# Assumption Grammars for Generating Dynamic VRML Pages

Anima Gupta

*Department of Computer Science*
*University of North Texas*
*P.O. Box 311366*
*Denton, Texas 76203, USA*
E-mail: agupta@cs.unt.edu
WWW: http://www.cs.unt.edu/~ agupta

Paul Tarau

*Department of Computer Science*
*University of North Texas*
*P.O. Box 311366*
*Denton, Texas 76203, USA*
E-mail: tarau@cs.unt.edu
WWW: http://www.cs.unt.edu/~ tarau

**Abstract**

We describe a Prolog to VRML mapping allowing generation of dynamic VRML pages through CGI and server side Prolog scripts. BinProlog's Assumption Grammars (a form of multi-stream DCGs with implicit arguments and temporary assertions, scoping over the current AND-continuation) are used to mimic VRML syntax and semantics directly, without using a preprocessor. The resulting scripting language allows quick development of integrated knowledge processing and high quality data visualization applications. Using BinProlog's multi-threading, we describe a design integrating in a self-contained Prolog application a Web Server, a Data Extraction module and an Assumption Grammar based VRML generator.

**Keywords:** *Automatic Software Generation, Internet Programming, Logic Programming, Prolog, Definite Clause Grammars, Assumption Grammars, Virtual Worlds*

## 1 Introduction

VRML (Virtual Reality Modeling Language) [13,10,2,11] is a de facto Internet standard for 3D visualization. Its applications range from modeling unique

objects like the International Space Station, to shared virtual reality and 3D visualization of complex data.

VRML does not provide extensive programming language constructs, and using its embedded JavaScript-like scripting language is cumbersome. Although an External Authoring Interface [22] allows controlling VRML worlds from Java, the combination of multiple layers running within a browser makes EAI applications extremely fragile.

Prolog is a flexible and powerful AI language, particularly well suited for knowledge processing, Internet data extraction, planning, machine learning, hypothetical reasoning etc.

This has motivated us to combine the two technologies to allow the integration of our Prolog based Internet data extraction tools and VRML's 3D visualization and animation capabilities.

In this paper we describe a Prolog to VRML mapping, which allows generation of dynamic VRML pages through CGI and server side Prolog scripts, following the technique introduced in [15].

Assumption Grammars [20,5] are multi-stream DCGs with implicit arguments and temporary assertions, scoping over the current AND-continuation [1]. We will use them to mimic VRML syntax and semantics with an Assumption Grammar based BinProlog code generator. The resulting scripting language will allow quick development of integrated knowledge processing and high quality data visualization applications. Using BinProlog's multi-threading, we will integrate in a self-contained Prolog application a Web Server, a Data Extraction module, and an Assumption Grammar based VRML generator. From the user's point of view our integrated set of tools looks like a dynamic VRML page which reacts to user input through a set of HTML forms and VRML anchors [2]. This allows the user to provide some parameters or click on sensor enabled VRML areas and see as a result the output of our server side VRML generator, displayed by the browser as a 3D animation.

## 2 Overview of Assumption Grammars

### *2.1 Assumed code, intuitionistic and linear implication*

Intuitionistic `*/1` adds temporarily a clause usable in later proofs, through calls to `-/1`. Such a clause can be used an indefinite number of times, mostly like asserted clauses, except that it vanishes on backtracking. Its scoped version `=>` (intuitionistic implication [8] ) as used in

```
Clause=>Goal or [File]=>Goal
```

[1] This allows producing and consuming backtrackable prolog assertions as a means to exchange information between arbitrary points of the parsing process.

[2] Hyperlinks attached to VRML nodes.

makes `Clause` or the set of clauses found in `File` available only during the proof of Goal. Both vanish on backtracking. Clauses are usable an indefinite number of times in the proof, i.e. for instance

```
?- *a(13),-a(X),-a(Y)
```

will succeed, as `a(13)` is usable multiple times, and therefore matches both `a(X)` and `a(Y)`.

