FLOOD: A Planning Framework for Reasoning with Linguistic Data

Curtis Huttenhower

Friday, December 12, 2003

Abstract

Open domain question answering is slowly broadening its horizons, expanding from simple factoid questions to encompass broader and more complex queries. Recent areas of interest include scenario-based question answering, incorporating existing domain knowledge during question analysis, various types of inference, and intelligently processing subquestions. FLOOD (standing for Fluent, Linguistic, Object-Oriented, and Dynamic) provides a system within which these problems may be explored. It consists of three parts:

A Text Processor which integrates existing tools to provide an end-to-end semantic parse. This begins with simple sentence breaking and tokenization and ends with semantic role filling. The Text Processor is organized as a separate module applicable to other tasks outside of FLOOD.

The FLOOD Planning Platform, an environment for authoring and utilizing simple linguistic planning domains and algorithms. The platform handles routine tasks such as domain and state space management while leaving specifics up to individual reasoning modules; the primary goals of the platform are to provide a convenient and flexible interface for reasoner and domain development.

A Question Answering Reasoner operating within the context of the Planning Platform. The current implementation is based on the FLECS algorithm (Veloso and Stone, 1995) and performs simple inference and subquerying tasks based on the output of the text processor.

Acknowledgements

I would like to thank my adviser, Eric Nyberg, and my committee members, Robert Frederking and Scott Fahlman. They have been extremely patient and helpful during development of this thesis, particularly in light of the additional chaos and time constraints I introduced into the process, and there is perhaps no one better than Eric for keeping an implementation-focused thesis on track. I would like to extend my thanks to Teruko Mitamura and the entire JAVELIN research team as well, particularly Benjamin Van Durme for his aid and suggestions during development. With their help, the FLOOD project came to be and will continue to provide us with new challenges in the future.

Finally, I would like to thank my wife, Gwendolyn. She has been a tremendous help throughout the entire process, from putting up with my initial confusion to proofreading the final product. She's also asked that I use the terms "patient" and "long-suffering", and she certainly deserves them. Fortunately, even if I'm faced with another large project in the future, our wedding will never again fall in the middle of it.

Contents

1 Introduction

In the past several years, open domain question answering has developed from a near-impossible problem into an active field supporting a variety of approaches [66, 67, 68, 69]. While initial efforts focused on limited factoid questions, recent work has expanded into more complex areas such as the incorporation of domain knowledge, question answering scenarios, and deeper semantic analysis of the language of the question and its answers [48, 47, 43]. To further the integration of these advanced techniques into the existing body of question answering work, it is necessary to provide broad and flexible environments within which experimentation and analysis can take place. In support of this expansion, a combination of solutions is presented here to allow the analysis and manipulation of text in the rich manner necessary for complex question answering.

1.1 Overview

This thesis, referred to herein as FLOOD (standing for Fluent, Linguistic, Object-Oriented, and Dynamic, to be discussed later), is an attempt to provide an environment allowing the integration of complex linguistic techniques into question answering architectures in support of complex question answering. As such, it is useful to present a brief overview of current approaches to question answering (more detail is presented in Section 2.3.1). This will provide a framework within which an outline of the FLOOD system may be presented.

1.1.1 Question Answering

A common approach to complex question answering breaks the problem down into a number of stages [48]. These tend to proceed in the following general manner:

- Question analysis. This stage varies in its depth and complexity, but is commonly used to generate a collection of basic properties of the question: its general form ("what", "how many", etc.), its expected answer type (a phrase, a number, a list, etc.), potentially important keywords, and so forth [27].
- Candidate answer retrieval. Based on the keywords and answer types generated during question analysis, classical information retrieval techniques are applied to produce documents and/or passages containing potential answers. These bodies of raw text provide input for more complex answer generation.
- Information extraction. This is an extremely broad category, perhaps the most varied in modern question answering. It may be described as any technique attempting to pair candidate answers with the question analysis, and can range from statistics on raw tokens to full semantic processing and unification [16].
- Answer generation. A less well-defined aspect of question answering, this can be anything from simply cleaning up the results of information extraction to a complex control structure performing feedback analysis and iteration [28, 43]. In general, its purpose is to ensure that some candidate answers have been found by information extraction that fit the requirements of question analysis and to represent these answers in a consistent manner.

1.1.2 FLOOD

FLOOD consists of three distinct units, each of which will generally be discussed separately in this text:

- The Text Processor (TP). This module is responsible for analyzing text from a raw, unprocessed form up to a level in which it is sufficiently well-understood to be used be the rest of the system. This is done by providing a standardized environment within which existing individual parsers (sentence separators, part-of-speech taggers, etc.) are leveraged to produce linguistic information, which is then processed into a normalized form. Analysis begins at the level of unformatted text and, for the purposes of FLOOD, produces a collection of high-level outputs such as theta roles and entity tagging. The TP is constructed as an independent module which could easily be used for other applications outside of FLOOD.
- The Planning Platform (PP). The Planning Platform provides an environment within which linguistic information can be analyzed after it has been produced by the Text Processor. The PP itself contains no knowledge of linguistic information or question answering: it is purely a planning environment that has been constructed to be flexible and general enough for use in complex question answering. It provides a number of features of general interest for basic planning problems:
  - Abstraction. The PP is meant to provide an abstraction layer between planning algorithms and the data over which they operate. It provides basic services such as domain and state space management so that planning algorithms can be developed and used more simply and rapidly.
  - Object-orientation. All information within a planning domain is represented as a collection of strongly-typed objects and properties. This also means that integration with standard programming languages for application of planning algorithms in a larger context is quite simple.
  - Fluent data. For support of continuous planning algorithms [40], all data can be represented as fluent (non-binary, e.g. real or character data) values.
  - Dynamic objects. The PP supports dynamic object creation and deletion by plan actions to support creation of planning algorithms aware of these operations [71].
- The Question Answering Reasoner (QAR). This is a specific planning algorithm and domain developed for use within the PP for application in complex question answering. This encapsulates all of the knowledge that is specific to language or question answering and provides the core implementation capable of receiving a question and answer passages (in processed form) and generating candidate answers.

To contrast these three units with the standard components of a question answerer as described above, the Text Processor provides functionality similar to that of question analysis, and the Question Answering Reasoner operating within the Planning Platform serves as an information extraction layer. Thus, FLOOD can be thought of as a subcomponent of a question answering system which provides portions of question analysis and, later on, a list of candidate answers (as would an information extractor).

1.2 Goals

With this functionality in mind, several question answering applications of FLOOD are likely already evident. However, since many aspects of the system are modular, it is useful to consider each of the three subcomponents and their goals in a broader context.

1.2.1 Text Processor

- First and foremost, the Text Processor must accept raw text and produce a level of analysis sufficiently deep for further processing by the QAR. This includes, but is not limited to, theta role filling, entity tagging, and synonym expansion.
- The TP is not intended to implement parsing algorithms itself, but to integrate existing parsers of a variety of types. It must, however, provide a unified form for processed data (e.g. syntactic parses from one component parser must appear practically indistinguishable from those from another parser using them for later processing) and a unified interface by which component parsers may be integrated.

- The TP must be modular, allowing the addition of new component parsers with minimal effort. This includes parsers which provide identical types of analyses as existing parsers (e.g. tokenization, constituent parses, etc.) and, to a lesser extent, new but easily adaptable types of output.
- The TP must be arranged as a modular component usable outside of FLOOD. Its input, output, and runtime formats must not be tied to the overall FLOOD system.

1.2.2 Planning Platform

- The Planning Platform must provide as many of the universal requirements of a planning system (such as domain and state space management) as possible without imposing unnecessary restrictions on planning algorithms implemented within it. For example, the PP domain model allows fluent data to be used, but it does not require component planners to understand or utilize fluent data.
- The PP must support input from the Text Processor and output from the Question Answering Reasoner to achieve FLOOD's overall goal of complex question answering. This means that the platform must be broad enough to support any functionality needed by the QAR; in particular, its domain model must be rich enough to support the representation of the TP's linguistic analysis as used by the QAR.
- To provide adequate functionality as a modern planning environment for simple experimentation with planning algorithms, the PP must support full object-orientation (both in its domain model and in its interfaces with the external environment), fluent (non-binary) data in its domain model, and dynamic object creation and deletion in plan actions.
- The PP must provide an adequate environment for development, monitoring, and debugging of component planning algorithms (such as the QAR). This includes a graphical environment for runtime analysis of the planning process.
- Like the TP, the PP must be sufficiently modular to operate outside the context of the FLOOD system. This again means that its inputs, outputs, and runtime must not be tied to other components of the FLOOD system (although as stated, they must also be sufficiently general that the question answering requirements of FLOOD can be satisfied within their constraints).

1.2.3 Question Answering Reasoner

- The QAR must be developed within the Planning Platform, taking advantage of its domain representation and runtime environment. It must accept data based on the Text Processor output, analyze it, and produce results compatible with FLOOD's functionality as an information extraction system for question answering. Note that this does make the QAR specific to FLOOD and not generally useful as a modular component.
- For the purposes of this thesis, the QAR must produce interesting behavior on at least a toy-sized corpus of interest to complex question answering. In particular, it must produce novel correct answers over a small corpus (to be discussed below) not obtainable by an existing QA system.

1.3 Text processing

The Text Processor can be thought of as a meta-parser which accepts text and outputs a parse built up out of the results of various component parsers. For example, a single sentence provided as input might produce tokenization, part-of-speech tagging, and various syntactic and semantic parses as output, with each component of the output generated by sub-parsers whose outputs are collected and normalized by the TP. From the outside, the Text Processor can thus behave as a black box that provides exactly what its name implies; if an application (such as FLOOD) needs textual analysis, it feeds raw text into the TP

and receives a collection of structured information as output. From within, the TP is a modular pipeline passing increasingly varied information from one modular sub-parser to the next, each adding its own data and potentially using the output of earlier processors. The TP is constructed to represent a variety of information in a normalized form:

- Document, paragraph, sentence, and token separation
- Per-token annotations such as part-of-speech, entity tagging, and morphological analysis
- Constituent and functional syntactic parses
- Semantic parses based on argument structure and theta roles
- Arbitrary additional information can be added in the form of structured properties; any component of the TP output may have zero or more named properties attached to it, each of which can contain scalar data or additional child properties

1.4 Planning

Unlike most tools currently available for planning, the Planning Platform does not represent a particular planning algorithm [14, 51, 64, 2, 72, 45]. Instead, it is a collection of utilities necessary for planning intended to make implementation of and experimentation with planning algorithms as simple and convenient as possible. The PP provides a tool which can be used by applications (again, such as FLOOD) which need reasoning (like text analysis) as a component. For example, FLOOD requires reasoning over processed text. To do this using the Planning Platform, it implements a specific planning algorithm and domain (the QAR), which it then loads using the PP. It provides input based on the output of the Text Processor and then instructs the Planning Platform to execute the loaded planning algorithm. The PP is thus an interface and runtime for planning algorithms and the systems which use them. As an example, services which it provides include:

- A standardized domain representation; this can load a planning domain from a variety of file formats and provide it to a planning algorithm in normalized form
- A standardized state space representation, initially containing objects and data as loaded from the domain and adapting them as plan actions are executed
- A planning runtime which allows a planning algorithm to incrementally generate new plan steps or to select and execute actions
- A programmatic interface for integration of reasoning systems into external applications without any changes necessary in planning algorithm implementation; this provides an interesting opportunity for modular interchange of planning algorithms
- A graphical interface to facilitate development and debugging of planning algorithms and domains

1.5 Question answering

The Question Answering Reasoner represents a specific planning algorithm and domain implemented within the Planning Platform. In some sense, this represents the core of FLOOD as it applies to complex question answering; from a high-level perspective, the QAR is the portion of FLOOD which accepts questions and generates answers. The planning algorithm of the QAR is implemented as an augmented version of the FLECS algorithm [65], since this proved to be quite appropriate for both the form of the Planning Platform and the requirements of complex question answering. The flexibility of the PP allows this basic planning algorithm to be augmented with details specific to question answering and to be integrated with an accompanying QA domain. Since FLOOD must operate on open domain questions, the

domain of the QAR contains representations of objects which generally correspond to common question and answer types: people, places, entities, and so forth. To facilitate more complex question answering, it also contains representations for subquestions and general relationships between entities. Given the Planning Platform's emphasis on object orientation, this domain could easily be adapted through subclassing to provide domain-specific knowledge.

