Extending CHR with objects under a variety of inheritance and world-closure assumptions

Marcos Aurélio A. da Silva[1], Jacques Robin[1]

Centro de Informática – Universidade Federal de Pernambuco
Recife – PE – Brazil {maurelio1234, robin.jacques}@gmail.com

Abstract. In this paper, we present CHORD (Constraint Handling Object-oriented Rules with Disjunction) an Object-Oriented (OO) extension of CHRD. Syntactically, CHORD integrates two versatile dual programming and knowledge representation languages: Flora, a hybrid OO rule-based language and CHRD. A CHORD program extends CHRD by allowing Flora-style object constraints (o-constraints) in the head, guard or body of its rules. Orthogonal to its rules, a CHORD program also contains semantic assumption directives, an innovative construct that maximizes semantic versatility. Each directive defines a point along one dimension of the space of semantic assumptions made by various OO and rule-based languages in order to complete the knowledge explicitly specified in them by complementary knowledge left implicit by the programmer. Among others, these assumptions specify what kind of world closure and inheritance is desired.

1 Introduction

In this paper, we present CHORD (Constraint Handling Object-oriented Rules with Disjunction) the first bona-fide Object-Oriented (OO) rule-based constraint programming language. CHORD integrates most advanced features of the OO paradigm with those of the rule-based and constraint based paradigms. It thus aims at bringing together within a single language the well-known benefits of objects for programming-in-the-large within a systematic software engineering process, with the unique facilities of rules and constraints for programming-in-the-small intelligent applications that require advanced embedded automated reasoning services. Fast prototyping such applications requires a language that supports not only Turing-complete programming but also formally well-founded declarative Knowledge Representation (KR).

Syntactically, CHORD integrates the cores of two exceptionally versatile dual programming-KR languages: Flora [14] a hybrid OO rule-based language and Constraint Handling Rules with Disjunctions (CHRD) [1], a hybrid rule and constraint based language. A CHORD program extends CHRD by allowing Flora-style object constraints (o-constraints) in the head, guard or body of its rules, Logically, these o-constraints are conjunctions of atomic constraints, each one defining or referencing one structural or behavioral feature of a class or object.

Orthogonal to its rules, a CHORD program also contains *semantic assumption directives*. This innovative construct maximizes the semantic versatility. Each directive defines a point along one dimension of the space of semantic assumptions made by various OO and rule-based languages in order to complete the knowledge explicitly specified in them by complementary knowledge left implicit by the programmer. The combination of leaving semantics largely orthogonal to syntax, with the integration of three powerful declarative paradigms, allows CHORD to elegantly encode and unify programs and knowledge bases from a wide variety of programming and KR languages based on objects and/or constraints and/or rules.

Compared to imperative OO languages such as Java, C#, C++ or Python, CHORD presents the advantage of being a dual programming and KR language that provide as built-in, within its execution platform, a powerful, general purpose rule-based constraint solving inference service. This allows fast prototyping intelligent systems. It also presents the advantage of possessing a simple formal declarative semantics in Classical First-Order Logic (CFOL), which allows to directly use consolidated theorem proving technology to verify properties of a CHORD implementation. Compared to Flora and other OO rule-based languages such as JESS [5] and ILOG Rules[1], CHORD presents the advantage of possessing a configurable semantics that can support various inheritance strategies together with not only the fully closed-world assumption, but also the selectively, partially closed-world assumption or the fully open-world assumption [11].

The rest of the paper is organized as follows. In Section 2, we overview CHRD, which rules, relational constraints, declarative CFOL semantics and operational semantics are largely reused by CHORD. In Section 3, we show how CHORD's syntax extend CHRD's syntax with Flora-style objects. In Section 4, we describe the semantic assumption space covered by CHORD together with the directives used in a CHORD program to choose a point in this space that indicate to the CHORD engine which semantic interpretation to choose for the OO rules of the program. In Section 5, we discuss the current CHORD engine implementation which translates a CHORD program into a semantically equivalent CHRD program which can in turn be executed on any CHRD platform such as SWI-Prolog. In Section 6, we compare CHORD with closely related language families, namely (a) dual programming and KR OO rule-based languages with a formal semantics and (b) programming languages that pioneered the integration of objects with rules and constraints. In Section 7, we conclude by highlighting the contributions of our research and discuss its possible future directions.

