

# Tasks graph and data dependencies

Benoît Perrot <[benoit@lrde.epita.fr](mailto:benoit@lrde.epita.fr)>

Technical Report *n°0311* - July 2003

**Towards an object-oriented logic** The aims of this report is to analyze existing solutions to express dependency between tasks, in order to formalize a powerful and reusable system of dependency expression. From “make” utility to Prolog, it collects the best of each well-known dependency schema in order to bring an object oriented flavor to first order logic.

**Vers une logique orientée objet** L’objectif est ici de formaliser l’expression orientée données des dépendances entre tâches, afin de construire un schéma puissant, efficace et réutilisable d’ordonnement de tâches. Après analyse des outils dédiés les plus connus sera présentée une logique orientée objet dont le but est de formaliser l’expression des dépendances.

## Keywords

Tasks Graph, Make Utility, Object-Oriented Logic, F-Logic



Laboratoire de Recherche et Développement de l’Epita  
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France  
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22

[lrde@epita.fr](mailto:lrde@epita.fr) – <http://www.lrde.epita.fr>

# Contents

<b>Introduction</b>	<b>1</b>
<b>1 Exploring dependencies</b>	<b>3</b>
1.1 Name-driven schema	3
1.1.1 Design	3
1.1.2 Implementation : LRDE's Tiger compiler task system	3
1.1.3 Solving	3
1.1.4 Discussion	4
1.2 Data-driven schema	4
1.2.1 Design	4
1.2.2 Implementation : "make" utility	4
1.2.3 Solving	5
1.2.4 Discussion	6
1.3 Automated deduction	6
1.3.1 Analogy	6
1.3.2 Implementation : Prolog	7
1.3.3 Solving	7
1.4 Keeping the best of each methods	8
<b>2 Object Oriented Logic</b>	<b>9</b>
2.1 Intuition	9
2.1.1 Reasoning within a logic class	9
2.1.2 Reasoning between objects	10
2.2 Formal definition	10
2.2.1 Class	10
2.2.2 Instantiation	11
2.2.3 Origin and garbage	12
2.2.4 Converters	12
2.2.5 Dependency theory	12
2.3 Resolution algorithm	12
2.3.1 Solving domain	12
2.3.2 Principle	15
2.3.3 Considerations	15
<b>3 Proposed syntax</b>	<b>16</b>
3.1 Alphabet	16
3.2 Formulas	16
3.2.1 Classes	16

3.2.2	Converters . . . . .	17
<b>4</b>	<b>Applications</b>	<b>19</b>
4.1	Describing LRDE's Tiger compiler tasks . . . . .	19
4.1.1	Default constructor : building a token stream . . . . .	19
4.1.2	Converters : building an AST and displaying it. . . . .	19
4.2	Describing "make" rules . . . . .	20
4.2.1	Factored rules . . . . .	20
4.2.2	PHONY rules . . . . .	20
4.2.3	Complex example : librairies . . . . .	20
<b>5</b>	<b>Conclusion</b>	<b>21</b>

# Introduction

## Context

The current quest for flexibility and reusability of software led programmers to consider component based design for software development. Increasing reuse and decreasing development time are both the origin and the benefit of programs' modular design.

The common process is to develop libraries separately and to inter-connect them through their predefined interfaces. In simple cases, this kind of inter-connection may be resolved statically: each component call others in an order that is uniquely determined at compile-time. But in some cases, the combination of components can only be built dynamically: following its inputs and the outputs wanted by the user, a program organizes its internal components at runtime, to produce the desired outputs. In this model, a resolution engine that has to deal with inter-component dependencies must be integrated in the software. Its purpose is to "understand" given inputs and wanted outputs and to consider modules dependencies, in order to establish a valid combination of components.

To permit such an engine to succeed in building a valid execution path, developers must clearly describe program tasks and their dependencies.

## Dependency expression

The goal is here to define a formalism and an adapted syntax to write task dependencies in an efficient way: the schema must come with a good expressivity and a low setup cost. To build such a schema, the real needs have to be extracted: what expression power must provide the schema? What constraints may it bring? etc. A formal framework must then be proposed to design these needs, and a grammar must be set up to express this formalization.

This way, two dual expression schemas will first be presented and discussed: a name-driven expression and a data-driven one. Then an analogy between dependencies resolution and logical deduction will be set, leading the discussion toward an object-oriented logic. Finally, a syntax will be proposed and use cases will be described.

