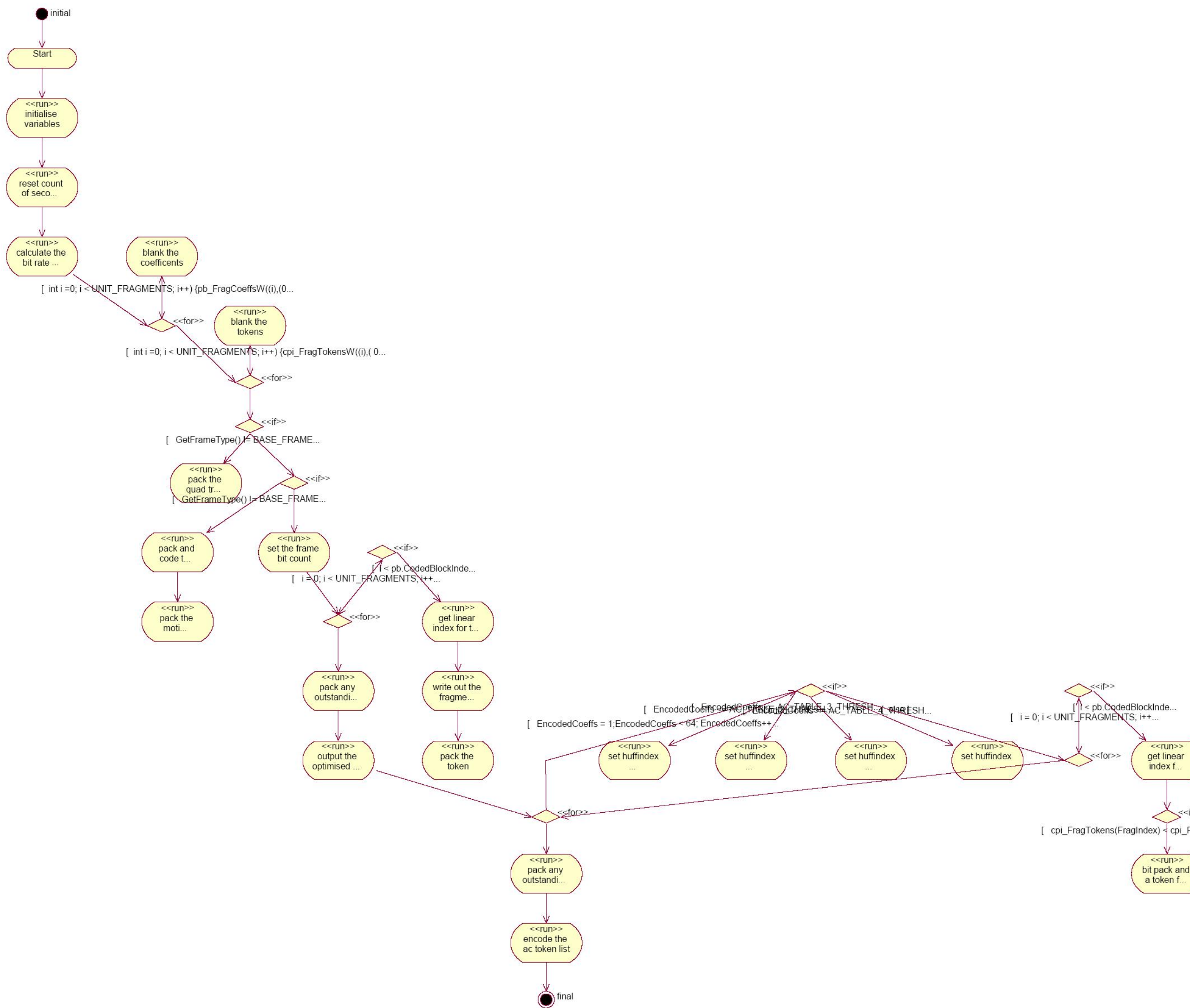


Figure 4.58: Activity Diagram of the PackCodedVideo module

The Ac tokens are then optimised and packed followed by any outstanding EOB tokens and finally outputted using the appropriate entropy tables.

TransformQuantizeBlock

This is a submodule of the QuadCodeComponent Block which codes a plane of the frame. The purpose of the block is to transform and quantise a block of data. Refer to Figure 4.59 for the class diagram.

As can be seen this block performs the transform and quantise process with the assistance of a number of lower level modules like the SUB8, SUB8_128, fdct_short, quantize, and MotionblockDifference which perform the functions of subtracting 2 8X8 blocks, either from each other or by 128, performing dct on the data and quantising the data or computing motion block residues.

The activity diagram is shown in Figure 4.60. First initilisations are performed and pointers initialised to the relevant buffers. Depending on whether it is a Y plane or UV plane various parameters are set. The coding mode for the block is then found out and set followed by the selection of the quantiser matrix and other plane related values. If the coding mode uses motion vectors we call the motionblockdifferences module to find block differences otherwise we call either the SUB8 or the SUB_128 module to find block differences depending on coding mode. Finally a 2D DCT transform is performed on the data and that transformed data is quantised.

ClearDownQFragData

This is a very simple sub module of the PackCodedVideo module. Its function is to clear down the data structure that is used to store quantised dct coefficients for each block. Shown in Figure 4.61 and Figure 4.62 are its static and dynamic UML descriptions respectively.

Figure 4.59: Class Diagram of the TransformQuantizeBlock module

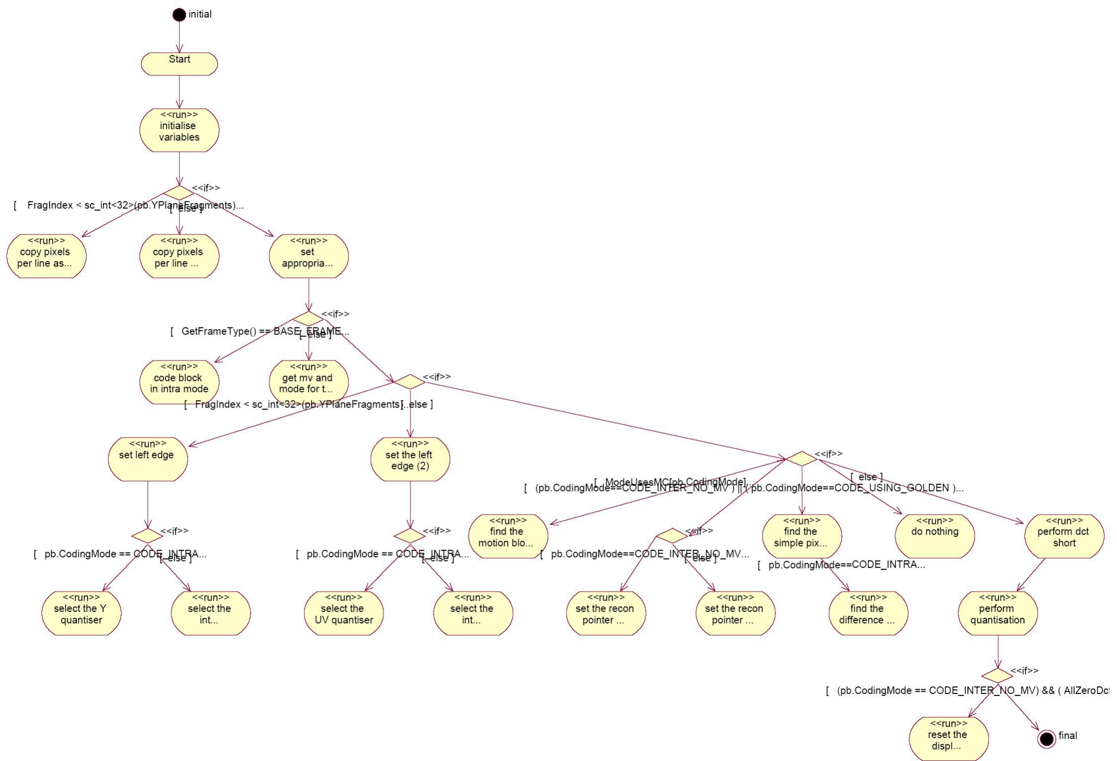


Figure 4.60: Activity Diagram of the TransformQuantizeBlock module

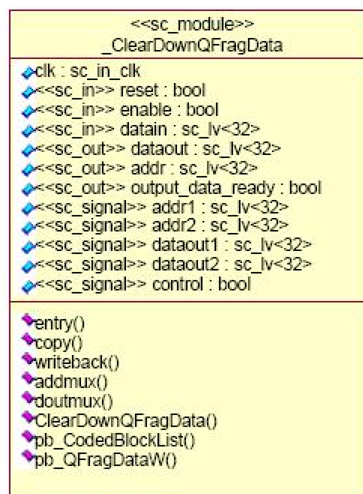


Figure 4.61: Class Diagram of the ClearDownQFragData module

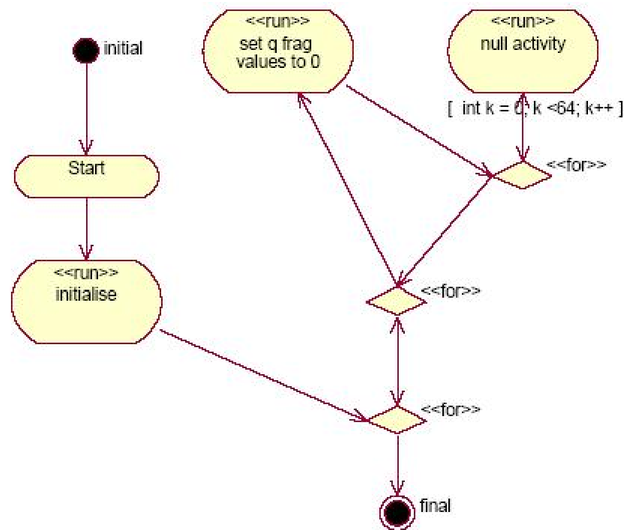


Figure 4.62: Activity Diagram of the ClearDownQFragData module

Its a very simple activity diagram. We merely run through the quantised fragment data buffer and set the values to zero.

EncodeDcTokenList

The EncodeDcTokenList is a sub module of the PackCodedVideo which as described earlier is responsible for taking the encoded token lists etc. and creating an output bitstream. This module in particular performs the function of outputting the DC token list using the selected entropy method. Refer to Figure 4.63 for the class diagram and Figure 4.64 for the activity diagram.

This is a leaf node as can be seen.

We start with initialisations. We then analyse the token list to see which is the best entropy table to use by first working out the number of bits required with each table option and then working out which table option is best for the Y plane. The DC Huffman table choice is then added to the bitstream. We proceed to do the same for the UV plane and finally encode the token list.

