

AM06: Fleet Description Language

draft v0.9

Adam Megacz
megacz@cs.berkeley.edu

October 16, 2006

1 Introduction

FDL is based principally on Alain Martin's *Communicating Hardware Processes* [Mar87], [Mar95] and, which in turn was derived from CSP [Hoa04] and Dijkstra's *Guarded Commands* [Dij75].

Briefly, FDL's semantics are the same as those of CHP, except that the "probe" operator can only appear within a guard, and FDL has only a single form of guarded selection. FDL also extends the semantics of CHP with additional channel types and data types.

The metagrammar used in this paper is detailed in [AM06].

2 Basics

A FDL program consists of a single top-level definition/declaration list:

```
Program = DefDeclList
```

A definition/declaration list is a list of one or more declarations or definitions:

```
DefDeclList = (Definition | Declaration)+
```

Declarations specify the type of some identifier; definitions specify its implementation (or value):

```
Declaration = Id ":" Type  
Definition = Id "=" Statement
```

Where `Id` is the production for identifiers (variable names)

```
Id      = [_a-zA-Z] [_a-zA-Z0-9]**
```

The single underscore identifier is a special identifier used to indicate that a value does not matter.

2.1 Ambiguity and Indentation

A *line* is a maximal sequence of characters not including a newline (`\n`). The *indentation* of a line is the number of contiguous space () characters at the start of the line.

When indentation is not considered, the grammar for FDL is deliberately ambiguous. That is, there are strings for which there exist multiple valid parse trees. It is the programmer's job to eliminate this ambiguity with parenthesis or indentation; programs with more than one valid parse tree are rejected.

The FDL grammar includes an additional rule: if a definition or declaration spans more than one line, *no line may have indentation less than that of the first line*. When parsing, the FDL parser simply will not consider parsing alternatives that would involve a definition or declaration whose indentation *decreases*.

3 Statements

The send operator, which can appear both backwards and forwards, evaluates its `Expr` argument and sends the resulting value along the `Channel` argument:

```
Channel = Id

Statement = ...
           | Expr  "!"->" Channel
           | Channel "<-" Expr
           ...
```

The receive operator reads a value from its `Channel` argument and binds that value to the channel name when executing the corresponding `Statement`:

```
Statement = ...
           | "?" Channel "->" Statement
           ...
```

3.1 Composite Statements

Statements can be composed sequentially (with `;`) or in parallel (with `||`):

```

Statement = ...
  | Statement ";" Statement
  | Statement "||" Statement
  ...

```

Additionally, a “where” clause can appear after any statement to define terms within the statement:

```

Statement = ...
  | Statement "where" DefDeclList
  ...

```

3.2 Guarded Statements

The receive operator shown earlier is actually a special case of the more general *guarded command* construct:

```

Statement = ...
  | "?" GuardedCommand ("|" GuardedCommand)*
  ...

GuardedCommand = Guard "->" Statement

Guard = Trigger ("," Trigger)*

Trigger = "?" Channel
  | Expr

```

A statement can be a list of guarded commands separated by the vertical bar. An optional vertical bar can be added at the beginning of the command; this is usually done when laying out the guards vertically.

Each guarded command is a statement protected by a set of *triggers*. A trigger can either be an expression which evaluates to a boolean or a Channel preceded by a question mark. A guard is *enabled* when all of its Expr triggers evaluate to true and data is ready to be read from each of its Channel triggers.

When a guarded command set is executed, the process will wait until at least one guard is enabled. It will then *nondeterministically* choose one of the enabled guards, read a value from each of its trigger channels, and execute its statement. Within this statement, the identifier corresponding to each trigger channel will be bound to the value read from that channel.

For example, the following program will read from b as well as either a or c. It will then sum the two values read and send that value along out:

```

program =
  | ?a, ?b  ->  a+b !-> out
  | ?b, ?c  ->  b+c !-> out

```

Note that if both *a* and *c* arrive before *b* arrives, the arrival of *b* will trigger a nondeterministic choice between the two alternatives.

It is important to note that a channel can only be used as an expression (ie a value) in a statement which it acts as a trigger for. In particular, the following is invalid:

```
program =
| ?a, ?b  ->  c !-> out    // Invalid
| ?b, ?c  ->  a !-> out    // Invalid
```

This is the principal difference between the use of guarded commands in CHP and FDL. To summarize: in FDL, checking if a channel has data and reading (consuming) that data is an atomic action; in CHP it is not.

4 Non-Deterministic Composition

Note that FDL is a language for specifying *programs*, not *sets of behaviors*. For this reason there is no unrestricted *nondeterministic choice* operator such as the $+$ of CCS and the \sqcap of CSP.