# Chapter 1

## Exploring dependencies

### 1.1 Name-driven schema

#### 1.1.1 Design

The easiest way to express tasks and their dependencies is to assign a name to each task, such that no other task may have the same, and to write the name of tasks on which they depend.

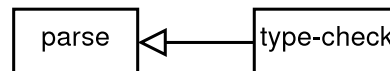
This name-driven schema consists in explicitly describing a dependency graph by an adjacency list: a name of a task is a vertex of this graph, and each named dependency is an edge between two vertices.

#### 1.1.2 Implementation: LRDE's Tiger compiler task system

The tasks engine implemented in LRDE's Tiger compiler in C++ uses a name-driven approach.

A C++ class `Task` aggregates a name, a call-back function and a list of names that represent dependent tasks. Each time a `Task` is instantiated, it registers itself to a `TaskRegister`, which role is to collect and resolve dependencies. In practice, a developer might write that task "type-check" depends on "parse" this way:

```
Task type_check( "types-check",  
                type_check_fun,  
                "parse" );
```



When invoked by the user, the `TaskRegister` will execute "parse" then "type-check" if needed.

#### 1.1.3 Solving

This explicit dependency graph may be resolved by a trivial and well-known algorithm, a topological sort, which is the "linear ordering of all vertices of a directed graph  $G$  such that if  $G$  contains an edge  $(u, v)$ , then  $u$  appears before  $v$ " (Cormen et al., 2002) and has an acceptable complexity of  $O(V + E)$ , with  $V$  the number of vertices of the graph and  $E$  the number of its edges (depth-first search complexity).

### 1.1.4 Discussion

#### Advantages

- Simple to implement: a simple graph data type and a trivial algorithm are the only needs of a solver for such a schema.

#### Drawbacks

- Error-prone: the low-level of information forbids automatic check system.
- Low expression power: the whole dependency graph must be explicitly written.
- Some constructions cannot be expressed. Consider the task “scan-trace” that enables the trace mode of the scanner: it does not have any dependency (because it does not need other tasks to have been done) and “parse” cannot have it for dependency (because else “scan-trace” would not be optional). So “scan-trace” cannot depend on “parse” and reflexively in this syntax, and the influence of “scan-trace” on “parse” cannot be expressed.

## 1.2 Data-driven schema

### 1.2.1 Design

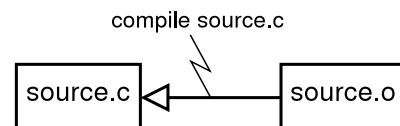
A clever way to express dependencies is to describe a task by the entities it needs and produces.

This data-driven schema is similar to a chemistry equation: a reaction (a task) needs reactants to build products.

### 1.2.2 Implementation: “make” utility

The “make” utility uses a data-driven approach: it allows developers to describe dependencies between files. For example, a binary file “source.o” depends on a source file “source.c”, these informations describing a pseudo-task “compile source.c”:

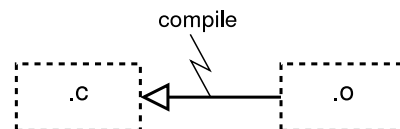
```
## Compile source.o from source.c
source.o: source.c
    $(CC) -c $<
```



#### Enhancement

The “make” utility also provides a way to factor dependency expression; the following rule means that a binary file (which extension is “.o”) may be built from a source file with the same name and the extension “.c”:

```
## Compile a '.o' from its '.c'
.c.o:
    $(CC) -c $<
```



Through this rule, the user does not have to write all similar dependency rules between a source file and its binary: the utility will infer that it must launch the factored task “compile .c into.o” to product a “.o” from a “.c”.

### 1.2.3 Solving

The topological sort will also fit in this case; the problem here is more to build the graph.

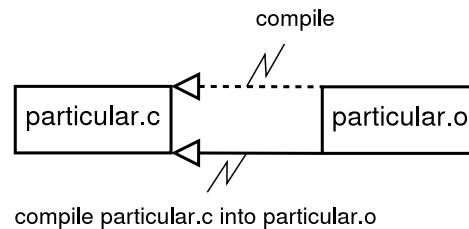
The engine must support factored rules and deal with ambiguities or priorities: it must be able to find a general task from a particular result. Comparing to the first approach, this engine may have to add edges and vertex to the dependency graph when solving it.