Figure 4.63: Class Diagram of the EncodeDcTokenList module

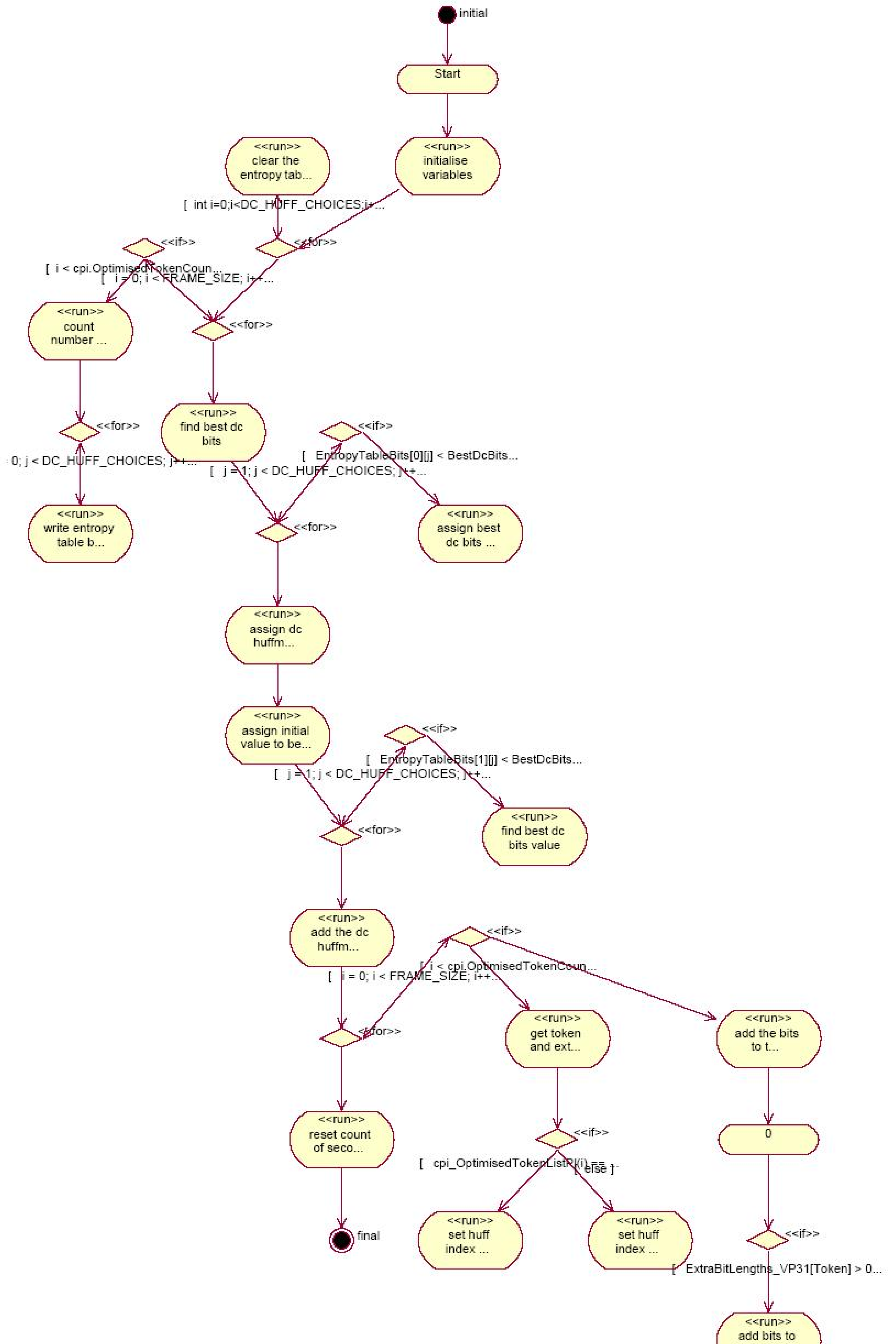


Figure 4.64: Activity Diagram of the EncodeDcTokenList module

EncodeAcTokenList

The EncodeAcTokenList is another sub module of the PackCodedVideo described earlier. This module performs the function of outputting the AC token list using the selected entropy method.Refer to Figure 4.65 for the class diagram and Figure 4.66 for the activity diagram.

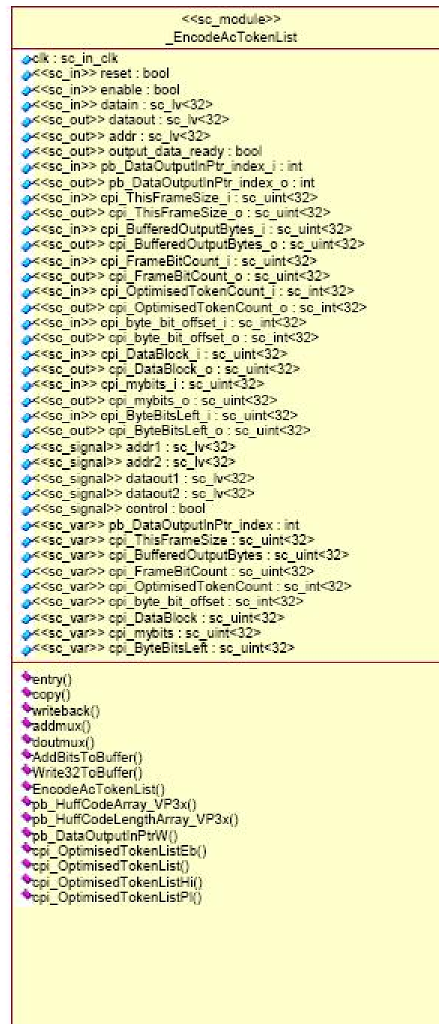


Figure 4.65: Class Diagram of the EncodeAcTokenList module

This is a leaf node as can be seen.

We start with initialisations. We then analyse the token list to see which is the best entropy table to use by first working out the number of bits required with each table option and then working out which table option is best for the Y plane. The AC-Y Huffman table choice is then added to the bitstream. We proceed to do the same for

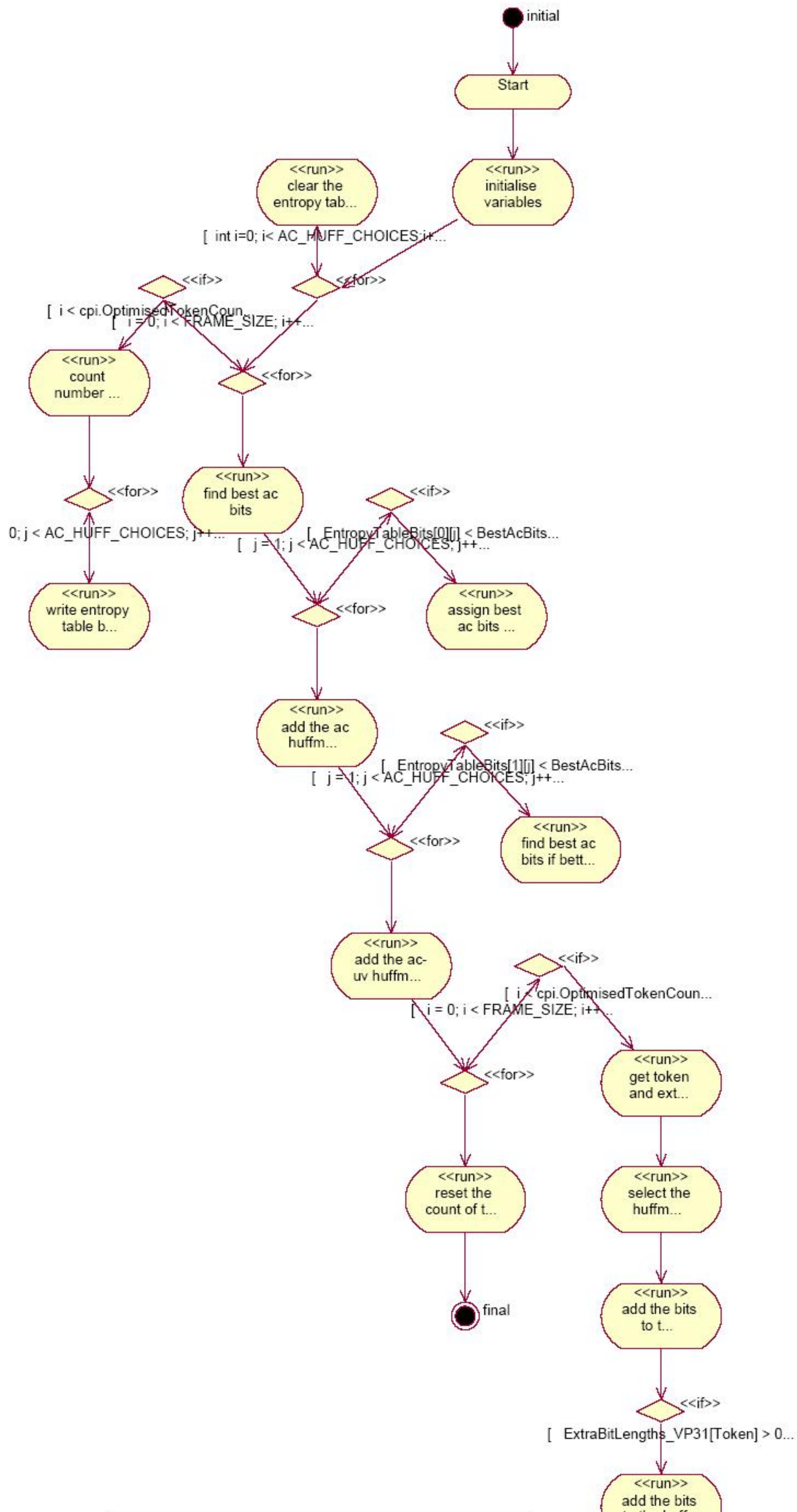


Figure 4.66: Activity Diagram of the EncodeAcTokenList module

the UV plane and finally encode the token list.

PackAndWriteDFArray

This module is a sub-module of the PackCodedVideo module and packs and writes the list of coded/uncoded blocks.Refer to Figure 4.67 for the class diagram and Figure 4.68 for the activity diagram.

This is a leaf node and has no children.

The workspaces and variables are initialised as usual. We then traverse each block of each macroblock in each superblock. We check and assign the status of whether it is fully/partially/not coded. We then first code the list of partially coded Super-Block using an RLE scheme. Next, the partially coded blocks are skipped and the fully coded blocks are encoded. Finally the block flags are coded.

PackModes

This is another sub module of the PackCodedVideo module which encodes and packs the mode list.Refer to Figure 4.69 for the class diagram and Figure 4.70 for the activity diagram.

This is a leaf node and has no children.

After initialisations are performed, a frequency map for the modes in this frame are built. The modes are then arranged from the most to the least frequent. All the schemes are then traversed and a bit score is found for each scheme which basically means the total bits to code using each available scheme. The best scheme is then outputted into the buffer. Finally the Mode list is packed and encoded.

