NL—a Network Language

Language Report

Jörn W. Janneck

DRAFT

ASTG Technical Memo
Xilinx DSP Division
June 26, 2006

Contents

Chapter 1

Introduction and design goals

This memo describes the Network Language, NL. It is a language for expressing algorithms that compute directed graphs among *entities* with *ports*. The first use of this language is the description of dataflow networks. In that case, the entities are dataflow actors, and the connections between them are FIFO channels that connect an output port of one actor to an input port of another.

The design of NL is guided by the following design goals:

- **generality**—The language must allow the description of arbitrary structures, and of arbitrary algorithms computing graph structures.
- **expressiveness**—The complexity of the description should bear a reasonable relation to the complexity of the described graph constructing algorithm. Regular structures should have simple descriptions associated with them. In particular, constant graph structures should be very straightforward to describe.
- **target independence**—Even though the language is targeted at describing dataflow networks, it should be general enough to accommodate the description of other structures of the same kind, such as circuits.

Note: In this draft version of the memo, reference will be made to syntactic elements that are not further explained here. The description of these elements can be found in the "CAL Language Report" [**?**].

Chapter 2

Describing a network class

A *network class* is the fundamental unit of specification in NL. It represents a named construct that maps a number of *parameters* onto a network of *entities*.

Network → **network** QualID ['[' TypePars ']'] '(' Pars ')' IOSignature '→'
{ Import } [VarDeclSection] { Network }
[EntitySection] [StructureSection]
(**end**| **endnetwork**)

IOSignature → [PortDecls] '==>' [PortDecls]

PortDecl → [Type] ID { '(' Expression ')' }

Import → (SingleImport | GroupImport) ';'

SingleImport → **import**[**type**| **entity**| **var**] QualID ['=' ID]

GroupImport → **import**[**type**| **entity**| **var**] **all** QualID

The header of a network class description establishes names for type and value parameters, as well as the port signature of the network, i.e. lists of named and optionally typed input and output ports.

It also includes import statements, which introduce names from specified packages, making them visible to the current network class description. There are separate namespaces for types, entity classes, and variables containing ordinary values, thus an import statement has to define which kind of name is being introduced (type, entity, or variable). If that specification is omitted, **var** is assumed by default.

The header is followed by a number of optional sections.

VarDeclSection → **var** VarDecls

The variable declaration section introduces new variable names, binding them to the values computed by evaluating the corresponding expression.

The variable declarations may be followed by network class definitions. These network classes can be used to instantiate sub-network entities within the current network only.

EntitySection → **entities** { EntityDecl }

The entity section serves to instantiate entities and give them names by which they may be referred to when building the structure in the following section.

An entity can either be *atomic* or *composite*. An atomic entity is only connected to other entities. A composite entity is effectively a subnetwork, consisting of other atomic and composite entities. It can be *flattened* into its containing network, effectively resolving it and absorbing its constituent entities into the containing network. Recursively flattening all composite entities in this manner will eventually lead to a "flat" network that only contains atomic entities.

StructureSection → **structure** { StructureStmt }

The structure section establishes connections between ports. The ports that can be connected here belong either to the network itself, or to the contained entities.

Chapter 3

Creating entities

The **entities** section serves to create the entitites that make up the network, and to give them names by which they can be referred to when building the network structure.

EntitySection → **entities** { EntityDecl }

EntityDecl → [EntityType] ID '=' EntityExpr ';'

EntityExpr → ID '(' Expressions ')' [AttributeSection]
| **if** Expression **then** EntityExpr **else** EntityExpr **end**
| '[' EntityExprs ['→' Generators] ']'

An entity declaration either refers to a single entity or to a (possibly nested) list of entities. Correspondingly, the expressions creating entities or such entitiy lists are constrained to be either direct instatiations (an entity name applied to a list of arguments), a conditional entity expression, or a list comprehension with entity expressions as its element expressions.

AttributeSection → '{' { AttributeDecl } '}'

AttributeDecl → ID '=' Expression ';'
| ID ':' Type ';'

Any basic instantiation expression may be followed by an *attribute section*. An attribute section adds an arbitrary list of named values or types to an entity. These can be used to control the behavior of the tools that process the result of the network language, adding additional information to each entity.

Chapter 4

Building network structure

StructureSection → **structure** { StructureStmt }

StructureStmt → Connector '- - >' Connector ';' [AttributeSection]
| **foreach** Generators **do**{ StructureStmt } **end**
| **if** Expression **then** { StructureStmt } [**else** { StructureStmt }] **end**

The structure section consists of a sequence of *structure statements*. Each of these statements will evaluate to a (possibly empty) set of directed port-to-port connections. Such a connection may start at an input port of the network or an output port of any contained entity, and it may end at an output port of the network or an input port of any contained entity.

There is a **foreach** loop construct, as well as a conditional structure statement.

Connector → PortRef
| EntityRef '' PortRef

EntityRef → ID { '[' Expression ']' }

PortRef → ID { '[' Expression ']' }

Chapter 5

Type system