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 to work with.

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

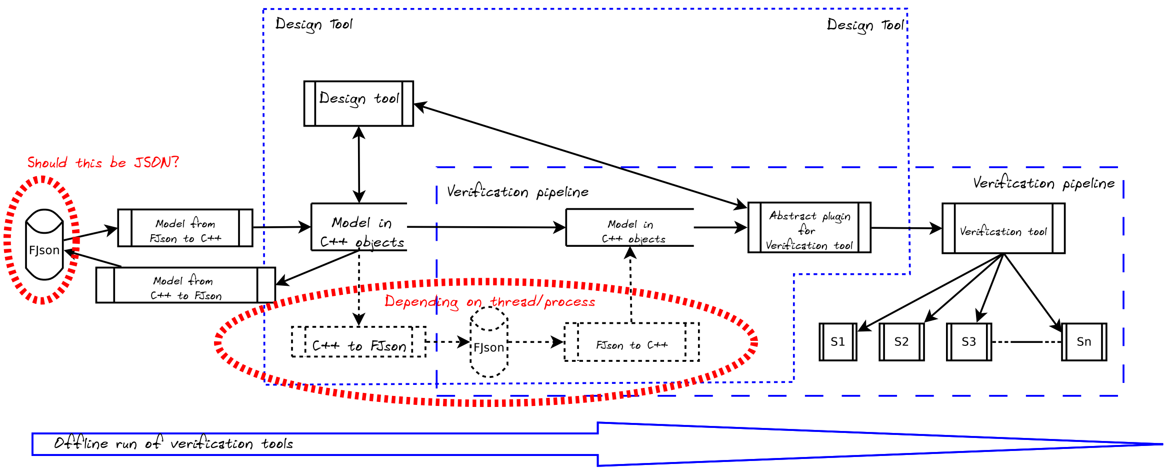
Chapter 2

Architecture Overview

The system as a whole includes the design tool, the verification pipeline and the specific verification tools. If we document the design tool only, we leave out essential parts of the system. TBD.

Where to document the verification pipeline and the verification tools? Maybe the pipeline in general (without VT specifics) should be documented here. Maybe a template for specific VT documentation would come in handy. TBD.

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?

Where to flatten the model: in the verification tool or in the design tool (plugin)? Could a VT work with composites?

2.2 Graphical Design Tool Dynamics

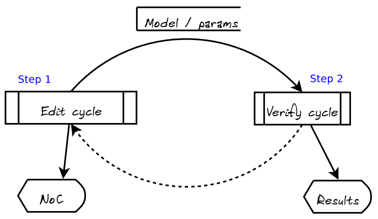


Figure 2.1: Dynamic process in overview

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.