2 Introduction to CHRD

A CHRD rule base is composed by a set of CHRD rules. The following CHRD rule defines the `append(X,Y,Z)` constraint:

```
r1 @ append(X,Y,Z) <=>
```

[1] ILOG Business Rules Management Systems. http://www.ilog.com/products/businessrules/index.cfm

```
(X = [], Z = Y);(X = [H|L1], Z = [H|L2], append(L1,Y,L2)).
```

In this rule, `Z` is a list composed by the elements of the list `X` followed by the elements of the list `Y`. If `append(X,Y,Z)` holds, we have two options: (i) `X=[]` and, therefore, `Z=Y`; or (ii) `X` is a list in the form `[H|L1]`, and thus, `Z` is composed by `H` followed by `L1` and then followed by the elements in `Y`.

There are three kinds of rules in CHRD: simplification, propagation and simpagation. The simpagation rules are the most general category of rules and they have the following form $r@H_k \backslash H_r \Leftrightarrow G|B.$, where `r` is an identifier for the rule, H_r and H_k are the heads of the rule, G is the guard and B is the body. If the guard is `true`, it can be omitted. The operational semantics of such rule is that if H_k and H_r are found in the constraint store and the guard G is entailed by it, the constraints in H_r should be removed and the constraints in the body B should be added to the constraint store. If H_k is empty, this rule is called a *Simplification Rule*, and this part of the rule is omitted. On the other side, if H_r is empty, this rule is called a *Propagation Rule*. In this case, the second part of the head of the rule is omitted and the $\Leftrightarrow$ is replaced by the symbol $\Rightarrow$.

The declarative semantics of a rule base is the conjunction of the declarative semantics of each rule. For each kind of rule, its abstract syntax and its declarative and operational semantics are described in the following table:

Rule	Abstract Syntax	Declarative Semantics	Operational Semantics
Simplification	$r@H \Leftrightarrow G\|B$	$\forall \bar{x}(G \rightarrow (H \leftrightarrow \exists \bar{y} B))$	$\langle H \rangle \mapsto_G \langle B \rangle$
Propagation	$r@H \Rightarrow G\|B$	$\forall \bar{x}(G \rightarrow (H \rightarrow \exists \bar{y} B))$	$\langle H \rangle \mapsto_G \langle H, B \rangle$
Simpagation	$r@H_k \backslash H_R \Leftrightarrow G\|B$	$\forall \bar{x}(G \rightarrow (H_k \wedge H_R \leftrightarrow \exists \bar{y} H_1 \wedge B))$	$\langle H_k, H_R \rangle \mapsto_G \langle H_k, B \rangle$

In this table $\bar{y}$ denotes the set of variables that appear in the rule body B but do not appear anywhere else in the rule and $\bar{x}$ the set of the remaining variables.

Example 1. Let us suppose that the current state of the constraint store is $append([1],[2],Z)$. The actual execution of this rule base is represented by the following set of transitions:

$$\begin{aligned}
&\langle append([1],[2],Z) \rangle \mapsto_{r_1} \\
\langle [1] = [], Z = [2] \rangle \,|\, &\langle [1] = [1|[]], Z = [1|L2], append([],[2],L2) \rangle \mapsto_* \\
&\langle [1] = [1|[]], Z = [1|L2], append([],[2],L2) \rangle \mapsto_{r_1} \\
&\langle [1] = [1|[]], Z = [1|L2], [] = [], L2 = [2] \rangle \\
|\, \big\langle [] = [H|L1], L2 = [H|L2'], append(L1,[2],L2') \big\rangle \mapsto_* \\
&\langle [1] = [1|[]], Z = [1|L2], [] = [], L2 = [2] \rangle
\end{aligned}$$

The notation $\langle G_0 \rangle \,|\, \ldots \,|\, \langle G_n \rangle$ represents the alternative execution states. The transition $\mapsto_{r_1}$ represents the application of r_1 and the transition $\mapsto_*$ the removal of failed states. In the initial state we add the constraint $append([1],[2],Z)$ to

the initial constraint store, the rule r_1 is applied and then we get two alternative execution states, notice that the first one is failed because of the constraint $[1] = []$. The next step removes this failed execution state. The next step applies the rule r_1 to the constraint $append([], [2], L2)$ and we obtain again two alternative execution states. This time, the second one is failed because of the constraint $[] = [H|L1]$. The next step removes this failed state and we get to the final state, where no other rule is applicable. From this final state we can thus conclude $Z = [1, 2]$.