The linear assumption operator `+/1` adds temporarily a clause usable *at most once* in later proofs, through use of `-/1`. This assumption also vanishes on backtracking. Its scoped version `-:` (linear implication [9,7]) as used in

```
Clause-:Goal or [File]-:Goal
```

makes `Clause` or the set of clauses found in `File` available only during the proof of Goal. Assumptions vanish on backtracking and are accessible only inside their stated scope. Each clause is usable at most once in the proof, i.e. for instance

```
?- +a(13),-a(X),-a(Y).
```

fails as `a(13)` is usable only once and therefore consumed after matching `a(X)`.

One can freely mix linear and intuitionistic clauses and implications for the same predicate, for instance as in:

```
?-a(10)-:a(11)=>a(12)-:a(13)=>(-a(X),-a(X)).
X=11 ;
X=13 ;
no
```

At the time a(X) is called with `-a(X)` we have the assumed facts a(10), a(11), a(12) and a(13), as implications are embedded left to right. Given that X is consumed twice, only the values 11 and 13 assumed with intuitionistic implications `=>` will be returned as solutions. The (two) attempts to consume twice the values `a(10)` and `a(12)`, assumed with linear implication, will produce no solutions.

### *2.2 The Assumption Grammar API*

*Assumption Grammars* [20,5] are logic programs augmented with linear and intuitionistic implications scoped over the current continuation, and implicit multiple DCG accumulators [3].

[3] An accumulator is a sequence of chained logic variables, for instance S1,S2,S3 as used in the DCG rule `a-->b,c` which expands to `a(S1,S3):-b(S1,S2),c(S2,S3)`. Assumption Grammars provide the equivalent of an arbitrary number of such variable chains - through and internal implementation using backtrackable destructive assignment, instead of the program transformation approach used to implement single stream DCGs.

The complete Assumption Grammar API consists of 3 assumption operators:

```
*A: adds intuitionistic assumption A
+A: adds linear assumption A
-A: matches an assumption in current scope or fails
```

and 3 DCG-stream operators.

Implicit DCG-streams can be seen as providing implicit input and output sequence arguments. While operationally equivalent to program transformation based DCGs, no preprocessing is required, as terminals interact with the implicit DCG stream directly, through a set of BinProlog built-ins:

```
#<Tokens : initializes/unifies the DCG stream with a list of Tokens
#Token: matches or inserts a Token in the implicit DCG stream
#>State: returns the current State of the implicit DCG stream
```

The following pattern explains the use of implicit accumulators in matching/recognition mode:

```
?- #<[a,b,c,d],#A,#B,#>LeftOver.
A=a,
B=b,
LeftOver=[c,d]
```

For generation mode we start with an unbound value `Xs` for the implicit DCG stream, then we bind it to list elements `a,b,c` by stating that `a,b,c` are terminals.

```
?- #<Xs,#a,#b,#c,#>[].
Xs=[a,b,c]
```

The combination of DCG stream operations and assumptions is particularly well suited for generating hierarchical data structures where properties hold for selected subobjects.

## 3 An Overview of VRML

VRML'97 is a hierarchical scene description language which is human readable [13,2]. The VRML scene is composed of **nodes** and **routes**. Each node can have a unique name for routing or re-use, a type, a number of associated events (both incoming and outgoing), a number of fields which hold the data for the node. In addition, group nodes can have many other nodes as children. One special node (script) can hold a Javascript program to control behavior in the scene. Group nodes can hold many other nodes, associating them in some fashion. Positioning of objects in the world is achieved through the **transform** node. The transform node applies **scale** operations, **rotations**

and **translations** to its children. To achieve complex repeated transforms, transform nodes may be nested. Sensors can be thought of in two categories, those associated with geometry and those for sensing time. The sensors can respond to various conditions such as proximity to an avatar, mouse clicks, mouse movement or whether an object is in the field of view of the user. Each interpolator node has a set of **keys** and a set of **keyValues**. When a **set_fraction** eventIn is received, the values are interpolated to get the correct **value_changed** eventOut. Prototypes are the mechanism used for expanding the VRML language. A prototype has a new type name followed by a list of fields and events in brackets. Interacting with VRML worlds or animating them will cause events to be generated. These are sent from one node to another. Here are some typical event sequence patterns: usually a Sensor of some kind generates an event to start the animation. Sometimes processing of events is required which can be done through a simple script node. For non-reactive subcomponents (for instance to run an animation or to supply a stream of events for a color interpolation) Time Sensors can be used.