#### Priorities

Consider the following rules, where the binary file “particular.o” requires more compilation flags than others:

```
## Compile a '.o' from a '.c'
.c.o:
    $(CC) -c $<

## Compile 'particular.o'
## from 'particular.c'
particular.o: particular.c
    $(CC) -ansi -Wall -c $<
```



On this little example, the engine will give the priority to the more explicit rule (the second one).

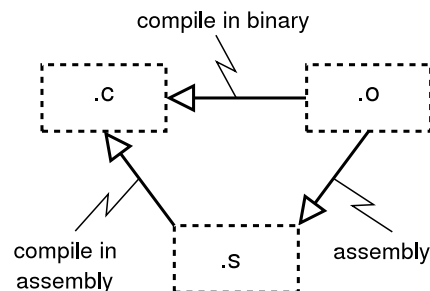
#### Ambiguities

On the following example:

```
## Compile a '.o' from a '.c'
.c.o:
    $(CC) -c -ansi -Wall $<

## Compile a '.o' from
## its assembly source file
.s.o:
    $(CC) -c $<

## Build an assembly file '.s'
## from its C source file
.c.s:
    $(CC) -S $<
```



The engine will arbitrarily choose the shortest path to build a binary “.o” file from its corresponding “.c”, and will use the first rule. If it is removed, another feature of the “make” utility allows it to prefer one of its predefined rule to user’s.

### 1.2.4 Discussion

#### Advantages

- Factorization of transforms: as detailed above, a “make” rule may define or use reusable schemes of transformation. This way, a powerful Makefile may be written in a few rules.
- Emphasis on generated data flow: by using a data-driven approach of dependency, an explicit data-flow graph is described. Such a graph brings a lot of informations and allows liveness analysis, which would be useful to detect unused or dead entities.

#### Drawbacks

- Factored rules need a powerful engine: such an engine must be able to deal with incomplete, ambiguous rules.

## 1.3 Automated deduction

### 1.3.1 Analogy

A simple analogy may be set between a graph dependency problem and a logical deduction.

An entity (a reactant or a product) may indeed be considered as a logical axiom; then the tasks, which are describes with these entities are nothing more than Horn clauses. A dependency graph may so be seen as a logical theory.

#### Example

This way, a simple Makefile might be written:

<pre>## Compilation main.o: main.c     \$(CC) -c \$&lt; sub.o: sub.c     \$(CC) -c \$&lt;  ## Link exe: main.o sub.o     \$(CC) main.o sub.o -o \$@</pre>	<p>“Theory”:</p> <pre>main.c → main.o sub.c → sub.o main.o ∧ sub.o → exe</pre> <p>Knowledge:</p> <pre>→ main.c → sub.c</pre> <p>To prove:</p> <pre>→<sup>?</sup> exe</pre>
---	--

Which means: “to prove exe, first prove main.o and sub.o, knowing main.c and sub.c”, also “to compile exe, first compile main.o and sub.o, from main.c and sub.c”.

### 1.3.2 Implementation: Prolog

Prolog (PROgramming in LOGic) is a logic programming language; it differs from others because it is a declarative language, which works on non-numeric objects.

A program in Prolog consists in a database of axioms and rules and a set of queries. Prolog tries to answer each query using logical algorithms on the database.

#### Example

The last Makefile may be written in Prolog:

Database: <pre>main_o:- main_c. sub_o:- sub_c. exe:- main_o, sub_o.  main_c. sub_c.</pre>	Query: <pre>?:- exe.</pre>
---	----------------------------

### 1.3.3 Solving

A common way to prove first order logic theorem giving a theory is to use the cut elimination. It allows to deduce from two clauses a new one, leading to the wanted theorem.

#### Formal description

In a formal way, consider two clauses

$$\begin{cases} C_1 = \bigwedge p_i \rightarrow \bigvee q_j \\ C_2 = \bigwedge r_i \rightarrow \bigvee s_j \end{cases}$$

If  $\exists(a, b), p_a = s_b$  then consider the new clause

$$C' = \bigwedge_{i \neq a} p_i \bigwedge r_i \rightarrow \bigvee_{j \neq b} q_j \bigvee s_j$$

#### Example

Let the set of atoms of a first order logic theory be  $\{main\_o, sub\_o, exe\}$ . Let the theory be constituted by the clauses

$$\frac{main\_o \wedge sub\_o \rightarrow exe}{\rightarrow main\_o}$$

The result of the cut elimination will be

$$sub\_o \rightarrow exe$$

That is: knowing that main\_o and sub\_o leads to exe and given main\_o, it is proved that sub\_o leads to exe.