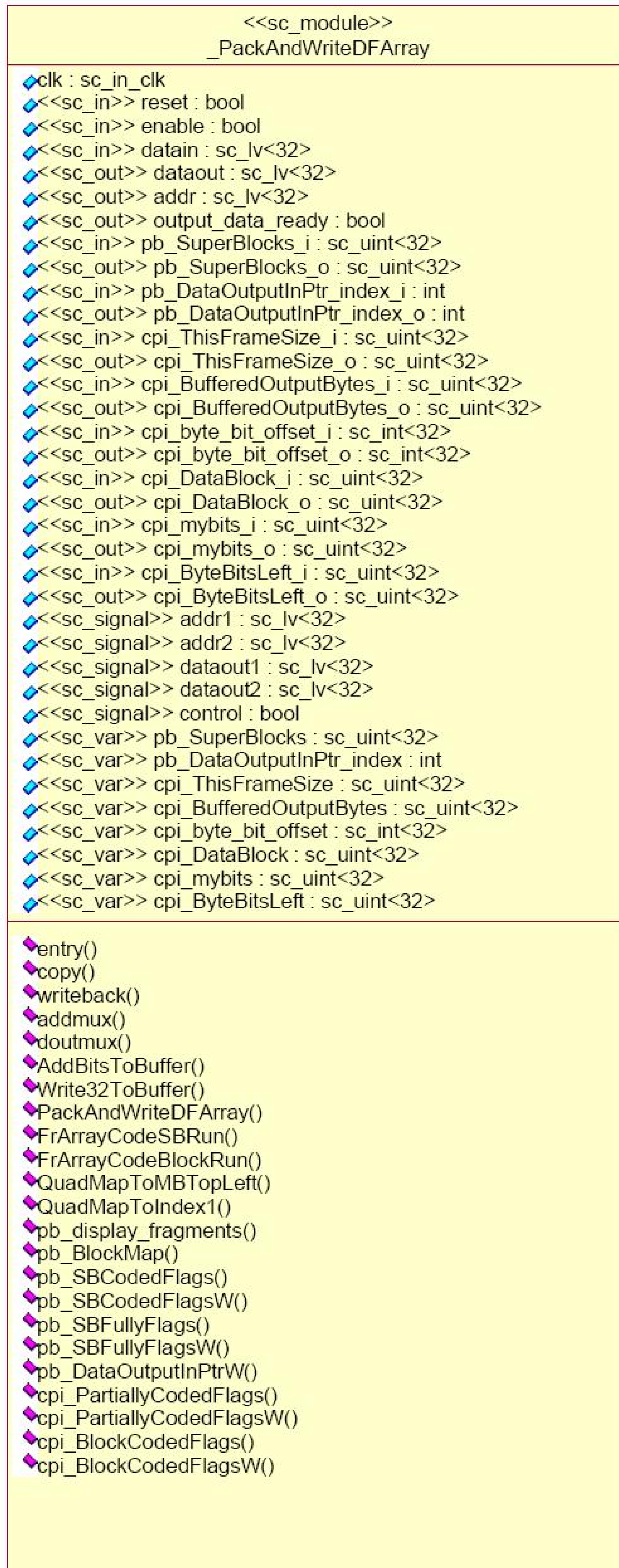


Figure 4.67: Class Diagram of the PackAndWriteDFArray module

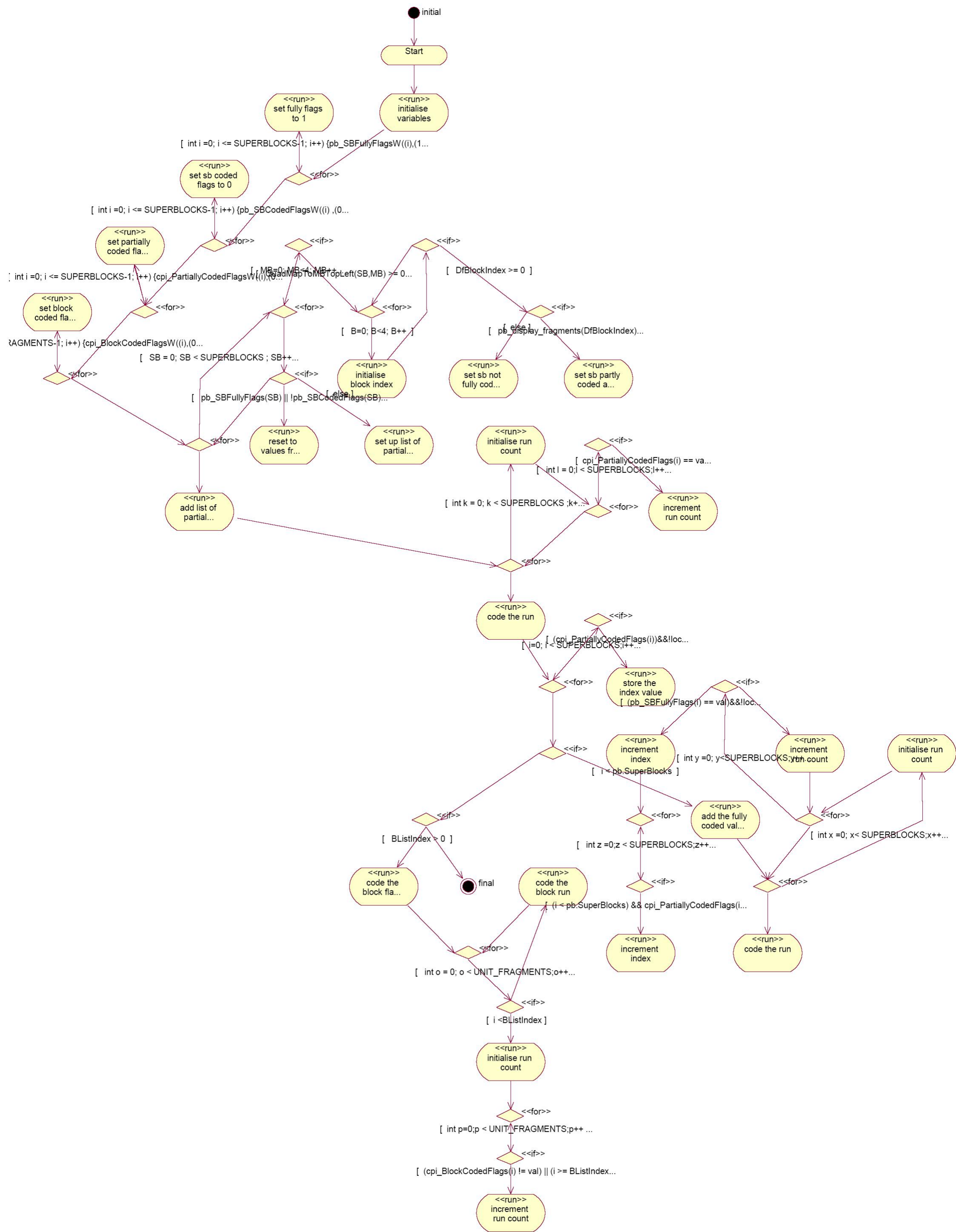


Figure 4.68: Activity Diagram of the PackAndWriteDFArray module

Figure 4.69: Class Diagram of the PackModes module

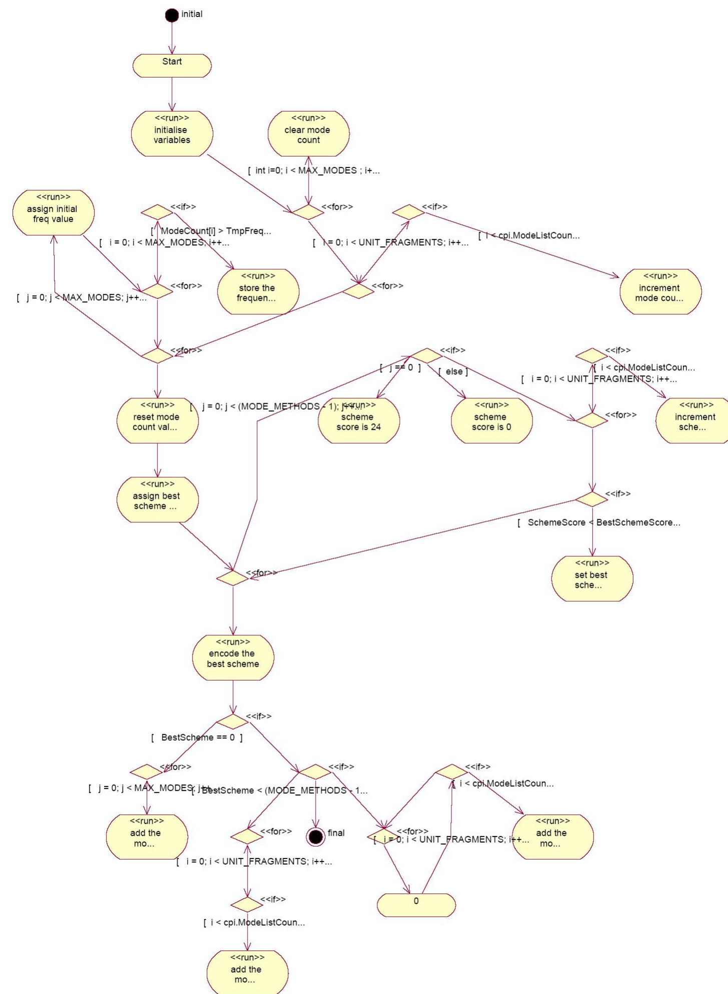


Figure 4.70: Activity Diagram of the PackModes module

PackMotionVectors

This is another sub module of the PackCodedVideo module which encodes and packs the motion vector list.Refer to Figure 4.71 for the class diagram and Figure 4.72 for the activity diagram.

This is a leaf node and has no children.

After initialisations are performed,a coding method is first chosen. We then select an entropy table and then pack and encode the motion vectors.

PackToken

This is another sub module of the PackCodedVideo module which packs a token for the given fragment.Refer to Figure 4.73 for the class diagram and Figure 4.74 for the activity diagram.

This is a leaf node and has no children.

After initialisations are performed, the record of what coefficient we have got up to for this block is updated and the encoded token is unpacked back into the quantised data array. The record of tokens coded and where we are in this fragment is updated followed by the counts of tokens coded. If the token is an EOB token, the run length is incremented followed by checks on the run length size. If the maximum size is exceeded the EOB run is packed and encoded. If the token is not an EOB token, we mark out which plane the block belonged to, then note the token, extra bits and huffman table in the optimised token list.

SUB8

This is a very simple module used to subtract 2 8x8 blocks. It is a sub module of the transformquantise block described earlier. Shown in Figure 4.75 and Figure 4.76 are its static and dynamic UML descriptions respectively.

