# Progress report on the ARC Project: Creating logical models of Gothic cathedrals[1]

**Charles Hollingsworth**
Institute for Artificial Intelligence

**Stefaan Van Liefferinge**
**Rebecca A. Smith**
Lamar Dodd School of Art

**Michael A. Covington**
**Walter D. Potter**
Institute for Artificial Intelligence

**The University of Georgia**
**Athens, GA 30602 U.S.A.**

**September 15, 2011**

[1]This research benefited from the generous support of a Digital Humanities Startup Level I Grant from the National Endowment for the Humanities (Grant Number HD5110110), a University of Georgia Research Foundation Grant, and support from the University of Georgia President's Venture Fund.

**Abstract**

This working paper reports the results so far of the ARC Project (Architecture Represented Computationally), whose goal is to reproduce in computer form the architectural historian's mental model of the Gothic cathedral. In addition to being major monuments of cultural heritage, Gothic cathedrals are particularly well suited for logical analysis because they have a definite logical structure and a definite set of component parts.

The ARC software system, still in its infancy, is the result of close collaboration between architectural historians and artificial intelligence researchers. It is designed to accept representations written in a subset of English and translate them into the logic programming language Prolog, then perform reasoning to draw conclusions about the structure of the cathedral being described.

**Résumé en français**

Ce rapport de recherche décrit les derniers résultats du projet ARC (Architecture Represented Computationally), un projet visant à reproduire par ordinateur le modèle visuel et mental produit par un historien d'art lorsqu'il décrit une cathédrale gothique dans un récit ou une analyse architecturale. La construction des cathédrales gothiques représente un fait majeur dans l'histoire de l'architecture et l'importance de ces bâtiments est un fait établi. De plus, la cathédrale gothique présente aussi un aspect qui en fait un sujet particulièrement bien adapté pour une analyse logique. Son architecture est en effet structurée suivant une méthode logique et ses composants architecturaux forment un ensemble clairement défini.

Le logiciel ARC, encore en phase initiale, est le fruit d'une collaboration étroite entre historiens de l'architecture et chercheurs en intelligence artificielle. Ce logiciel acceptera des descriptions utilisant un sous-ensemble de la langue anglaise afin de les traduire en langage de programmation Prolog pour ensuite raisonner et tirer des conclusions concernant la cathédrale en question.

# Contents

# List of Figures

# Chapter 1

# Overview of the ARC system

## 1.1 Introduction

The ARC project (for **A**rchitecture **R**epresented **C**omputationally) is designed to assist architectural historians and others with the task of gathering and using information from architectural descriptions.[1] The architectural historian is confronted with an overwhelming amount of information. Even if we restrict ourselves to Gothic architecture (our primary area of interest), any given building has probably been described dozens, if not hundreds, of times. These descriptions may have been written in different time periods, using different vocabularies, and may describe the same building during different stages of construction or renovation. Descriptions may be incomplete or even contradictory. An architectural historian should be able to extract necessary information about a building without encountering anything contradictory or unclear.

To facilitate information gathering, the ARC research group propose a logic-based knowledge representation for architectural descriptions. Descriptions of various cathedrals would then be translated into this representation. The resulting knowledge base would be used to give intelligent responses to queries, identify conflicts among various descriptions, and highlight relationships among features that a human reader might have missed.

## 1.2 Why Gothic?

In addition to being major monuments of cultural heritage, Gothic cathedrals are particularly well-suited for logical analysis. The structure of Gothic follows a logical form. Despite variations, Gothic cathedrals present a number of typical features, such as pointed arches, flying buttresses, and a plan on a Latin cross (Figure 1.1). The repetition of elements like columns and vaulting units allows for more succinct logical descriptions (Figure 1.2). And the historical importance of Gothic means that a wealth of detailed descriptions exist from which we can build our knowledge base.

[1] This chapter draws heavily on the author's previous work in [Hollingsworth2011].

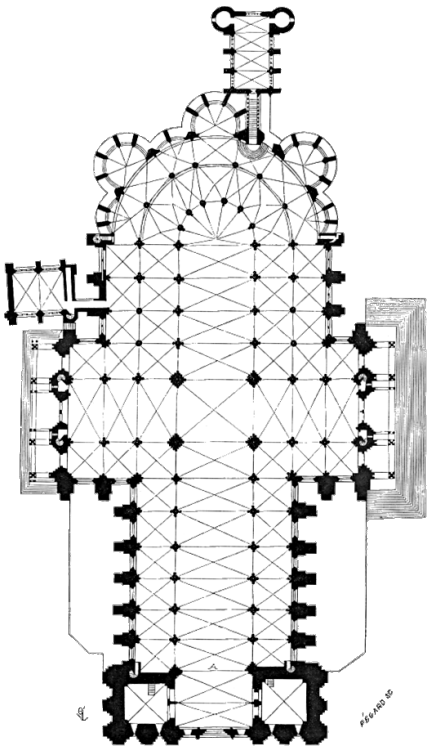

Figure 1.1: Example of a cathedral ground plan (Chartres, France), from [Viollet1854]

The study of Gothic cathedrals is also important for cultural preservation. Some cathedrals have been modified or renovated over the years, and their original forms exist only in descriptions. And tragedies such as the 1976 earthquake which destroyed the cathedral in Venzone underscore the importance of architectural information. A usable and versatile architectural knowledge base would greatly facilitate the task of restoring damaged buildings.

## 1.3 Outline of the ARC system

The outline of the ARC system is the result of close collaboration between architectural historians and artificial intelligence researchers. While the system is still in its infancy, the complete ARC system will have three distinct modes of interaction, to be used by three different types of user. We will refer to these modes as superuser mode, administrator mode, and user mode. The superuser mode will be used to write and edit a generic model for Gothic architecture that will serve as background information prior to dealing with any specific descriptions. The administrator mode will be used to enter the details of particular buildings. The purpose of the user mode will be to allow end users to submit queries to the knowledge base.