3 The Syntax of CHORD

CHORD extends CHRD with a set of predefined *object-oriented* constraints (the so-called *o-constraints*) that appear in the rule heads and in the rule bodies. There are essentially 4 kinds of o-constraints:

- $o : c$, whose meaning is: the object o is an instance of the class c.
- $s :: g$, whose meaning is: the class s is a subclass of the class g.
- $o[a = v]$, whose meaning is: the value of the attribute a of the object o is v.
- $c[a *= v]$, whose meaning is: the value of the attribute a to be inherited by instances of the class c is v.

This extension is called *Core CHORD*. On top of it we build *Full CHORD* which extends first one with syntactic sugar that is translated into *Core CHORD* by the compiler. The first set of syntactic sugar is the *o-molecules* that are sets of atomic o-constraints condensed into only one constraint like `o:c[a=t,b*=u]` that is represented in *Core CHORD* by the conjunction `o:c, o[a=t], o[b*=u]` Another sort of syntactic sugar is the path expressions. They appear as values inside constraints in a CHORD program like in `p(a.b.c.d)` and they can be translated into conjunctions of constraints such as `p(D), a[b=B], B[c=C], C[d=D]`.

A CHORD program is a set of rules annotated with at most *one* semantic directive (for the whole program), that defines the semantics of its o-constraints. This directive has the following syntax:

```
semantics [
        s1 [ p11, ..., p1n ],
        ...,
        sm [ pm1, ..., pmn ]
  ].
```

Notice that semantic assumptions may optionally have parameters. Let us take the following directive as an example:

```
semantics [
       simpleInheritance,
       classes [a, b, c]
].
```

It specifies that in the annotated CHORD rule base: (i) only *simple inheritance* is allowed, i.e., every class must have at most one direct superclass; and (ii) there are only three classes: a, b and c and no other classes.

4 The Semantic Assumptions Taxonomy

In this Section, we propose a general ontology for the implicit semantics assumptions of knowledge representation and programming languages that include constructs for classes and objects, optionally integrated with constructs for constraints and rules. We present it completely in an intuitive manner as UML[9] diagrams and natural language definitions.

To illustrate our ontology, we consider the knowledge base shown in Fig. 1 as UML class and object diagrams annotated with OCL[10] constraints. In this example we define the class `GroupMember` with two attributes: `pacifist`, a boolean and `color` a string. It has two subclasses: `ReligionMember` and `PartyMember` which redefine the attributes defined in the superclass.

The `ReligionMember` class redefines `pacifist` as being derived from the `pacifist` attribute of the `Religion` of the member. The `PartyMember` redefines `pacifist` as being derived from the `pacifist` attribute of the `Party` of the member. They both redefine `color` and set it to `'red'` by default. The class `President` is a subclass of both `ReligionMember` and `PartyMember` and redefines its attributes. It defines two OCL invariants: (i) the `pacifist` attribute should be equal to `party.pacifist` and (ii) the `oppositeParty` should not be the same as the `party`. It also defines the default value of `color` to `blue`.

We define four instances in our model. `nixon` is a `President` with `religion` set to `quaker` (which is pacifist) and `party` set to `republican` (which is not pacifist).It also sets the value of the attribute `impeached` as `true`. We also have the instance `john` of the class `Vice`. Note that we slightly abuse the UML/OCL notation to provide a visual rendering of an example that covers constructs covered that are not acceptable in UML/OCL. In particular, the definition of the `impeached` feature in the nixon object instance of a class where such feature is not defined is not allowed in UML/OCL. Similarly for the definition of the `john` as an instance of a non-defined class `Vice`.

In Fig. 2, we display the overview of our space of semantic assumptions. We divide assumptions space into two orthogonal dimensions: $WorldAssumption$ and $InheritanceAssumption$.

By $WorldAssumption$, we mean the distinction between closed and open world [11]. More specifically, we focus on the *closure* of the set of classes, instances and features of a language. In a language where $closedClasses = true$ the `john` instance would be valid only if the `Vice` class was declared. In a language where $closedInstances = true$, the OCL invariant on the attribute `oppositeParty` of the class `President` would not be valid, since there is only one declared instance of `Party`. In a language with $closedInstanceFeatures = true$, the slot `impeached` of the instance `nixon` would be forbidden, since it is not declared in the class diagram. In a language with $closedClassFeatures = false$,