VRML's hierarchical space representation suggests a natural mapping to Prolog terms. Interestingly enough, both VRML and Prolog describe *atemporal* structures - space in the case of VRML and logical axioms in the case of Prolog. To some extent, VRML's declarative **space** representation and Prolog's declarative **truth** representation share the same difficulties when faced with the problem of representing time and change. Not surprisingly, VRML's event propagation mechanism (needed for animation) has a strong declarative flavor. In fact, it is quite similar to Prolog's DCG argument chaining: events are propagated with ROUTE statements between chained nodes.

### *3.1 Syntactical similarities between Prolog and VRML*

VRML's concrete syntax is closer to languages like Life [1] than to Prolog, i.e. all arguments are *named*, not positional. However, by using compound terms tagged with VRML's attribute names and some user defined operators (like `@`) we can easily mimic VRML syntax in Prolog.

#### *3.1.1 Syntax mapping of some key idioms*

Our mapping is so simple that it does not even need a preprocessor. Instead, we use Prolog's reader with some prefix and infix operator definitions such that a standard Prolog reader accepts constructs like `@[...]` and `@{...}` as terms. As the same applies to Assumption Grammars, this pseudo-VRML notation turns out to be just plain Prolog. The details of the mapping are as follows:

```
Prolog          VRML
-------------------------
```

```
  @{  } ===>   {  }
  @[  ] ===>   [  ]
  f(a,b,c) ==> f a b c
  a is b   ==> a 'IS' b
  a=b      ==> a b
-----------------------
```

The key idea behind this mapping is to allow writing unrestricted VRML code in Prolog, while being able to interleave it with "compile time" Prolog computations. The following example illustrates how to build pseudo VRML in Prolog, in the (interesting) case when a VRML PROTO construct is generated, which in turn can be used to model multiple instances of VRML objects.

```
proto anAppearance @[
   exposedField('SFColor')=color(1,0,0),
   exposedField('MFString')=texture@[]
]
@{
   'Appearance' @{
       material='Material' @{
         diffuseColor is color
       },
       texture='ImageTexture' @{
         url is texture
       }
    }
}.
```

The reader will notice that the resulting VRML code is in fact a close syntactic variant of its Prolog template.

```
#VRML V2.0 utf8

PROTO anAppearance
[
  exposedField SFColor color 1 0 0 ,
  exposedField MFString texture []
] {
  Appearance { material
    Material { diffuseColor IS color }
    texture ImageTexture {
      url IS texture
    }
  }
}
```

### *3.2 Building Prolog Macros*

While VRML's PROTO constructs allow significant code reuse, it is sometimes preferable to build simpler VRML code through Prolog "macro"-like constructs, which, depending on their parameters, will expand to multiple VRML node instances.

```
shape(Geometry):-
  % based on scoped assumptions (=>)
  default_color(Color),
  toVrml(
     aShape @{
       geometry(Geometry@{}),
       Color
     }
  ).

sphere:- shape('Sphere').
cone:- shape('Cone').
cylinder:- shape('Cylinder').
box:- shape('Box').
```

### *3.3 Using Prolog macros*

We have added a ‘ (backquote) notation to force evaluation of Prolog macros by our generator.

```
group @{
  children @[
    ‘transform(
       translation(0,10,4),scale(2,2,2),rotation(0,0,0,0),
       [‘sphere, ‘cone]
     ),
    ‘transform(
       translation(5,0,0),scale(1,3,6),rotation(0,1,0,1.5),
       [‘box]
     )
  ]
}.
```

The generated code looks as follows:

```
Group {
  children [
    Transform {
```

```
      translation 0 10 4 scale 2 2 2 rotation 0 0 0 0
      children [ # aShape is a VRML 2.0 PROTO!
        aShape { geometry Sphere{} color 0.7 0.8 0.8
        } ,
        aShape { geometry Cone{} color 0.6 0.3 0.9
        }
      ]
    } , .....
  ]
}
```