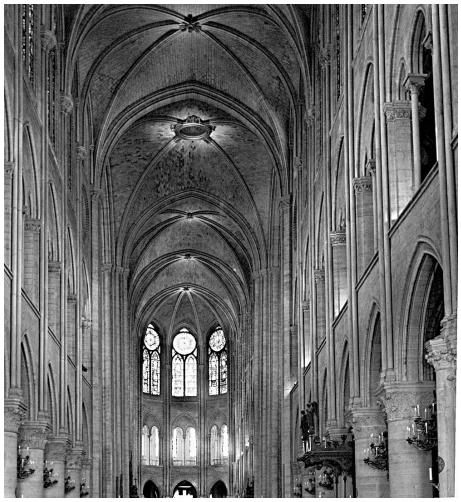

Figure 1.2: Nave of Notre Dame de Paris, showing the repetition of elements. (Photograph by S. Van Liefferinge)

### 1.3.1 Superuser mode

A small set of superusers will be able to create and edit the generic model of a Gothic cathedral. This will consist of information about features generally considered typical of Gothic (such as the cruciform ground plan and use of pointed arches) as well as more common-sense information (such as the fact that the ceiling is above the floor). These are facts that are unlikely to be explicitly stated in an architectural description because the reader is assumed to know them already. Individual descriptions need only describe how a particular building differs from this generic model. The generic model will be underdetermined, in that it will remain silent about features that vary considerably across buildings (such as the number of vaulting units in the nave).

The generic description will be written in ARC English, a small subset of English specifically designed for capturing information about the structure of Gothic cathedrals, and translated into the logical programming language Prolog. The goal is not a complete implementation of English semantics, but rather a form of natural language programming, in which the computer is able to extract its instructions from human language. Anyone reasonably familiar with architectural technology should be able to work on the generic description in ARC English without the steep learning curve of a programming language. An example of ARC English code is given in Figure 1.3.

### 1.3.2 Administrator mode

The administrator mode is used to input information about particular buildings, as opposed to Gothic cathedrals in general. When an administrator begins an interactive session, the generic model designed by the superuser is first read into the knowledge base. The administrator simply describes how the particular cathedral in question differs from the generic

A column is a type of support. Every column has a base, a shaft, and a capital. Most columns have a plinth. The base is above the plinth, the shaft is above the base, and the capital is above the shaft. Some columns have a necking. The necking is between the shaft and the capital.

Figure 1.3: Sample listing of ARC English.

model, using the same architectural description language. We would also like for the administrator mode to accept real-world cathedral descriptions in natural language rather than ADL. This is a nontrivial task, and complete understanding is likely a long way away. In the short term, the system should be able to scan a description, identify certain salient bits of information, and allow the administrator to fill in the gaps as needed. To illustrate the problem of understanding real-world descriptions, we present the following excerpt from a description of the Church of Saint-Maclou:

> The nave arcade piers, chapel opening piers, transept crossing piers, and choir hemicycle piers are all composed of combinations of five sizes of individual plinths, bases, and moldings that rise from complex socles designed around polygons defined by concave scoops and flat faces. All the piers, attached and freestanding on the north side of the church, are complemented by an identical pier on the opposite side. However, no two piers on the same side of the church are identical. [Neagley1998] p. 29.

There are important similarities between this description and our own architectural description language. We see many key entities identified (*nave arcade piers, chapel opening piers, etc.*), as well as words indicating relationships between them (*composed, identical,* etc.) Even if complete understanding is not currently feasible, we could still use techniques such as named entity extraction to add details to our model.

### 1.3.3 User mode

The user mode will consist of a simple query answering system. Users will input queries such as "How many vaulting units are in the nave at Saint-Denis?" or "Show me all cathedrals with a four-story elevation." The system will respond with the most specific answer possible, but no more, so that yes/no questions might be answered with "maybe," and quantitative questions with "between four and six", depending on the current state of the knowledge base. Unlike web search engines, which only attempt to match particular character strings, our system will have the advantage of understanding. Since descriptions are stored as a logical

knowledge base rather than a string of words, we can ensure that more relevant answers are given.

## 1.4 Outline of the thesis

Chapter 2 will give a review of previous work relevant to the ARC project, and demonstrate the unique contributions this project makes. Chapter 3 details the knowledge representation used in ARC: how architectural features are represented in the Prolog knowledge base, and how inferences are derived from them. Chapter 4 describes ARC English, the natural language programming system that uses a specialized subset of English to write architectural descriptions. Finally, chapter 5 gives some suggestions for further research.

# Chapter 2

# Related work

## 2.1 Logic and architecture

Perhaps the most important previous work in developing logical models for architecture was [Mitchell1990]. This work introduced the concept of buildings having a logical structure, with architectural forms following rules similar to the rules of grammar for natural languages. It also described how concepts such as types, properties, and inheritance could be applied to architectural description, and discussed the role of logical inference in architecture.

There have been a number of attempts at modeling architecture based on logical rules. Yong Liu and others [Liu2008, Liu2010] used rules based on L-System grammars to represent Chinese architecture. Wonka et al. [Wonka2003] designed a system for automatic modeling of architecture using split grammars. However, neither of these approaches used a semantics that was closely tied to natural language. Coyne and Sproat [Coyne2001] designed a system that produced graphical models of scenes based on natural language descriptions. This system was not specifically concerned with architecture, and its emphasis was on producing graphical rather than logical representations.

## 2.2 Knowledge representation and logic programming

The knowledge representation presented in chapter 3 builds on much previous work in using logic programming to capture English semantics. Of particular value was [BlackburnBos2005], which gave a detailed account of expressing Montague semantics in the Prolog logical programming language. This work aimed more at capturing the general semantics of English, however, and was not particularly suited for an architecture-specific domain.

The knowledge representation used in ARC was influenced by discourse representation theory, as presented in [KampReyle1993]. Several previous works on using DRT for knowledge representation were useful, including [CovingtonNute1988], [CovingtonSchmitz1988], and [Izzo1993].

Much use was also made of defeasible or nonmonotonic reasoning. A good overview of this topic from a formal logic standpoint is [Antoniou1997]. The specific implementation of

defeasible reasoning used, Donald Nute's d-Prolog, was described in [CovingtonNute1997] and [Nute2003].

## 2.3 Natural language programming

ARC English, the domain-specific subset of English presented in chapter 4, is an example of natural language programming. Hugo Liu and Henry Lieberman have been among the most prolific advocates of natural language programming, and several of their works were helpful. [LiebermanLiu2005] presented a case study in which non-programmers (small children, in fact) were asked to describe a simple video game in English. This work illustrated the usefulness of natural language programming in helping non-programmers instruct computers. In other works, these authors describe representing programs in a subset of natural language [LiuLieberman2004, LiuLieberman2005].

The task of using unrestricted natural language (as opposed to a specialized subset) for programming was described in [Vadas2005]. This work was particularly relevant because the authors adopted discourse representation theory for their semantics. They used the freely available C&C Parser to translate English sentences into discourse representation structures, and then translating these into code. I opted not to use this technique, however, because the general-purpose C&C Parser was unacceptably slow and did not always produce the best discourse representations. A simpler, handwritten parser was used instead.