The guarded commands in the previous section do, however, offer a form of nondeterminism when two guards are enabled at the same time.

5 Channels

Channels must be declared before they are used. Currently, the following channel types are supported:

```
PrimitiveType = "void" | "boolean" | "int"[0-9]+

Type = PrimitiveType
| PrimitiveType "event"
| PrimitiveType "fifo"
| PrimitiveType "reg"
| PrimitiveType "signal"
```

Appendix A: Comparison to CSP

Pure CSP, as described in Hoare's book *Using CSP* [Hoa04] is a bit too general for the purpose of describing ships. In fact, CSP is not so much a programming language as a generalized notation for describing sets of traces – sometimes without suggesting a concrete procedure for generating those traces. For example, the unguarded, nondeterministic choice operator (\sqcap) in CSP does not give enough information to implement a process; it only helps in describing *a set of behaviors* to which a given process' behavior may or may not belong.

CSP has three forms of choice (\square , \sqcap , and $|$), which need not be guarded, and are occasionally equivalent to each other. Currently FDL provides only a single choice operator and leaves the question of whether or not that choice is deterministic to other mechanisms [TA06].

CSP's generalized recursion (μ) requires potentially unbounded storage, and therefore is not appropriate for designing hardware. However, the [mutually] recursive `DefDeclLists` of FDL can be translated into special cases of the μ -construct.

FDL does not include a notion of “failure” as a primitive; therefore the interrupt (Δ), catastrophe-recovery (\ddagger), and restartable (\dot{P}) operators of CSP are not meaningful.

CSP employs implicit communication in many places, whereby communication/synchronization occurs between processes which happen to share syntactically identical names – traditional lexical scoping is not used. This is quite handy when describing small systems such as the weave of Petri Net snippets [Ben96]. However, it is this author's judgement that communication dependencies should be made *explicit* in languages designed for large systems. For this reason, the channel-based communication model of CHP replaces the hiding (\backslash), interleaving ($|||$), and subordination ($//$) operators of CSP.

The chaining ($>>$) operator of CSP is strictly syntax sugar and is omitted.

To summarize: like many other domain-specific languages for concurrent systems, FDL chooses what is essentially a subset of CSP that can be completely translated to hardware. This subset is substantially similar to CHP.

On Determinism

Hoare uses a special definition of the word “nondeterministic” – by his definition, a process whose choices are observable but *not determined* by its inputs is still considered to be “deterministic.”

Throughout this document, I use the definition favored by the functional programming languages community: a function or process is deterministic if its outputs are *determined* exclusively by its inputs and nothing else.

References

- [Mar87] Alain J. Martin, *Programming in VLSI: From Communicating Processes to Self-timed VLSI Circuits*. Proc. UT Year of Programming Institute on Concurrent Programming, March 1987, Addison-Wesley(1990). Caltech Computer Science Technical Report 5257:TR:87.
- [Mar95] Alain J. Martin, *Communicating Hardware Processes*.
<http://www.cse.ttu.edu.tw/~cheng/courses/async/readings/chp95.pdf>
- [JU93] Mark B. Josephs and Jan Tijmen Udding, *An overview of DI algebra for delay-insensitive circuits*. In Proc. Hawaii International Conf. Systems Sciences, volume I. IEEE Computer Society Press, 1993.
- [Hoa04] C. A. R. Hoare, *Communicating Sequential Processes* (2004)
<http://www.usingcsp.com/cspbook.pdf>
- [Dij75] E. W. Dijkstra, *Guarded commands, nondeterminacy and formal derivation of programs*, Communications of the ACM, Vol. 18, No. 8, 1975, pp. 453-458. 239.
<http://portal.acm.org/citation.cfm?doid=360933.360975>
- [TA06] Tachio Terauchi and Alex Aiken *A Capability Calculus for Concurrency and Determinism*. In Proceedings of the 17th International Conference on Concurrency Theory (CONCUR 2006), August 2006.
<http://http.cs.berkeley.edu/~tachio/papers/concur06-capcon-full.pdf>
- [Ben96] Benko, Igor. *Weaving Petri Nets*. IB-96-01.
<http://research.cs.berkeley.edu/project/fleet/wiki/files/petri-net-weave.pdf-dir.pdf>
- [AM06] Megacz, Adam. *Scannerless Boolean Parsing*. Proceedings of LDTA'06, Electronic Notes In Theoretical Computer Science 1368.
<http://www.megacz.com/research/papers/megacz-ldta06.pdf>