Thus, the QAR can be thought of as a combination of two entities: a planning algorithm based on FLECS and customized for question answering, and a planning domain containing classes, objects, and actions appropriate for QA. These are loaded by FLOOD through the PP, augmented on a per-question basis with question and passage information as analyzed by the TP, and executed by the PP to produce candidate answers for the given question.

2 Background

The concept of FLOOD originally arose during work towards a proposal for the HALO [70] advanced question answering project. HALO, sponsored by Vulcan Inc., is an ongoing effort to develop knowledge-based systems capable of answering novel questions given an initial set of information in which the answers are not explicitly present. For example, the HALO pilot program provided several chapters from a chemistry textbook and challenged participants to develop a system capable of answering basic stoichiometry and acid/base questions - none of which were present verbatim in the text.

A moderate amount of background research was performed during the proposal submission phase of the HALO pilot, and part of this effort was directed through the PRODIGY planning architecture [64]. It rapidly became clear that an extremely broad platform would be required to address the difficult issues raised by HALO, and it was assumed that the rich history of PRODIGY would provide sufficient resources for a prototype reasoning system. Several issues arose, however:

- With respect to systems and integration, PRODIGY represents something of a legacy issue. Its codebase has been minimally updated for the past decade, making it extremely difficult to interoperate with other software developed for HALO.
- The most common ways of addressing stoichiometric problems all require the supposition of intermediate formulas; likewise, acid/base problems tend to suppose intermediate reactions or solutions. This led to the concept of creating and then deleting entities (dynamic object creation and deletion as discussed above), for which PRODIGY provides essentially no support.
- While stoichiometry is often discrete, acid/base and other equilibria are inherently continuous. Maintaining fluent data in PRODIGY is also extremely difficult.

These impediments encountered in initial experiments with PRODIGY proved to be representative of most existing large-scale planning systems. They provided a foundation for the features of the FLOOD Planning Platform.

Once work with HALO was completed, development of FLOOD continued within the JAVELIN [48] question answering system. JAVELIN is an end-to-end question answerer with a focus on complex question types (relationships, scenario-based dialogues, etc.) Investigation of FLOOD's usefulness within JAVELIN coincided with a set of general investigations into requirements for complex question answering in JAVELIN. These eventually led to the features which make up the FLOOD Question Answering Reasoner.

Similarly, JAVELIN has historically used only shallow parsing techniques to answer (primarily factoid-based) questions. To further JAVELIN's complex question answering in general, and to integrate FLOOD into its question answering pipeline, more sophisticated linguistic analysis was necessary. The TP thus developed both to generate input for FLOOD from JAVELIN's intermediate data and to provide the rest of the JAVELIN system with various types of textual analysis.

2.1 Significance

Most of the goals of the FLOOD system focus on development of useful components which are otherwise absent from the research community. The Text Processor, for example, represents a modular, end-to-end parsing system. While any number of parsers performing specific tasks have been developed for research purposes, most of these are highly focused and perform merely a single step of the analysis necessary to convert raw text into a fully analyzed form. Similarly, the Planning Platform provides a tool of intermediate complexity between toy planning implementations and highly intricate and specific planning algorithms. On one hand, the focus of this thesis is to fill these gaps in the overall research environment to allow parsing and reasoning to be used as more convenient, modular units in future work.

On the other, the goal of FLOOD is to advance complex question answering by providing a proof-of-concept for the use of deep parsing and reasoning through planning for QA. Applications of planning to QA have previously been limited to control flow [47]; FLOOD revisits the concept in a new manner by using a planner to manipulate the linguistic content itself. This provides a largely unexplored environment almost ideally suited to the challenges of complex question answering; domain knowledge, scenario context, and inference can all be easily adapted to the framework of a planning domain. Likewise, the Text Processor brings a level of linguistic analysis to the QA process which has to date been utilized only minimally (but with great success) [16, 43].

Thus, the overall goals of FLOOD are to advance current work in complex question answering by the application of deep linguistic analysis and planning and to aid future work in any field requiring parsing or planning as experimental components.

2.2 Challenges

As with all other aspects of the FLOOD system, each of the three primary subcomponents presents its own unique difficulties. Overall, FLOOD's biggest challenge is perhaps one of integration: how may three largely independent modules best be merged and leveraged to provide an increase in question answering capabilities? Once this is achieved, how may FLOOD in turn be integrated into a full question answering system such as JAVELIN? These practical questions must be addressed in tandem with the generally more theoretical issues faced by the subcomponents implementing FLOOD's overall functionality.

2.2.1 Text processing

The Text Processor represents the most traditional aspects of the system; all types of parsing which it addresses, from sentence separation to semantic analysis, have been studied in great depth. The focus for the Text Processor is one of normalization: how best to represent linguistic data and the process of parsing so that they may both be used constructively by client programs. Additionally, like FLOOD as a whole, the TP has its own challenges of integration. The fact that its subparsers have generally been quite well-studied means that they exist in a variety of forms and implementations, some of which tend to be quite eccentric. Just as a theoretical challenge for the TP is to specify a unified representation for this data, it is a practical challenge to collect it from parsing implementations themselves.

2.2.2 Planning

Planning as a field has also been studied in extensive depth, and it is not a goal of the FLOOD Planning Platform to produce significant new information in this area. Rather, it is the responsibility of the PP to integrate what is already known about planning in theory and present it in a practical, useful manner for use by planning algorithm implementations. The PP must support this vast body of existing planning knowledge and integrate it into a modern platform for planner development in such a way as to provide newly developed algorithms with maximum flexibility and minimum overhead. In the context of FLOOD, it must specifically provide the support and capabilities necessary for use by the QAR.

2.2.3 Question answering

The Question Answering Reasoner presents perhaps the most obvious difficulties for FLOOD: this is the deepest layer of the system, and upon it falls the responsibility of accepting questions and producing end results in the form of candidate answers. The QAR must operate within the framework of the PP and, for integration with JAVELIN, it must interact with an external question answering system correctly when providing answers or presenting subquestions. From a planning perspective, and interesting key requirement is controlling the runtime search space. Since much of the QAR's input is automatically generated by the TP and several of its actions (e.g. subquestions) are extremely expensive, care must be taken as to which actions are executed during planning and in what order.

2.3 Previous work

As with other aspects of FLOOD, related previous work can most easily be presented as it applies to each of the three main constituents. As a whole, the most applicable related systems are those which perform other types of integration of linguistic information for question answering. The LCC system [26, 43], for example, uses a type of shallow semantic parse over which a sophisticated theorem prover is run. This provides a similar level of analysis as is generated in FLOOD - it is somewhat less descriptive and far more rigorous - but fails to provide the level of flexibility yielded by FLOOD's combination of analysis and planning. Similarly, several other recent question answering systems have integrated semantic information in a variety of ways, generally also focusing on rigorous (and thus somewhat restricted) manipulation of shallow semantic data [39, 1, 75]. This may be contrasted to FLOOD's intended use within JAVELIN [47], which focuses of flexibility and modularity at the potential expense of algorithmic rigor.

2.3.1 Question answering

The foundations of modern question answering were laid by some of the earliest work in natural language processing - or, for that matter, in computer science as a whole. Systems such as SHRDLU [73], BASEBALL [23], and particularly LUNAR [74] all represent early implementations of question answerers, some of them quite sophisticated even by modern standards. Their primary restriction lay in their limited domains; LUNAR provided an excellent representation of lunar geology, but was quite incapable of analyzing questions outside of this subject. These systems were of key interest to early researchers, spawning over a dozen variations as early as 1965 [56]. Question answerers of this original type, analyzing relatively limited questions over specific domains, provided the basis for modern systems such as QUALM [38], which is based on reading comprehension of specific paragraphs, and various database query front ends [30, 53, 62].

Open domain question answering as it is currently understood had its genesis in the Message Understanding Conferences during the early 1990s [24]. These were based on an early form of information extraction, focusing on the location and understanding of particular entity types (people, places, etc.) within bodies of text. This task at first tended to be approached in a purely template-based manner [36, 37], with later systems slowly incorporating more linguistic analysis [58]. Simultaneously, the Text Retrieval Conference (TREC) began as a standardized evaluation of information retrieval techniques [44]. This combination eventually developed into what is currently perhaps the best known large-scale question answering evaluation, the TREC question answering track, initiated at TREC-8 in 1999 [66]. Subsequent conferences have continued to demonstrate a larger and more sophisticated collection of question answering systems, as well as expanding from simple entity discovery to more and more complex types of questions [67, 68, 69].

Current state-of-the-art question answering systems take a variety of approaches. The LCC system [43], which has performed highly in recent TREC evaluations, uses a largely logic-based approach relying on a combination of entity extraction, syntactic parsing, shallow semantic parsing, and sophisticated feedback loops to evaluate performance. IBM's question answering system [28], on the other hand, implements a largely statistical approach based on supervised training of question/answer pairs using an interlingua-based intermediate representation. JAVELIN [48] itself, the QA system of primary importance for FLOOD,

implements a modular architecture in which individual subcomponents are controlled by a planner, providing more holistic performance.

2.3.2 Parsing

The body of previous work relating to parsing is far too extensive to recap adequately here, but an attempt will be made to review each of the primary services addressed by the Text Processor. Each parser used in the TP tends to encompass a particular subfield of natural language processing as described below.

Sentence separation Sentence separation by maximum entropy classification over terminal punctuation is implemented in the MXTerminator system as described in [54]; this implementation is currently used by the Text Processor. Other approaches include neural network classification of punctuation in combination with part-of-speech tagging [49] and tagging terminal and non-terminal punctuation as separate "parts-of-speech" [41].

Tokenization The TP currently uses a simple rule-based tokenizer as originally implemented for the LaSIE system [20]; this is a minor enhancement over pure whitespace tokenization based on nearby characters (punctuation, etc.) that has the advantage of being extremely fast. More complex approaches include finite-state methods as described in [10].

Part-of-speech tagging The CLAWS system as originally described in [35] is currently used for part-of-speech tagging within the TP; this is particularly advantageous given its probabilistic output and extremely rich tag set. Other approaches include transformation-based methods such as [7], which may be included in the TP in the future.

Morphological analysis Morphological analysis has not yet proved to be a critical component of the Text Processor, and the simple rule-based approach described in [42] have to date proven effective. This again has the advantage of being extremely fast without the simplistic drawbacks of [52], while avoiding the greater performance costs of more advanced two-level methods [33].

Named entity tagging BBN IdentiFinder, a well-known commercial named entity tagger, is currently used by the TP for open domain entity tagging [3]. This provides extremely broad coverage at some expense of accuracy; more specialized methods such as maximum entropy classification over textual features [6] or neural network noun phrasing [63] have also been considered for future use.

Syntactic parsing This is perhaps the broadest field covered by a Text Processor component. Currently, the TP uses the RASP statistical parser [8] and the dependency-based Link grammar parser [57] to provide two very different types of constituent and functional parses. These were chosen to provide a balance of breadth, robustness, coverage, and depth of parsing. Other statistical approaches are described in [11] and [13], and a largely rule-based system in [61].

Semantic parsing One aspect of semantics not directly related to parsing is sense disambiguation, in part addressed by the use of WordNet [18] within the Text Processor. In addition to this information, more complex semantic parses are provided by integration with the FrameNet database [29] and, to a lesser extent, the Lexical Concept Structure lexicon [15]. The former is a relatively straightforward enhancement of standard theta roles based on large-scale text annotation, while the latter is a lexically-driven specification intended largely as an interlingua for machine translation. Other frame-based methods such as PropBank [31] have also been considered as future additions to the TP.

Most other research systems for full end-to-end parsing focus on speech rather than written text [21, 34]. Similar problems have arisen in other fields, however, leading to similar approaches for integration of subprocessors into a single normalized system [12].