One of the more successful and well-developed implementations of natural language programming to date was Graham Nelson's Inform 7, a system for writing interactive fiction (a genre of computer games also known as text adventures). These games take the form of second-person narratives in which a variety of locations, objects and characters are described to the player, who then issues instructions to the game in English. The motivations and outline of the Inform 7 system are presented in [Nelson2006]. This work was a great influence on ARC English, as one of the primary tasks of interactive fiction—describing physical objects and their properties—is also ARC's goal. Nelson's paper also helped inspire some of the guidelines for natural language programming found in section 4.2.

# Chapter 3

# Knowledge representation in the ARC system

## 3.1 Introduction

Knowledge representation is at the heart of the ARC system. One of the distinguishing features of ARC is that it produces building descriptions in the form of logical predicates, as opposed to 3D models or other formats more commonly used with architecture. It is therefore important that we determine exactly what information we wish to capture, and how it is to be represented.

While architectural knowledge representation is not a simple task, certain features of the domain make it less daunting than more general knowledge representation. Perhaps the biggest simplifying factor is that buildings are more or less static objects. Were we describing living agents, or even machinery, we would have to find ways to render action verbs, describe change over time, and so forth. Buildings, however, can be treated as "frozen" in a moment of time. While buildings do change over time, with features being added or removed, the change is gradual enough that we need not describe it as a temporal process. Instead, we can treat various time-slices of a cathedral as complete and separate buildings—say, Chartres Cathedral in 1260 versus Chartres Cathedral in 1325.

Without the need to describe change over time, our task is reduced to that of naming the entities that make up a cathedral, and describing their properties and how they relate to each other. Nevertheless, there are a number of issues that must be resolved before we can produce a logical representation of a cathedral. The following sections address a few of these issues.

## 3.2 Referencing entities

Before we can go about describing objects, we must first be able to refer to them. In natural languages such as English, there are two main ways we can refer to something: by a proper name (e.g., "Chartres Cathedral"), or by a common noun phrase (e.g. "the south

transept"). For our purposes, proper names are of limited use. Indeed, the only object in a Gothic cathedral that is likely to be referred to by a proper name is the cathedral itself. All other objects will be referred to in terms of the type of object they exemplify, or the larger features of which they form a part.

Fortunately, both types of names are easy to render in Prolog. Proper names correspond to simple Prolog atoms, such as `chartres` or `notre_dame_de_paris`. Common names can be represented as a Skolem function indicating the type of which an object is an instance, the larger object of which it is a part, and an index: `nave_inst(chartres,1)` to refer to the nave of Chartres, for example. Since the cathedral itself is the only proper name, it can be treated as the top level, with all other objects being either parts of the cathedral as a whole, or parts of those parts, and so forth.

The convention I have adopted for Skolemizing the names of entities is as follows: The functor consists of the object's type plus `_inst` (for "instance"). The first argument of the Skolem function indicates the owner (the larger object of which this object is a part), and the second argument is a numerical index. Thus `vaulting_unit_inst(nave_inst(chartres,1),2)` refers to the second vaulting unit of the first (and only) nave of Chartres Cathedral. Skolem functions are recursive, and any object's Skolem function describes the entire chain of ownership all the way up to the cathedral itself.

Skolem functions may seem unwieldy, but as they are generated automatically by the software the user need not deal with them directly. Having a systematic way of generating the names of objects in fact makes things easier on the user, as it prevents him or her from having to come up with a unique name for each object in the cathedral.

It should be noted that any Skolem function that follows this format is valid, even if it refers to an absurd object. Thus `triforium_inst(nave_inst(amiens,23),17)` is a valid object name, though it makes no sense to speak of the seventeenth triforium of the twenty-third nave of Amiens. The ability to refer to objects that may not in fact exist is useful in dealing with the architectural equivalent of "donkey sentences": we want to be able to say things like "For every column that has a necking, the necking is above the shaft." Representing this sentence logically requires us to say something about the relationship between an instance of "necking" and an instance of "shaft", though the former does not exist for every column.

## 3.3 Types

The set of types is fixed, and should correspond more or less to the list of entries in any decent glossary of Gothic architectural terms: nave, transept, column, vaulting unit, and so forth. Types are implemented as one-place predicates; thus if we wanted to assert that

Colin is a column, we would assert `column(colin)`.[1] In this respect, types are similar to properties (as described in section 3.4).

Types differ from properties, however, in that the former have the capability of bestowing existence on entities. As explained in section 3.2, any Skolem function of the proper form is a valid name. What separates a real entity like the nave of Saint-Denis from a nonsense entity like the twenty-fourth clerestory of Chartres is that the knowledge base will contain a type assertion of the former: `nave(nave_inst(saint_denis,1))`. The knowledge base also contains a list of all the valid types, so that in order to find out which objects actually exist, we simply run through the list of types and query each one to find out what entities of that type exist. This feature will be very useful when we are trying to discover whether our knowledge base is complete.

## 3.4 Properties and attributes

Attributes can be thought of as adjectives, which either do or do not describe a given object. They correspond to Boolean values or one-place Prolog predicates, and can describe the color, shape, material, or other features of an object. For example, `engaged(colin)` indicates that Colin the column is an engaged column, and `wood(doris)` indicates that Doris the door is made of wood.

Of course, treating each adjective as a boolean value has its limits. We might also like to have something like the enumerated types found in some programming languages, where we can define a many-valued property called `color` which can have values such as `red` or `green`. Thus to say that Colin is green we would assert `color(colin,green)`. But in fact these two approaches need not be mutually exclusive. We can make assertions of the type `green(colin)`, but make sure our knowledge base contains a set of rules like those in figure 3.1:

```
color(X,green) :-
green(X).

color(X,blue) :-
blue(X).

color(X,red) :-
red(X).
```

Figure 3.1: Creating properties from various attributes

[1] As stated above, only the cathedral itself will be called by a proper name. However, in the examples I will sometimes use proper names, such as Colin the column or Vernon the vaulting unit, to avoid having to type out a long Skolem function. Once again I remind the reader that the user will never need to enter Skolem functions by hand, so their unwieldiness is not a hindrance to usability.

Even better, rather than laboriously enumerating the rules by hand, we could do something like figure 3.2:

```
color_value(green).
color_value(red).
color_value(blue).

color(X,Color) :-
color_value(Color),
Functor =.. [Color,X],
call(Functor).
```

Figure 3.2: More elegant code for creating properties

Thus we can make simple assertions like `green(colin)` while still being able to submit queries such as `color(colin,X)`.