## 1.4 Keeping the best of each methods

The needs that were first requested was:

- high expression power
- high factorization power

After studying existing schemas and their respective implementations, new criteria raised:

- A data-driven approach must be preferred, because it implies an explicit description of the data-flow graph which is useful in liveness analysis.
- A logic flavor should be used, because it comes with a strong mathematical environment.
- The language must come with a syntax that allows to define factored expressions.

## Chapter 2

# Object Oriented Logic

The object-oriented logic was first described in [Amir \(2001\)](#). The approach presented here use the same original idea but define a quite different system, less theoretic but best fitting the needs expressed precedently.

### 2.1 Intuition

The original idea behind the object-oriented logic described in [Amir \(2001\)](#) is that a logical theory may be considered as a graph of smaller theories that communicates through their interface.

The interests of such a decomposition are the speed and the expression power improvement. By splitting a big logic theory in smaller ones, a deduction engine will be able to consider independently each sub-theories and do local deductions without having to work on all atoms and clauses. The object-oriented flavor brings the so-desired expression power to logical theory: hierarchies, factorization, polymorphism are well-known object-oriented features and will simplify the design of logical theories.

#### 2.1.1 Reasoning within a logic class

A logic class is equivalent to a small logic theory; it mainly differs from a vocabulary aspect:

- atoms of the theory become attributes of its associated class;
- clauses of the theory become methods of the class;
- proving a theorem consists in obtaining an attribute, by applying a chain of methods

The expression “small logic theory” stands for a logic theory which is minimal in terms of atoms and clauses but semantically homogeneous.

#### Example

- A logic class `CoffeeMachine` with its set of attributes = {is-plugged, is-switched-on, is-running, has-water, has-coffee}
- The theory of `CoffeeMachine`:  
 $\text{is-plugged} \wedge \text{is-switched-on} \rightarrow \text{is-running}$

Solving “running CoffeeMachine” so becomes:

$$\frac{\text{is-plugged} \wedge \text{is-switched-on} \rightarrow \text{is-running}}{\text{is-switched-on} \rightarrow \text{is-running}}$$

### 2.1.2 Reasoning between objects

The logic classes are instantiated in objects that are linked through a set of converters:

- a converter (or constructor) is a clause between atoms of objects
- a default constructor is a converter between the origin  $\top$  and an object
- a destructor is a converter between an object and the garbage  $\perp$

#### Example

- A class Coffee with its set of attributes = {is-warm, is-cool}
- “warm Coffee” constructor:  
CoffeeMachine.{is-running  $\wedge$  has-water  $\wedge$  has-coffee}  $\rightarrow$  Coffee.is-warm
- Making “warm Coffee” from scratch

$$\frac{\begin{array}{l} \rightarrow \text{CoffeeMachine.is-plugged} \\ \text{CoffeeMachine.is-switched-on} \rightarrow \text{CoffeeMachine.is-running} \end{array}}{\text{CoffeeMachine.\{is-running} \wedge \text{has-water} \wedge \text{has-coffee}\} \rightarrow \text{Coffee.is-warm}}$$

$$\frac{\text{CoffeeMachine.\{is-running} \wedge \text{has-water} \wedge \text{has-coffee}\} \rightarrow \text{Coffee.is-warm}}{\text{CoffeeMachine.\{is-switched-on} \wedge \text{has-water} \wedge \text{has-coffee}\} \rightarrow \text{Coffee.is-warm}}$$

## 2.2 Formal definition

### 2.2.1 Class

A class  $C$  is defined as a pair of a set of attributes  $\mathcal{A}_C$  and a set of activation methods  $\mathcal{M}_C$ .

$$C = (\mathcal{A}_C, \mathcal{M}_C)$$

#### Attributes

A class  $C$  comes with a set of attributes  $\mathcal{A}_C$ , each may be affected by the use of an element of  $\mathcal{M}_C$ .

The internal state of an instance of a class  $C$  is entirely determined by the current interpretation of the attributes it aggregates. An attribute may have two exclusive states, true or false. Its default state is false; it may be enabled (set to true) through the use of an element of  $\mathcal{M}_C$ .