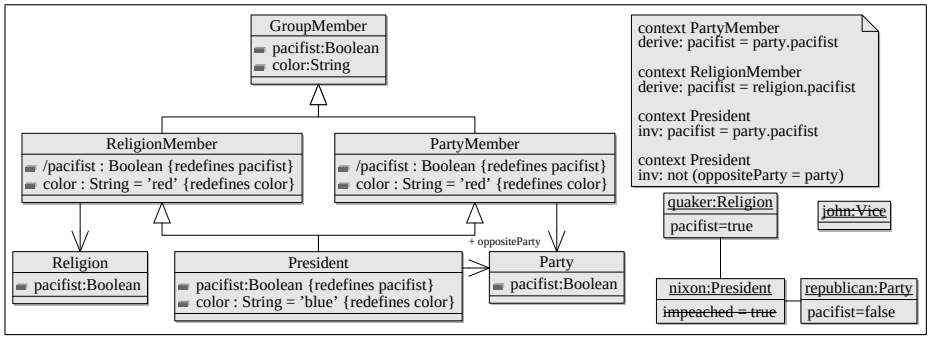

Fig. 1. UML Example

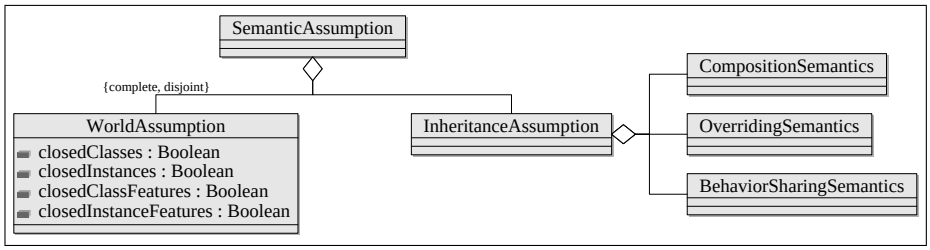

Fig. 2. Semantic Assumptions: Overview

an inference engine would use the slot `impeached` in `nixon` to deduce that some of its superclasses defines this attribute, even though it is not made explicit in the diagram.

The other dimension we explore in our ontology is the *InheritanceAssumption*, that we have divided into three sub-dimensions: *CompositionSemantics*, *OverridingSemantics* and *BehaviorSharingSemantics*, which are detailed in Fig. 3.

The *CompositionSemantics* (on the left side in Fig. 3) defines the way classes can be connected by the means of the subclass relationship. It is partitioned into two kinds of *CompositionSemantics*: the *SimpleInheritance* and the $MultipleInheritance$. as indicated by the UML constraints {*disjoint, complete*}). In the first case, we allow each class to be a direct subclass of only one other class. In the second one, there is no such restriction. In the example in Fig. 1, the class `President` would be invalid if we have a language with $SinpleInheritance$.

In case of $MultipleInheritance$, we need to specify a strategy to resolve potential conflicts, which may arise when the same feature is defined in several superclasses with incompatible signatures or values. In our ontology, we divide $ConflictResolutionStrategy$ into four disjoint subclasses: the $ValueBased$, the $SourceBased$, the $PriorityBased$ and the $ExplicitDefinitionBased$. These

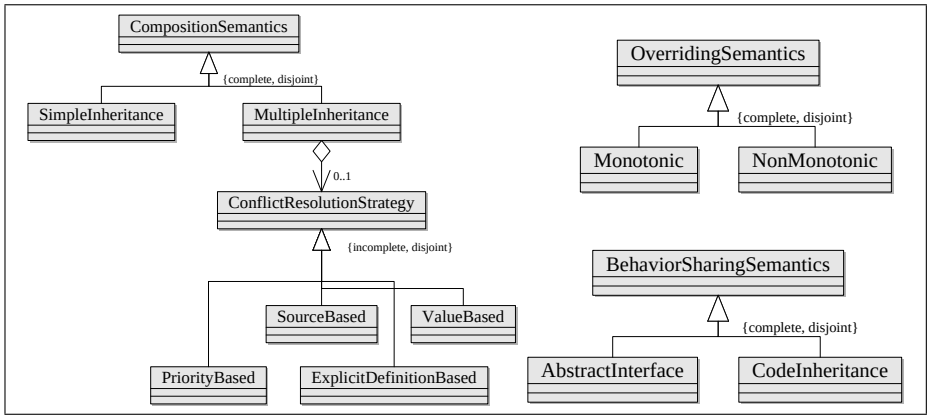

Fig. 3. Inheritance Assumptions: Composition, Overriding and Sharing Semantics

subclasses form are incomplete division in the sense that other strategies, not covered here, may be used by other languages.