## 3.5 Relations

Where properties correspond to one-place predicates, relations correspond to two-place predicates. They represent various relationships between two entities, such as spatial relationships (`above`, `below`), relations of ownership (`has`), or functional rellationships (`supports`, `crosses`). Thus `above(colin,herbert)` signifies that Colin is above Herbert, while `has(colin, plinth_inst(colin,1))` signifies that colin has a plinth.

In order for our queries to return the desired results, the knowledge base will need to contain certain rules for making inferences about relations. For example, some relations are the inverse of others: if X is above Y, then necessarily Y is below X. Relations can also be transitive: if we know that X is above Y and Y is above Z, we know that X is above Z.

The "above" relation is an example of a partial ordering, which is a particularly common kind of relation for describing the spatial arrangement of objects. Other partial ordering relations might include "left of," "right of," "behind," and so forth. These relationships are transitive, nonreflexive (nothing is above itself), and antisymmetric (if X is above Y, Y cannot be above X).

Care must be taken when adding statements to the knowledge base not to violate the properties of relations. That is, if we have previously asserted that the shaft is above the plinth and the capital is above the shaft, we cannot subsequently assert that the plinth is above the capital without violating antisymmetry. To ensure this, rather than adding facts to the knowledge base directly using Prolog's `assert` predicate, we must implement an `assert_if_consistent` predicate that tests an assertion to see if any violations of the relation's properties would result, adding it to the knowledge base only if no violations are found.

The task of checking for contradictions is complicated somewhat by the fact that we are typically not asserting that a singular object is above or to the left of another singular object. When we say "the capital" is above "the shaft," we mean that, for each column, that column's capital is above that column's shaft. Thus instead of asserting a simple fact like `above(capital,shaft)`, we would need to assert a rule such as that in figure 3.3.

```
above(X,Y) :-
column(Z),
has(Z,X),
has(Z,Y),
capital(X),
shaft(Y).
```

Figure 3.3: Prolog representation of "The capital is above the shaft"

In other words, if X and Y both belong to the same column, and X is a capital and Y is a shaft, then X is above Y. Thus when checking for the consistency of an assertion of this type, we cannot just search the existing facts in the knowledge base to see whether any of them directly contradict our assertion. Instead, we need to add our rule to the knowledge base, systematically test its consequences (by querying `above` to get an exhaustive list of what is above what), and then retract the rule if a contradiction is found.

Another complication stems from the inductive nature of definitions of partial ordering relations. For example, the `ancestor` predicate that is encountered in introductory Prolog courses has a two-part definition. The first part is the base case: if X is the parent of Y, then X is the ancestor of Y. The second part is the inductive step: if X is the parent of Z and Z is the ancestor of Y, then X is the ancestor of Y. Thus the definition of `ancestor` depends on the existence of another predicate, `parent`, which is a special case of the former (the "immediate ancestor"). When the knowledge base is built, one would simply assert who is the parent of whom, and the more general ancestor relations would be derived from this knowledge.

Unfortunately, natural language does not always preserve the distinction between the more general relations and the particular, immediate relations. An English-language description of a column, for example, might say "The shaft is above the plinth, the necking is above the shaft, and the capital is above the necking." It is never explicitly said what is immediately above what, though the reader probably assumes that the shaft is immediately above the plinth and so forth. However, nothing prevents us from saying something like "The capital is above the plinth." We need to be able to assert that one object is above another without any definite commitment as to whether or not this relation is immediate. To accomplish this, we can make use of defeasible reasoning (about which more is said in section 3.6. If one object is asserted as being above another, we will consider it defeasibly true that it is immediately above it (provided we do not already know of objects lying between the two). This assertion is defeated as soon as another object is asserted to be below the former but above the latter.

## 3.6 Specific versus Generic Descriptions

A complete description of a cathedral would consist of the set of all and only those propositions that hold true for that cathedral. These propositions would include those properties possessed by each entity, and those relations held by each pair of entities. Since the knowledge base contains lists of all properties and relations, a complete description could be produced simply by querying each entry in those lists.

However, only a description of a specific cathedral can be complete. The generic description that is defined in superuser mode is necessarily incomplete. Since it represents only the features common to all Gothic cathedrals, it must be mute regarding any features that differ from cathedral to cathedral. Or, if not mute, at least not inflexible. In the generic description, some facts will be said to hold most of the time, others only rarely.

Thus while a specific cathedral description can be implemented entirely in Prolog, we need more expressive capability to render the generic description. For this I have chosen to use d-Prolog, Donald Nute's implementation of defeasible (or non-monotonic) reasoning in Prolog. In addition to normal Prolog rules, d-Prolog allows for defeasible rules (rules that apply most of the time, unless defeated by a stronger rule) as well as explicit negation (rather than simply negation-as-failure). Thus if we wanted to say that "most columns are round," we could assert the defeasible rule `round(X) := column(X)`, which would be defeated if our knowledge base contained the strict facts `column(colin)` and `neg round(colin)`.

Conflict between defeasible rules (when a given rule is both defeasibly true and defeasibly false) can be minimized because of the hierarchical nature of the knowledge base. Every entity must belong to a type or a subtype, and the rules defining subtypes (such as `support(X) :- column(X)`, "all columns are supports") are all strict rules. d-Prolog allows us to assume that more specific rules are superior to more general rules, so that a defeasible rule about columns would overrule a contrary defeasible rule about supports.

Even with the specificity condition, it is still possible for two defeasible rules to conflict if neither is more specific. This is not necessarily a bad thing, because it can tell us whether a given description is sufficiently detailed. If a purportedly complete cathedral description does not contain enough information for our inference engine to determine the truth of a given claim, an architectural historian reading the description might not be able to do so either.

It is not just the properties of, and relations between, entities that vary from cathedral to cathedral. Different cathedrals contain different sets of entities. A cathedral may or may not have a triforium, there may be different numbers of vaulting units in the nave, columns may or may not have neckings, and so forth. We need a way to say that certain entities might or might not exist. Fortunately, this is not difficult. Recall that most entities are identified by Skolem functions, but a given Skolem function is not taken to refer to an actually existing entity unless it has been assigned to a type. A rule such as `triforium(triforium_inst(X,1)) := cathedral(X)`, literally "For most cathedrals there exists a triforium", can still be defeated if we know that a particular cathedral has no triforium. Entities can thus be asserted into and out of existence; with apologies to Immanuel Kant, in ARC, existence *is* a predicate.