2.3.3 Planning

Planning algorithms have a long and rich history. Planning as a field began in the late 1960s and early 1970s, including as perhaps the first concrete example the General Problem Solver [46]. This development culminated with the introduction of the STRIPS [19] algorithm. The limitations of this algorithm were quickly apparent; the representational capabilities of its domain model were extremely limited (operating only within a closed world, for example), operator representations were similarly restricted, and, most importantly, it did not take advantage of partial order planning. Incremental improvements to address this deficiency led to ADL [50], which introduced a representation for quantification, and the eventual development of partial order planners such as UCPOP [51] and PRODIGY [64]. Many algorithms relating to adaptation and learning derived from the PRODIGY algorithm, including the FLECS [65] algorithm. FLECS is uniquely suited to many of the features captured by the FLOOD planning platform: it deals explicitly with choice between plan extension and simulated action execution, which (like PRODIGY) makes up the core of FLOOD's planning process.

The FLOOD Planning Platform is based most directly on the PRODIGY 4.0 framework [64]. PRODIGY represents a mature planning algorithm integrated into a full platform for experimentation, while the PP is only a framework at the prototype stage, but many of the founding concepts for the Planning Platform arose from PRODIGY. While PRODIGY represents one of the broadest planning platforms currently available, modern planning algorithms tend to be dominated by small and fast systems such as MIPS [17] and Graphplan [5]. These, however, tend to represent specific planning algorithms rather than full planning environments. A more similar project is I-X [60], which originally began as O-Plan [14]. I-X aims to extend and modernize O-Plan in much the same way that the Planning Platform is a modernized shadow of PRODIGY, although in the case of I-X there is a strong focus on planning for operations management. Much of I-X is still under development.

3 Methods

The FLOOD system was implemented almost entirely in Java using the Eclipse IDE[1]. This promoted ease of development and future extensibility at some expense in performance; where necessary, certain submodules were implemented in C++ for purposes of interoperability or performance. If necessary, this could easily be replicated for increased performance in other modules in the future.

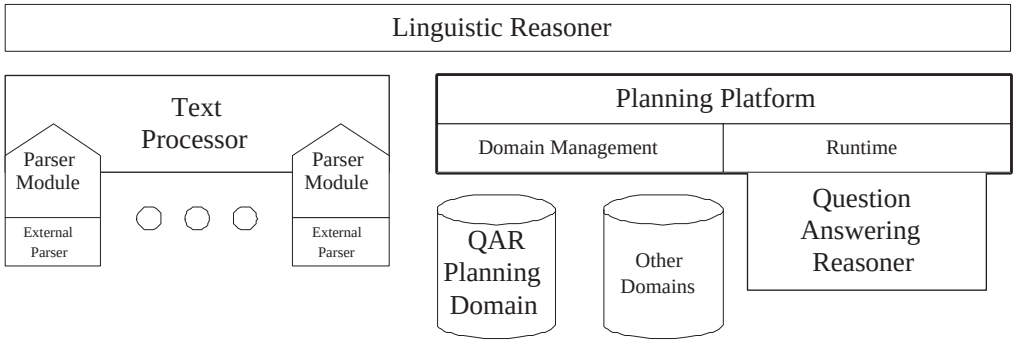
An overview of the entire FLOOD system architecture and its relationship to the JAVELIN QA system can be seen in Figure 1. For purposes of interoperation with JAVELIN, a thin layer referred to as the Linguistic Reasoner (LR) controls access to the PP (and thus the QAR) and translates output from the TP into a form useful by the QAR. The TP and the LR both communicate with external clients through an XML interface (discussed in detail below), and the LR passes input into and receives output from the PP/QAR through the Planning Platform's own programmatic APIs.

3.1 Text processing

As described in the overview above, the TP can be though of as a pipeline which accepts raw text, annotates it with increasingly complex linguistic analyses, and outputs a structured parse containing a hierarchical XML representation of this added data. There are three main groups of entities relevant to this process:

[1] http://www.eclipse.org

Figure 1: High-level representation of the overall FLOOD architecture



- The control structure. This is a network server which accepts incoming requests for parsing, runs the appropriate sub-parsers, and returns the results to the client.
- Data elements. One of the primary purposes of the TP is to format many types of linguistic data into a single, unified format. To achieve this, it specifies a rigorous set of data elements representing linguistic analysis results at every level of detail, from paragraph separation to semantic role filling.
- Parsers. Again, to achieve this level of normalization, the TP also defines a strict interface to which component parsers must conform. This means that existing parsers can be controlled in a simple and uniform fashion while new ones can be added with minimal effort.

3.1.1 Assumptions

At a high level, the goal of the TP is to make several small, stepwise parses look like one large, end-to-end parser. To do this, it must force interoperability in two main ways: through runtime controls for parsing modules and through a normalization framework for data. Each of these requirements imposes limitations on the processors operating within the TP; they must be able to deal with the pipelined flow control of the TP's runtime behavior, and they must be able to convert from and to the formats provided by the TP data elements (as described in Section 3.1.3). A brief discussion of the ramifications of these restrictions - both good and bad - follows below.

Control structure restrictions A TP parsing module is guaranteed to be called at most once per parsing request. It is currently the responsibility of the caller to guarantee that data dependencies are satisfied (e.g. that a sentence separator is called before a tokenizer), but this is not inherent to the framework. A parsing module will always be provided with all data currently available in the hierarchy built up during TP processing, although it will be "focused" on the data requested by the module's input specification. Finally, while dependencies on properties (non-typed, optional hierarchy members) are discouraged, they are possible when necessary (e.g. for morphological data).

Few of these restrictions, if any, should have any effect on component parsing modules. It is the intent of the TP to constrain its modules mainly in their data formatting, providing a homogeneous output, rather than in their runtime operation. The major limitation of the current TP runtime implementation is that it prohibits parallelization of components within a single parsing request, which is sometimes possible (for example, if syntactic parses from two different, independent parsers are requested). This is not necessarily inherent in the framework, but it would be nontrivially difficult to work around it given the current implementation.

Data element restrictions The biggest restriction imposed by the TP is easily the uniformity of its data model. In a true end-to-end parser, the components and the data model would be interdependent, and each

could adapt as necessary to the requirements of the other. Since the TP integrates multiple heterogeneous data sources, however, each module must be able to adapt its inputs and outputs to the data structure required by the TP as a whole. While the Text Processor is arranged to be as general as possible, making this restriction as minimal as possible, it is also necessary to impose certain requirements on the final form of the data to allow uniform input, output, and processing.

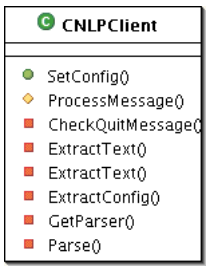
The main elements of the TP data hierarchy follow patterns drawn from the general body of natural language processing techniques. The constituent structure of a syntactic tree, for example, will almost certainly follow the patterns required by the TP framework no matter what its programmatic source. Like XML, the TP's data structure is intended to be as general as possible - but also like XML, certain data types will eventually fail to fit conveniently into the provided structure. The property data type originally arose to cover these circumstances; at worst, any TP processor can encode private data as untyped, structured properties, allowing even the most restricted tools to make some use of the framework.

3.1.2 Control structure

The control structure of the TP is captured by the *CNLPClient* class. A UML description of this class can be seen in Figure 2. The responsibilities of this class are relatively straightforward:

- It accepts incoming parse requests, consisting of a piece of text and a list of desired parsers by which this text should be analyzed (both encoded as XML).
- It runs the requested parsers sequentially based on the client's request.
- It returns the resulting data elements to the client (also encoded as XML).

Figure 2: *CNLPClient* UML digram



The overall behavior of this class can be configured by an XML property file as specified by Apple Computer (see Appendix E). This includes elements such as the network port and which parsers to run by default (see the input XML specification below for details).

Input XML Requests to the TP are typically made with XML of the form shown in Figure 3. The indentation flag is provided simply for cosmetic formatting, and each element in the list of parsers corresponds to a named sub-parser as described below. If a partial parse is already available, any serialized TP data element may be substituted for the raw text block in the request and the requested parsers will begin operation with the given object(s) instead of simple text.

3.1.3 Data elements

Each data element in the Text Processor descends from either one or both of two key interfaces, *INLElement* and *INLCollection*. The latter defines nearly all of the functionality required of a TP data element, while the former inherits from it and adds the minimal requirement that its implementations correspond to some unit

Figure 3: Format of a parsing request as sent to the Text Processor

```
<parse>
   <options>
      <key>indent</key>
      <true/>
      <key>parsers</key>
      <array>
         <string>mxterminator</string>
         <string>rasptok</string>
         <string>claws</string>
         <string>morpha</string>
         <string>rasp</string>
         <string>bbn</string>
      </array>
   </options>
   <text>Please parse this sentence.</text>
</parse>
```

within an original body of text. This difference can be seen, for example, in contrasting a token (an *INLElement* which typically corresponds to a single word in a body of text) with a tokenization (an *INLCollection* which simply contains zero or more tokens).

Below these two primary interfaces sit two corresponding base implementations (*CNLElement* and *CNLCollection*) in addition to a host of classes implementing specific data types. The inheritance structure of these classes can be seen in Figure 4. More useful, perhaps, are their theoretical relationships, presented in Figure 5. The TP models data during processing as a tree of *INLCollections*; the root of the tree is a *CText* element originally containing only a block of text. Child elements are added in a descending order as described in Figure 5; each parser may add either child elements or property annotations to existing elements in this growing tree. Thus, each element in the tree depends only on the information in its ancestors; a tokenization, for example, depends on a sentence separation, but two functional parses from different sources are independent. After all requested parsers have been run, data is returned to the client by simply serializing the root element of the tree, which now includes as children any data elements and properties generated during parsing.

As can be seen in the XML fragments below, each data element is given a unique numerical ID and annotated with a source (the name of the parser which created it) and zero or more properties (special data elements described below). In any XML emission when the full body of an element has already been serialized, it may be abbreviated using only its first line and ID number; this greatly reduces the size of serialized output in addition to making cyclic references safe. An example of the complete output for a single TP parse can be found in Appendix C; note that it is made up of a simple accumulation of the XML fragments shown for each data type below.

CText A simple string of text; contains zero or more *CPassage* children.

CPassage Minimally structured block of text analogous to a paragraph or passage; contains zero or more *CSentence* children.

CSentence A single sentence; contains zero or more *CTokenization* children.