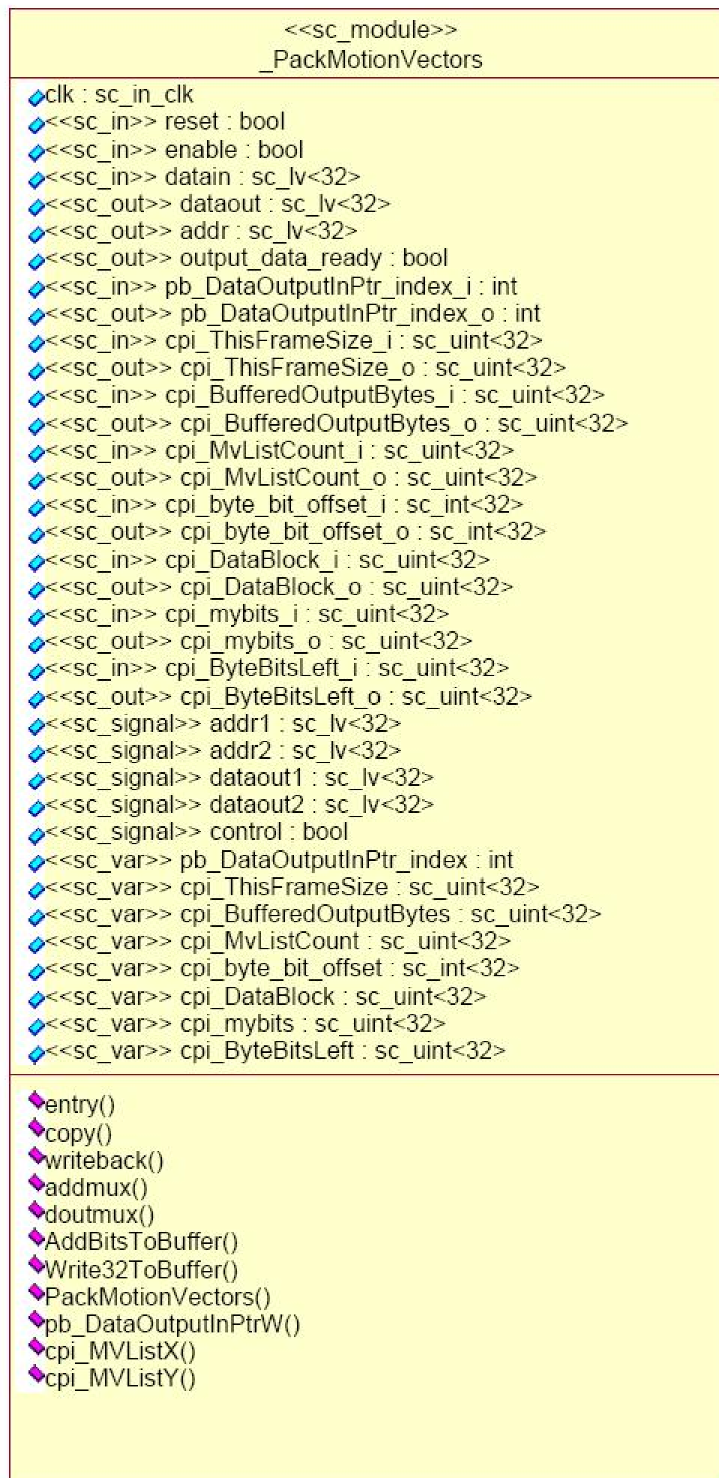


Figure 4.71: Class Diagram of the PackMotionVectors module

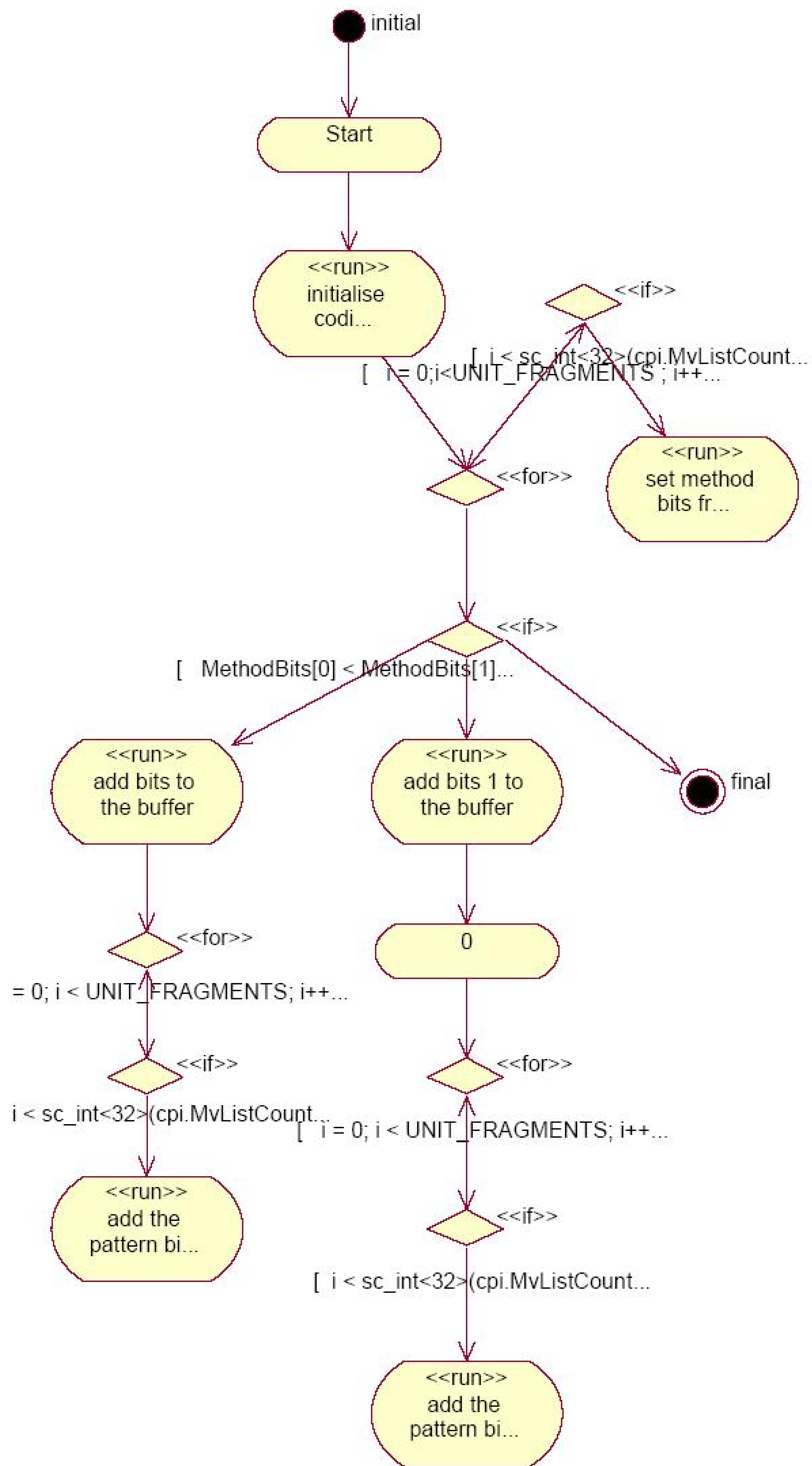


Figure 4.72: Activity Diagram of the PackMotionVectors module

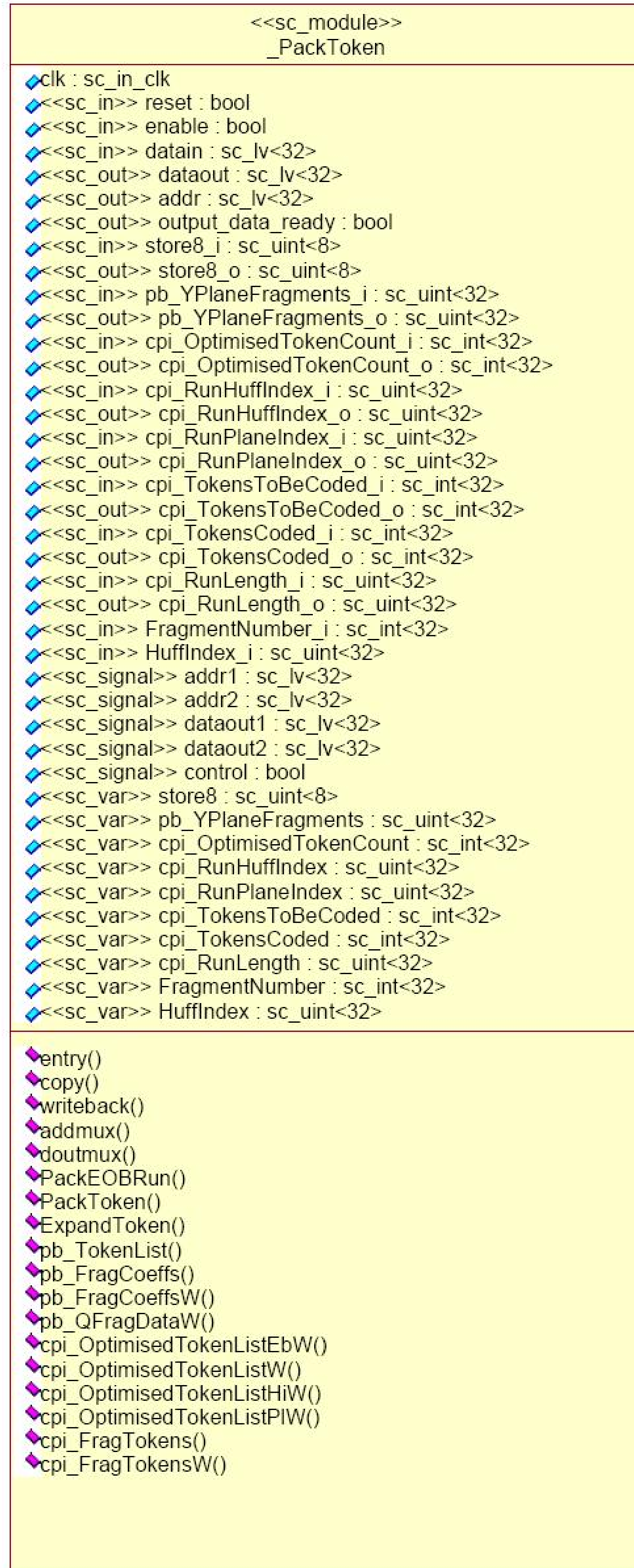


Figure 4.73: Class Diagram of the PackToken module

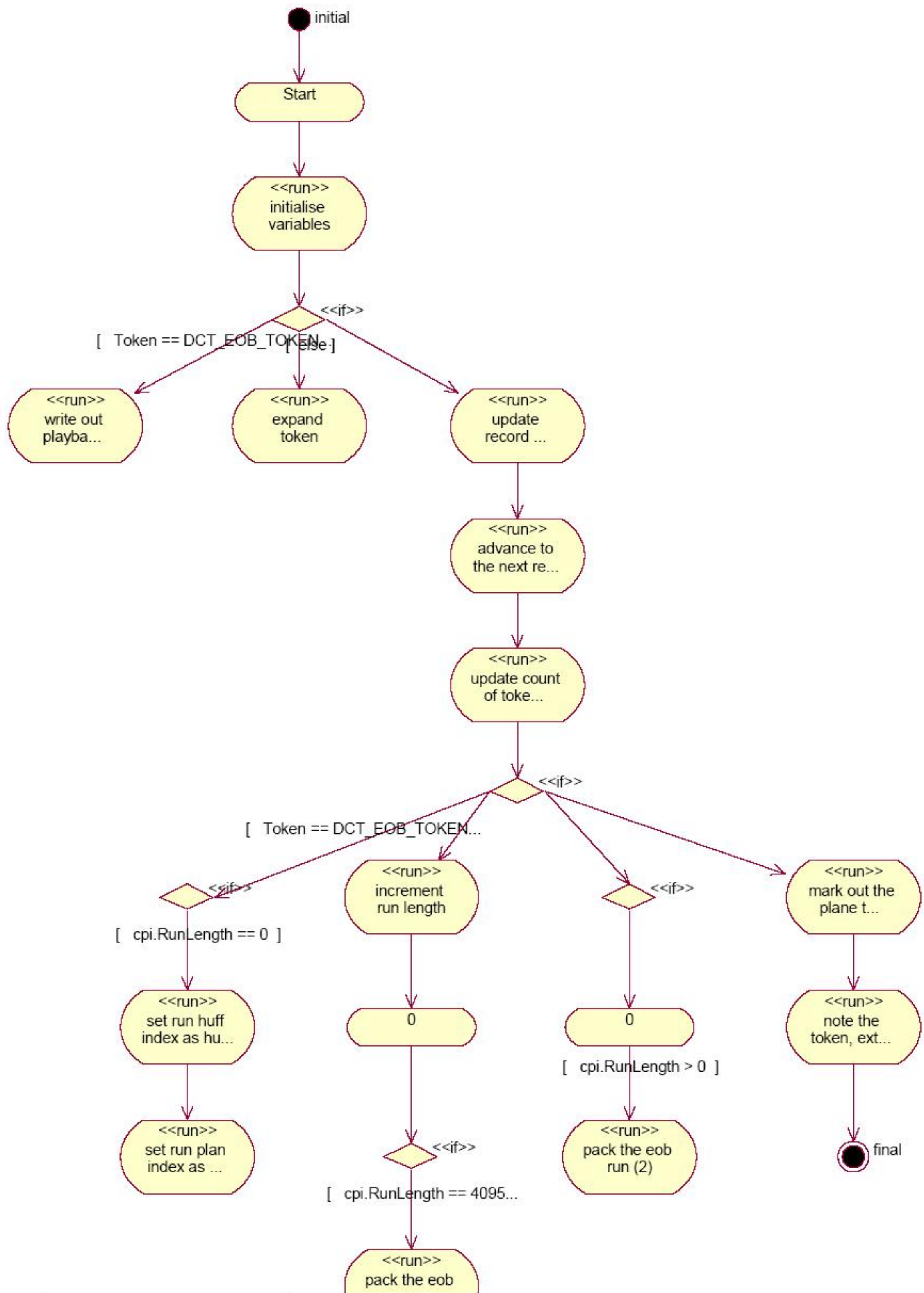


Figure 4.74: Activity Diagram of the PackToken module

Figure 4.75: Class Diagram of the SUB8 module

This is a leaf node and has no children.

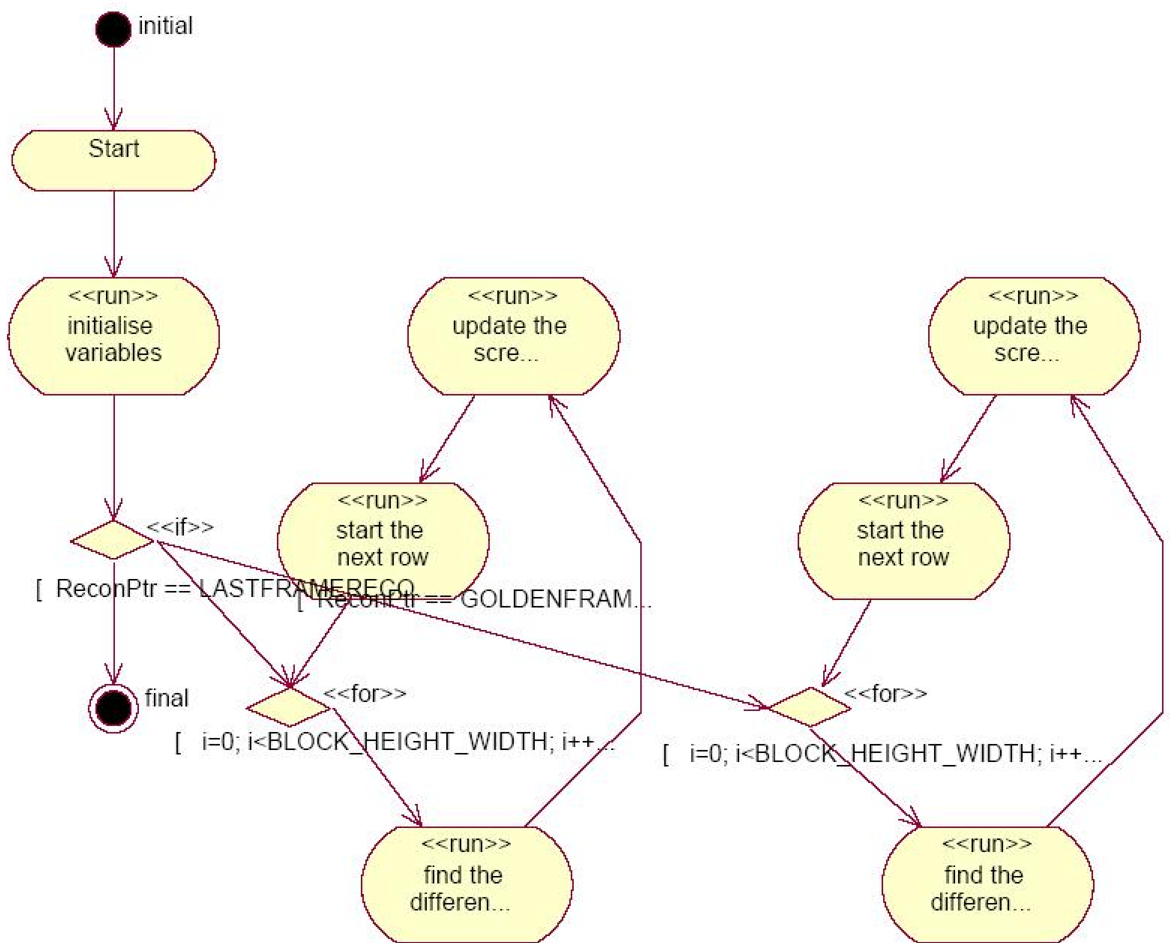


Figure 4.76: Activity Diagram of the SUB8 module

The function runs through each pixel of the block and finds the difference between the pixels of the two blocks and stores the result in a resulting "Difference block" which is then used as an input to the DCT process.