Entities whose numbers may vary are also easy to implement. To say that a nave has between five and seven bays, we would assert the existence of bays one through five using strict rules, and bays six and seven with defeasible rules. Finally, we would include a rule explicitly negating the existence of any bays with index greater than seven.

Another advantage of using defeasible reasoning is that it can help with the extraction of information from real-world descriptions. By using d-Prolog's `@` operator to query the various types, properties, and relations, we can find out what information about the cathedral is known for certain and what is still doubtful. We can then mine real-world descriptions for unknown information, or for potential defeaters of facts that are only defeasibly true. For example, if we cannot prove whether or not a given cathedral has a triforium, this tells us we need to search the cathedral description for any mention of a triforium. Or, if we know that most columns have neckings, we can search to see if any columns are explicitly mentioned as lacking neckings. By searching only for those things about which we are in doubt, we avoid having to interpret and understand every sentence of a description.

# Chapter 4

# ARC English: Natural-language programming for architectural knowledge

## 4.1 Natural language programming

*Natural language programming* (not to be confused with *natural language processing*) refers to the use of natural language, rather than a formal programming language, to issue instructions to a computer. While some programming languages, such as COBOL or Applescript, are superficially similar to natural languages, they still have the strictly limited syntax and vocabulary typical of traditional programming languages. Natural language programming, on the other hand, suggests interacting with computers in true natural language.

In natural language programming, programming takes place in a true subset of a natural language. This means that programming language staples like curly braces, reserved words (such as "printf"), and other features not found in natural language should not be required in natural language programming. Vagueness and ambiguity should be tolerated, so that information that would need to be provided explicitly in a traditional programming language can be left out with no effect on performance. Any given concept or instruction can usually be expressed a number of different ways. As a result, much more processing is required of the computer than with traditional programming languages. However, less effort should be required on the part of the human programmer.

Architectural description is not the same as programming. Whereas a program is a set of instructions for the performance of a task, an architectural description simply details the properties of physical objects. Nevertheless, there are enough similarities for a comparison between architectural description and natural language programming to be useful. Since the architectural description language used in the ARC project will ultimately be translated into a logical knowledge base, our task could be considered a form of natural language logic programming.

## 4.2 Some guiding principles for architectural natural language programming

At this point it is important to review some of the goals of the ARC project. We want it to be used by architectural historians and others who will not necessarily have a strong background in formal logic or computer programming. We want to be able to describe the mental model of a Gothic cathedral that such historians possess, that allows them to understand architectural descriptions. They must therefore be able to add facts to the knowledge base as easily and as naturally as possible, as though they were simply giving a verbal description. Once our generic model is complete, we want to be able to model particular cathedrals based on real-world architectural descriptions. The language in which the generic model is written should not be vastly different from that of other architectural descriptions.

The following are some guiding principles that will help us achieve these goals.

- *It is up to the computer to figure out what the user means, rather than up to the user to figure out what the computer wants to hear.* Since our target user base consists of non-programmers, they should not be required to learn a new programming language. Nor should they be expected to spend a lot of time trying to figure out how to express some concept so that the computer will accept it.

- *There should be no "syntax errors" or invalid inputs.* Nothing that is accepted in natural language should be rejected by the computer. This includes typographical errors and other small mistakes that people can easily cope with. When the user's input is such that the computer cannot even make a guess as to the intended meaning, the computer should not simply declare that an error has occurred. Rather, it should attempt to elicit clarification, perhaps by asking the user to rephrase or suggesting alternate phrasings.

- *More than one input string can correspond to the same output.* For example, "A column is a type of support," "All columns are supports," and "Every column is a support" all correspond to `support(X) :- column(X).`

- *The same input string might represent more than one output.* The sentence "All columns are supports" might have different interpretations, depending on context: does it mean all columns, everywhere, or all columns in the nave (or whatever part of the cathedral is being discussed)?

- *The computer should be able to figure out what to ignore.* Most programming language have special punctuation or syntax that separate comments from code. In natural language, however, authors might make an aside or offer inessential information without clearly delineating it as such. The computer should be able to ignore unimportant words and extract what is needed. In the architectural domain, this might mean

skipping over historical or sociological notes and concentrating solely on structural information.

- *Syntax and vocabulary should not be fixed.* The parser should not have a predefined vocabulary list detailing which words are or are not acceptable. Instead, the users should be able to add vocabulary from within the language itself, for example by saying "A 'column' is a type of support."

  Furthermore, the parser should be able to incorporate novel types of sentence structure into its rules. For example, suppose the user enters the unknown sentence, "For every column there is a capital." The computer might ask the user to rephrase, at which point the user enters the understood sentence "Every column has a capital." A new rule might then be added to the parser, so that future sentences of the form "For every X there is a Y" are interpreted as "Every X has a Y".

## 4.3 ARC English

The natural language programming environment I have developed for the ARC project is known as ARC English. This is a true subset of English used for expressing information about the structure of Gothic cathedrals. While ARC English does not contain every sentence that would be found in a real-world architectural description, every relevant fact about the structure of a Gothic cathedral should be expressible in ARC English. The primary use of ARC English will be for writing the generic cathedral description in superuser mode.

The built-in vocabulary of ARC English is very small, and consists almost entirely of function words ("is", "has", "every", "some", and so forth). No architectural terms are built in, because they are to be defined within ARC English itself, through statements such as "A 'column' is a type of support." This gives the superuser complete freedom in writing the generic description, and means that this description will contain human-readable definitions of all architectural terms. The limited built-in vocabulary would also make the task of internationalization simpler, should we someday wish to implement ARC French or ARC German.

Descriptions written in ARC English are translated into Prolog statements. The SWI-Prolog program for making this translation consists of a tokenizer and a parser, the latter implemented mainly through DCG rules. Using SWI-Prolog's module system, we can maintain two separate knowledge bases: the parser knowledge base, which contains the parsing rules; and the target knowledge base, which contains the rules that are produced as a result of the translation. For example, to process the ARC English statement "'Colin' is a column," which introduces the proper name Colin as referring to an entity of the column type, the parser would do two things. First, it would add a rule to the parser's knowledge base declaring 'Colin' to be a proper noun, so that any future occurrences of the term 'Colin' would be recognized as such. Second, it would add the rule `column(colin)` to the target knowledge base, declaring Colin to be a column.

The following subsections give a detailed outline of the syntax of ARC English that has been developed so far.