Figure 4: High-level TP UML class diagram

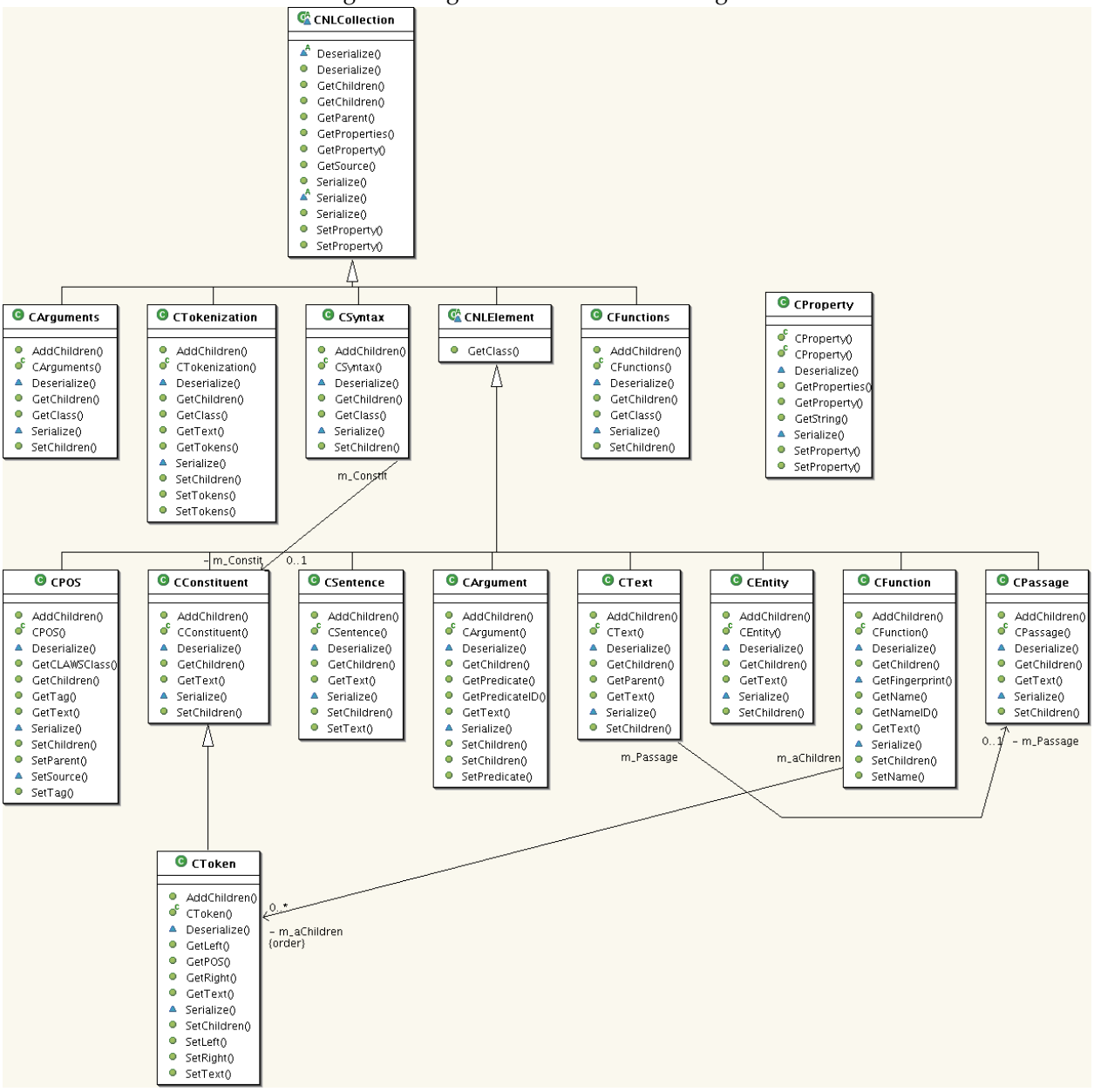


Figure 5: Conceptual relationships of TP data elements

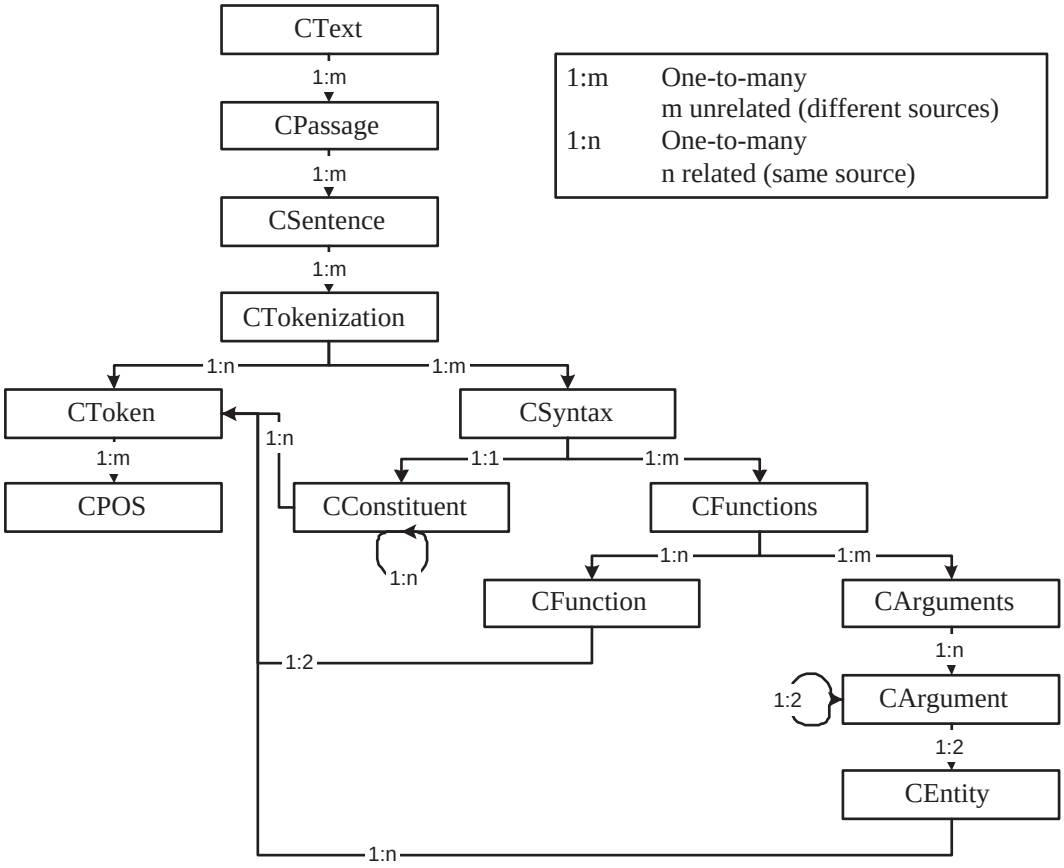


Figure 6: *CText* XML

```
<text id="[#]" source="[str]">
    [properties]
    [text]
    [children]
</text>
```

Figure 7: *CPassage* XML

```
<passage id="[#]" source="[str]">
    [properties]
    [children]
</passage>
```

Figure 8: *CSentence* XML

```
<sentence id="#" source="[str]">
    [properties]
    <text>[text]</text>
    [children]
</sentence>
```

CTokenization A container element holding a list of zero or more tokens; contains these *CToken* elements as children in addition to zero or more *CSyntax* elements.

Figure 9: *CTokenization* XML

```
<tokens id="#" source="[str]">
    [properties]
    [token children]
    [syntax children]
</tokens>
```

CToken A single word (e.g. "car") or multi-word entity (e.g. "log in"); contains zero or more *CPOS* children. Token boundaries are numbered within a sentence beginning at zero, so the leftmost token begins with a left index of 0 and a right index of 1. This allows disambiguation of overlapping multi-word tokens.

Figure 10: *CToken* XML

```
<token id="#" source="[str]">
    [properties]
    <left>[#]</left>
    <right>[#]</right>
    <poss>[children]</poss>
    <text>[text]</text>
</token>
```

CPOS A part-of-speech containing no children. These could conceivably be annotated as simple properties, but promotion to a full data element allows for convenient notation of multiple parts-of-speech on the same token with different probabilities (which are themselves annotated as properties). The current implementation of *CPOS* is somewhat customized for the CLAWS tag set, but this could easily be rearranged to incorporate either richer or simpler tag sets.

CSyntax A container element containing as children a single, optional constituent parse (a tree rooted with a single *CConstituent*) and zero or more functional parses (each a *CFunctions* element).

CConstituent Represents a single node in a constituent tree; each node has zero or more *CConstituent* children, with constituent types provided as property annotations (e.g. "NP", "VP", etc.) Note that *CToken* elements are also considered to be *CConstituent* elements, such that the leaf nodes in a constituent tree will always be tokens.

Figure 11: *CPOS* XML

```
<pos id="[#]" source="[str]">
    [properties]
    [tag]
</pos>
```

Figure 12: *CSyntax* XML

```
<syntax id="[#]" source="[str]">
    [properties]
    [constituent child]
    [functions children]
</syntax>
```

CFunctions A container element (much like *CTokenization*) holding a set of zero or more *CFunction* elements in addition to zero or more semantic parses (*CArguments* elements).

CFunction Represents a single syntactic functional relation within a sentence (e.g. the subject of "ate" is "Mary"); contains either two or three *CToken* children, the latter in cases with headed phrases (e.g. "of" in a prepositional phrase or "that" in a relative clause). Each function is given one of approximately 25 possible types as outlined in Table 1; these were drawn from a variety of sources and arranged to be as general as possible without losing specificity.

CArguments A container element holding a set of zero or more *CArgument* relations.

CArgument A single semantic argument (or theta role) within a sentence (e.g. the agent of "ate" is "Mary"); contains exactly two *CEntity* children in addition to a predicate as described in Table 2. In the case of a ROOT predicate, the second *CEntity* child is replaced by a *CToken*. Like the function types discussed above, these predicate types were collected from a variety of sources to provide a balance of flexibility and specificity. Note that as with most theta role systems, these are not mutually exclusive; a PATIENT may also be a DESTRUCT, etc.

CEntity This is currently used to tie individual tokens to arguments, but it is intended for use with reference resolution algorithms to map sets of tokens to single entities; it contains one or more *CToken* children.

CProperty A property is neither an *INLCollection* nor an *INLElement*, but a special dictionary-like data structure used to attach additional, non-essential information to data elements. Each element may have

Figure 13: *CConstituent* XML

```
<constituent id="[#]" source="[str]">
    [properties]
    [children]
</constituent>
```

Figure 14: *CFunctions* XML

```
<functions id="[#]" source="[str]">
    [properties]
    [function children]
    [arguments children]
</functions>
```

Figure 15: *CFunction* XML

```
<function id="[#]" source="[str]">
    [properties]
    <name>[function type]</name>
    <head>[third child]</head>
    [first two children]
</function>
```

Table 1: *CFunction* types

Name	Description	Example
NULL	Default; no function	
QUANT	Quantification of a determiner	All the dogs were there.
SMOD	Special modifier (idiomatic, etc.)	We don't go out as much now.
NMOD	A noun-noun modifier	We used a garden hose.
PMOD	Prepositional phrase	The dog in the yard barked.
OBJ	The direct object of a verb	I patted the dog's head.
XCOMP	A complement lacking a subject	The dog was in the yard.
QOBJ	Object of a question	He asked what she said.
COMPL	A complement containing a subject	The dog barked when he saw me.
CONJ	A conjunction	I laughed and the dog barked.
VMOD	An adverbial modifier	The dog barked loudly.
CCOMPL	A comparative complement	The dog likes bacon more than I do.
DET	A determiner	The dog barked.
NMWE	A noun-noun compound	The dog is named Horatio Jones.
INF	An infinitive verb (containing "to")	The dog wanted to chase the car.
AUX	An auxiliary verb	The dog might like some bacon.
MWE	An idiom or other multi-word entity	He is the head of state.
RELCL	A relative clause	The dog who chased me was big.
NEG	Negation marker	The dog is not done playing yet.
SUBJ	Subject of a verb	The dog barked.
POSS	Relates a possessive marker and phrase	That is the dog 's bowl.
MOD	Adjectival modifier	The big dog barked.
PART	Verb particle	The dog stood up.
COMMACL	A subclause inserted using commas	The dog, the owner of whom I know, is big.
SUBJCL	A relative clause with an implied subject	I saw the dog mentioned in the paper.

Figure 16: *CArguments* XML

```
<arguments id="[#]" source="[str]">
    [properties]
    [children]
</arguments>
```

Figure 17: *CArgument* XML

```
<argument id="[#]" source="[str]">
    [properties]
    <predicate>[predicate]</predicate>
    [children]
</argument>
```

Table 2: *CArgument* predicates

Predicate	Description	Example
ROOT	Ties a semantic entity to one or more tokens	
CONJUNCTION	Joins a phrase and a conjunction	I run and the dog barks.
MEMBER	Indicates membership in a set	I categorized the book as fiction.
AGENT	The conscious performer of an action	I categorized the book as fiction.
THEME	An unchanged undergoer of an action	I categorized the book as fiction.
GOAL	The end location of a path	I went to the store.
INSTRUMENT	An object used to perform an action	I hit the nail with the hammer.
LOCATION	The physical location of an action	I ate in the cafeteria.
POSSESSION	Indicates ownership	I ate Joe's apple.
SOURCE	The start location of a path	I walked from home to the store.
BENEFACTOR	Entity for whom an action is performed	I gave an apple to Joe.
INFORMATION	Data transmitted by a informative action	I said hi to Joe.
PROPERTY	A transient modifier of an entity	I thought the concert was nice.
PURPOSE	The desired outcome from an action	I ate to avoid starvation.
PREDICATE	A permanent modifier of an entity	My given name is Curtis.
PARTICLE	A verb particle	I suddenly woke up.
EXPERIENCER	The perceiver in an informative action	I saw a beautiful sunrise.
PERCEIVED	The perceived in an informative action	I saw a beautiful sunrise.
TIME	The time of an event	I woke up at 9:00 AM.
MODIFIER	A transient modifier of an event	I ate quickly.
MANNER	The way in which an event proceeds	The show finished with a bang.
PATIENT	The modified undergoer of an event.	I ate an apple.
DURATION	The length in time of an event	I ate for three hours.
CONSTRUCT	A non-focus object created by an event	She mixed them into a potion.
PATH	The location through which an event proceeds	I walked home through the woods.
ENVIRONMENT	The non-locational surroundings of an event	I cooked the onions in butter.
REASON	The initial motivation for an event	I ate because I was hungry.
ACTOR	The non-conscious performer of an action	The rain fell from the sky.
DESTRUCT	A non-focus object destroyed by an event	I replaced the curtains with blinds.
DEGREE	The amount or severity of an event	The temperature fell three degrees.
LOCALIZATION	An additional specifier added to a location	I ate at the table in the kitchen.