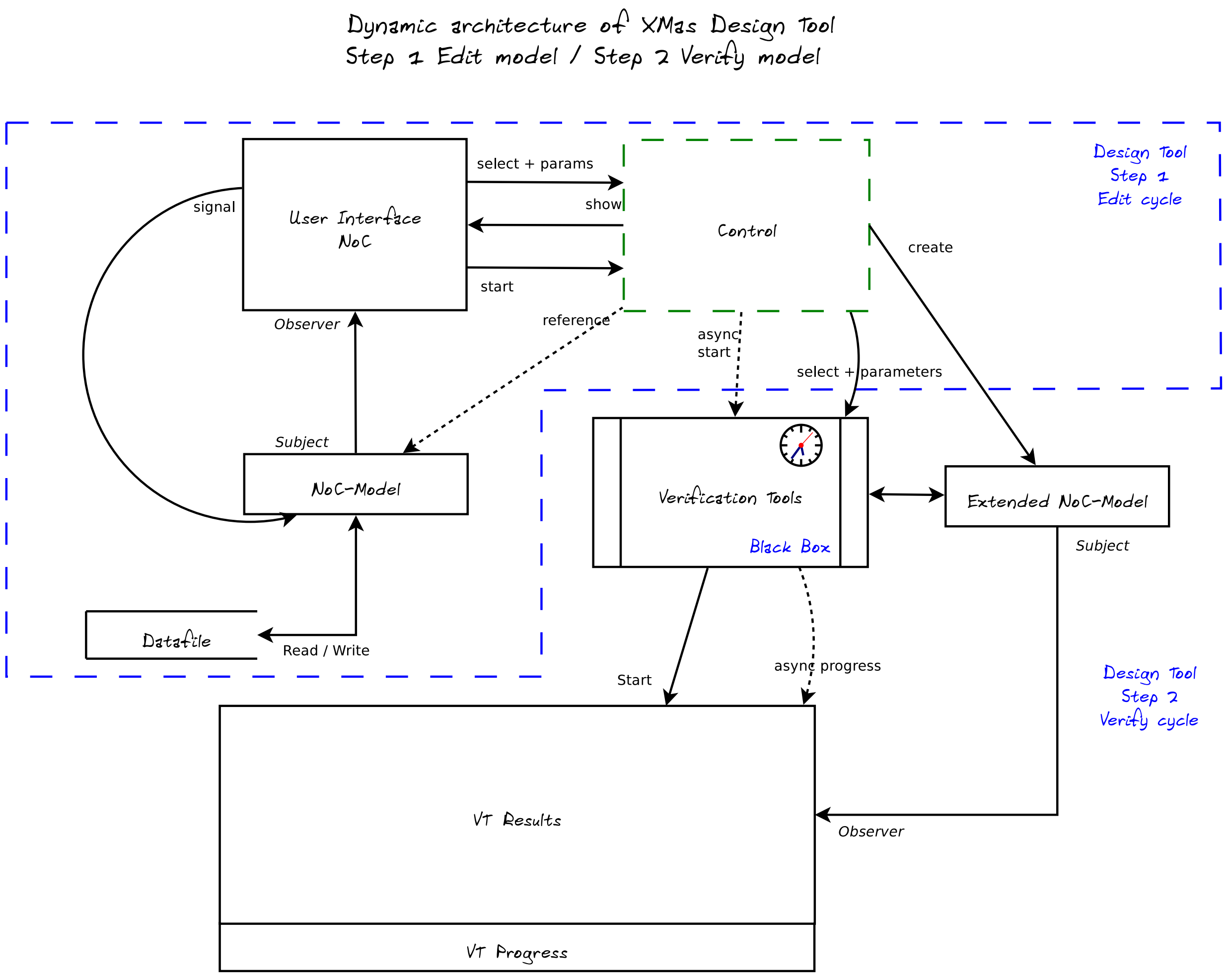


Figure 2.2: Dynamic process of editing and verifying

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 The design tool needs a reference to each verification tool in order to have the user select the verification tools to run. This information whether stored in the code or stored in a file, we call the tool repository. Each tool needs an entry in the tool repository.

Tool 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 directly influence the structure of the component. An example is a number n that specifies the number of primitives in a spidergon topology (see []).

These parameters need special care when passing through to the verification tools. whether and how to flatten is still subject to discussion.

Control in a loosely coupled way An way to implement the verification repository is to have each verification tool register with a controller. The registration involves informing the controller about function, and parameters of the specific verification tool. The graphical editor will ask the controller for available verification tools. The verification tools should also provide a dialog for entering the parameters. 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

We need a (non-programmatic) way of defining new components, whether primitive or composite.

2.6 Model repository

Some way of storing models.

Chapter 3

Architecture Design Guidelines

3.1 Leading requirements

The following list of leading requirements is meant to guide the creation and maintenance of design guidelines and design decisions. The most important requirements are first and foremost leading arguments in any comparison of alternatives.

Portability leads We need the design tool to run equally well on all defined platforms and possibly platforms not (yet) defined. For that reason portability is a leading requirement.

Ease of use seconds We need the design tool to assist in the design of NoC and the development of verification tools with respect to NoC designs. For that reason the design tool should facilitate these activities as much as possible.

Maintainability third The ease of maintenance was one of the primary requirements and as such should guide future design decisions.

Performance follows Under normal circumstances performance arguments should not lead. However, unacceptable performance will override all other considerations, because unacceptable performance disrupts use of the tool.

Remark What is acceptable performance is subjective.