In the example in Fig. 1, in the class `President` the features `color` and `pacifist` conflict. In the $ValueBased$, if the multiple feature definitions provide the same *value* the inheritance is allowed. In our example, the `President` class would inherit $color = red$ if this feature were not redefined in this class. In the $SourceBased$, the inheritance is always denied if there is a conflict.

In the $PriorityBased$ the source is chosen by the means of a priority between the conflicting superclasses. In our example, the `pacifist` from the class `ReligionMember` could be preferred to the one of the class `PartyMember`. In the $ExplicitDefinitionBased$, the user is responsible for designing a model without conflicts. In our example, UML forces the class `President` to override the conflicting attributes of its superclasses.

The $OverridingSemantics$ (on the center in Fig. 3) defines whether the *overriding* is allowed ($NonMonotonic$) or not ($Monotonic$). In languages with classical FOL semantics, *overriding* is usually not allowed. In our example, the features containing the keyword `redefines` are, in fact, overriding the feature of same name in the superclass.

The $BehaviorSharingSemantics$ (on the right side in Fig. 3) defines how features are shared with their subclasses. In the $AbstractInterfaceInheritance$, only the feature signature is inherited, whereas in the $CodeInheritance$, the feature signature and code are both inherited by the subclasses.

In the next table we map every class in our ontology into its concrete syntax in the semantic directive to be used in a CHORD program. The assumptions marked with a "–" are used by default.

Assumption		Syntax
Composition Semantics	SimpleInheritance	`simpleInheritance`
	MultipleInheritance	–

	ValueBased
	SourceBased
	PriorityBased
	ExplicitDefinitionBased
		`valueBasedConflictResolution`
		`sourceBasedConflictResolution`
		`priorityBasedConflictResolution`
		`explicitConflictResolution`

Overriding Semantics	Monotonic	`noOverriding`
	NonMonotonic	–
BehaviorSharing Semantics	AbstractInterface	–
	CodeInheritance	`code[(c,[f*])*]`

WorldAssumption		`classes [c*]`
		`classesHierarchy [(s,g)*]`
		`objects [o*]`
		`classFeatures [(c,[f*])*]`
		`objectFeatures [(o,[f*])*]`
		`localValues [(o,[f*])*]`
		`inheritables [(c,[f*])*]`

Some assumptions require a list of parameters to be provided. The *CodeInheritance* requires the user to define the set of features to which the code inheritance should be enabled. The list of possibilities in the *WorldAssumption* allow the user to selectively close the world for part of its model. The *classes* directive closed the set of classes, the *classesHierarchy* directive closes the graph formed by the hierarchy of classes in the model, the *objects* directive closes the set of objects in the model, the *classFeatures* and the *objectFeatures* directives close the set of features (operations and attributes) in an object or in a class, the *localValues* directive closes the set of features that are defined locally in the model (i.e. that should not be defined by inheritance) and the *inheritables* directive closes the set of features that are marked as inheritable[2] .

5 The CHORD Inference Engine

In this Section we analyze the implementation of an inference engine for CHORD on top of the SWI Prolog 5.6.61 and that is available in `http://chord.sf.net/`. It consists of approximately 1500 lines of Prolog and CHRD code, including the code for the trailing rules. The inference engine is composed by a Runtime and a Compiler. The function of the Runtime is to extend the SWI Prolog default console runtime with special commands to interact with the CHORD inference

[2] At first sight the assumption directives for partially closing the world may seem to verbosely declare facts that could be automatically be infer from the rules in the program, but remember that in some programs part of the model may exist that are not referred in the program rules.

engine, i. e., for compiling CHORD programs, running the engine with a provided goal, providing help and etc.

The function of the Compiler is to read CHORD rule bases and generate the equivalent CHRD rule base for them, this final rule base is obtained by:

1. Translating *Full CHORD* into *Core CHORD*.
2. Interpreting every o-constraint as a user-defined constraint, e.g., the constraint $a : b$ is interpreted as the binary constraint $: (a, b)$.
3. Translating each assumption in the semantic directive into a CHRD rule base (notice that parameterless assumptions are translated by adding a fixed set of rules to the final rule base, but the other ones will generate a different set of rule depending on the parameters). This set of rules is called *Trailer Rules*.
4. Appending the set of Core Rules.