SUB8_128

This is a very simple module used to subtract 128 from each pixel of a 8x8 block. It is a sub module of the transformquantiseblock module described earlier. Shown in Figure 4.77 and Figure 4.78 are its static and dynamic UML descriptions.

This is a leaf node and has no children.

The function runs through each pixel of the block and finds the difference between the

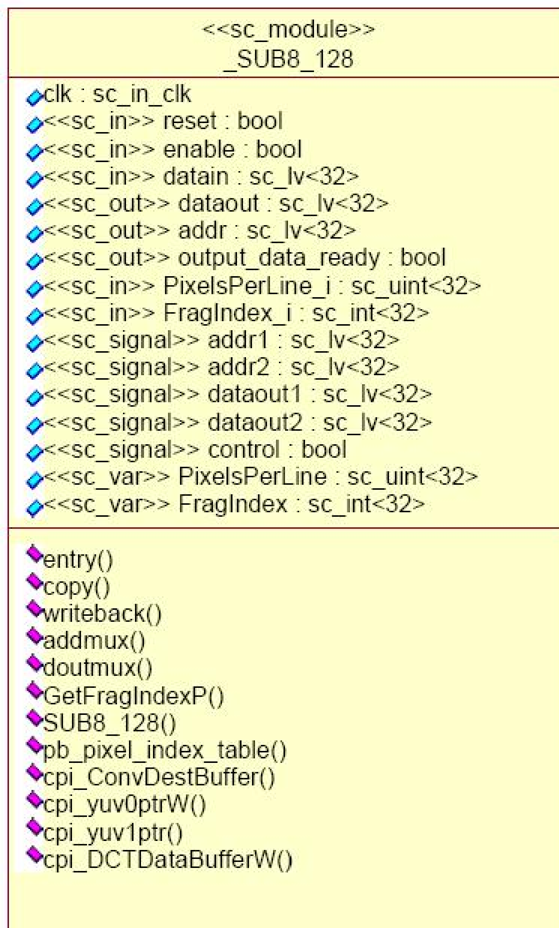


Figure 4.77: Class Diagram of the SUB8_128 module

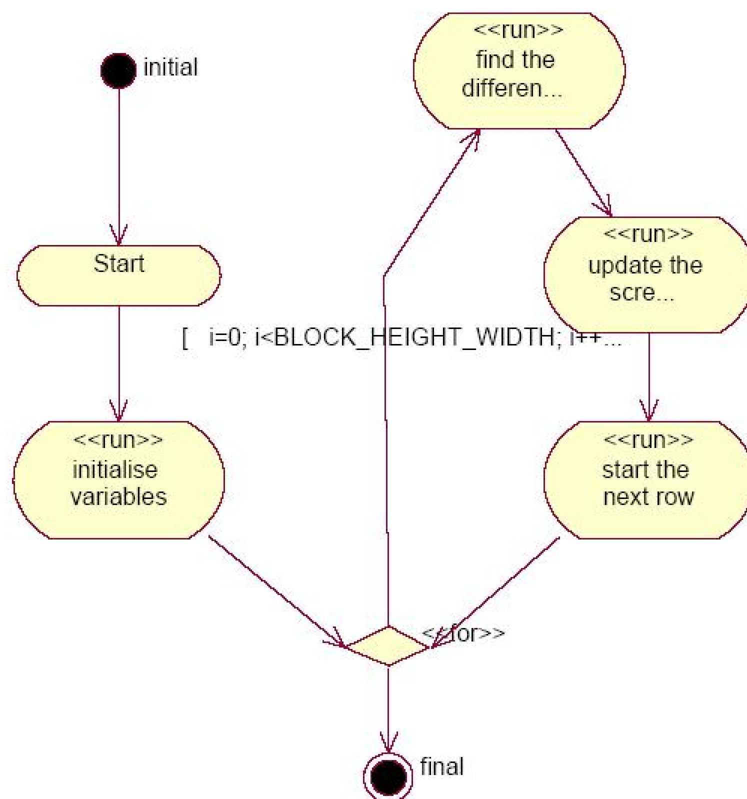


Figure 4.78: Activity Diagram of the SUB8_128 module

pixel and 128 and stores the result in a resulting “Difference block” which is then used as an input to the DCT process. This converts the data to 8 bit signed (by subtracting 128)and reduces the internal precision requirements in the DCT transform.

fdct_short

The fdct_short module performs a DCT transform on the data. It is a submodule of the transformquantiseblock module. The UML diagrams are shown in Figure 4.79 and Figure 4.80.

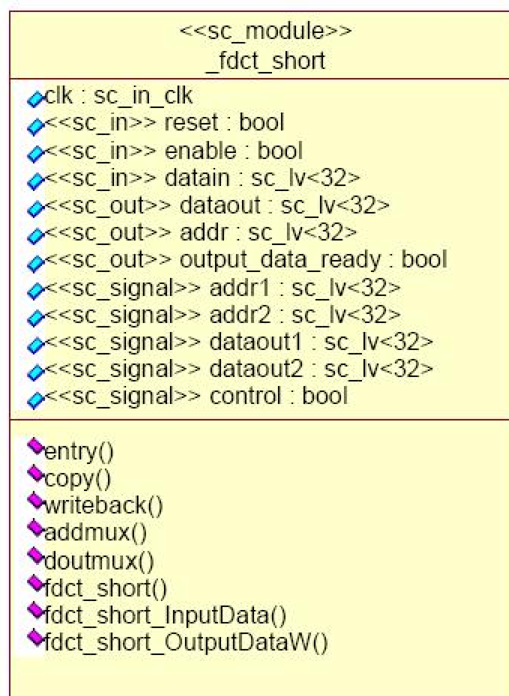


Figure 4.79: Class Diagram of the fdct_short module

This is a leaf node and has no children.

The activity diagram describes the transform. We first start with the rows. Firstly some common sums and differences used in the calculation are calculated. Following this some commonly used product terms are calculated. We then define inputs to rotation for outputs 2 and 6 and apply rotation for outputs 2 and 6. The same is repeated for 1,7 and 3,5. The exact same procedure is now repeated with the columns.

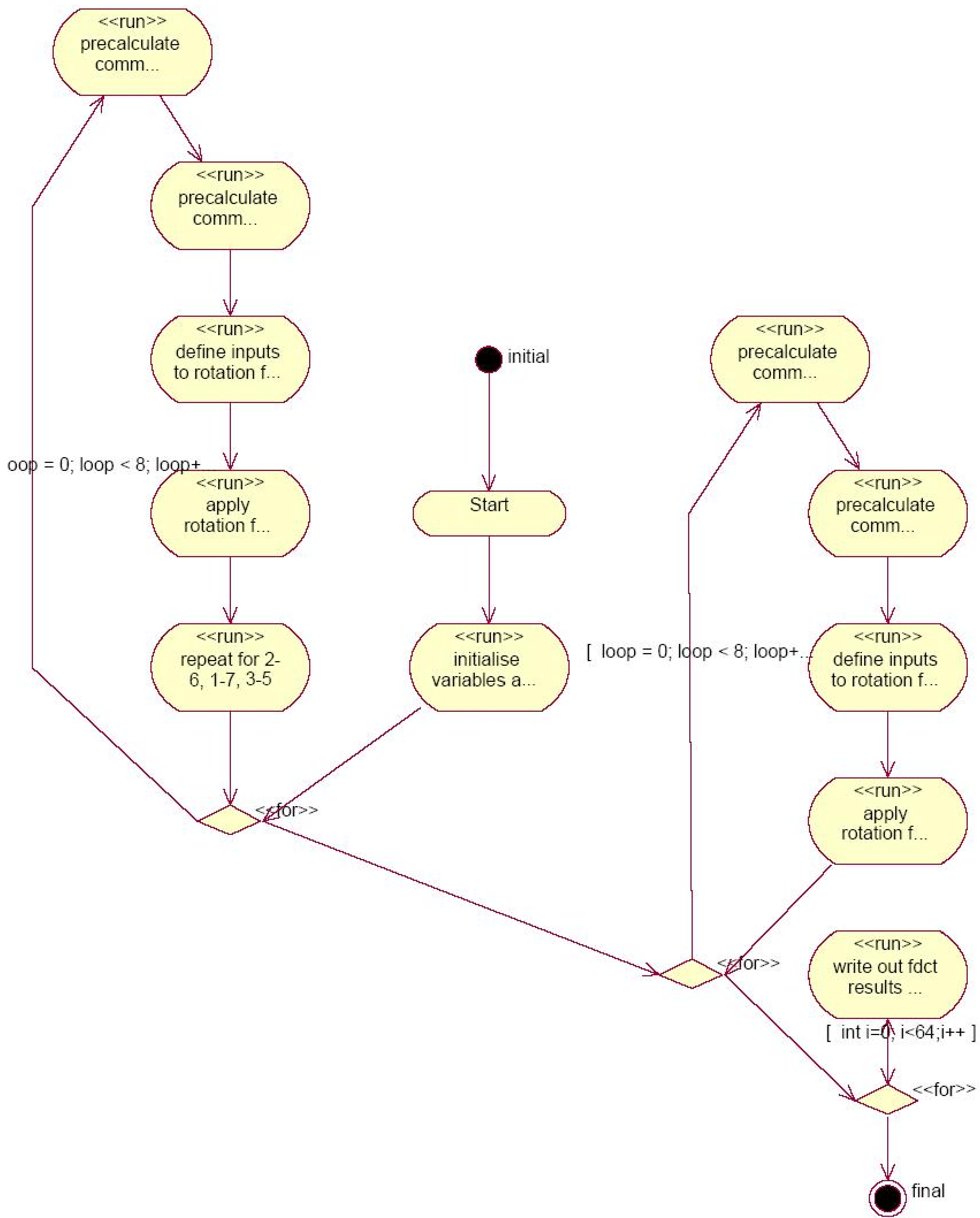


Figure 4.80: Activity Diagram of the fdct_short module

quantize

The quantize module quantizes a block of pixels by dividing each element by the corresponding entry in the quantization array. Output is in a list of values in the zig-zag order. The module acts on the block to by quantized and outputs the quantized values in zig-zag order. It is a submodule of the transformquantiseblock module. The UML diagrams are shown in Figure 4.81 and Figure 4.82.

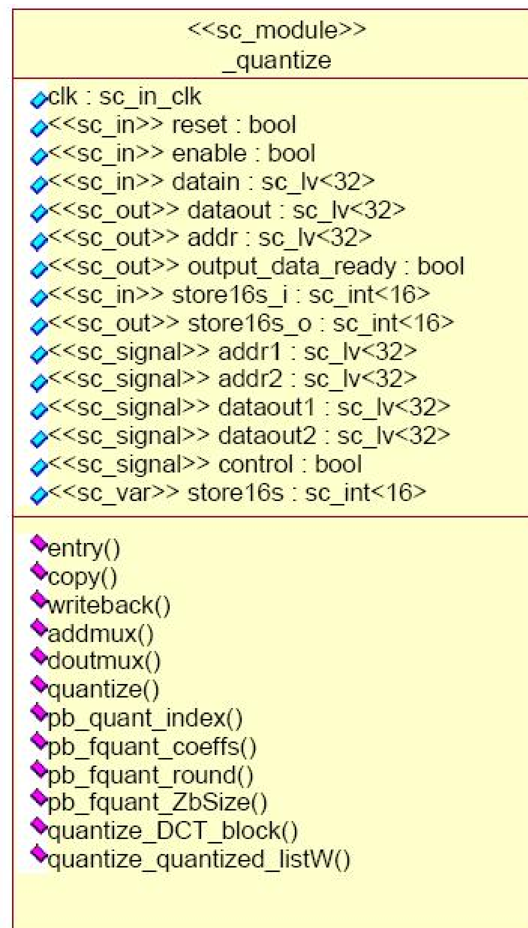


Figure 4.81: Class Diagram of the quantize module

This is a leaf node and has no children.

The activity diagram describes the module. We simple go through each pixel in the block and multiply it with the quantising coefficient with the quantised values being capped at +/-511.