**Using assumptions in the VRML generator macros**

The following example illustrates the use of a scoped implication to color *blue* a complete sub-object (cone) expressed itself as a macro.

```
def saucer = ‘transform(
   translation(0,1,1),scale(0.6,0.2,0.6),rotation(0.5,0.5,0,1.5),
   [
     ‘sphere,
     % this can used to propagate a color over a region
     ‘(color(blue)=>>cone)
   ]
 )
```

### *3.4 The Assumption Grammar based VRML Generator*

Our generator is implemented as a predicate **toVrml/1** which takes a Pseudo-VRML tree and expands it into an implicit argument DCG stream. The (simplified) code looks as follows:

```
toVrml(X):-number(X),!,#X.
toVrml(X):-atomic(X),!,#X.
toVrml(A@B):-!,#indent(=),toVrml(A),toVrml(B).
toVrml(A=B):-!,toVrml(A),toVrml(B).
toVrml(A is B):-!,#indent(=),toVrml(A),#'IS',toVrml(B).
toVrml({X}):-!,#'{',#indent(+),toVrml(X),#indent(-),#'}'.
toVrml(‘X):-!,X.
toVrml(X):-is_list(X),!,
  #'[',#indent(+),vrml_list(X),#indent(-),#']'.
toVrml(X):-is_conj(X),!,vrml_conj(X).
toVrml(T):-compound(T),#indent(=),vrml_compound(T).
```

A final step is performed by a simple post-processor which applies indentation hints and formats the actual VRML output. VRML'97 ROUTES are structurally similar to DCGs: they are used to *thread* together streams of

change expressed as event routes. While a DCG-like notation could have been used to generate such ROUTE statements automatically, we have preferred to use the simpler syntax mapping mechanism as it is closer to what VRML programmers expect.

# 4 Web Application architectures

## *4.1 CGI script based VRML generation*

We have integrated our VRML generator with a BinProlog based CGI scripting toolkit [17]. This allows generation of dynamic VRML content returned to the browser as a result of a POST-method call (Fig 1).

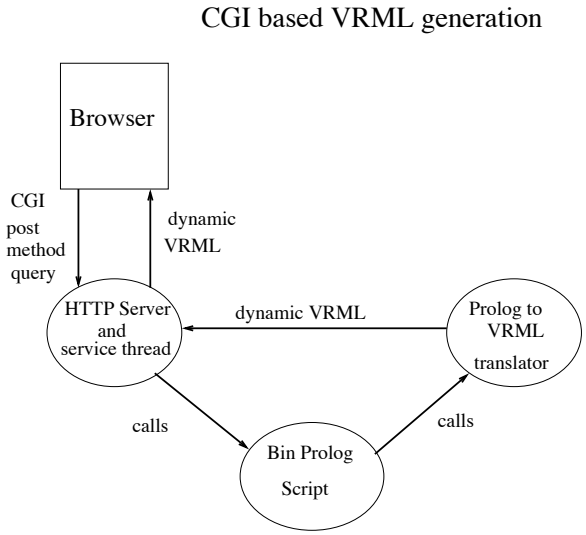


Fig. 1. A CGI based Architecture

The process of generating a VRML page with a CGI script is as follows:

- *the user clicks on an HTML form or VRML anchor*
- *a CGI script is invoked by the HTTP server*
- *the BinProlog based VRML generator is activated*
- *the dynamic VRML page is sent back to the client*
- *the browser's VRML plugin displays the results*

A CGI based demo of our generator implementing this architecture is available at `http://www.binnetcorp.com/BinProlog`.

Persistent server state can be maintained using BinProlog's remote predicate calls to a persistent BinProlog server or through some higher level networking abstractions like mobile threads [19] or remote blackboards [6].

*4.2 Server Side Prolog based VRML generation*

By using the BinNet Internet Toolkit [17] we can further simplify this CGI-based architecture as shown in Fig. 2). Instead of calling a CGI script each time, a form is sent to the server with a SUBMIT request (POST or GET) to provide a dynamic VRML page. We can then simply embed the VRML generator in the server. As in most modern Web server architectures (Apache, Microsoft, Sun etc.), a new BinProlog *thread* is spawned inside the server process for each request. Server state can be shared between multiple users and made persistent by having a backup thread periodically writing the server's state to a Prolog dynamic clause file. This feature is particularly important for shared virtual world applications like LogiMOO [21].