Figure 18: *CEntity* XML

```
<entity id="[#]" source="[str]">
    [properties]
    [children]
</entity>
```

zero or more properties; each property consists of a name and either a piece of scalar data (a string) or a list of zero or more properties. This simple structure allows for extremely detailed data to be attached when it might not otherwise fit into the TP's data element hierarchy. For example, properties are used to provide probabilities for part-of-speech annotations and to describe WordNet synset hierarchies for particular verbs and nouns.

Figure 19: *CProperty* XML

```
<property name="[name]" source="[str]">
    [text]
or
    [children]
</property>
```

3.1.4 Parsers

Each parser is an implementation of the *INLProcessor* interface. Simply implementing this interface imposes three key constraints:

- The parser must provide a name. This is a simple string which must not be identical to the name of any other parser currently in use by the TP.
- The parser must describe what type of element it consumes. This is the most specific type of data it will require, i.e. the lowest level of the TP element hierarchy which it will use. For example, a tokenizer will consume sentences, while a semantic parser will (most likely) consume functional parses. Even if a parser utilizes more than one type of element during processing, it should specify only the single deepest element in the TP hierarchy that is needed, reading other element types from this bottommost element's ancestors.
- The parser must provide a list of element types which it produces during parsing. As opposed to the single consumed element type, this list should include any element at any level of the TP data hierarchy which will be produced by this parser. If a parser only produces properties and attaches them to preexisting data elements, this list may be empty. Note that a processor should not describe inconsistent output types (i.e. ones that skip levels in the element hierarchy); if a processor consumes a sentence and outputs a constituent parse, it should also output a tokenization.

Parsers are added to the Text Processor by implementing an *INLProcessor* wrapper around their core binaries. This can be done in any manner - each parser's Java implementation of *INLProcessor* might call directly into other Java code, launch a child process and read from standard output, or communicate over a network connection. The Text Processor is completely unaware of this process and requires only that the parser attach its output into the growing data element hierarchy. Each currently implemented parser is briefly described below.

bbn This parser wraps BBN IdentiFinder [3] to provide named entity (NE) annotations on a per-tokenization basis. Named entities may span multiple tokens, so each tokenization is analyzed separately and annotations are added as properties with a particular NE type (as defined in [3]) and token span (using the token indexes described above). Communication is performed over a network socket using IdentiFinder's native network communications protocol.

claws This parser is an interface to the CLAWS part-of-speech tagger [35]. The CLAWS tag set was adopted nearly verbatim for the normalized representation used in *CPOS*, so little normalization of the parser output is necessary. The parser consumes the text of a *CTokenization* and attaches zero or more *CPOS* elements to its constituent *CToken* elements, each annotated with a probability property. Communication is performed using standard input and output channels with the CLAWS binary launched as a child process.

framenet This parser is a relatively complex interface to the data contained in the FrameNet database [29]. Since FrameNet itself is merely a large corpus of sentences annotated with frame-based semantic relations, it is the responsibility of the parser rather than the tool to perform role filling for novel sentences. This is done using a combination of heuristics:

- The FrameNet corpus is preprocessed using the TP's Link parser (described below) to produce functional parses. Each sentence's semantic annotations are aligned with its functional parse to produce a mapping from a semantic annotation to one or more functional relations. For example, in the sentence, "I ate an apple," the functional relation "SUBJ ate I" would be aligned with the FrameNet annotation "Ingestor ate I".
- FrameNet captures semantic roles as elements within frames, each frame containing one or more words. For example, the "Ingestion" frame contains elements such as "Ingestor" or "Ingestibles" and corresponds to words such as "eat" or "gobble". The TP's framenet parser maps each combination of frame and element onto one of its standardized semantic roles. So for example, "Ingestor ate I" becomes "AGENT ate I".
- When a new sentence is encountered, its functional parse elements are organized into a directed graph. Each node in the graph corresponds to a word (one of the first two children of a *CFunction*) and edges are optionally labeled (with the third child of the corresponding *CFunction*, if present). Each node's word is retrieved from FrameNet using any available part-of-speech information; if a successful lookup is performed, the functional parses added to the lexicon by preprocessing are unified against the functional information in the new sentence currently being processed.
- For any successful unification, the semantic annotations (and thus the *CArgument* roles) corresponding to the lexical functions are attached to the new, unannotated functions. All possible combinations of functional-to-semantic mappings are considered (this remains tractable because of the relative sparsity of FrameNet) and a score is assigned to each based on A) the number of "core" frame elements covered by the mapping and B) the number of identical functional-to-semantic mappings present in the lexicon. This scoring algorithm is described explicitly in Algorithm 1.
- The highest scoring mapping is retained and its component semantic roles added to the TP's data hierarchy as *CArgument* elements.

Communication with FrameNet is performed by another layer of preprocessing which digests the XML-encoded database into a more compact, binary form containing the same data. This binary representation is loaded by the TP's framenet parser at runtime and used to extract the functional patterns, frame descriptions, and counts necessary for performing the tasks listed above on a per-word basis.

Algorithm 1 FrameNet scoring algorithm

```
Score( Mapping )
   If this is a hard-coded mapping with no FrameNet entry
      Assign Mapping a default score of 1
   Else
      Let T_M be the total number of frame elements assigned by Mapping
      Let T_E be the total number of frame elements in Mapping's entry
      Let C_M be the number of core frame elements assigned by Mapping
      Let C_E be the number of core frame elements in Mapping's entry
      Let P_M be the number of functional patterns in Mapping's entry
         identical to Mapping's pattern
      Let P_E be the total number of functional patterns in Mapping's entry
      Assign Mapping a score of (T_M/T_E)^2 - ((C_E-C_M)/C_E)^2 + (P_M/P_E)^2
```

lcs The LCS lexicon [15] is wrapped by a TP parser in much the same was as the FrameNet lexicon. Experimentation revealed that the format and content of FrameNet data is significantly more conducive to the types of semantic output needed by FLOOD, though, so the LCS parser is at best a prototype implementation. Communication is again performed by loading the LCS lexicon data into a binary format and performing a heuristic mapping into *CArgument* roles.

link The Link grammar parser [57] is wrapped by this TP parser in a relatively intricate manner to normalize Link output into *CFunction* grammatical functions. Since the functionality of the Link parser is offered as a C API, it is first wrapped by a C++ server which attaches this API to a network socket. This server accepts per-sentence parse requests in the simple XML format shown in Figure 20 and produces an XML encoded version of the raw Link output (shown in Figure 21). Like *CNLPClient*, this server can be provided with basic configuration such as a network port, maximum number of parses to produce, and a per-sentence timeout through the use of an XML configuration file.

The link *INLProcessor* then takes this server's output, in which syntactic dependencies are encoded as any one of over 100 different link types, and normalizes it into *CFunction* elements. Some of these mappings are simple; an O link always creates a single OBJ function between its source and its target, for example. Others are complex and context-sensitive, particularly those for which Link has particularly specialized link types (numerical expressions, times, distances, monetary values, etc.) Many of these mapping rules are implemented on a case-by-case basis to appropriately analyze specific linguistic constructs, and a full specification of the rules is beyond the scope of this document.

Figure 20: Link server input XML

```
<Sentence>
   [text]
</Sentence>
```

morpha This parser wraps the morphological analysis component of the RASP toolkit [42]. This is a simple, extremely fast analyzer built around character-based lexical rules. It consumes tokens and their parts-of-speech to produce morphological annotations in the form of a root and a list of any prefixes or suffixes; these are then attached to the input *CToken* elements as properties.

mxterminator MXTerminator is a maximum entropy based sentence separator, classifying potential sentence boundaries as terminal or non-terminal punctuation [54]. This parser communicates with MXTermi-

Figure 21: Link server output XML

```
<parse>
    <words>
        <word>[text]</word>
        ...
    </words>
    <linkage>
        <link>
            <label>[link]</label>
            <left>[#]</left>
            <right>[#]</right>
        </link>
        ...
    </linkage>
    ...
</parse>
```

nator directly through its Java interface, consuming *CPassage* objects and attaching *CSentence* elements to them as children.

rasp This parser encapsulates the central parsing tool of the RASP toolkit [8]. It consumes several types of lower-level annotations - sentence separation, tokenization, morphological information, and parts-of-speech - to produce both constituent and functional parses for a sentence. In the TP, this means that it must be provided with a fully annotated *CTokenization*, to which it attaches *CSyntax* children containing a *CConstituent* tree and (optionally) *CFunctions* objects. While RASP's constituent parses tend to be quite good, its functional parses were consistently much less detailed and accurate than those produced by the Link parser; thus, this TP parser tends to be largely unused. Communication is performed by spawning a child process and communicating using standard input and output.

rasptok This parser encapsulates the tokenizer from the RASP package [20], consuming single sentences (*CSentence* objects) and attaching a *CTokenization* child to them containing a set of *CToken* children. Communication is performed by spawning a child process and communicating using standard input and output.

wordnet This parser extracts various information from the WordNet synset hierarchy [18] and attaches it to *CToken* objects in the form of structured properties. The general format of this data can be seen in Figure 22; currently, only basic synset ancestors are provided, but other information could easily be added to the parser's output. Communication is performed using the Java WordNet Library (JWNL) [2].

3.2 Planning

The FLOOD Planning Platform, like the Text Processor, is implemented in Java as a set of modular subcomponents. Before reaching its current incarnation, a prototype system was constructed using Lisp and the PRODIGY planning environment [64]. As mentioned above, this led to the original conclusion that a separate planning platform was necessary, but it also provided an excellent testbed for the development of the overall PP architecture. Testing with real complex questions as they arose in the context of HALO demonstrated the need for certain representational capabilities in the domain model for both data and actions; for example, to balance chemical equations, an action language must be flexible enough to sacrifice