Example When given a choice between alternatives with equal portability and ease of use consequences, the maintenance arguments should lead at the cost of performance provided the performance seems acceptable.

Agreement On 6th December 2014 Bernard agreed to this priority ordering.

3.2 High level design guidelines

C++ constructs lead We strive to use C++ of the latest standard, currently C++ 2011. This implies the use of the C++ library including containers.

Boost We use BOOST as our goto library for portability. For mechanisms that would otherwise require platform dependent code we use BOOST. Messaging, process control and interprocess communication is all directed through BOOST.

fltk We use the GUI library FLTK (pronounce "`fulltick`") for all GUI code. This takes care of most of the performance concerns for a graphical user interface.

OpenGL Where we need extra drawing power we use OpenGL or the fltk gl extentions. This takes care of the remaining performance concerns that might exist.

3.3 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.

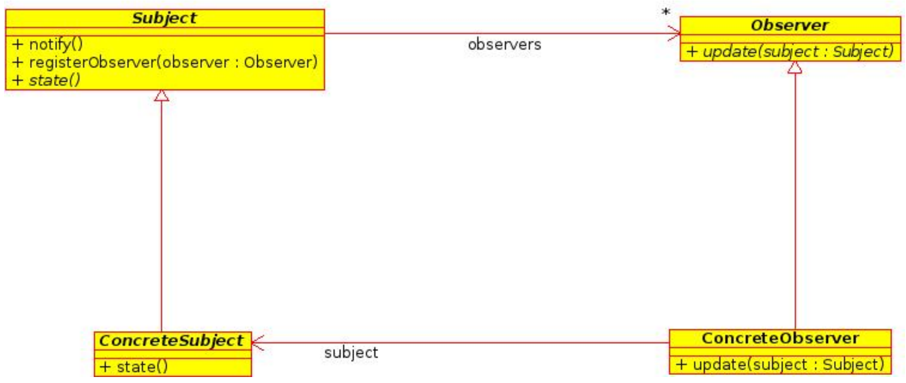

Subject
+ notify()
+ registerObserver(observer : Observer)
+ state()
observers
*
Observer
+ update(subject : Subject)
ConcreteSubject
+ state()
subject
ConcreteObserver
+ update(subject : Subject)

Figure 3.1: Observer pattern as described in GoF

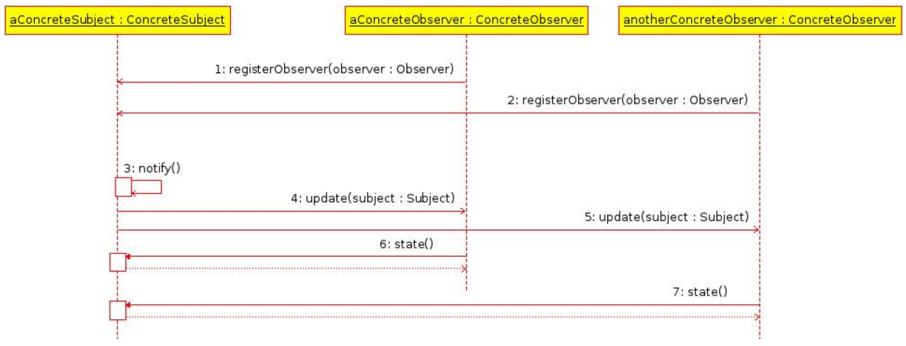

aConcreteSubject : ConcreteSubject
aConcreteObserver : ConcreteObserver
anotherConcreteObserver : ConcreteObserver
1: registerObserver(observer : Observer)
2: registerObserver(observer : Observer)
3: notify()
4: update(subject : Subject)
5: update(subject : Subject)
6: state()
7: state()

Figure 3.2: Observer pattern applied

3.4 Architecture High Level Diagrams

To do

3.5 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.