The Core Rules specify the semantics of the o-constraints in a way that is not dependent on the semantic assumptions. For example, the following rules are part of the core rules:

```
% 1. taxonomy transitivity
A::B, B::C ==> A::C.
A:B,  B::C ==> A:C.

% 2. monotonic feature inheritance
O:C, C[F*= V] ==> O[F=W].
```

The first group of rules defines the transitivity of the $:: (\cdot, \cdot)$ and of the $: (\cdot, \cdot)$ relationships and the second one defines that if O is an object of a class C that defines an inheritable value V for the feature F, then there exists a value W for this feature in O. The value of such feature depends on the semantic assumptions, e.g. this inheritable value may be *overridden* by a more specific superclass of O.

To illustrate the Trailer Rules, we present the translation of the `simpleInheritance` assumption used in the previous example:

```
X::Y, X::Z ==> (Y::Z ∨ Z::Y).
X:Y,  X:Z  ==> (Y::Z ∨ Z::Y).
```

These rules state that if a class or an object has two superclasses these superclasses should be related, i.e., one of them should be more specific than the other, avoiding multiple inheritance.

Example 2. Let us consider the following CHORD program:

```
semantics [ ].
facts <=> nixon : republican.
```

In this program we define the object `nixon` and the class `republican`. We say that `nixon` is an instance of `republican`. If we run this program with the `facts, nixon : party` initial constraint store, we are going to get only one final store, with the following state:

```
nixon : republican, nixon : party
```

If we change the `simpleInheritance` semantics directive and run this example with the same initial constraint store we are going to get three final states:

```
S1: nixon : republican, republican :: party
S2: nixon : party, party :: republican
S3: nixon : party, republican :: party, party :: republican
```

Notice that assuming the simple inheritance changed our final state. Since we added two classes to `nixon` in the constraint store, there are only three ways to restore its consistency:

1. Assume that `party` is a super class of `republican`
2. Assume that `party` is a sub class of `republican`
3. Assume that `party` and `republican` are the same class by stating that one is a subclass of the other

6 Related Work

6.1 OO Rule based Languages

F-logic. F-logic [14] is an extension of Prolog with object-oriented predicates on top of Well-Founded Models [6] with Negation-As-Failure (NAF)[7] to represent the default reasoning involved in inheritance. In terms of our Semantic Assumptions Ontology, F-logic employs the Closed World Assumption with Multiple Inheritance using Source Based Conflict Resolution Strategy, Overriding and Code Inheritance.

A rule has a head and a body, with the following concrete syntax: "`head :- body.`". A rule with a "`true`" body can be written as "`head.`" and it is called a *fact*. The body of a rule may be represented by a conjunction of formulas with `f0,...,fn` as concrete syntax. F-logic extends the set of terms with the so-called F-Atoms and the F-Molecules. The first group contains terms that express *atomic* informations about the object model, e.g, `O:C` means `O` is an instance of `C`, `C::S` means `C` is a subclass of `S`, `O[F->V]` means `V` is the value of the feature `F` in the object `O` and `C[F*->V]` means `V` is the value of the feature `F` to be inherited by instances of the class `C`.

The following example illustrates how we can represent F-logic Knowledge Bases in CHORD.

Example 3. Let us consider the following F-logic program:

```
nixon:quaker.
nixon:republican.
quaker[policy->pacifist].
republican[policy->hawk].
```

It defines two classes for the `nixon` object: `quaker` and `republican` which define different inheritable values for the `policy` feature: `quaker` defines its `policy` to `pacifist` and `republican` defines its `policy` to `hawk`. In this case, `nixon` should inherit neither `pacifist` nor `hawk` for `policy`, since there is a conflict between its subclasses.

The CHORD rule base that implements this F-logic program follows:

```
% semantic assumptions:
%         closed world,
%         multiple source based inheritance
semantics [
        overriding,
        sourceBasedMultipleInheritance,
        objects [nixon],
        classes [quaker, republican],
        classesHierarchy [ ],
        localValues [ (nixon, []) ]
].

facts <=> nixon : quaker, republican[policy *= hawk],
          nixon : republican, quaker[policy *= pacifist].
```

The final constraint store for an initial constraint store containing only the `facts` constraint is:

```
quaker[policy*=pacifist], nixon[policy=_]
republican[policy*=hawk],
nixon:republican
nixon:quaker
```

The value for the `policy` feature of `nixon` remains undefined agreeing with the expected F-logic semantics of this rule base: the inheritable values defined by its superclasses for this feature conflict and therefore nixon should not inherit any value for it.