### 4.3.1 Types

As mentioned above, the ARC English parser is equipped with a minimal vocabulary. At first, the list of types is empty. The user must issue type definitions, either for top-level types or subtypes. To define a top-level type, simply say {`Indefinite article`} {`type name`} `is a type.` (For example, `A support is a type.`) To define a subtype, say {`Indefinite article`} {`subtype name`} `is a type of` {`type name`}`.` (e.g., `A column is a type of support.`)

When a type or subtype is defined, its name is added to the lists of types in both the parser knowledge base and the target knowledge base. When a subtype is defined, an additional rule is added to the target knowledge base expressing the subtype relationship. For the above example, `A column is a type of support`, the rule `support(X) :- column(X)` would be added.

The addition of a new type or subtype also occasions the addition of a rule to the parser knowledge base defining the plural of that type name. At present, the default is simply to define the plural as the singular plus 's', though rules to handle other regular plurals might be added. Because this rule does not work for every type name, the user can override the default plural by saying `'{Plural form}'is the plural of '{type name}'.` (e.g., `'Triforia'is the plural of 'triforium'.`)

To establish the existence of an entity identified by a proper name, one would write {`Proper name`} `is a` {`type`}`.` (For example, `Colin is a column.`) This would add the given name to the list of proper names in the parser knowledge base, and add the rule `column(colin)` to the target knowledge base.

Finally, sentences of the form `Every` {`type`} `has a` {`type`} or `Every` {`type`} `has` {`number`} {`plural type`} (e.g., `Every column has a shaft` or `Every vaulting unit has four columns`) establish both existence and ownership of type members. This type of sentence is used for entities identified by Skolem functions rather than proper names, since those entities can only be identified by their type and owners. `Every column has a shaft` adds two rules to the target knowledge base (figure 4.1).

```
shaft(shaft_inst(X,1)) :- column(X).
has(X, shaft_inst(X,1)) :- column(X).
```

Figure 4.1: Prolog representation of "Every column has a shaft"

The first rule establishes the existence of a shaft instance for every column, while the second establishes that said shaft instance belongs to that column. Note that we have not yet declared the existence of any shafts! Only if at least one column is known to exist would these rules establish the existence of a shaft.

### 4.3.2 Properties and attributes

To declare a new attribute, we need only say `'{Attribute}'is an attribute.` This adds the name of the attribute to the lists of possible attributes in both the parser and target knowledge bases. Once an attribute has been defined, it can be assigned to entities. For example, `Colin is a column. 'Red'is an attribute. Colin is red.`

Properties are similarly defined: `'{Property}'is a property.` Properties range over attributes, and there are many possible ways to express the relationship between a property and its attributes. The following are equivalent:

- `'Color'is a property whose attributes are red, green, and blue.`
- `'Red', 'green', and 'blue'are attributes. The property 'color'comprises red, green, and blue.`
- `'Red', 'green', and 'blue'are attributes. 'Color'means red, green, or blue.`

A statement of the form `Every {type} has the property {property}` indicates that every entity of the given type must possess exactly one of the attributes belonging to {property}. For example, `Every door has the property color` means that every object of type `door` must possess one of the attributes `red, green,` or `blue`.

### 4.3.3 Relations

Relations in ARC English correspond to prepositions or preposition-like phrases in English. Examples include "above," "below," "in front of," "to the left of," "in," "beside," and so forth. They are translated into Prolog as two-place predicates whose arguments are both entities, and they indicate some sort of relationship between the two entities. These relationships are typically, though not necessarily, spatial.

Relations can also have properties, similar to mathematical relations. Such properties might include symmetry, transitivity, reflexivity, or the inverses of these. A relation is defined through a sentence of the following type: `'{Relation name}'is a {list of properties} relation.` For example, `'Above'is a transitive, irreflexive, antisymmetric relation.`

Relations can have an opposite, as with `above` and `below`. To define opposites, one simply says `'Below'is the opposite of 'above'.`

Recall that in section 3.5 we found that many relations have both an immediate form (such as the `parent` relation) and a more general form (such as the `ancestor` relation). Whenever a new relation is defined, it will be assumed to be a general relation, and its immediate form is assumed to have the same name, prefixed by `immediately.` So, for example, when we define `above`, we also define `immediately above`. It may be desirable to override this, allowing us to say, e.g., `'Atop'is the immediate form of 'above'.`

It is not yet clear whether we need the capability to restrict the domains of relations. So far we have assumed that any entity can potentially relate to any other. But it makes no sense to say that the triforium is to the left of the clerestory, or that the nave is inside some column's plinth. We might want to be able to say something like `'Above'is a relation`

`between parts of a column` and so forth. However, implementing this feature would add considerable complexity at questionable benefit, since the superuser would be unlikely to make such category mistakes.

### 4.3.4 More complex determiner phrases

So far we have only seen how attributes are given to objects in the special case of proper nouns, e.g. `Colin is round.` But recall that only the cathedral as a whole is likely to be given a proper name. Other entities will only be referred to in terms of their type, owners, or other properties. An example of this type of reference was given earlier in introducing the `has` operation: `Every column has a capital.` We can assign attributes the same way: `Every column is red.`

But what if we don't want to say something about every member of a type? Since attributes correspond to adjectives, we can use them to select subsets of types. For example, `Every wooden door is red` assigns the attribute `red` to only those doors that possess the attribute `wooden`. Relations, which correspond to prepositions, can be used similarly: `Every column in the triforium is blue.`

This last example introduces a very common determiner that is nevertheless particularly difficult to implement: `the.` In the case of the triforium, of which there is going to be at most one per cathedral, "the" triforium obviously has only one possible referent. But consider the following code:

```
Every column has a shaft and a plinth.  The shaft is above the plinth.
```

What are "the" shaft and "the" plinth? Here they do not refer to unique objects, as there are many shafts and plinths in the cathedral. But they are introduced in the context of "every column." What we are really saying is something like "For each column, that column's shaft is above that column's plinth." The Prolog translation would be as in figure 4.2.

```
shaft(shaft_inst(X,1)) :- column(X).
plinth(plinth_inst(X,1)) :- column(X).
has(X, shaft_inst(X,1)) :- column(X).
has(X, plinth_inst(X,1)) :- column(X).
above(Y, Z) :-
column(X),
has(X,Y),
shaft(Y),
has(X,Z),
shaft(Z).
```

Figure 4.2: Prolog representation of "For each column..."

For each of these rules, the body of the rule essentially tells us what entities we are talking about (every column, or each column's shaft and plinth) while the head tells us something

about those entities (that it has a shaft, that one is above the other). The list of entities under discussion may persist from sentence to sentence. "The plinth" means "every column's plinth", even though "every column" was introduced in a previous sentence. Our level of analysis is not the isolated sentence, but the discourse as a whole.

