Chapter 1

Introduction

Document use The document is meant for developers new to this project and for maintainers considering a change. It allows a high level view and drilling down some to isolate the partition that needs change. Finally, the document is for programmers looking for design guidelines.

Document structure The document consists of the high level class diagrams, some sequence diagrams, some design patterns and finally the design guidelines and specific platform dependencies.

Document maintenance Some of the diagrams – especially the introductory architectural drawings – were created with DIA[1].

Most of the UML diagrams we created with UMBRELLO. Be careful to use the most recent version as possible. From the the menu option EXPORT ALL DIAGRAMS AS PICTURE or the equivalent option for one diagram one can create an image for this document.

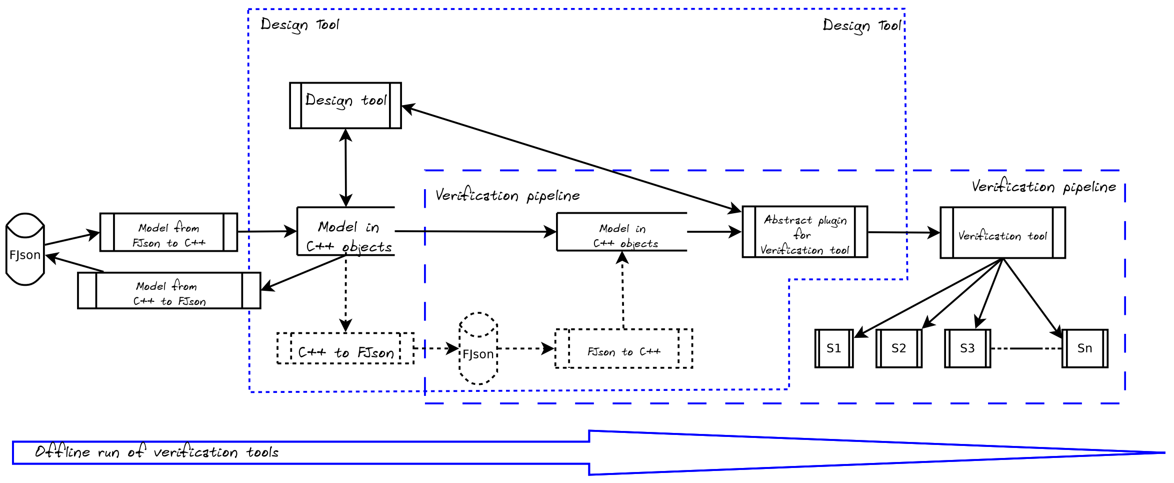
Naming Make sure to create names without spaces so the exported filenames will also be valid. Even though most OS's can work around spaces they still are a pain in the neck.

[1]you can find DIA at http://sourceforge.net/projects/dia-installer/

Chapter 2

Architecture Overview

2.1 Partitioning



The complete tool consists of a graphical design tool and a verification pipeline. The design tool runs online (in a graphical environment) and is meant to aid in designing a Network on Chip (NoC). The verification tools can run in the online (graphical) environment or in an offline (commandline) environment and are meant to do consistency and correctness checks on a model. The case of offline verification is useful for verifying a model that is too big to verify completely during graphical edit cycles.

Usage goal The goals for running the graphical editor or the verification pipeline may either be creating an NoC or testing a verification tool. In both cases the dynamics are the same.

Plugin architecture The verification tools and the graphical design toolkit communicatie through a plugin architecture. The runtimes for verification tools will vary – depending on the input – from several seconds to a few minutes or longer. For that reason the verification tools run relatively independently from the graphical design toolkit.

How will the plugin architecture overlap between the online and the offline versions of running the verification tools? I.e. how will this architecture reflect on the currently existing verification tools?