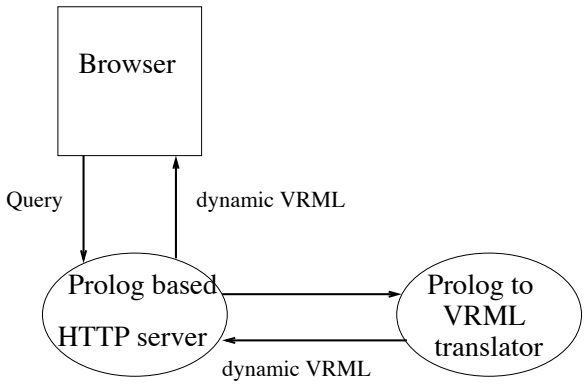


Fig. 2. Server Side Prolog based VRML Generation Architecture

To summarize, the process of generating a VRML page with a generator embedded in the Web server is as follows:

- *the user clicks on an HTML form or VRML anchor*
- *the VRML generator working as a server side Prolog thread is activated*
- *the dynamic VRML page is sent back to the client*
- *the browser's VRML plugin displays the results*

The multi-stream nature of Assumption Grammars is instrumental in embedding the VRML generator in ordinary CGI script processing - which uses its own DCG stream to parse input. The same thing happens in the case of the server side use - other DCG streams are used to parse elements of the server side HTTP protocol as well as to generate output. Assumption Grammars based I/O shares some similarities with Monadic [23] I/O - as present in functional languages like Haskell, [12]. For instance, by using Assumption Grammars we are able to separate the logic of sequencing the output from

formatting and writing to the output stream.

### *4.3 Combining the VRML Generator with Web Data Extraction and Persistent Server State*

A number of ongoing projects in our research group involve Web data extraction. A typical application architecture involves *blackboard* based agent coordination (Fig. 3).

#### *4.3.1 Linda Based Agent Coordination*

Our agent coordination mechanism is built on top of the popular Linda [4] coordination framework, enhanced with unification based pattern matching, remote execution and a set of simple client-server components merged together into a scalable peer-to-peer layer, forming a network of interconnected virtual places. The key Linda operations are the following:

```
out(X): puts X on the server
in(X):  waits until it can take an object
        matching X from the server
all(X,Xs): reads the list Xs matching X
        currently on the server
```

The presence of the `all/2` collector avoids the need for backtracking over multiple remote answers. Note that the only blocking operation is `in/1`. Typically, distributed programming with Linda coordination follows consumer-producer patterns with added flexibility over message-passing communication through associative search.

#### *4.3.2 Agent Coordination with Blackboard Constraints*

A natural extension to Linda is to enable agent threads with *constraint solving* for the selection of matching terms on the *blackboard*, instead of plain unification. This is implemented in Jinni [18] through the use of 2 builtins:

`Wait_for(Term,Constraint)`: waits for a Term on the blackboard, such that Constraint is true, and when this happens, it removes the result of the match from the blackboard with an `in/1` operation. Constraint is either a single goal or a list of goals `[G1,G2,...,Gn]` to be executed on the server.

`Notify_about(Term)`: notifies about this term one of the blocked clients which has performed a `wait_for(Term,Constraint)` i.e.

```
notify_about(stock_offer(aol,89))
```

would trigger execution of a client having issued

```
wait_for(stock_offer(aol,Price),less(Price,90)).
```

In a *client/server* Linda interaction, triggering an atomic transaction when data, for which a constraint holds, becomes available, would be expensive. It

would require repeatedly taking terms out of the blackboard, through expensive network transfers, and put them back unless the client can verify that a constraint holds. Our *server side* implementation checks a blackboard constraint only after a match occurs between new incoming data and the head of a suspended thread's constraint checking clause, i.e. an indexing mechanism is used to avoid useless computations. On the other hand, a mobile client thread can perform all the operations atomically on the server side, using local operations on the server, and come back with the results.