The equivalence of the translated CHORD rule base is given in terms of the three-valued object model represented by the F-logic rule base: every *positive* fact in the F-logic model, should be entailed by the FOL semantics of the translated CHRD rule base and every *negative* fact should turn it into an inconsistent base. For example, the initial constraint store `facts, nixon:X, X::republican` (i.e., exists `X` such that `nixon` is an instance of `X` and `X` is a subclass of `republican`) has no consistent final store, because according to the closed world assumption held by the rule base no such `X` class can exist.

Implementing F-logic's inheritance on top of CHRD is not so innovative as Kaeser & Meister have already tried to do it in [8]. In a few words, they implemented an inference engine for a subset of F-logic on top of CHRD forward propagation in order to compute the intended object model. When compared

to our work, their work just shows the feasibility of the idea while failing in providing a correct account for one of the most complex features of the language: the interaction between inheritance and deduction. In the present work, we do not intend to provide an inference engine for F-logic, but we want to provide F-logic's object-oriented capabilities to CHRD programmers.

6.2 OO Constraint Programming Languages

LAURE. LAURE [2] was developed in 1988 at Bellcore Labs. It aims to be used in designing *complex applications* in which AI needs to be embedded into the software. Its semantics is based on a CFOL sublanguage with equality. It distinguishes between objects (whose equality is based on object identity) and values (whose equality is based on equality of feature values) and supports only binary relations.

In terms of our Semantic Assumptions Ontology, LAURE employs the Open World Assumption with Multiple Inheritance and no Code Inheritance. Since there are no inheritable vales there are no conflicts (and thus no conflict resolution Strategy) and no overriding. We are going to use the following example to demonstrate the use of OO concepts in LAURE and how to translate them into CHORD:

Example 4. Let us consider this piece of code that is part of a constraint solver for the problem of automatic graphical layout:

```
% declaration 1
[define RECTANGLE class superclass {GRAPHICAL_OBJECT}]

% declaration 2
[define Xul attribute domain RECTANGLE,
        -> {0 -- MaxX},
        comment "x-coordinate of the upper left corner"]

% declaration 3
[define Yul attribute domain RECTANGLE,
        -> {0 -- MaxY},
        comment "Y-coordinate of the upper left corner"]
...

% declaration 4
[define DISJOINT class superclass {object},
          with   slot( r1 -> RECTANGLE),
                 slot( r2 -> RECTANGLE)]

% declaration 5
[define disjoint_spec constraint
     for_all (D DISJOINT),
```

```
    if   xul1 = Xul(r1(D)), xdr1 = Xdr(r1(D)),
         yul1 = Yul(r1(D)), ydr1 = Ydr(r1(D),
         xul2 = Xul(r2(D)), xdr2 = Xdr(r2(D)),
         yul2 = Yul(r2(D)), ydr2 = Ydr(r2(D)),
        then      [or ydr2 > yul1, ydr1 > yul2,
                      xdr1 < xul2, xdr2 < xul1]]
```

This code is divided into four declarations. The first one defines the `RECTANGLE` class as a subclass of `GRAPHICAL_OBJECT`. The second one defines `Xul` (x-coordinate of the upper left corner) as an attribute of the class `RECTANGLE` with range going from `0` to `MaxX` and the third one defines `Yul` (x-coordinate of the upper left corner) as an attribute of the same class with range going from `0` to `MaxY` (the definition of the coordinates of the other corners was removed from this example. The fourth declaration defines the `DISJOINT` class that associates, by the means of its slots, two different rectangles. Finally, the last declaration provides the semantics for the `DISJOINT` class as a constraint defining that if two rectangles are disjoint their areas should not overlap.

This LAURE program can be translated into the following CHORD program:

```
% semantic assumption: open world without overriding
semantics [ noOverriding ].

% translated declaration 1
facts ==> rectangle :: graphicalObject.

% translated declaration 2
X:rectangle[xul=Xul] ==> Xul > 0, Xul <= maxX.

% translated declaration 3
X:rectangle[yul=Yul] ==> Yul > 0, Yul <= maxY.
...

% translated declaration 4
facts ==> disjoint::object.
X:disjoint[r1=R] ==> R:rectangle.
X:disjoint[r2=R] ==> R:rectangle.

% translated declaration 5
D:disjoint ==> (D.r2.ydr > D.r2.yul, D.r1.ydr > D.r2.yul) ;
               (D.r1.xdr < D.r2.xul, D.r2.xdr < D.r1.xul).
```

Oz. The Oz [12] language was developed in the early 1990s at DFKI with the objective of being a high-level concurrent constraint programming language to overcome the problems in concurrent object oriented languages at the time. It borrows ideas from logic programming (such as logic variables and data structures), and from concurrent programming (such as cells). When compared to

CHORD's model of inheritance, Oz's is very much restricted. It is based based on a very procedural approach with low level hard-to-learn concepts.