[2] http://jwordnet.sourceforge.net

Figure 22: WordNet property XML

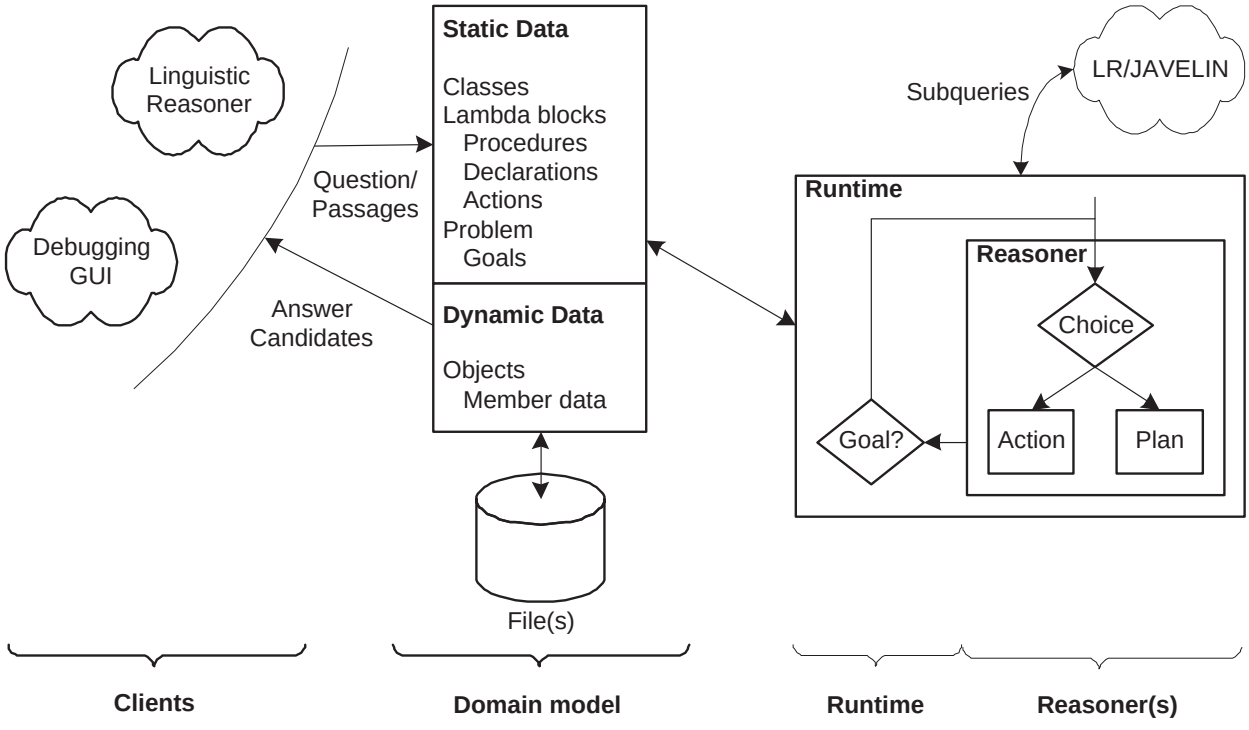
```
<property name="senses" source="wordnet">
 <property name="sense0">
  <property name="gloss">
   cultivate, tend, and cut back the growth of
   &quot;dress the plants in the garden&quot;
  </property>
  <property name="hypernyms">
   <property name="word0">snip</property>
   <property name="word1">clip</property>
   ...
   <property name="hypernyms0">
    <property name="word0">thin_out</property>
    <property name="hypernyms0">
     <property name="word0">reduce</property>
     <property name="word1">cut_down</property>
     ...
     <property name="hypernyms0">
      <property name="word0">decrease</property>
      <property name="word1">lessen</property>
      <property name="word2">minify</property>
      <property name="hypernyms0">
       <property name="word0">change</property>
       <property name="word1">alter</property>
      </property>
     </property>
    </property>
   </property>
  </property>
 <property name="sense1">
  <property name="gloss">
   weed out unwanted or unnecessary things;
   &quot;We had to lose weight, so we cut the sugar from our diet&quot;
  </property>
...
```

planning rigor for numerical calculations. Similarly, to be able to experiment with various planning approaches, there were clear boundaries where standard functionality (loading domains, performing state space maintenance, etc.) could be separated from the planning algorithms themselves.

This gave rise to the basic architecture of the PP, which can be thought of as consisting of three basic components as pictured in Figure 23:

- A domain model. This is purely a data model capturing a representation of objects, types, actions, etc. as they are commonly used in planning; it is also easily the most complex part of the Planning Platform, containing essentially an embedded programming language capable of representing objects, inheritance, and various types of procedural and conditional information.
- A runtime. This unit controls planning algorithms implemented within the PP (like the QAR). It offers ways to load them, provide them with domain information (based on the PP's domain model), and to allow them to find a successful plan in an incremental manner.
- An external API, exemplified by the PP GUI. The Planning Platform includes a graphical interface which can be used to load domains, execute reasoners, and view various aspects of a planner's internal state during execution. This interface makes use of the PP's external API, which can be used by other programmatic clients (like FLOOD) to embed use of a planning system within a larger application framework.

Figure 23: Planning Platform architecture



In general, the various portions of the Planning Platform are all organized using a COM-like interface framework. Objects tend to offer their functionality through interfaces, each of which corresponds to a single implementing class. This leads to some apparent duplication in views of the class hierarchy, but it also complements the overall design of the Planning Platform: components are forced to be extremely modular, and any implementation details which can be hidden are. This helps to reduce the complexity of what could easily be an overwhelmingly convoluted system, particularly within the PP domain model.

3.2.1 Assumptions

Like the Text Processor, the Planning Platform imposes several restrictions on planners which intend to take advantage of its framework. However, in this case, both the runtime and encoding restrictions are significant: planning algorithms must adapt to the restrictions of the PP runtime, while the domains over which they operate must fit within the PP domain model. To accommodate the PP's position as a platform for broad planning experimentation, as well as for tackling difficult AI and engineering problems such as question answering, both the runtime and the domain model have remained as unrestrictive as possible. In this case, though, a lack of restrictions can be as much of a liability as an abundance: most planners operate under strict modeling assumptions, which are not necessarily required in a PP algorithm or domain.

Control structure restrictions First, to operate within the PP runtime, a planning algorithm must be usefully separable into discrete iterations. In most cases, this is not an issue; planning algorithms are, to overgeneralize, complex adaptations of search algorithms, which tend to operate by incrementally working "further" or "deeper" into a space and backtracking as necessary. Most classical planning algorithms (such as GPS *** and its descendants, most notably Prodigy ***) do this recursively, but it is a relatively trivial task to transform this into incremental iteration instead.

Where this does cause issues is in any application of the FLOOD Planning Platform to "real world" planning tasks, such as detailed robot control. While an algorith operating within the context of the PP's runtime might be suitable for a very high level robot planning task, for example, the lack of timing control makes the PP completely unsuitable for realtime or continuous task planning. The FLOOD PP is intended primarily as a research and learning platform, to implement and examine planning algorithms and domains, and not for direct application in real-world contexts. Question answering is perhaps an ideal example of its target use, given the task's heavy research focus and concentration on end results rather than running time.

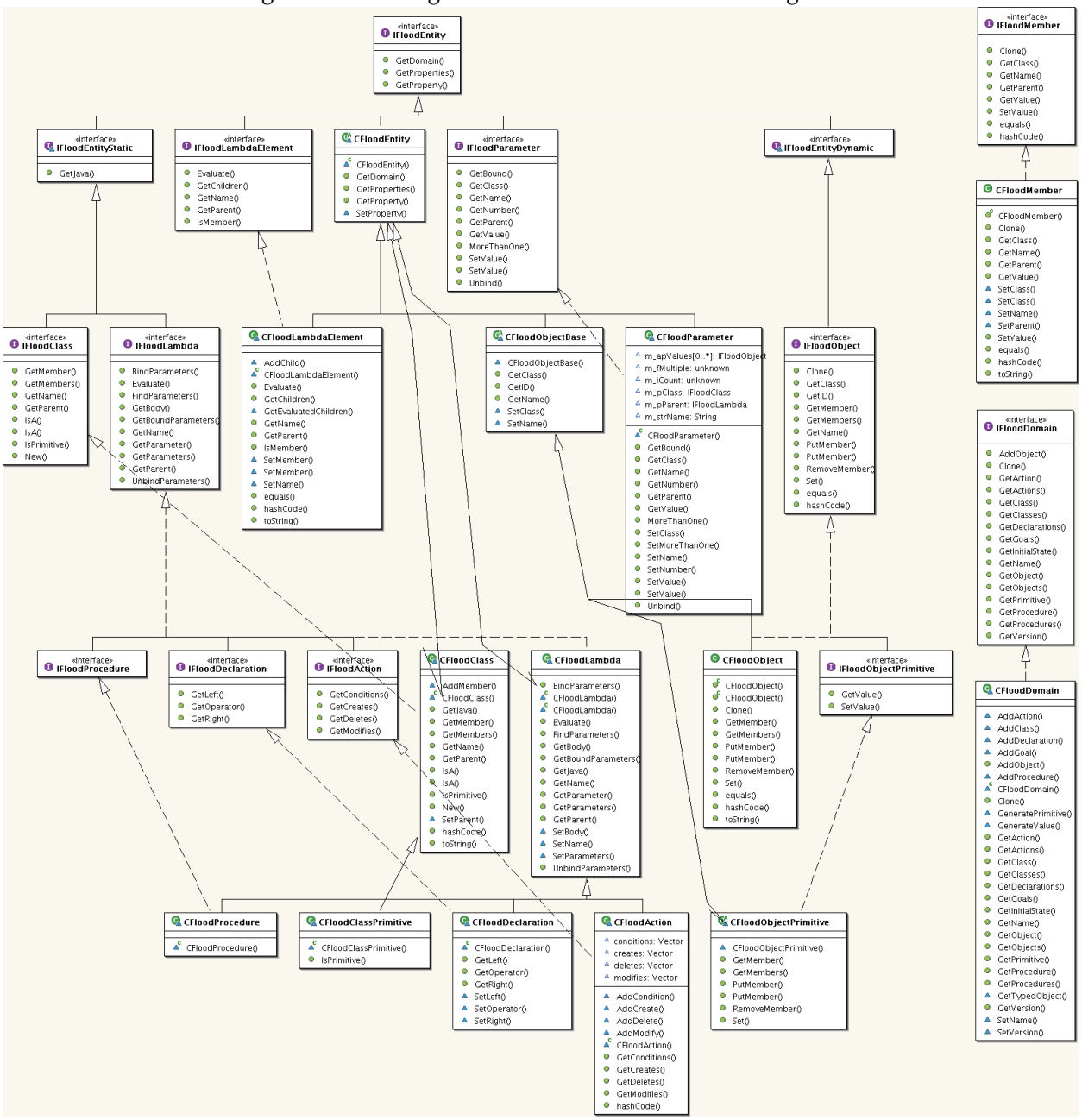
Data element restrictions The data restrictions of the PP are very light. If interaction with a specific planning algorithm is allowed, a planning domain can use anything up to and including raw Java constructs for data encoding and manipulation. In this case, however, it is overgenerality rather than overspecificity which incurs a price. Most planning algorithms operate by making assumptions such as a static universe (actions may not create or delete domain entities), purely boolean values, binary predicates, and so forth. The FLOOD PP makes none of these assumptions; it allows particular domains and planning algorithms to restrict themselves as necessary, but doing so is significantly more complex than it would be in a planning framework inherently constrained in an appropriate manner. This is most readily seen in the ease with which huge performance hits can be taken during planning, since a planning algorithm must explicitly make and take advantage of simplifying assumptions of its own accord if it is to gain from them.

3.2.2 Domain model

First, for reference, an example of a small PP domain can be found in Appendix B. Like the Planning Platform itself, the domain model can be further broken down into a set of modular components, briefly outlined in Figure 24. A domain is a collection of entities, each of which represents a piece of data (either pure data or procedural information) within a planning domain. Entities are broken down into two categories: dynamic entities which can be changed during planning (objects and their member data), and static entities which cannot be changed during planning (classes, lambda blocks, and the domain's goal statement). A brief overview of these entity types is given below, followed by a more detailed description of the domain model's layout within Java and the intended interactions of these entities.

- Classes. These are static entities providing type descriptions much like traditional classes in C++ or Java. A PP class consists of a name, an optional parent class, and zero or more name/type pairs; these names and types make up the member data present in objects of this class type. A small set of primitive classes is built into the domain model; these have no member data an operate as scalar data, much like integers or characters in standard programming languages.

Figure 24: Planning Platform domain model UML diagram



- Procedures. These are static entities consisting of little more than a named lambda block. In a FLOOD PP domain, a lambda block is essentially the same as it is in Lisp or Scheme: a functional unit that accepts zero or more strongly-typed arguments, performs some set of programmatic instructions when invoked, and returns a single result. Procedures are provided as a way of abstracting shared code away from other functional units (actions and declarations) within a planning domain.
- Actions. These are perhaps the core of a planning domain, static entities that are made up of augmented lambda blocks. Like procedures, they are named, but they also include a set of specifications that allows planning algorithms to manipulate them more intelligently. These include a list of objects potentially created, deleted, or modified by the action and a set of conditions which must be satisfied by the current state for the action to be applicable. The PP runtime supports execution of actions by a planning algorithm, in which case a transition to a new state occurs. A state is a collection of all available dynamic entities; the initial state consists of the entities as loaded from the original domain, and the current state changes as actions are executed by a reasoner.
- Declarations. Declarations can be thought of as conditional procedures. They are again static, named lambda blocks, but they contain not one but two collections of procedural information. If the "left" side of the declaration is ever true for a particular state, the "right" side is forced to execute. Declarations thus behave somewhat like control rules in PRODIGY.
- Objects. These are the dynamic entities of a domain. Each object consists of a class and, unless it is a primitive object, a list of name/value pairs. These names and values correspond to the names and types specified by the object's class (or one of its inherited ancestors).
- Goals. Each domain has exactly one goal set, which is a static collection of lambda blocks. These are considered to be satisfied for some state in which they all evaluate to a truth value, at which point the PP runtime stops execution of the current planning algorithm.