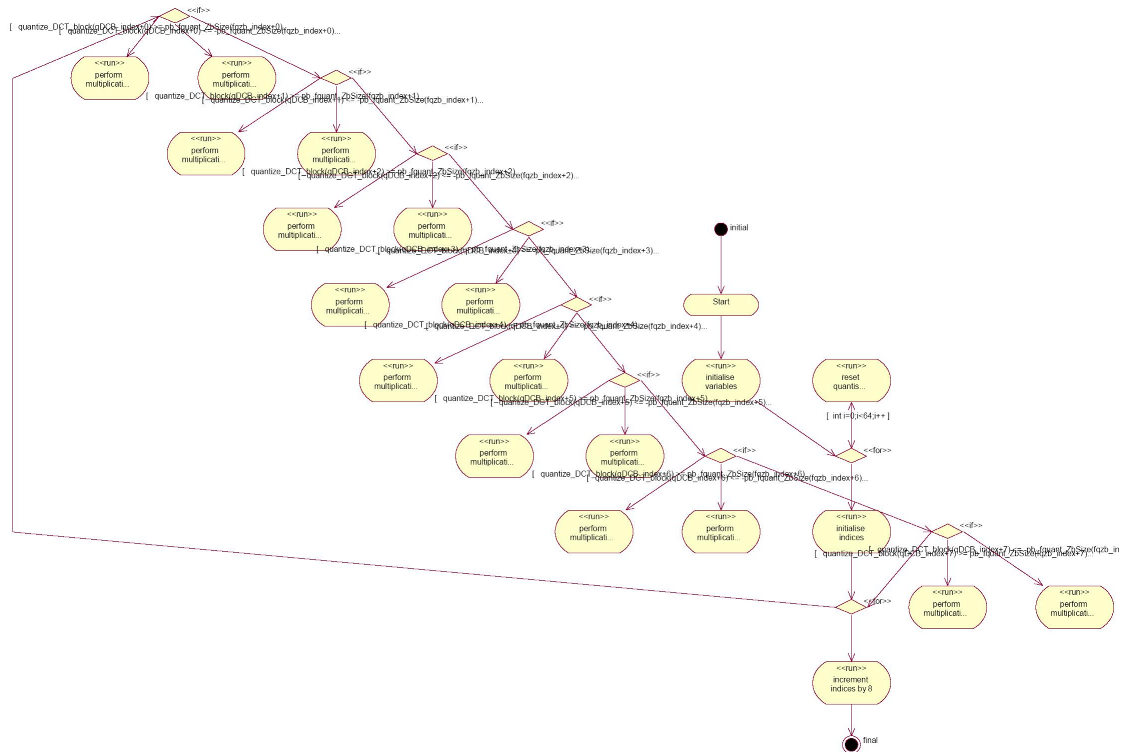


Figure 4.82: Activity Diagram of the quantize module

MotionBlockDifference

This is a very simple module used to compute the motion block residues, after filtering internal edges in the reference block. This is used when the coding mode involves a motion vector. It is a sub module of the transformquantiseblock module described earlier. Shown in Figure 4.83 and Figure 4.84 are its static and dynamic UML descriptions.

This module has two children the SUB8 module described earlier and the SUB8AV2 module for calculating block differences.

After initilisations the function calculates a baseline offset for the motion vector, works out the offset of the second reference position for 1/2 pixel interpolation and finally decides upon the reference pointers. If the motion vector offset is exactly pixel aligned we call the SUB8 module to find a simple difference. For fractional pixel Motion Vectors we call the SUB8AV2 module which uses two pixel values.

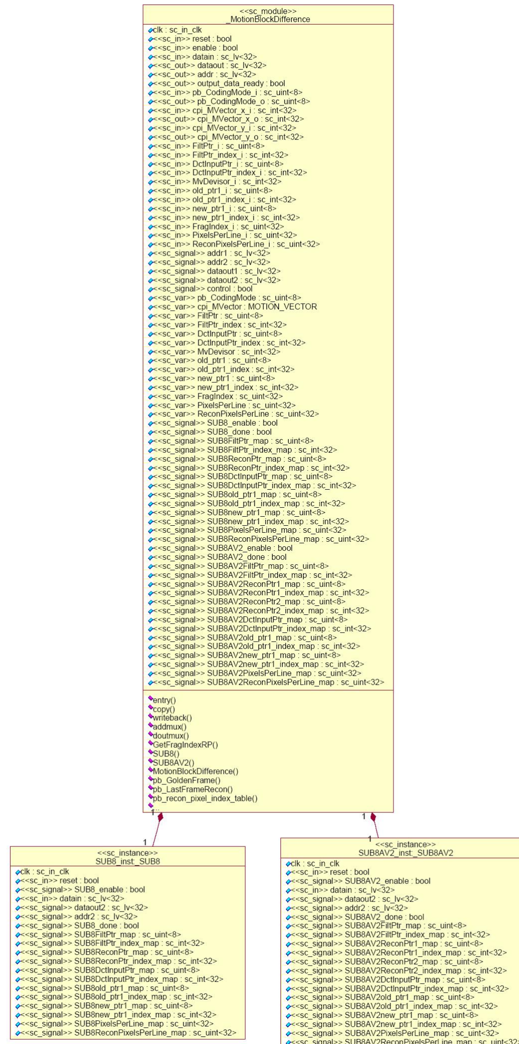


Figure 4.83: Class Diagram of the MotionBlockDifference module

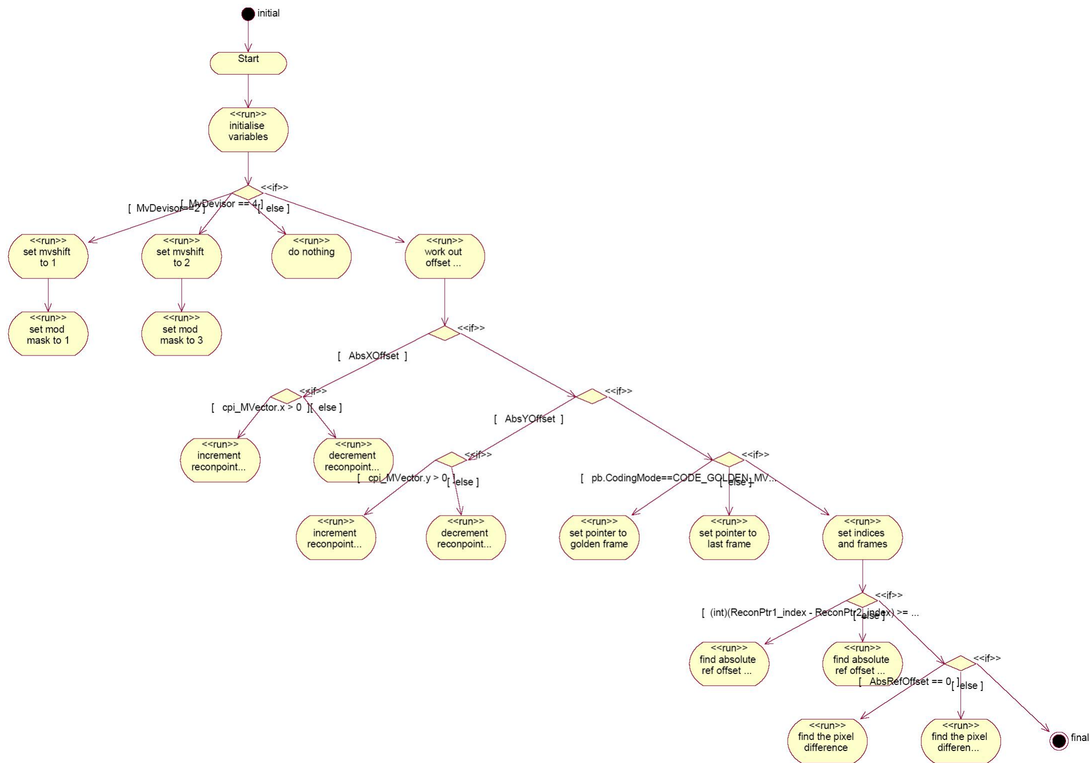


Figure 4.84: Activity Diagram of the MotionBlockDifference module

SUB8AV2

This is a very simple module used to subtract 2 8x8 blocks. It is a sub module of the MotionBlockDifference block described earlier. Shown in Figure 4.85 and Figure 4.86 are its static and dynamic UML descriptions.

This is a leaf node and has no children.

The function runs through each pixel of the block and finds the difference between the pixels of the source block and an average of two pixels from the reference block and stores the result in a resulting “Difference block” which is then used as an input to the DCT process.

RegulateQ

This is a submodule of the top most module VP3Encoder. If appropriate this function regulates the DCT coefficients to match the stream size to the available bandwidth