7 Conclusion

In this paper, we present CHORD (Constraint Handling Object-oriented Rules with Disjunction) the first bona-fide Object-Oriented (OO) rule-based constraint programming language. CHORD integrates most of the advanced features of the OO paradigm with those of the rule-based and constraint-based paradigms. It thus brings together within a single language the well-known benefits of objects for programming-in-the-large within a systematic software engineering process, with the unique facilities of rules and constraints for programming-in-the-small intelligent applications that require advanced embedded automated reasoning services.

The main innovative distinctive feature of CHORD over previous programming and KR languages that integrate objects with rules and/or constraints is that it allows the programmer to piecemeal choose, via header directives, which semantic assumption the CHORD engine should make about inheritance as well as the closeness of the universe of class names, class hierarchies, object names, class and object feature names and values. For a given CHORD OO rule base O and two different semantic assumption directives A1 and A2, the CHORD engine translates the CHORD program into distinct CHRD bases R1 and R2 before running them on the underlying CHRD engine that it relies on for constraint solving.

This makes CHORD uniquely versatile in terms of potential application fields. It can be used as a declarative programming language to implement constraint solvers in taxonomically rich domains. In can also be used as a knowledge representation language for semantic web ontologies, services and agents and expert systems in taxonomically rich domains. It also has potential as a model transformation language for model-driven engineering [13].

In future work, we intend to empirically evaluate the benefits of CHORD as compared to standard languages on both benchmark and real-world programs and knowledge bases in each of these application domains. We also intend to integrate CHORD with software components, one key programming-in-the-large feature that it still misses. This integration will feature additional constructs to encapsulate CHORD bases into components that can be assembled by connecting their provided and required services interfaces. One source of inspiration for this future integration is CHR^{at} which extends CHR with modules and rule-based deep guard entailment [4].

Remark. An extended version of this work can be found in [3], including complete source code of the Core and Trailer Rules and an in-depth description of the CHORD inference engine and of the case studies presented here (along with other ones).

References

1. Slim Abdennadher. A Language for Experimenting with Declarative Paradigms. *Second Workshop on Rule-Based Constraint Reasoning and Programming*, 2000.
2. Yves Caseau. Constraint Satisfaction with an Object-Oriented Knowledge Representation Language. *JAIPS*, 1993.
3. Marcos Aurélio Almeida da Silva. CHORD: Constraint Handling Object-oriented Rules with Disjunctions. Master's thesis, Universidade Federal de Pernambuco, February 2009. Universidade Federal de Pernambuco.
4. François Fages, Cleyton Mario de Oliveira Rodrigues, and Thrierry Martinez. Modular CHR with ask and tell. *Proceedings of the Fifth Workshop on Constraint Handling Rules (CHR 2008)*, 2008.
5. Ernert Friedman-Hill. *JESS in Action*. Manning Publications, 2003.
6. A. V. Gelder, K. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of ACM*, 38(3):620–650, 1998.
7. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.
8. Martin Kaeser and Marc Meister. Implementation of a F-logic Kernel in CHR. *Third Workshop on Constraint Handling Rules*, pages 33–48, 2006.
9. OMG. Unified Modeling Language (UML), Version 2.0. Technical report, Object Management Group Inc, May 2004.
10. OMG. OCL 2.0 Specification. Technical report, Object Management Group Inc, June 2005.
11. Raymond Reiter. *On Closed World Databases*. In Gallaire, H. and Minker, J., editors, Logic and Databases. Plenum Press, New York, 1978.
12. Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in oz. In Pascal Van Hentenryck Vijay Saraswat, editor, *Principles and Practice of Constraint Programming*, chapter 2, pages 29–48. The MIT Press, Cambridge, MA, May 1995.
13. Thomas Stahl and Markus Voelter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.
14. Guizhen Yang. *A Model Theory for Nonmonotonic Multiple Value and Code Inheritance in Object-Oriented Knowledge Bases*. PhD thesis, Stony Brook University, December 2002.