To keep track of the entities under discussion, we should make nested lists as things are named, and pass them on from sentence to sentence. For example, when the sentence "Every column has a shaft and a plinth" is processed, the parser could pass the list `[column(X), [shaft_inst(X,1), plinth_inst(X,1)]]` on to the next sentence. If that sentence mentions "the" shaft, the parser will first check the list that was passed to it to see if a `shaft_inst` appears in it. If one appears, the parser understands "the shaft" to refer to that shaft. If more than one appears, the user is asked to clarify. If no shaft instance appears in the list, the parser checks the knowledge base as a whole. (We need not pass an instance of `nave`, for example, because there is only one nave per cathedral, so it is always clear what "the nave" refers to.)

Finally, we need to draw a distinction between "each" and "every". If the user makes a statement about "every" column, this probably means all columns anywhere, and so "every" is interpreted globally. If, on the other hand, the user mentions "each" column, this implies that a certain group of columns has just been mentioned, and the statement applies only to them. The parser would search the list of entities that was passed to it from the last sentence in order to see whether any columns are mentioned.

# Chapter 5

# Suggestions for further research

## 5.1 Space

So far it has been assumed that entities will be physical objects, such as columns and arches. However, architecture is about space, and it is useful to be able to refer to empty spaces and reason about them. We may think of the nave, for example, as being composed of columns and vaults, but if I say I am standing in the nave, I mean I occupy an empty space that is delineated by those objects.

As far as I can determine there is no "object-oriented" bias built into ARC English or its knowledge representation. One should be able to declare spaces to be entities as easily as one can declare objects to be entities, and assign them properties and relations just the same. However, as the generic description is developed, it may be discovered that spaces require special treatment. It may be useful to incorporate some insights from spatial logic, a form of interval logic designed for reasoning about space [Randell1992].

## 5.2 Paraphrasing

Recall that in section 4.2, one of our goals was that the ARC English parser should not be as restrictive as a traditional programming language, and that it should be able to understand multiple phrasings of the same sentence. So far we have not met this goal, as the examples given here still prescribe strict sentence forms. This situation can be improved by hand, by adding new rules to the parser's grammar. By having architectural historians test the system, we can identify what types of input sentences they are likely to attempt to use that are not already understood, and add rules for those sentences.

However, we do not want to have to enter rules for all possible input sentences by hand. For one thing, it may be impossible to anticipate every way a user might express some bit of information. For another, hand-coding rules puts all the power (and responsibility) of improving the grammar in the hands of the programmer rather than the user. We already have ways for the user to expand the parser's vocabulary (through type definitions). Why not allow the user to expand the parser's grammar as well?

One obvious way to do this is to understand novel sentences as paraphrases of already valid sentences. For example, to tell the system that all members of a particular type have a certain attribute, the canonical expression is "Every type is attribute." For example, "Every column is round." But there are a number of obvious alternate ways to express this: "All columns are round," "Columns are round," "A column is round," "Anything that is a column is round," "If something is a column, it is round," and so forth.

Note that all of these sentences contain the words "column" (a type) and "round" (an attribute), plus some function words. If we tell the system that all these sentences translate to `round(X) :- column(X)`, it should be able to figure out how to process all similar sentences. The methods for doing so are simply those used in statistical machine translation, wherein sentences of one language are identified as counterparts of sentences in another language. Indeed, the task of interpreting ARC English can be seen as a machine translation task: we are translating from a less restrictive subset of English (the sentences the user is likely to enter) into a more restrictive subset (the canonical dialect of ARC English).[1]

## 5.3 Numerical values

The properties and attributes covered so far have allowed only Boolean or enumerated values. However, it will be useful to allow numerical values as well. We do not simply wish to know that the south spire of Chartes is "tall." We want to be able to say that it is 103 meters high. The most obvious way to do this would be to allow a new type of relation that corresponds either to a two-place predicate (whose arguments are an entity and a numerical value) or a three-place predicate (whose arguments are an entity, a numerical value, and a unit of measurement).

Numerical values are probably more useful in dealing with specific cathedrals than with our generic description, as there were not many Gothic architectural features that were built to a standard size. As a result, the generic description will be more likely to contain comparative measurements like "taller than," "half as long as," and so forth. These comparative statements can be translated into numerical constraints and added to the knowledge base. SWI-Prolog includes modules for dealing with numerical constraints in both the real and integer domains, and these should be sufficient for handling comparisons of the sort found in the generic description.

## 5.4 Dealing with real-world descriptions

While descriptions of particular cathedrals could be added using the same ARC English instructions as the generic description, our goal is to be able to extract this information directly from real-world descriptions. This task might involve using machine translation techniques to translate the unrestricted English descriptions into ARC English. However, this is probably not the optimal approach, since such a system would need to be trained. A

[1] For a good overview of statistical machine learning techniques, see [Koehn2010].

large corpus of English-language architectural descriptions would have to be assembled, and then translated into ARC English by human experts. Furthermore, translating entire texts would be redundant, as such texts typically contain historical or aesthetic details and other information irrelevant to our task.

Fortunately, the nature of the ARC knowledge base suggests an approach for deciding exactly what information needs to be extracted from the text. Recall that in section 3.6, we saw that there is a systematic way of determining which facts about the cathedral are known with certainty, and which are unknown, or only tentatively assumed. All we need is a list of facts that we wish to know (the number of stories, the length of the nave, whether the columns have neckings) and a way to identify statements of these facts in real-world texts.

One way to do this might involve simple pattern matching. For example, to determine the number of stories in the elevation, a description could be searched for the phrase "{number}-story elevation". There are two problems with such an approach, however. One is that the programmer will have to hand-code an exhaustive list of patterns to search for. The other is that such a simple search might miss important details, like the presence of the word "not" or the fact that the sentence in question was referring to a different cathedral.

A better plan might involve machine learning. Several architectural descriptions could be studied by architectural historians, and passages that give facts about particular features could be identified and annotated. A machine learner could then be trained to recognize such sentences and the facts they describe. This method would require a corpus of architectural descriptions, and a lot of laborious annotation. It would probably still be less laborious than hand-coding all the rules. Both this approach and the hand-coded approach are likely to have problems with accuracy, and so the extraction of information from real-world texts should probably be supervised by architectural experts and not completely automated.