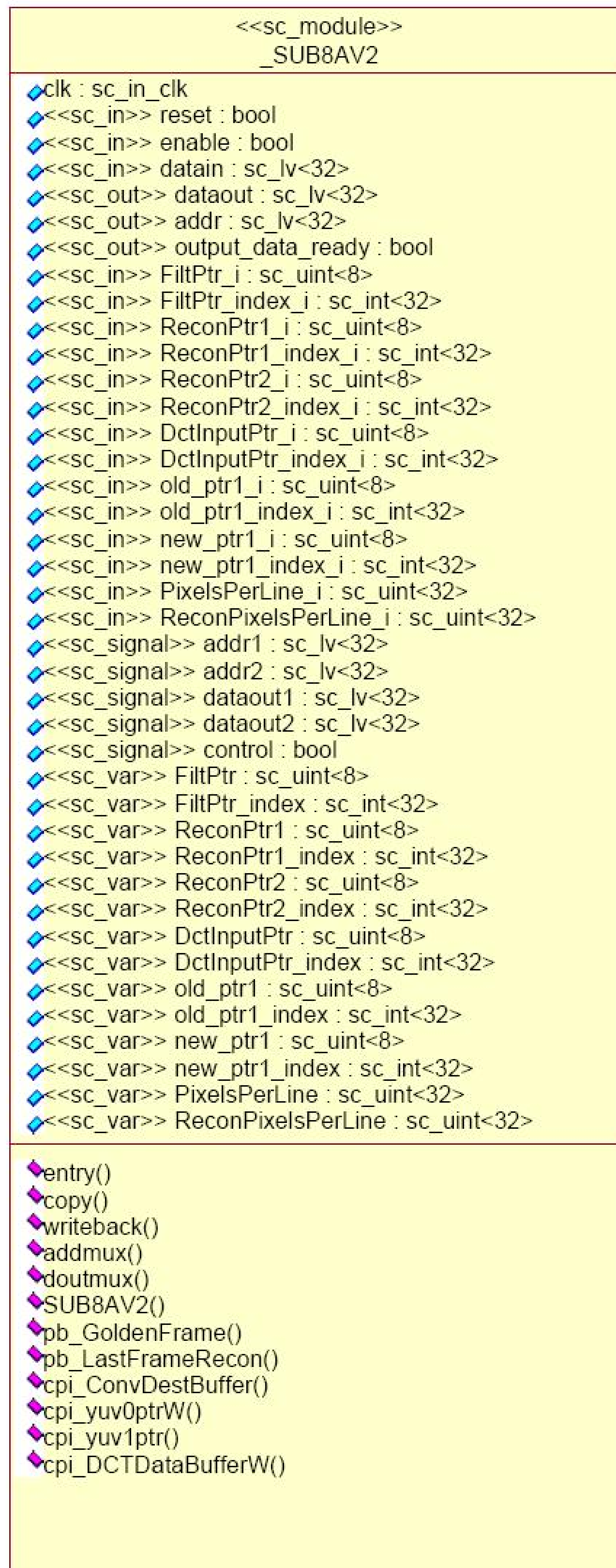


Figure 4.85: Class Diagram of the SUB8AV2 module

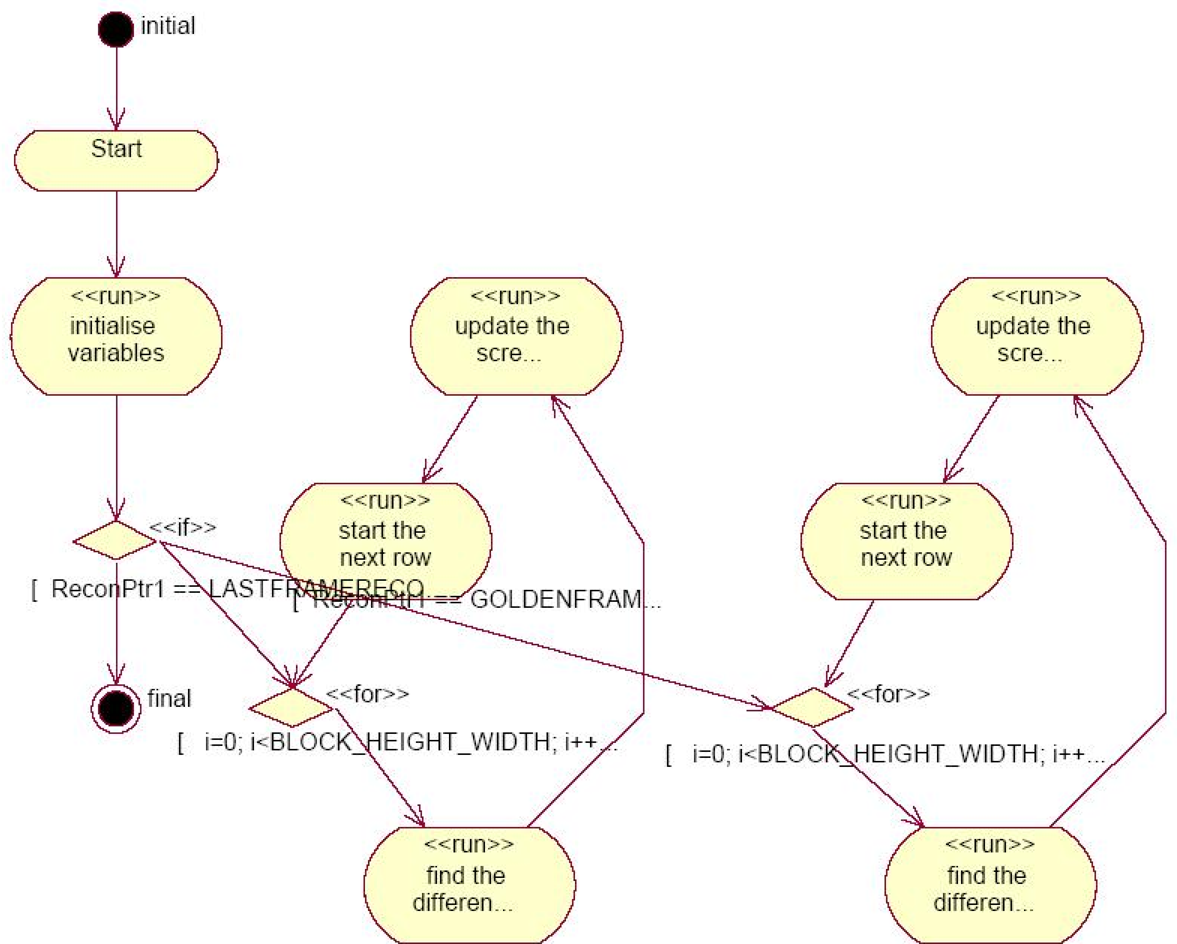


Figure 4.86: Activity Diagram of the SUB8AV2 module

(within defined limits). Shown in Figure 4.87 and Figure 4.88 are its static and dynamic descriptions.

This is a leaf node and has no children.

The function after performing initilisations, traverses the whole Q table and searches for the best Q for the target bitrate, by getting an estimate of the bytes per block that will be achieved at the given Q. After finding the optimal Q value, it applies range restrictions for key frames. Finally, if the quantiser value has changed it is re-initialised.

UpRegulateDataStream

This is a submodule of the top most module VP3Encoder. This function uses up spare bandwidth when not much is going on to refresh quality. Shown in Figure 4.89 and Figure 4.90 are its static and dynamic descriptions.

This is a leaf node and has no children.

As is seen above, after initilisations, the number of blocks in an MB is deducted from the recover block count. This compensates for the fact that once we start checking an MB we test every block in that macro block. Then Up regulate blocks last coded at higher Q. Finally, If we have still not used up the minimum number of blocks and are at the minimum Q then run through a final pass of the data to insure that each block gets a final refresh.

Verification and Synthesis

The modules were translated to SystemC, the lower level details filled in and the design was simulated. Pure 640X480 frames captured using a Creative NX Pro webcam were written to a file. These were then encoded using the encoder and later decoded by the original VP3 decoder. The images were successfully reproduced and the encoder was found to be functioning as per expectations.

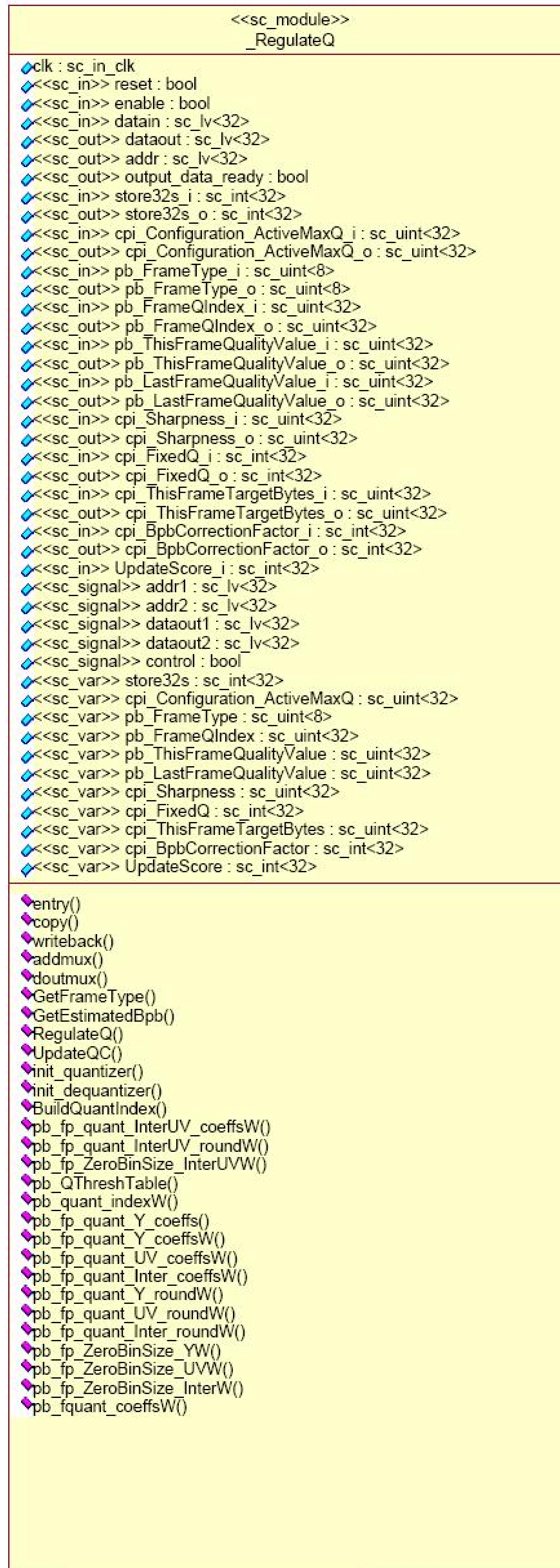


Figure 4.87: Class Diagram of the RegulateQ module

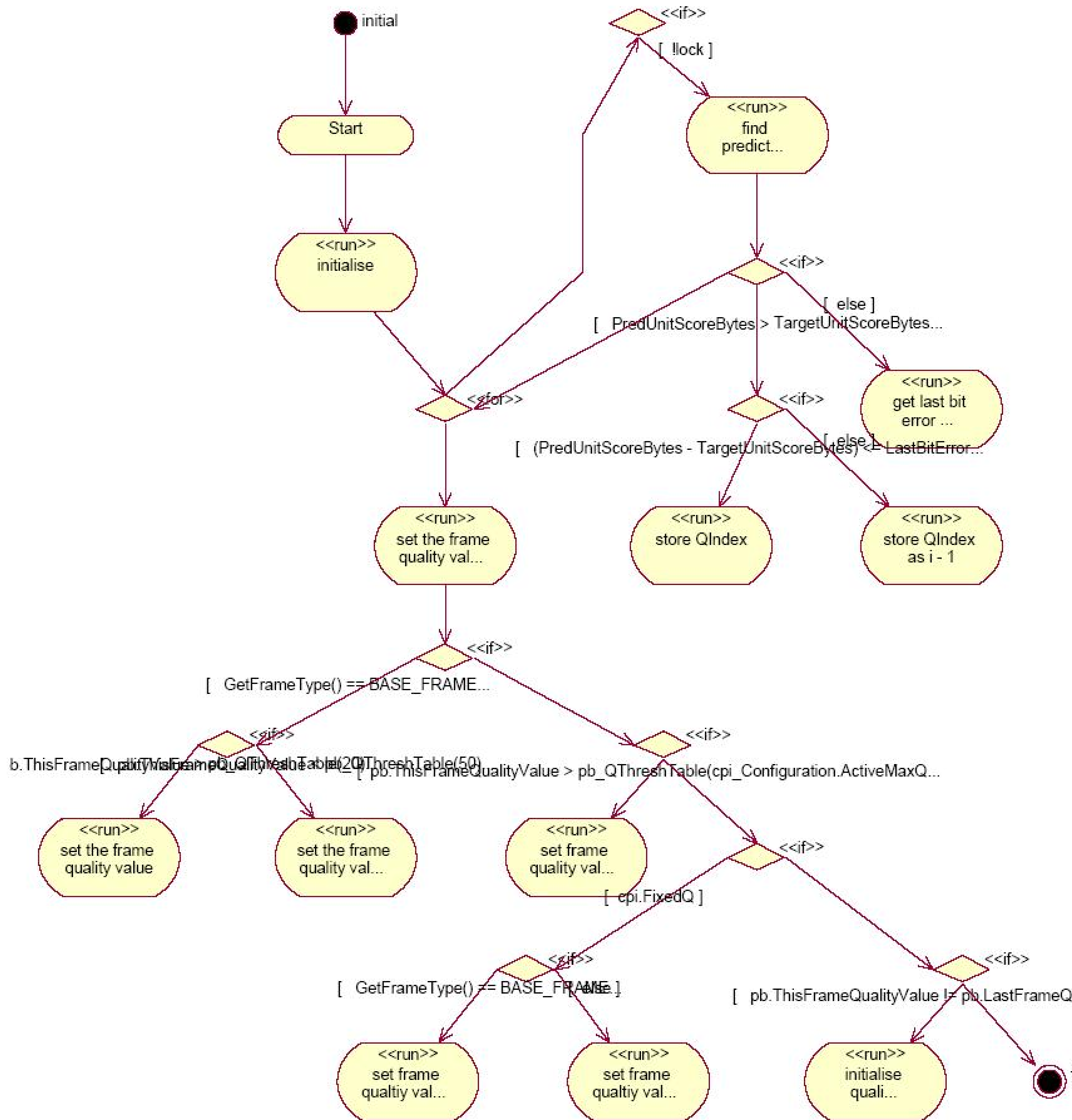


Figure 4.88: Activity Diagram of the RegulateQ module

Figure 4.89: Class Diagram of the UpRegulateDataStream module

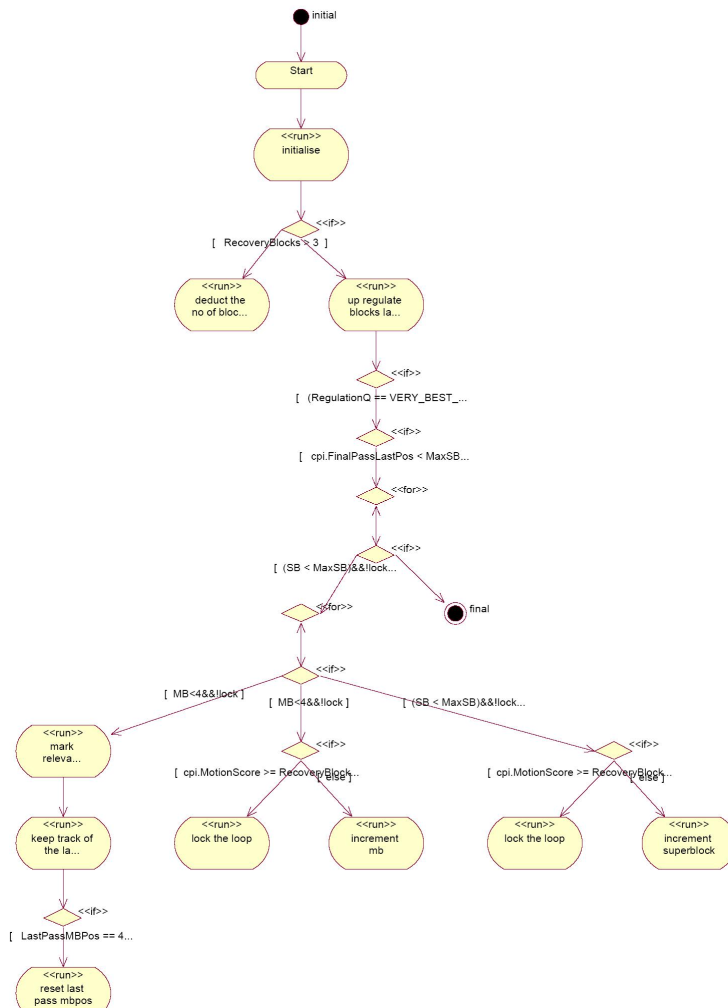


Figure 4.90: Activity Diagram of the UpRegulateDataStream module

Setting	Value
Technology library	tc6a_cbcore.db
Synthetic library	dw01.sldb, dw02.sldb
Map Effort	low
Clock Period	10 ns
I/O Mode	superstate
hlo_resource_allocation	area_only

Table 4.12: VP3 Encoder synthesis settings

The individual modules were all synthesised using Synosys Cocentric compiler with the settings shown in Table 4.12. A summary of the synthesis results obtained are given in Table 4.13 along with area of a Theora video encoder being developed for an FPGA device.