### Clauses

A clause  $c \in \mathcal{C}_C$  is defined as a tuple of premises and conclusions (Horn's clauses), which are respectively a finite conjunctions and a finite disjunctions of elements of  $\mathcal{A}_C$ :

$$\mathcal{C}_C = \{(p_1, \dots, p_m, q_1, \dots, q_n) \in \mathcal{A}_C^{m+n}, \bigwedge_{1 \leq i \leq m} p_i \rightarrow \bigvee_{1 \leq j \leq n} q_j\} \text{ with } (m, n) \in \mathbb{N}^2$$

A clause abstracts a dependency rule within a class: the left-side of the Horn's clause (the premises) represents the prerequisites of the rule, the right-side (the conclusions) the potentials products of the rule.

### Clause semantic: activation methods

An activation method  $m$  associates a clause  $c \in \mathcal{C}$ , which stands for a signature, to an instruction  $i$ , which is an element of an abstract instruction set  $\mathcal{I}$ . An activation method gives a semantic to a clause by associating an instruction to it.

$$\begin{cases} \mathcal{M}_C \subset (\mathcal{C}_C \times \mathcal{I}) \\ m \in \mathcal{M}_C, m = (c, i) \text{ with } c \in \mathcal{C}_C \text{ and } i \in \mathcal{I} \end{cases}$$

An activation method allows to change the internal state of a class by affecting one or more attributes of an object. The premises of the signature clause define what attributes need to be activated to run this activation method. The conclusions define what attributes the activation method enables (set to true).

For convenience, the set of signatures of  $C$   $\{c \in \mathcal{C}_C, \exists i \in \mathcal{I}, (c, i) \in \mathcal{M}_C\}$  will be noted  $\mathcal{S}_C$ . Remark that  $\mathcal{S}_C \subset \mathcal{C}_C$ .

### Inheritance

A class  $C$  is a subclass of  $B$  if and only if the set of  $B$ 's attributes is included in the set of  $C$ 's and the set of signatures of  $B$  is included in  $C$ 's one.

$$\begin{aligned} \text{The class } C = (\mathcal{A}_C, \mathcal{M}_C) \text{ is a subclass of } B = (\mathcal{A}_B, \mathcal{M}_B) \\ \iff \begin{cases} \mathcal{A}_B \subset \mathcal{A}_C \\ \mathcal{S}_B \subset \mathcal{S}_C \end{cases} \end{aligned}$$

This formula simply means that a subclass inherits the attributes and the signatures of its parent.

### 2.2.2 Instantiation

Class may be realized in objects by instantiation. This artefact of oriented object flavor allows to use the methods and the attributes of a class. A class describes what attributes are manageable, and what methods may be used; an object owns an internal state (an interpretation of the attributes) on which methods may be applied.

An object  $o$  is an instance of a class  $C$  if and only if the set of  $o$ 's attributes is included in the set of  $C$ 's (and reciprocally) and the set of signatures of  $o$  is included in  $C$ 's one (and reciprocally), and  $o$  comes with an interpretation of the attributes of  $C$ :

$$\begin{aligned} \text{The object } o = (C, \delta) \text{ is an instance of } C = (\mathcal{A}_C, \mathcal{M}_C) \\ \text{with } \delta : \mathcal{A}_C \rightarrow \{true, false\} \end{aligned}$$

### 2.2.3 Origin and garbage

Origin  $\top$  and garbage  $\perp$  are two special builtin singleton classes. Their purpose is mainly to provide the ability to express default construction (conversion from nothing,  $\top$ ) and destruction (conversion to garbage,  $\perp$ ).

They do not have any attribute or method and do not need to be instantiated.

### 2.2.4 Converters

#### Signature

The signature  $s \in \mathcal{S}$  of a converter is defined as a set of needed classes, which required internal state may be specified, and a set of produced classes, which initial internal state may also be specified:

$$\mathcal{S} = \left\{ \left( \bigwedge_{i,j} C_{i.p_j} \rightarrow \bigvee_{i,j} C_{i.p_j} \right) \text{ with } \forall i, C_i \in \mathcal{C}, C_{i.p_j} \in \mathcal{A}_{C_i} \right\}$$

Remark that for an  $i$  fixed,  $(\bigwedge_j C_{i.p_j})$  corresponds to the description of an internal state of  $C_i$ .