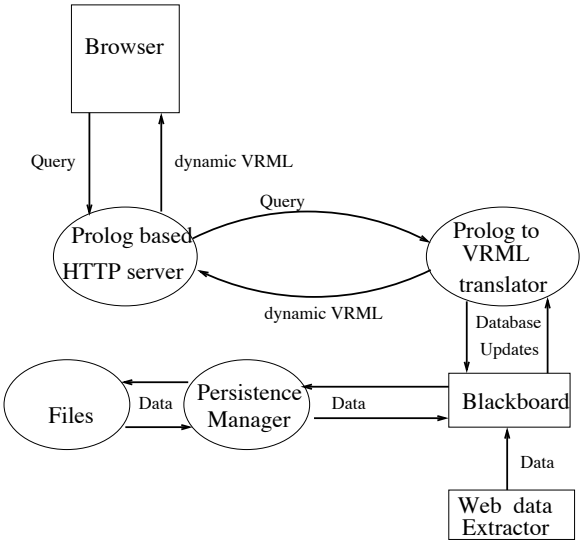


Fig. 3. Prolog HTTP Server With VRML Generator, Web Data Extraction and Persistent State

In a typical application (see Fig 3), a Web based data extractor fetches stock quotes and related historical information, which is visualized using our VRML generator. In particular, before committing to a stock market transaction, a visual animation of the projected stock marked dynamics can help human users to quickly validate the proposed buying or selling action. Trader agents, watching for triggers expressed as blackboard constraints run in the server process on separate threads. They allow expressing complex conditions for buy or sell transactions, far beyond the *stop* and *limit* transactions conventional online brokerages can offer. The state of the server (Web data and user agents) is made persistent through a separate thread which periodically saves terms on the blackboard and thread suspension records on the server, to Prolog dynamic clause files.

# 5 Related Work

Dynamic generation of HTML Web pages with Prolog CGI scripts has been pioneered by the Madrid group's Pillow library [3]. Currently most free and commercial Prologs offer basic support for CGI programming. Among them the BinNet Internet Toolkit [17] offers extensive server and client side tools, ranging from a Prolog based Web server supporting SSI (Server Side Includes) written in Prolog, to a programmable Internet Search Engine (spider). The first description of a Prolog term to VRML mapping (of which the one used in this paper is an extension) has been presented in [15]. This seems to be the model also followed (using different implementation techniques) by [14], which also pioneers the use of VRML visualization in constraint logic programming. Dynamic VRML manipulation using a Java + EAI + VRML plugin + Prolog interpreter in Java architecture is part if the BinNet Jinni demos - available online at `http://www.binnetcorp.com/Jinni`). This prototypes a shared virtual world - allowing synchronized manipulation of multiple VRML objects by human users or Jinni based Prolog agents. Global state kept on a server ensures that all users see the same VRML landscape.

An important difference between the approach described in this paper and [14] is the ability to use assumptions to parameterize *deep* components of VRML trees. For instance, in [14] a code transformer needs to be written to replace cylinders by cones in the Prolog sources of a VRML page. We can achieve the same effect (see subsection 3.3) by expanding a generic macro `shape` and then pass the actual value (`cone` or `cylinder`) as an intuitionistic assumption. At the time the macro is called, the effect of the assumption will generate either a cylinder or a sphere, depending on the assumption.

The approach described in this paper also differs from [14] as we use Assumption Grammars - as outlined in [15]. Among the advantages of Assumption Grammars over conventional DCGs:

- no preprocessor is needed, therefore VRML syntax is emulated simply by adding a few Prolog operator definitions
- defining a reentrant macro language can be achieved more easily than with repeated calls to the DCG term expander
- the use of assumptions and (scoped) implications allows injecting modifications of colors and various other attributes in the VRML code

On the downside, as only BinProlog [16] and Jinni [18] support Assumption Grammars at this time, portability of our generator is quite limited.