2.2 Graphical Design Tool Dynamics

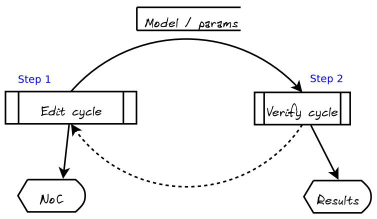


Figure 2.1: Dynamic process in overview

The graphical editor of NoC consists of a toolkit containing the XMAS primitives, the composites available in separate libraries, the graphical window, and the commands to prepare and execute the verification tools. The process of designing an NoC consists of alternating an edit cycle with a verify cycle (see figure 2.1).

Step 1 Edit cycle The editor adds components (primitives or composites) to the NoC diagram filling in their parameters and connecting components as necessary. Once satisfied the editor switches to prepare for the verfication cycle.

Step 2 Verification cycle Firstly, the editor chooses the verification tools to be run. Secondly, the editor fills in the relevant parameters for running the selected verification tools. Finally, the editor starts the verification processes causing the controller to copy the parameters and the current NoC model and start the verification tools selected. See figure 2.2 for a detailed illustration.

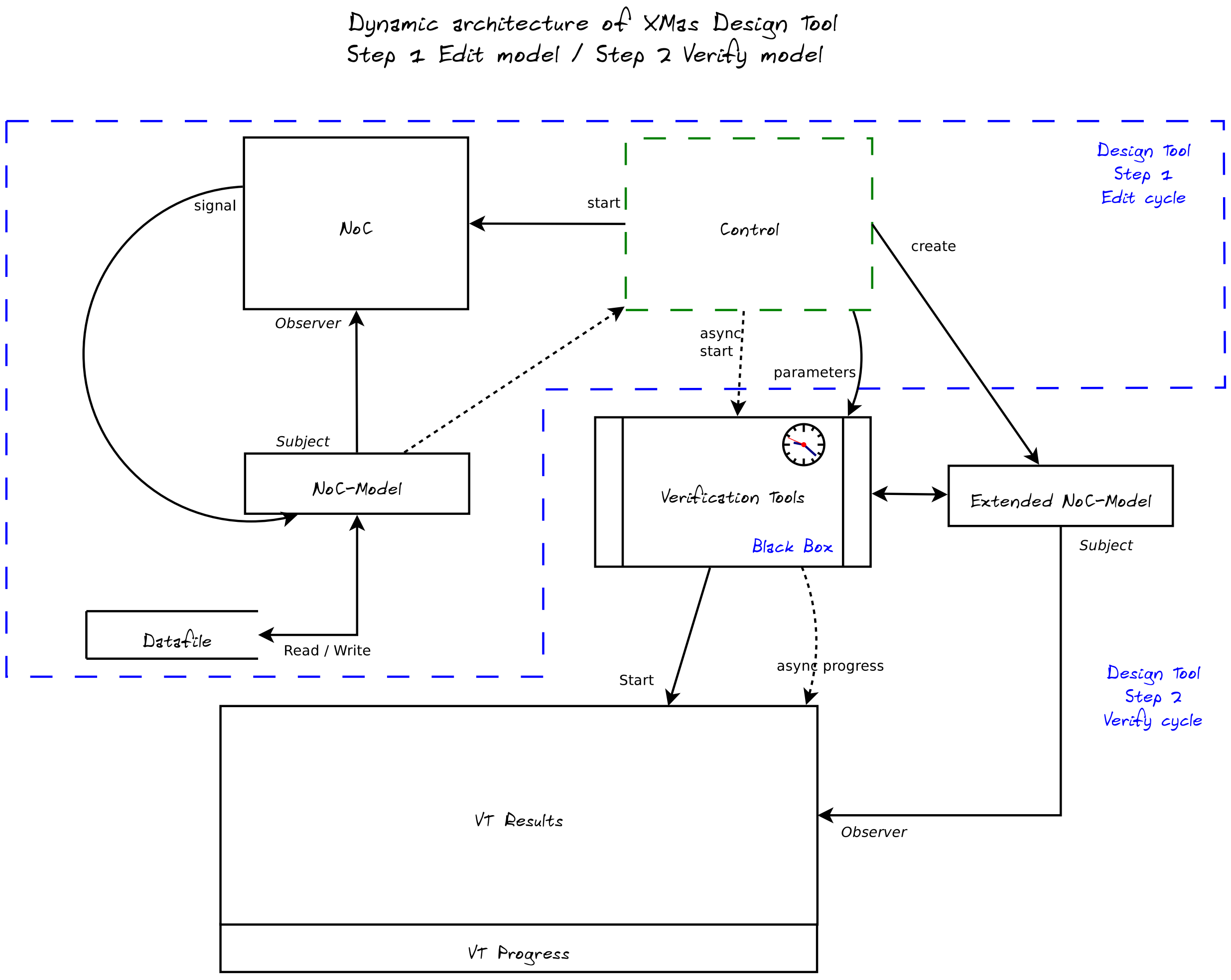


Figure 2.2: Dynamic process of editing and verifying

Verification results The verification tools will show their results and their progress while running. Depending on the verification tool, the results could be graphical or textual or both. The plugin will show either textual or graphical output allowing for addition textual or graphical markings on the network drawing.

Remark that the verification tools run in their own thread or process depending on whether the output will be shown in the design UI thread or an independent UI thread. TBD

Communication pattern The user interface implements the observer pattern with respect to data changes. Both the design tool and the verification plugin use this pattern to propagate changes in the model to the user interface.

2.3 Verification pipeline

Scope. The verification pipeline itself is not in scope for the graphical editor. The connection to the verification pipeline is through the plugin architecture, where a plugin is wrapped in a C++ class that derives from VerificationTool.

2.4 Verification Tool repository

Tool repository Each tool needs to register with a tool repository from where a control object could load the objects and obtain the necessary meta information.

Meta information Each tool needs certain parameters filled in. The tool should provide a method of entering this data both through a graphical user interface and through providing a file with input.

Generic meta information Some parameters have no influence on the structure of the component, other than specifying the number of entries in a queue. Other parameters alter the structure in a way that changes the network in size. An example is a number n that specifies the number of primitives in a spidergon topology (see []).

Design details Each verification tool must register with the VTController or the controller. The registration involves informing the VTController about function, and parameters of the specific verification tool. The graphical editor will ask the VTController for available verification tools. This way the connection between the graphical design tool and the verification pipeline is as loose as possible while keeping control over the dynamics in the process.