#### Converter

A converter  $V$  associates a signature  $s$  to an instruction  $i$ . It abstracts the concept of class constructor and destructor; it establishes a dependency link between two classes.

$$\begin{cases} \mathcal{V} \subset (\mathcal{S} \times \mathcal{I}) \\ v \in \mathcal{V}, v = (s, i) \text{ with } s \in \mathcal{S} \text{ and } i \in \mathcal{I} \end{cases}$$

### 2.2.5 Dependency theory

A dependency theory  $D$  is a pair of a set of classes and a set of converters defined on these classes:

$$\begin{cases} D = (\mathcal{C}, \mathcal{V}_D) \\ \{\top, \perp\} \subset \mathcal{C} \end{cases}$$

Remark that builtin classes, origin  $\top$  and garbage  $\perp$ , are in  $\mathcal{C}$ .

## 2.3 Resolution algorithm

### 2.3.1 Solving domain

In object oriented logic, a dependency problem consists in a query, abstracted by a tuple of desired states of objects, which must be solved on a database, consists in instantiated and default-constructible objects and the dependency theory. A complete object oriented theory is schematized in 2.1.

The database on which the query must be solved has two levels of abstraction. The highest consists of the converters and the classes (2.2). The second one consists of the methods that enable the properties of each objects.

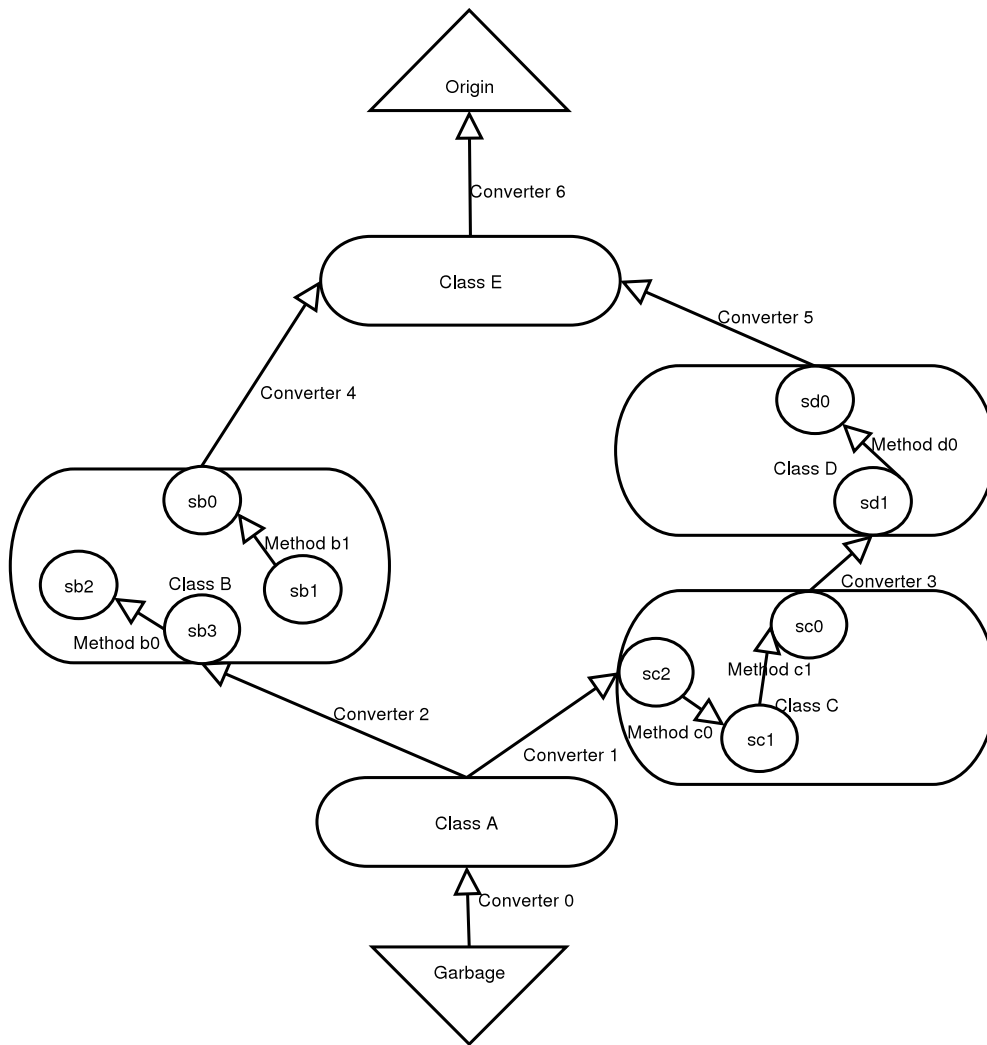


Figure 2.1: A simple but complete object oriented logic theory

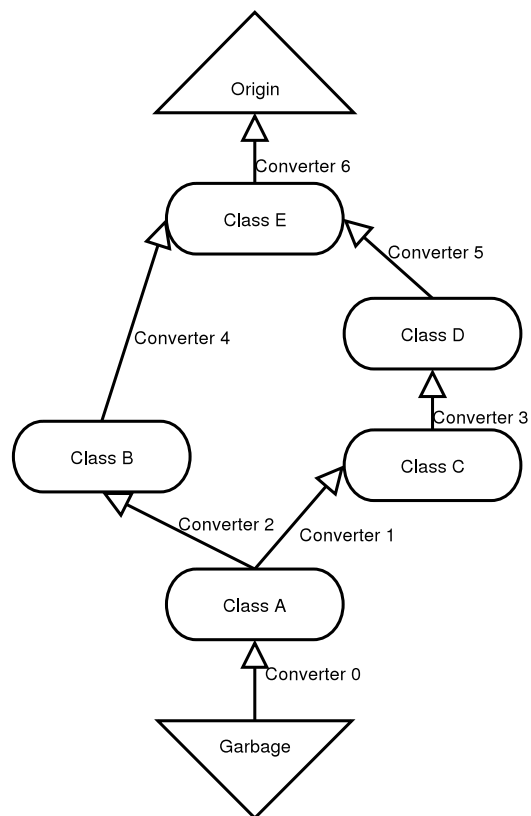


Figure 2.2: High level of an object oriented logic theory

### 2.3.2 Principle

In an object-oriented logic system, the resolution of a dependency request may be done in two steps:

1. a minimization of the search space is done by identifying the classes that are concerned with the request (inter-class reasoning)
2. a whole cut elimination algorithm is performed in the restricted class set obtained (intra-class reasoning).

#### Inter-class reasoning

The first step works at the higher level of abstraction: it just considers the classes and the converters, forgetting the required internal states of each objects.

This limitation leads to a simpler dependency graph of sub-theories. The idea is then to build a set of paths of converters from the requested object to instantiated objects and/or to the origin  $\top$ .