The total gates for all modules is close to 700K comparable to the recently developed theora video encoder (based on vp3 also) that sits in a 1000k FPGA device used in the Elphel reconfigurable camera project. The exact gate count for that design is not available.

4.3 A Comparison with HDL Design Flow

The typical HDL design flow is shown in Figure 1.11. As can be seen, the flow starts with RTL/Behavioural description. It then proceeds to the Functional simulation, followed by translation and optimisation and finally the netlist.

In Figure 4.91, an example design flow using VHDL is shown(**?**). It can be described as follows.

1. Write a design description in the VHDL language. This description can be a combination of structural and functional elements
2. Provide VHDL-language test drivers for your VHDL simulator. These drivers supply test vectors for the simulation and gather output data.

Module	ports	nets	cells	refs	comb. area	non-comb. area	net inter-con area	total cell area	total area	gate count	slack
ClearDownQFragData	101	2765	2298	76	4180	4677	30198	8858	39056	3035	0.07
DPCMTokenizeBlock	253	5460	5080	159	7460	6254	53859	13714	67573	5822	0.00
EncodeAcTokenList	613	25293	24765	177	30162	28564	221993	58726	280720	25881	0.00
EncodeDcTokenList	613	23398	22870	182	28223	25864	209903	54087	263991	25953	0.00
fdct_short	101	52691	51744	193	91424	32375	602122	123814	725922	57844	0.01
GetFOURMVExhaustiveSearch	1011	8811	8050	167	13593	13413	104315	27006	131322	9924	0.01
GetHalfPixelSumAbsDiffs	487	8191	7737	121	10366	14246	93617	24612	118229	8469	0.00
GetInterErr	237	10414	10017	147	17762	12487	132612	30250	162863	11923	0.00
GetIntraError	165	2700	2384	81	6818	3244	40045	10062	50107	4138	0.45
GetMBInterError	527	2736	2368	69	6737	4320	43678	11057	54736	4326	0.07
GetMBIntraError	295	1853	1597	62	2090	3367	19492	5458	24950	1763	0.61
GetMBMVExhaustiveSearch	1035	6654	6173	112	8365	11174	74911	19540	94451	6939	0.00
GetMBMVInterError	1277	7198	6772	115	8156	12107	79278	20264	99542	7089	0.00
GetNextSumAbsDiffs	269	12089	8428	230	24433	5895	122381	30329	152710	5288	0.03
GetSumAbsDiffs	237	5189	4777	101	7212	7063	55897	14276	70173	5202	0.01
MotionBlockDifference	1009	3280	2668	75	26889	7653	102342	34542	136885	10683	0.01
PackAndWriteDFArray	549	29195	27041	268	40279	29387	271877	69667	341545	30735	0.00
PackCodedVideo	3011	9537	7932	138	12209	11593	90648	23802	114451	9718	0.00
PackModes	549	20005	19531	173	23517	24541	182857	48058	230915	20942	-0.73
PackMotionVectors	549	40235	39771	207	40775	71777	384351	112549	496904	40902	0.00
PackToken	621	9406	8717	149	11726	17084	106497	28811	135308	9643	0.01
PickIntra	341	4944	4620	104	33396	10213	136073	43611	179684	13803	0.00
QuadCodeComponent	1303	5727	5014	97	5604	10156	55882	15761	71643	5112	0.06
QuadCodeDisplayFragments	4219	54796	52566	231	74776	58164	535152	132962	668093	56928	0.00
quantize	133	16692	15827	163	24898	24331	189480	49230	238710	18981	0.00
RegulateQ	693	36343	34847	197	130463	58027	686330	188528	874821	77595	0.00
SUB8	333	5224	4732	89	6723	8999	57683	15723	73406	5049	0.07
SUB8_128	165	9644	8398	123	13725	14972	104258	28698	132956	9990	1.69
SUB8AV2	365	8408	7906	117	10795	13624	94623	24419	119043	8425	0.00
TransformQuantizeBlock	1514	3537	2934	93	3131	8771	39459	11903	51362	3477	0.00
UpdateFrame	6697	35367	31454	208	123140	45373	575179	168538	743693	61991	-0.37
UpRegulateDataStream	741	17416	16839	167	47476	27995	252960	75472	328432	26627	0.00
VP3Encoder	6644	62217	58631	234	93864	110668	722422	204548	926955	74834	-0.01
Theora FPGA Video Encoder (4i2i)	-	-	-	-	-	-	-	-	-	sits in 1000K device	

Table 4.13: Results obtained from synthesis of VP3Encoder modules

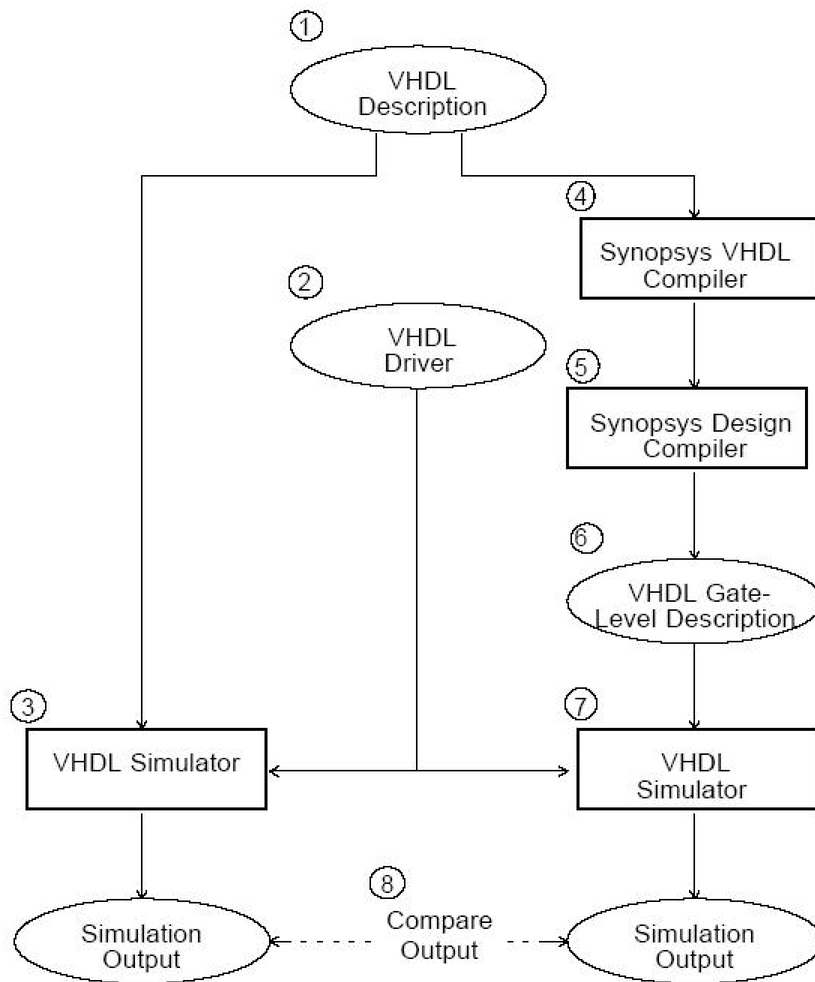


Figure 4.91: VHDL Design Flow Example using the Synopsys VHDL compiler

3. Simulate the design by using your VHDL simulator to verify the accuracy of the description.

4. Synthesize the VHDL description with VHDL Compiler. VHDL Compiler performs architectural optimizations, then creates an internal representation of the design.

5. Use the Synopsys Design Compiler to produce an optimized gate-level description in the target ASIC library. You can optimize the generated circuits to meet the timing and area constraints you want. This optimization step must follow the translation step (step 4) to produce an efficient design.

6. Use the Synopsys Design Compiler to output a gate-level VHDL description. This netlist-style description uses ASIC components as the leaf-level cells of the design. The gate-level description has the same port and module definitions as the original high-level VHDL

HDLs provide higher level constructs than gate-level design. The enable easier verification. They also enable automatic conversion of these technology independent high-level contructs like entities, if-else and other C-like control structures, processes/'always' blocks, variables/signals/wires/registers, into gates with the help of compilers.

In case of complex designs with a large number of modules design is gaining an ever-increasing importance. This is where design constructs such as those specified by UML (class diagrams, activity diagrams/statecharts) help. The design flow proposed in this thesis is shown in Figure 2.1.

As can be seen the flow begins with the requirement specification, functional specification, and scenarios from which the UML model contructs are generated. These UML models are then translated into synthesisable SystemC contructs (which are at the same level as traditional HDLs) thus reducing the time taken from design to HDL level.

While HDLs are still important at the low level, this method helps to define the control structure and static structure of the design which is increasingly becoming more complex to capture. The method is therefore a means to improve productivity and accuracy at high level. Low level optimisation for area and speed does not fall within the realm of the methodology.

In terms of coding time, this methodology reduces the design, and translation to HDL time. It also reduces the number of design iterations through greater focus on high level specification.

Chapter 5

Conclusion

The work of this thesis covered three main areas namely, the definition of UML syntax for describing hardware, building of a UML-SystemC translator to act as a UML model-compiler to produce synthesisable SystemC code, and creating designs based on the syntax , using the tool created. Three main ways of representing designs were defined, class diagrams for static structures, and statecharts and activity diagrams for dynamic structures. A mixture of these elements was employed depending on the nature of the design.

The UML based methodology was found to be an effective method in designing small as well as large digital systems where preliminary conceptualisation and analysis played a crucial role. The method was tested on a number of designs, the largest of which was the VP3 video encoder. The results obtained are comparable to that obtained through traditional methods. This method does not obviate the need for low level optimisation. It merely provides tools for a more streamlined approach for defining designs with an elaborate control structure.

While this effort is a crucial link in the design flow, the upstream requirement specification and analysis is equally important. In fact it is imperative to find ways to automate generation of UML diagrams from user requirements and scenarios (which is

another area of work in the DSA lab) as very large systems mean very many diagrams, most of which involve large amounts of repetitive work and which must be automated for greater productivity. Many UML tools provide scripting options for this purpose which must be exploited to achieve greater gains in time.

Bibliography

El Mustapha Aboulhamid, Mike Baird, and Bishnupriya Bhattacharya et al. *SystemC 2.0.1 Language Reference Manual, Revision 1.0.* Open SystemC Initiative, 2003.

F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, Jurecska A, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach.* Kluwer Academic Press, June 1997.

G de Jong. A uml-based design methodology for real-time and embedded systems. In *Proceedings of the Design Automation and Test in Europe (DATE)02*, pages 776–778, 2002.

C Drosos, D Metafas, and G.Papadopoulos. A uml-based methodology for the system design of a wireless lan prototype. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC04)*, pages 45–51, 2004.

D. Drusinsky and D. Harel. Using statecharts for hardware description and synthesis. In *IEEE Trans. on CAD*, pages 798–807, 1989.

Tom Fitzpatrick. Systemverilog for vhdl users. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE04)*, pages 1530–1591/04, 2004.

W. Fornaciari, P. Micheli, F. Salice, and L. Zampella. A first step towards hw/sw

partitioning of uml specifications. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, page 10668, 2003.

D. Harel. Statecharts: A visual formalism for complex systems. In *Science of Computer Programming*, pages 231–274, 1987.

Grant Martin. Uml for embedded systems specification and design: Motivation and overview. In *Proceedings of the Design Automation and Test in Europe (DATE)02*, pages 773–775, 2002.

OMG. *OMG Unified Modeling Language Specification.* OMG, 2003.

Zhu Qiang, Matsuda Akio, Kuwamura Shinya, Nakata Tsuneo, and Minoru Shoji. An object-oriented design process for system-on-chip using uml. In *Proceedings of the 15th international symposium on System Synthesis*, pages 249–254, 2002.

Terry Quatrani. *Visual Modeling with Rational Rose 2002 and UML.* Addison Wesley Professional., 2002.

Inc. Synopsys. *Synthesizable abstraction levels and Cocentric support.* Synopsys, Inc., Jan 2002a.

Inc. Synopsys. *Describing Synthesizable RTL in SystemC TM.* Synopsys, Inc., Nov 2002b.

Chen Xi, Xu Ningyi, and Zhou Zucheng. A methodology for systemc algorithmic model verification applying matlab. In *ASIC, 2003. Proceedings. 5th International Conference on*, pages 294 – 297, 2003.