IFloodDomain/CFloodDomain The IFloodDomain interface is the top-level representation of everything contained within a PP domain. This includes all static entities - classes, procedures, actions, declarations, and goals - and all dynamic data, both initial and subsequently created objects and member values. This interface is used by the GUI and the runtime to access information about a domain. Conceptually, the domain and all entities within it make up a miniature programming language; the PP currently supports two on-disk representations of this language (one based on XML for programmatic manipulation and the other Lisp-like for convenience and brevity), both of which are mapped into the object model described below.

IFloodClass/CFloodClass This interface/implementation pair represents types of objects that may appear within a planning domain. As in standard object-oriented programming, a PP class consists of typed properties (for example, a property might be named "zip_code" and be of type "number"), an optional parent class from which additional members are inherited, and a class name. Since PP classes do not contain methods, they are more closely analogous to C++ structs or Lisp classes than to C++ or Java classes.

IFloodLambda/CFloodLambda A lambda object represents a set of functional operations in a PP domain, exemplified by procedures or actions. Lambda blocks are represented similarly to their Lisp or Scheme incarnations: they are sets of functional predicates in which certain lexical elements (lambda parameters) may be replaced with specific values when evaluated at runtime. Parameters are specified lexically (by name and type) within the domain, while their values are passed in on a case-by-case basis at runtime. Alternately, and perhaps more simply, lambda objects can be thought of as an object model over functions, in which each functional unit is represented as a data object which is evaluated on request to return a specific result.

IFloodParameter/CFloodParameter Within the PP domain model, parameters to lambda blocks are represented both lexically and at runtime as *IFloodParameter* objects. These begin as a simple name/type pair and are given a value each time a specific lambda block is evaluated. This allows the object model for the lambda block (consisting of a tree of *IFloodLambdaElement* objects as described below) to reference a single *IFloodParameter* object generated when they are constructed; when they are evaluated, that parameter will have been given a value, which they can simply extract using the Java object model and use to complete their evaluation.

IFloodLambdaElement/CFloodLambdaElement Each *IFloodLambda* block is made up primarily of a collection of *IFloodLambdaElement* objects. As is a common representation for lambda block components in Lisp or Scheme, each lambda element is either a scalar value with no children (a parameter reference, variable reference, or constant value) or a functional application (a procedure name or child lambda element) with zero or more arguments (also lambda elements). This means that each lambda block becomes a single tree of lambda elements; to execute, it binds its parameters and evaluates each lambda element in succession, returning the result of the final lambda element.

CFloodPrimitives Built-in functional primitives in the PP domain description language are all collected as static methods of the *CFloodPrimitives* class. By using Java reflection, new primitives can be added in a strongly-typed manner simply by adding appropriately typed Java methods to this class. This includes many basic Lisp-like functions such as `push`, `equal`, `add` (an alphabetical replacement for `+`), etc.

IFloodProcedure/CFloodProcedure As mentioned above, procedures are simply named lambda blocks. Thus, *IFloodProcedure* inherits from *IFloodLambda* (and *CFloodProcedure* from *CFloodLambda*), adding essentially nothing in the process.

IFloodAction/CFloodAction A PP action represents a planning action that causes a transition to a new state when executed. This means that the domain is changed in some way: objects are created, deleted, or modified. Actions in the Planning Platform correspond to actions in the transitional planning sense; they are programmatic representations of the actions that would actually be available to an agent performing the plan, such as "MoveBriefcase" or "OpenDoor". In the PP, each action is a lambda block with an additional list of conditions (possibly empty, with each member a lambda element, all of which must be true in the current state for an action to execute), lists of object types and counts which may be created, and specific references to objects (from the parameter list or global namespace) which may be deleted or modified when the action is executed. As in the PRODIGY system, action execution is performed by the runtime when requested by a planning algorithm during forward-chaining planning; this is described in more detail below.

IFloodDeclaration/CFloodDeclaration As mentioned above, PP declarations are entities similar to procedures or actions, but containing two lambda blocks instead of just one. If the "left" side of the declaration is ever true for the current state (checked by the runtime after action execution causes a new state to be entered), the "right" side of the declaration must be executed. This can in turn produce a special modification of the state, not unlike the transition to a new state caused by action execution, but without the chance for choice or intervention by a planning algorithm. Declarations thus operate not unlike control rules in the PRODIGY architecture.

IFloodObjectPrimitive/CFloodObjectPrimitive Several classes are provided by the PP as primitives for domains to use. These include strings, numbers, booleans, and lists, in addition to pointers, classes, members, and objects, all allowing simple reflection to be performed within PP domains. Each of these primitive types is implemented by a Java class inheriting from *CFloodObjectPrimitive* (and thus *IFloodObjectPrimitive*), which provides basic operations available for any primitive type.

IFloodObject/CFloodObject These represent an instantiation of an *IFloodClass* just as a Java or C++ object represents an instantiation of its respective class. Objects in the Planning Platform domain model are simplified essentially to the level of structs or Lisp objects, containing only named data members with typed values (which, of course, may in turn be other objects with their own classes and data members).

IFloodMember/CFloodMember Members, like lambda parameters, are used both in bound and unbound forms. Without a value, a member is a name/type pair, used in *IFloodClass* objects to specify the allowable data members of an object. When used in an *IFloodObject*, members are (optionally) bound to a value of the appropriate type.

3.2.3 Runtime

The PP runtime provides an environment in which a planning algorithm (such as the QAR) can be loaded, communicate with a domain model, and execute to produce a satisfactory plan. On one hand, the runtime is minimally exposed to external clients; it is possible to load a domain and a reasoner into the PP, press the "Go" button (figuratively speaking), sit back, and wait for a completed plan to pop out. On the other hand, it is also possible to communicate with the runtime and the planning algorithm it controls to a relatively fine degree of detail; this is done, for example, by the PP GUI to provide debugging information for domain and reasoner developers. Thus, the runtime is largely encapsulated by a small number of classes, but they each present a rich selection of information to external clients.

During normal operation of the Planning Platform, the runtime classes represent a relatively self-contained collection of control structure and data. The internal *CFloodRuntime* class contains the core iterative algorithm which repeatedly asks a reasoner what it would like to do and takes action based on its answer, each iteration either executing a plan action (a forward-chaining step) or allowing the planning algorithm to manipulate its plan representation (a backward-chaining step). Planning algorithms themselves are implemented in Java using the *IFloodReasoner* interface, and they communicate with the runtime primarily through data contained in the *IFloodContext/CFloodContext* object. Most significant among this information is a directed graph of *IFloodState* objects. This graph begins with the initial state as specified by the objects within the domain model, and each action execution introduces an arc labeled with that action and leading to a state representing the configuration of objects generated by that action's results.

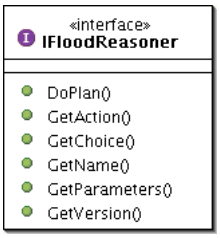
CFloodRuntime This is the core of the PP runtime. When a client asks for the PP to use some planning algorithm to satisfy the goals of a planning domain, the runtime executes the represented in Algorithm 2 until a successful plan is generated. The client can then query the planning algorithm or the final context and state to determine what actions were necessary or what configuration of data elements has been generated to satisfy the goals.

Algorithm 2 The PP runtime

```
Runtime( Reasoner, Context )
   While any domain goal remains unsatisfied,
      Let C be the result of Reasoner.GetChoice( Context )
      If C is an action execution (forward chaining step)
         Let A be the result of Reasoner.GetAction( Context )
         Let P be the result of Reasoner.GetParameters( Context, Action )
         Execute A using P as drawn from the current state in Context
         Let S be the new state generated by this action
         Add S to Context and make it current
      Else C is a plan step
         Call Reasoner.DoPlan( Context )
```

IFloodReasoner Any planning algorithm written for use within the Planning Platform is required to implement the *IFloodReasoner* interface. For a client to provide a new, domain-specific reasoner, they need only implement this interface and register the implementing class with the PP API. The PP runtime works iteratively in a series of steps or time slices. During each step, a reasoner performs two actions: it chooses to either execute an action (perform a forward-chaining plan extension) or to expand its plan representation (perform a backward-chaining plan extension), after which it either chooses an action and parameters or it performs the plan expansion. This process is outlined in Algorithm 2. For a reasoner implementation, the key methods required for this process are *GetChoice*, *GetAction*/*GetParameters*, and *DoPlan*. Each of these methods is provided with an *IFloodContext* upon execution which, as described below, allows the reasoner to communicate with the domain and the state graph as well as to store its own arbitrarily formatted data.

Figure 25: *IFloodReasoner* UML diagram



IFloodContext/CFloodContext The context is the primary data store for the PP runtime. Along with several pieces of internal data, it holds a link to the initial domain representation, the entire state graph (including the initial state and any new states and links generated during reasoner execution), any client-provided primitives (discussed below), and arbitrary plan-related information provided by an *IFloodReasoner* implementation. All of this is accessible to the runtime or the planning algorithm during execution; it can also be accessed by the external API (and thus the GUI) through a callback system, allowing interested clients to view domain and plan information in "real time" as it is manipulated by a reasoner.

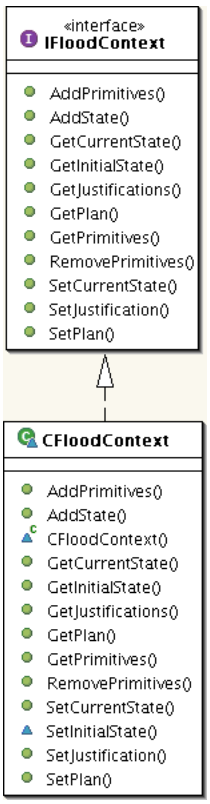
IFloodState/CFloodState A PP state captures a snapshot of a particular configuration of dynamic domain entities (objects and their data member values). Although the static data of a domain remains the same as actions execute, objects and their members may change. A state contains a copy of all objects and members at one point in a domain's lifetime. States are connected by directed links indicating their relationships, i.e. how one was formed from another by execution of an action by a planning algorithm. New states are formed exclusively by execution of lambda blocks (either actions or declarations) from the domain of which the state's objects are entities. Thus, when execution of an action creates, deletes, or modifies objects, a link is created from the current state (as maintained in an *IFloodContext*) to this new state, the new state is added to the context, and it is marked as current. Note that it is the responsibility of the context to ensure that duplicate states are not created, i.e. that new states identical to an existing state anywhere within the state graph are discarded and replaced by their existing counterparts.

IFloodPrimitives To further expand the flexibility of the FLOOD Planning Platform, clients may provide additional "primitives" just as they may provide domains and reasoners. The *IFloodPrimitives* interface allows a client to implement additional methods in Java which may be called by a PP domain just as the built-in primitives are (`push`, `member`, etc.) It consists of a single function that returns a set of methods to be used as primitives using Java reflection.

3.2.4 GUI and API

Despite the Planning Platform's internal complexity, it endeavors in its external interface to provide a simple, unified way for clients to quickly and easily integrate planning and reasoning into their applications.

Figure 26: *IFloodContext/CFloodContext* UML diagram



Outside of the more specific interfaces outlined above, the core of the PP's API consists of only a few classes and interfaces used to load a domain, load a reasoner, and tell the runtime to begin execution. Built around this interface is a rich graphical interface intended to make development and debugging easier for domain and reasoner authors. This currently has the capability to:

- Load a domain, validate its syntax, and provide a graphical overview of its static entities
- Load a reasoner and, optionally, additional primitive functions into the runtime
- View the dynamic entities present in any PP state, including the initial state as loaded with a domain
- View the state graph as maintained by the context at any point during runtime execution
- View a basic graphical representation of a reasoner's plan representation at any point during runtime execution

CFlood The *CFlood* class is the main top-level managerial interface for the entire Planning Platform. It is the first class that should be created and accessed by a client, and it exposes nearly all other public PP functionality through the *IFloodReasonerManager* and *IFloodDomainManager* interfaces. This allows the client to access the two main PP components - domain modeling and runtime reasoning - through a single, simple class.