This set of paths is reduced to a single element when there is no ambiguity in the resolution, *i.e.* there is a unique chain of converters to build the wanted object. In general case, it must be considered as bigger.

#### Intra-class reasoning

The second step deals with internal states of classes. Its purpose is to build a chain of methods within classes to realize a path of converters. The underlying interest is the disambiguation of the paths: when a chain between two required states of a class cannot be build, the path is invalidated.

A converter defines an entry point in a new built class and an output point of a used class; this way, a class in a path comes with two converters which define the initial state it will have and the final state it must reach. A cut elimination algorithm may be run using these data:

- the initial state and the methods within the considered class represent the theory on which...
- ... the final state, the conclusion of the theorem, must be proven.

### 2.3.3 Considerations

Some immediate considerations must be emphasized:

- the speed of the first step's core is dependent of the amount of classes: the more there are vertices, bigger will be the graph;
- the speed of the second step's core depends on the complexity of classes' internal: the more classes have attributes and methods, slower will be the cut elimination;
- the efficiency of the whole is dependent on the amount of connections between the classes.

For good efficiency, an object oriented logical theory must come with a balance between the amount of classes, the methods within, and their interconnections – exactly as in software.

# Chapter 3

## Proposed syntax

A more convenient syntax to manipulate object oriented logic is here proposed; it is based on [Kifer et al. \(1995\)](#).

### 3.1 Alphabet

The alphabet of our object-oriented logic language consists of:

- a set of common alphabetical words (identifiers)
- usual logical connectives: “&” (logical and), “|” (logical or), “->” (logical implication)
- is a (derivation) operators: “::” for class derivation, “:” for instantiation
- application operator: “@”
- owning symbol: “[” and “]”
- implementation tags: “{” and “}”

### 3.2 Formulas

#### 3.2.1 Classes

A class is entirely defined by the attributes it aggregates and the methods it implements. No dedicated statement is required to declare a classe: it suffices to write all its attributes and methods.