## 5.5 Responding to queries

Once the knowledge base has been filled with logical representations of various cathedrals, we need a way to access the information. This will probably be done through a simple web interface, in which questions such as "Which cathedrals have a three-story elevation?" or "How long is the nave of Saint-Denis?" are entered in a text box. These would be translated into Prolog queries and used to query the knowledge base. This task is similar to that of natural language database querying, an area of active research [Popescu2003].

# Bibliography

[Antoniou1997] Antoniou, Grigoris. *Nonmonotonic Reasoning.* The MIT Press, Cambridge, MA, 1997.

[BlackburnBos2005] Blackburn, Patrick and Johan Bos. *Representation and Inference for Natural Language: A First Course in Computational Semantics.* CSLI Publications, Stanford, CA, 2005.

[CovingtonNute1988] Covington, Michael A., Donald Nute, Nora Schmitz and David Goodman. "From English to Prolog via Discourse Representation Theory." ACMC Research Report 01-0024, The University of Georgia. URL (viewed May 5, 2011): `http://www.ai.uga.edu/ftplib/ai-reports/ai010024.pdf`

[CovingtonNute1997] Covington, Michale A., Donald Nute, and André Vellino. *Prolog Programming in Depth.* Prentice-Hall, Upper Saddle River, NJ, 1997.

[CovingtonSchmitz1988] Covington, Michael A., and Nora Schmitz. "An Implementation of Discourse Representation Theory." ACMC Research Report 01-0023, The University of Georgia. URL (viewed July 11, 2011): `http://www.ai.uga.edu/ftplib/ai-reports/ai010023.pdf`

[Coyne2001] Coyne, Bob, and Richard Sproat. "WordsEye: An Automatic Text-to-Scene Conversion System." In *ACM SIGGRAPH,* 2001. pp. 487-496.

[Hollingsworth2011] Hollingsworth, Charles, Stefaan Van Liefferinge, Rebecca A. Smith, Michael A. Covington, and Walter D. Potter. "The ARC project: Creating logical models of Gothic cathedrals using natural language processing." *Proceedings of the 5th ACL-HLT Workshop on Language Technology for Cultural Heritage, Social Sciences, and Humanities*, Portland, OR, USA, 24 June 2011. pp. 6368.

[Izzo1993] Izzo, Gregory J. "Incorporating Defeasible Reasoning into an Implementation of Discourse Representation Theory." ACMC Research Report AI-1993-06, The University of Georgia. URL (viewed July 11, 2011): `http://www.ai.uga.edu/ftplib/ai-reports/ai199306.pdf`

[KampReyle1993] Kamp, Hans, and Uwe Reyle. *From Discourse to Logic.* Kluwer, Dordrecht, the Netherlands, 1993.

[Koehn2010] Koehn, Philipp. *Statistical Machine Translation.* Cambridge University Press, Cambridge, UK, 2010.

[LiebermanLiu2005] Lieberman, Henry, and Hugo Liu. "Feasibility Studies for Programming in Natural Language." In *End-User Development,* H. Lieberman, F. Paterno, V. Wulf, eds. Kluwer Academic Publishers/Springer, the Netherlands, 2005.

[LiuLieberman2004] Liu, Hugo, and Henry Lieberman. "English: The Lightest Weight Programming Language of them *All*." From *LL4* (Workshop: Lightweight Languages 2004), MIT, Cambridge, MA, December 4, 2004. URL (viewed July 11, 2011): `http://ll4.csail.mit.edu/`

[LiuLieberman2005] Liu, Hugo, and Henry Lieberman. "Metafor: Visualizing Stories as Code." In *IUI'05*, San Diego, CA, January 10-13, 2005. ACM 1-58113-894-6/05/0001.

[Liu2008] Liu, Yong, Congfu Xu, Qiong Zhang, and Yunhe Pan. "The Smart Architect: Scalable Ontology-Based Modeling of Ancient Chinese Architectures." *IEEE Intelligent Systems,* vol. 23, no.1, Jan/Feb 2008. pp. 49-56.

[Liu2010] Liu, Yong, Yunliang Jiang, and Lican Huang. "Modeling Complex Architectures Based on Granular Computing on Ontology." *IEEE Transactions on Fuzzy Sustems,* vol. 18, no.3, June 2010. pp. 585-598.

[Mitchell1990] Mitchell, William j. *The Logic of Architecture: Design, Computation, And Cognition.* The MIT Press, Cambridge, MA, 1990.

[Neagley1998] Neagley, Linda Elaine. *Disciplined Exuberance: The Parish Church of Saint-Maclou and Late Gothic Architecture in Rouen.* The Pennsylvania University Press, University Park, PA, 1998.

[Nelson2006] Nelson, Graham. "Natural Language, Semantic Analysis and Interactive Fiction." URL (viewed May 5, 2011): `http://www.inform-fiction.org/I7Downloads/Documents/WhitePaper.pdf`

[Nute2003] Nute, Donald. "Defeasible Logic." In *Proceedings of the Applications of Prolog 14th International Conference on Web Knowledge Management and Decision Support* (INAP'01), Oskar Bartenstein, Ulrick Geske, Markus Hannebauer, and Osamu Yoshie (Eds.). Springer-Verlag, Berlin, Heidelberg, Germany, 2003. pp. 151-169.

[Popescu2003] Popescu, Ana-Maria, Oren Etzioni, and Henry Kautz. "Towards a Theory of Natural Language Interfaces to Databases." In *IUI'03*, Miami, Florida, 2003. ACM 1-58113-586-6/03/0001.

[Randell1992] Randell, David A., Zhan Cui, and Anthony G. Cohn. "A Spatial Logic based on Regions and Connection." In *Proceedings of the third National Conference on Principles of Knowledge Representation and Reasoning.* B. Nebel, C. Rich, and W. R. Swartout (Eds.). Morgan Kaufmann, Los Altos, 1992. pp. 165176.

[Vadas2005] Vadas, David, and James R. Curran. In *Proceedings of the Australasian Language Technology Workshop 2005,* Sydney, Australia, December 2005. pp. 191-199.

[Viollet1854] Viollet-le-Duc, Eugène-Emmanuel. *Dictionnaire raisonné de l'architecture française du XIe au XVIe siècle.* vol. 2. 1854-68. Libraries-Imprimeries Réunies, Paris. Image URL (viewed May 5, 2011): `http://fr.wikisource.org/wiki/Fichier:Plan.cathedrale.Chartres.png`

[Wonka2003] Wonka, Peter, Michael Wimmer, François Sillion, and William Ribarsky. "Instant Architecture." In *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2003,* vol. 22, issue 3, July 2003. pp. 669-677.