## 6 Conclusion

We have described a novel use of Prolog as a program generator for empowering special purpose languages without processing abilities like VRML. We have described two Web based architectures for generating dynamic VRML, supporting persistent server side and multi-user synchronization. Not only Prolog can be used as a macro language for building compact template files but typical AI components like Prolog planners can be used for realistic avatar movement in VRML worlds.

Our syntax-mapping based generator mechanism allows reuse of VRML programming skills, while the presence of Prolog as a powerful macro language provides an interesting synergy between Prolog's planning, scheduling, Internet data extraction ability and VRML's Internet ready animated 3D visualization.

## Acknowledgment

We thank the anonymous referees for their valuable comments and suggestions.

## References

[1] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3/4):195, 1993.

[2] D. R. J. L. Ames, Andrea L./Nadeau. *VRML 2.0 Sourcebook*. Wiley, Dec. 1996.

[3] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/CIAO Library for INTERNET/WWW Programming using Computational Logic Systems, May 1999. See http://www.clip.dia.fi.upm.es/Software/pillow/pillow.html.

[4] N. Carriero and D. Gelernter. Linda in Context. *CACM*, 32(4):444–458, 1989.

[5] V. Dahl, P. Tarau, and R. Li. Assumption Grammars for Processing Natural Language. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 256–270, MIT press, 1997.

[6] K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, Jan. 1996.

[7] J. S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, May 1994. Available as University of Pennsylvania Technical Reports MS-CIS-92-28 or LINC LAB 269.

[8] J. S. Hodas and D. Miller. Representing objects in a logic programming language with scoping constructs. In D. D. H. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 511 – 526. MIT Press, June 1990.

[9] J. S. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear logic. *Journal of Information and Computation*, 110(2):327–365, May 1994.

[10] K. Jamsa Kris A./Schmauder;Phil/Yee, Nelson/Jamsa. *VRML Programmer's Library*. May 1997.

[11] A. D. Martin McCarthy. *Reality Architecture: Building 3D Worlds in Java & VRML*. Feb. 1998.

[12] S. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. Technical report, Feb. 1999. available from: http://www.haskell.org/onlinereport/.

[13] G. B. Rikk Carey. *The Annotated Vrml 2.0 Reference Manual*. June.

[14] G. Smedbäck, M. Carro, and M. Hermenegildo. Interfacing Prolog and VRML and its Application to Constraint Visualization. In *The Practical Application of Constraint Technologies and Logic programming*, pages 453–471. The Practical Application Company, April 1999.

[15] P. Tarau. Logic Programming Tools for Advanced Internet Programming. In J. Maluszynski, editor, *Logic Programming, Proceedings of the 1997 International Symposium*, pages 33–34, MIT press, 1997. http://www.cs.unt.edu/ tarau/research/PapersHTML/ptut/art.ps.

[16] P. Tarau. BinProlog 7.0 Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp., 1998. Available from http://www.binnetcorp.com/BinProlog.

[17] P. Tarau. BinProlog 7.0 Professional Edition: Internet Programming Guide. Technical report, BinNet Corp., 1998. Available from http://www.binnetcorp.com/BinProlog.

[18] P. Tarau. Inference and Computation Mobility with Jinni. In K. Apt, V. Marek, and M. Truszczynski, editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48. Springer, 1999. ISBN 3-540-65463-1.

[19] P. Tarau and V. Dahl. High-Level Networking with Mobile Code and First Order AND-Continuations. *Theory and Practice of Logic Programming*, 1(1), Jan. 2001. to appear at Cambridge University Press.

[20] P. Tarau, V. Dahl, and A. Fall. Backtrackable State with Linear Affine Implication and Assumption Grammars. In J. Jaffar and R. H. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, Lecture Notes in Computer Science 1179, pages 53–64, Singapore, Dec. 1996. Springer.

[21] P. Tarau, K. De Boschere, V. Dahl, and S. Rochefort. LogiMOO: an Extensible Multi-User Virtual World with Natural Language Control. *Journal of Logic Programming*, 38(3):331–353, Mar. 1999.

[22] The External Authoring Interface Workgroup. VRML-EAI Specification. Technical report, 1999. available from http://www.vrml.org/WorkingGroups/vrml-eai/index.html.

[23] P. Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, pages 1–17, 1993.