##### Attributes

To write that a class *A* has an attribute *p*:

A[ p ]

**Methods**

To write that a class  $A$  provides a method  $m$  of signature  $p \wedge q \rightarrow r \vee s$  and instruction  $i$ :

```
A[ m @ p & q -> r | s ]
{
  i
}
```

Remark that the name of the method may be forgotten:

```
A[ p & q -> r | s ]
{
  i
}
```

is also a valid statement.

**Inheritance**

To write that a class  $B$  is a subclass of  $A$ :

```
B :: A
```

**Instantiation**

To write that  $a$  is an instance of a class  $A$ :

```
a : A
```

**3.2.2 Converters****Construction**

To write that a class  $D$  may be build with an initial state of  $v \vee w$  from a class  $A$  with a required state  $p \vee q$ , using a converter  $c$  implemented by  $i$ :

```
c @ A[ p & q ] -> D[ v | w ]
{
  i
}
```

Remark that a constructor may be anonym:

```
A[ p & q ] -> D[ v | w ]
{
  i
}
```

A more complex example, using more classes, would be:

```
c @ A[ p & q ] & B[ r & s ] -> C[ t | u ] | D[ v | w ]
{
  i
}
```

**Default constructor**

Default constructor, *i.e.* converters from origin may be written:

```
d @ -> A[ p | q ]  
{  
  i  
}
```

That is the default constructor  $d$  of  $A$  with an initial state of  $p \vee q$

**Destructor**

Destructor, *i.e.* converters to bottom may be written:

```
d @ A[ p & q ] ->  
{  
  i  
}
```

That is the destructor  $d$  of  $A$  with an required state of  $p \vee q$ .

## Chapter 4

# Applications

### 4.1 Describing LRDE's Tiger compiler tasks

#### 4.1.1 Default constructor: building a token stream

```
-> TokenStream
{
  echo "open_file"
}

TokenStream[scan_trace @
             -> scan_trace_p]
{
  echo "scan_trace"
}

TokenStream[parse_trace @
            -> parse_trace_p]
{
  echo "parse_trace"
}
```

1. Default constructor: from the origin, build a TokenStream, with no attributes set.
2. TokenStream methods: scan-trace and parse-trace may be used to modify the internal state of a TokenStream

#### 4.1.2 Converters: building an AST and displaying it.

```
parse @
      TokStream -> AST
{
  echo "parse"
}
```

```
display @
    AST -> StdOut
{
    echo "ast_display"
}
```

1. Convert a TokenStream with no special internal states into an AST.
2. Convert an AST to StdOut.

## 4.2 Describing “make” rules

### 4.2.1 Factored rules

```
## Source rules.
Source [c -> obj]
{
    echo "Compiling_c_into_object"
}

main, sub : Source[c]
```

- A class “Source” defined by the attributes “c” and “obj” and method that allows to activate “obj” when “c” is given
- Two objects “main” and “sub” which attribute “c” is enabled.

### 4.2.2 PHONY rules

```
phony @ main.c & sub.c
{
    echo "Dummy_dependence"
}
```

### 4.2.3 Complex example: libraries

```
Library [ objs -> lib]
{
    echo "Build_library"
}

main, sub: Source [c]
lib : Library [objs = (main.obj & sub.obj)]
```

Here some syntactic sugar is used to express the dependency of “lib” on a state of “main” and “sub”.

## Chapter 5

# Conclusion

This report presented a summary of common dependency expression schemas: from an easy C++ implementation to a Prolog program and through the “make” utility, different syntaxes have been described. By their diversity, they permit to extract critical aspects of a dependency system: flexibility, expression power, emphasized informations, etc.

These observations led to an object oriented approach of logic, which was defined as a structural sugar of usual first order logic. It was built to easily express dependency factorization by a logic class, common dependency by methods within classes or converters between them. The principle of a solving algorithm have finally been presented.

To complete this object oriented logic, a simple syntax was proposed, and some applications were detailed.

### Future works

Although the tools that have been presented here were oriented toward the applied aspect, they stands mainly technical. The natural following step would be to propose an effective implementation, to check the validity of the model and eventually to define some extension, like class recursive aggregation or temporal notions.

# Bibliography

Amir, E. (2001). Object-oriented first-order logic.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2002). *Introduction to Algorithms*.

Kifer, M., Lausen, G., and Wu, J. (1995). Logical foundations of object-oriented and frame-based languages.