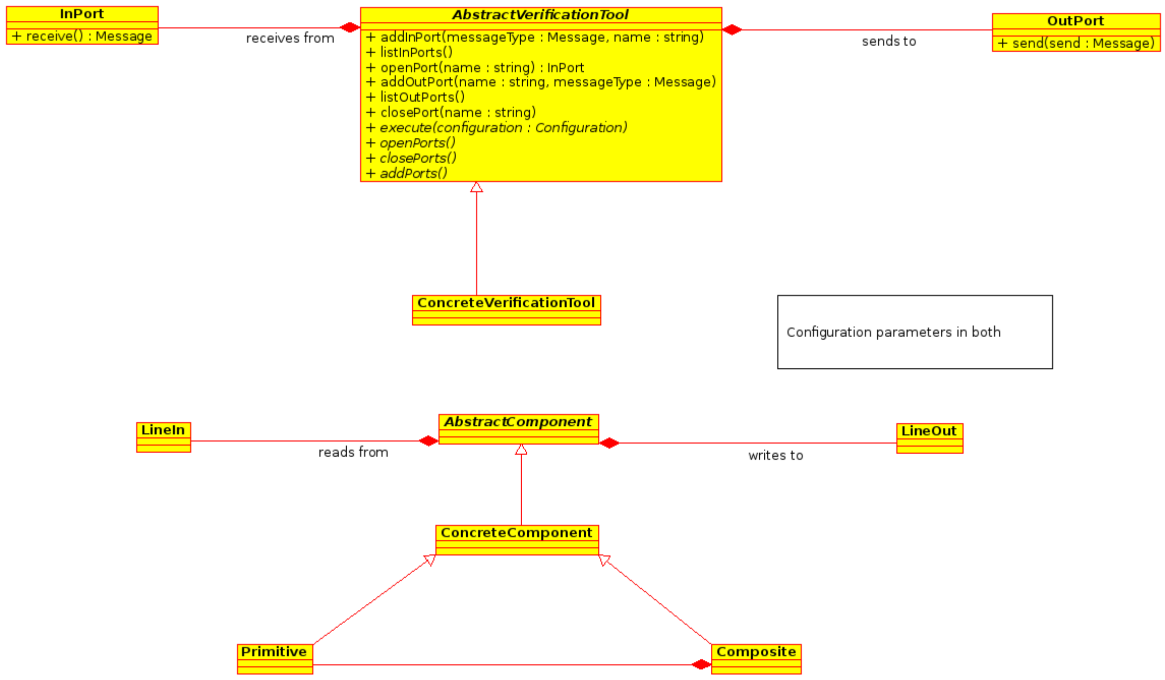


Figure 3.3: The plugin to build verification tools.

3.6 Language guidelines

3.6.1 C++ construct preferences

Our coding strives to use the best of C++ 2011 and to avoid coding that may be error prone. Sometimes a construct like

```
using typename = std::vector<std::string>
```

should enhance readability provided the `typename` chosen makes sense in the context. Table 3.1 lists the current coding preferences.

Prefer	**Avoid**	**Comment**
shared_ptr, unique_ptr	pointer	
weak_ptr		with specific reason
make_shared()	new, delete	

Table 3.1: Preferences in coding guidelines

Memory management Using smart pointers avoids many of the pointer pitfalls causing memory leaks to occur that might go undetected. When creating dynamic memory preferring `make_ptr` leads to code that avoids mixing the old paradigm (`new` and `delete`) with the new.

weak_ptr A `weak_ptr` we use only with a specific motivation because it requires extra work to make sure that we are referencing an existing smart pointer i.e. it has not been deleted yet.

allocator In most situations the library versions of `vector`, `list` and `string` should suffice and be performant enough for our goals. In case explicit memory management is necessary we could use the library class `allocator`. This ensures efficient allocation and deallocation of memory with knowledge of type information and separating allocation and construction of memory.

Coding style The file `CODING-STYLE` in the root of the git tree contains the coding style we adhere to.

We adopt the coding style used by the verification tools.

- C-style type specifiers (Type &x) instead of (Type& x)

  JK: Not my preference, but that's what the verification tools use

- mixed camelCase member names
- capital CamelCase type names
- indentation: 4 spaces (no tabs)
- opening braces on the same line
- don't introduce {} blocks for single statements

Prefer	**Avoid**

`if (x)`
`    foo();`
	`if (x) {`
	`    foo();`
	`}`

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