2.5 Component repository

2.6 Layer architecture

2.7 Plugin communication

2.8 Primitive toolkit communication

2.9 Toolkit repository

2.10 Model repository

2.11 Architecture High Level Diagrams

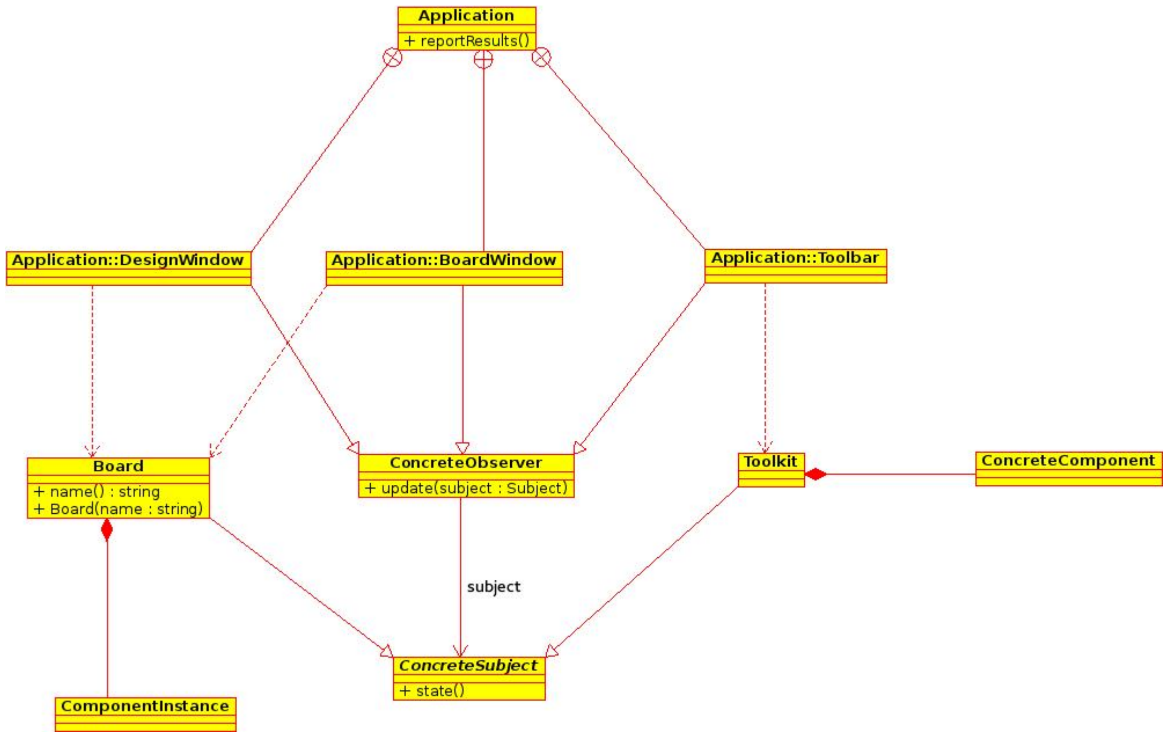


Figure 2.3: User Interface high level class diagram

Chapter 3

Architecture Design Guidelines

3.1 Observer pattern

We use our own observer pattern as described in the GoF. Figure 3.2 the pattern applies the principle. Any object wanting to observe one of the subjects, needs to derive from ConcreteObserver.

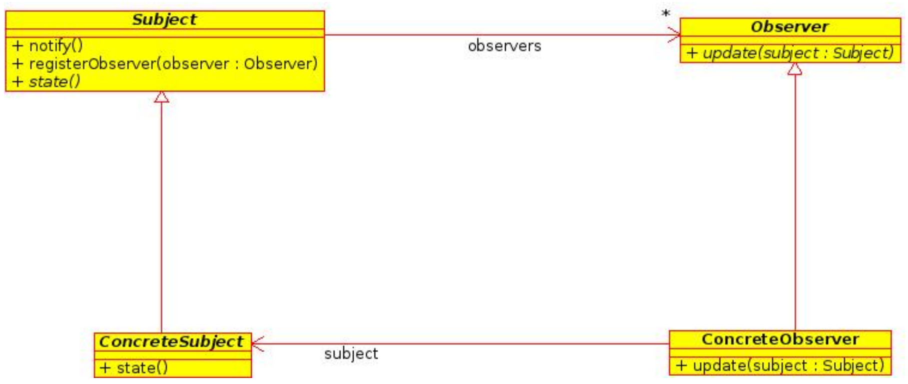

Subject
+ notify()
+ registerObserver(observer : Observer)
+ state()
observers
*
Observer
+ update(subject : Subject)
ConcreteSubject
+ state()
subject
ConcreteObserver
+ update(subject : Subject)

Figure 3.1: Observer pattern as described in GoF

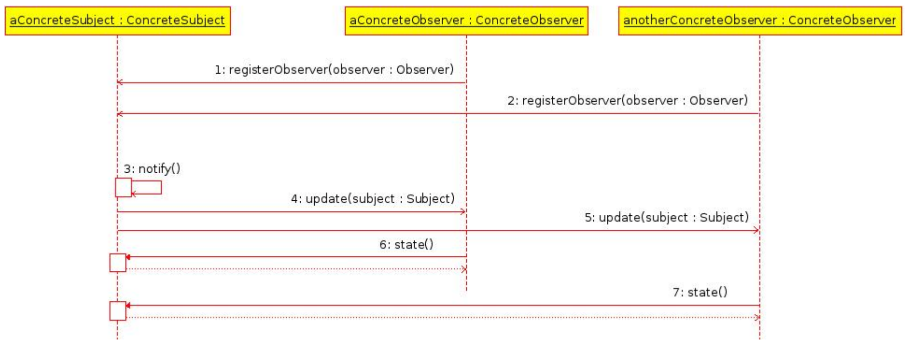

aConcreteSubject : ConcreteSubject
aConcreteObserver : ConcreteObserver
anotherConcreteObserver : ConcreteObserver
1: registerObserver(observer : Observer)
2: registerObserver(observer : Observer)
3: notify()
4: update(subject : Subject)
5: update(subject : Subject)
6: state()
7: state()

Figure 3.2: Observer pattern applied

3.2 Plugin construction

Any verification tool must override the ConcreteVerificationTool and register with the application in order to have the option of being executed on request.

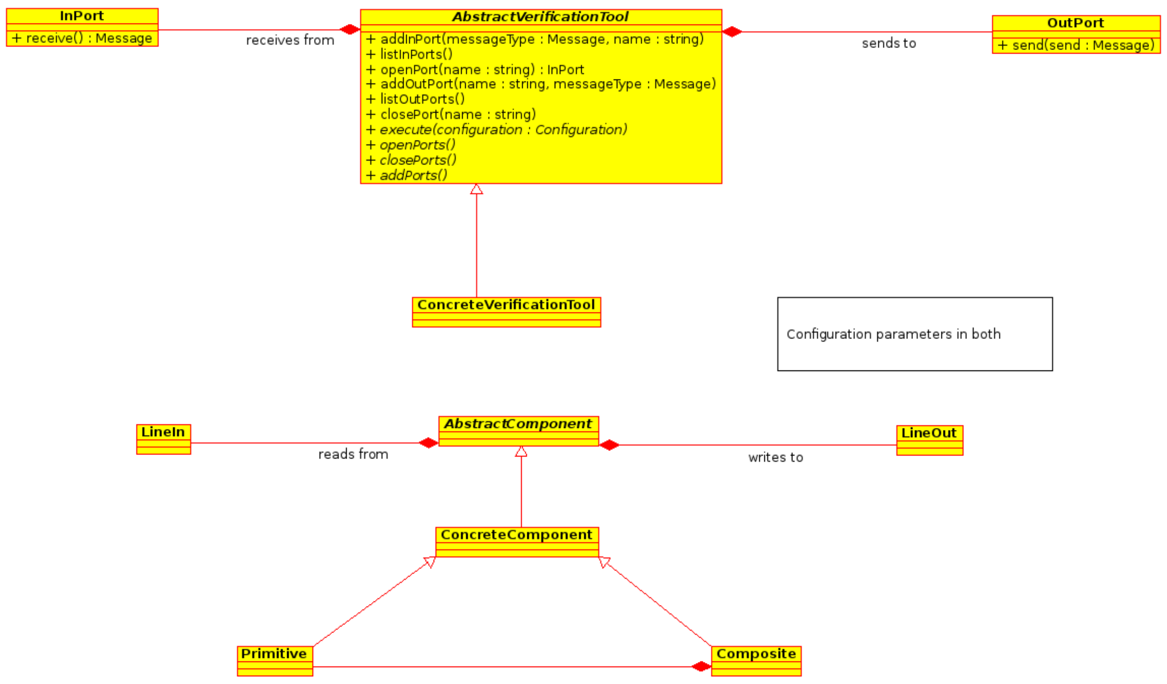


Figure 3.3: The plugin to build verification tools.

Chapter 4

Platform dependent constraints

4.1 Home of the software

4.2 Linux

4.2.1 Download and install

4.2.2 Regular build procedure

4.3 Macintosh

4.3.1 Download and install

4.3.2 Regular build procedure

4.3.3 Build limitations

4.4 MS Windows

4.4.1 Download and install

4.4.2 Regular build procedure