IFloodReasonerManager The reasoner manager interface allows a client to access and change the currently active reasoner. This can be used to provide an external planning algorithm implementation or to select one of the Planning Platform's built-in reasoners.

Figure 27: *IFloodState*/*CFloodState* UML diagram

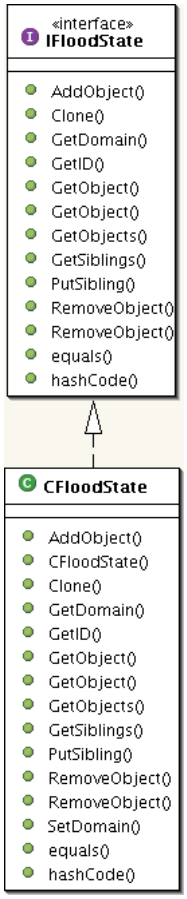
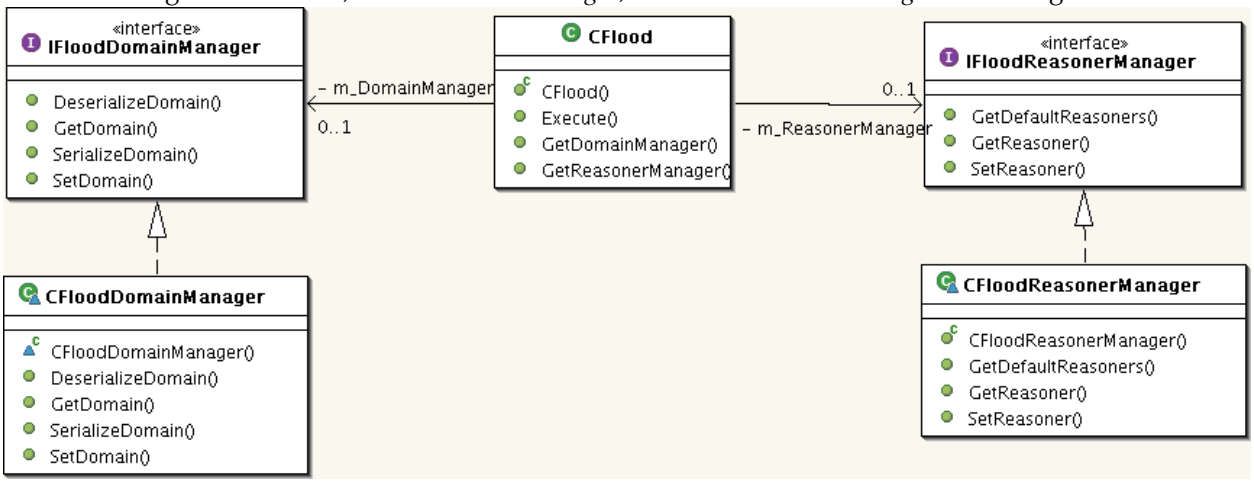


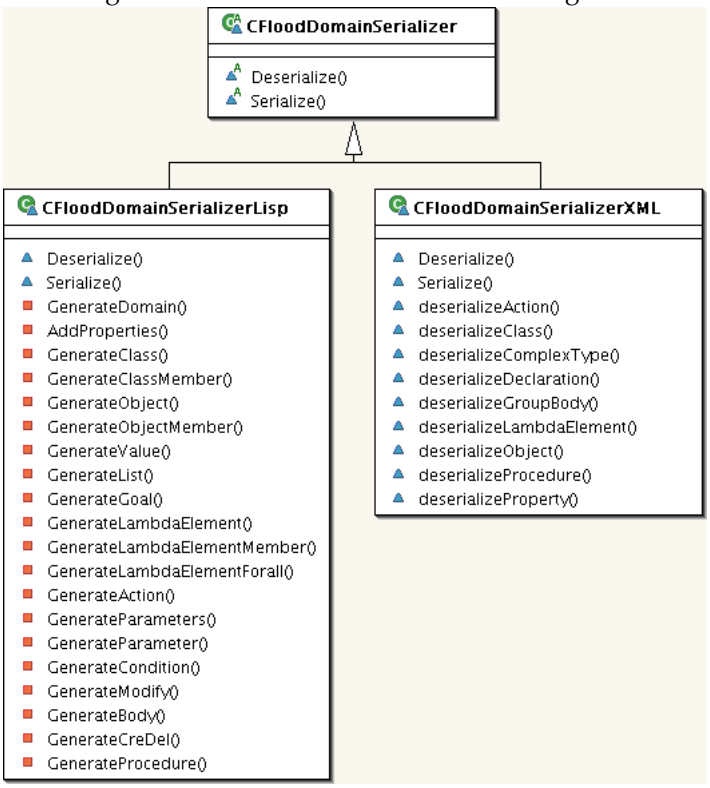
Figure 28: *CFlood*, *IFloodReasonerManager*, and *IFloodDomainManager* UML diagram



IFloodDomainManager The domain manager is a high-level interface through which the client registers domains, loads them from files, and begins any desired interaction with the domain model as described above. Most importantly, *IFloodDomainManager* allows a client to select the active domain to be used with a planning algorithm for reasoning by the PP runtime. It also provides utility functions that allow actions like serialization/deserialization to save or load domains on disk.

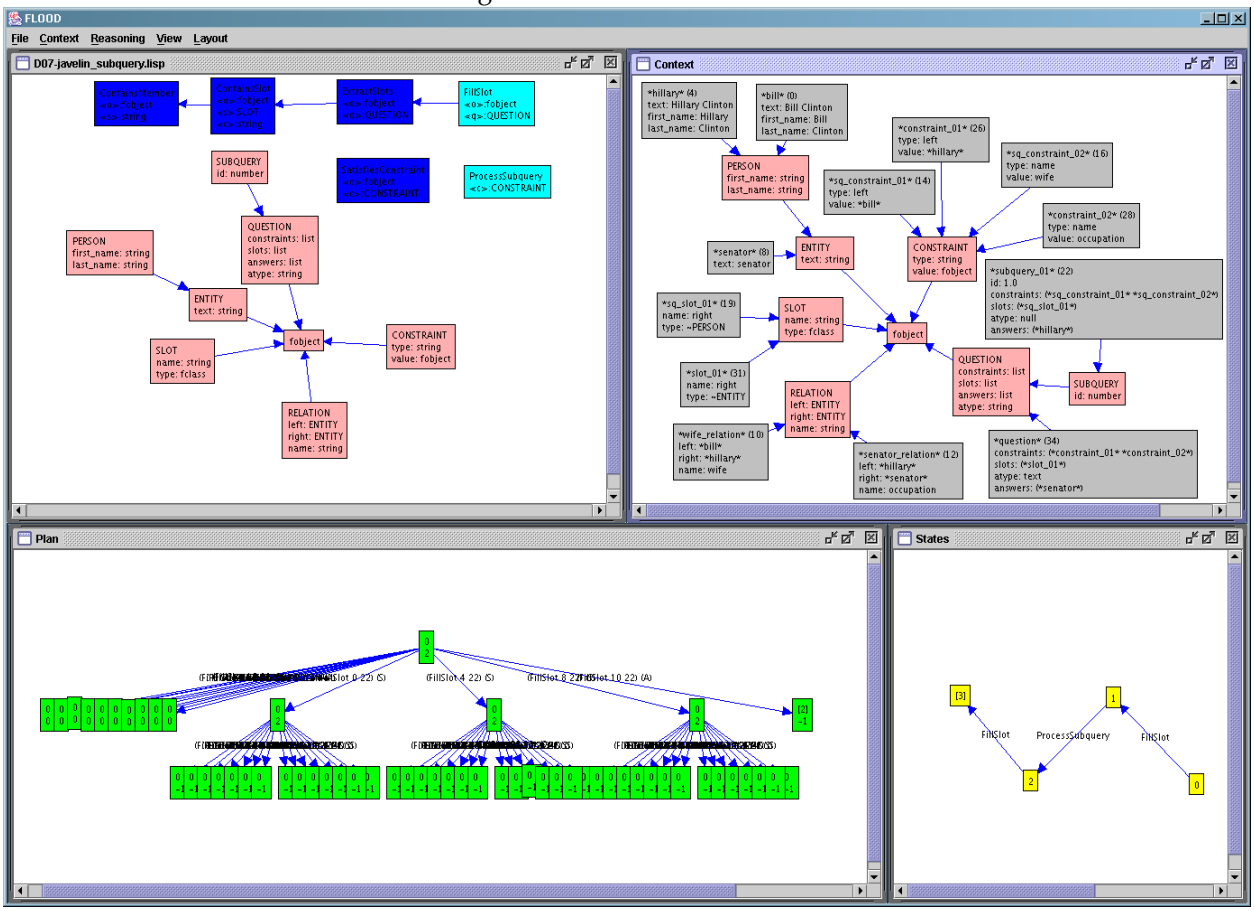
Domain serialization While general serialization functionality is exposed through the domain manager interface, its specifics are implemented by a small set of internal classes. The PP domain model can be represented on disk in either one of two equivalent formats. The first is an extremely verbose XML representation that is convenient for external programmatic manipulation of domains but horribly unpleasant to actually develop by hand. The second uses a much more compact Lisp-like syntax that is less computationally manipulable but far easier to author. This is the format shown in Appendix B. When given a file to load into the PP's domain object model, the domain manager attempts to automatically determine its format and loads the appropriate serialization manager, which then parses the file and creates an *IFloodDomain* with the appropriate contents.

Figure 29: Domain serialization UML diagram



GUI As described above, the PP GUI provides four main functions for use by domain and reasoner developers: viewing the static contents of a domain, viewing the dynamic contents of a state, viewing the entirety of the state graph generated at runtime, and viewing a simple form of the plan description generated at runtime. These four features are all demonstrated in the screenshot shown in Figure 30. In addition to these debugging functions, however, the GUI also provides a reference implementation of a Planning Platform client; it is unlikely that any external client would ever require more detailed interaction with the PP than does the GUI (the rest of the FLOOD system certainly doesn't). Thus, the GUI is not only a debugging tool for PP domains and reasoners, but a testbed for the PP itself and a platform within which new functionality can quickly be attached to an external interface and tested.

Figure 30: PP GUI screenshot



3.3 Question answering

The process of question answering in FLOOD is carried out by a combination of two components: the Question Answering Reasoner, which provides the core functionality necessary to reason over question components, and the Linguistic Reasoner, which provides a glue layer between the Text Processor, the Planning Platform, and the external question answering environment (JAVELIN). The control and data flow which occurs when FLOOD is being used by JAVELIN for complex question analysis can be seen in Figure 31. In temporal order, these steps are:

- JAVELIN's planning process decides that complex question analysis is necessary. At this point, basic question analysis has already been performed (a simple process extracting a basic question type, answer type, and keywords from the text) and passages containing potential answer candidates have been generated. This combination of data (the question text, its initial analysis, and the text of the answer candidate passages) is provided to the LR with instructions to produce an answer.
- The LR requests a full parse, from sentence separation through semantic role filling, from the Text Processor.
- The LR performs a relatively simple transformation on this semantic data to produce actions and entities suitable for inclusion in a Planning Platform domain.
- This planning domain information is merged with a basic, pre-existing QA domain (authored as part of the QAR). The LR instructs the PP to load this combined domain, the QAR, and to begin execution of the planning algorithm.
- The Planning Platform allows the QAR to perform whatever analysis is necessary to produce an answer to the question. This may include recursively invoking the external question answering system in order to answer a subquestion.
- Once planning is complete, the LR examines the resulting domain model and plan steps and transforms these into answer candidates. These are returned to the question answering system which invoked FLOOD, which can then process them just as it would answer candidates produced by traditional information extraction.

3.3.1 Linguistic Reasoner

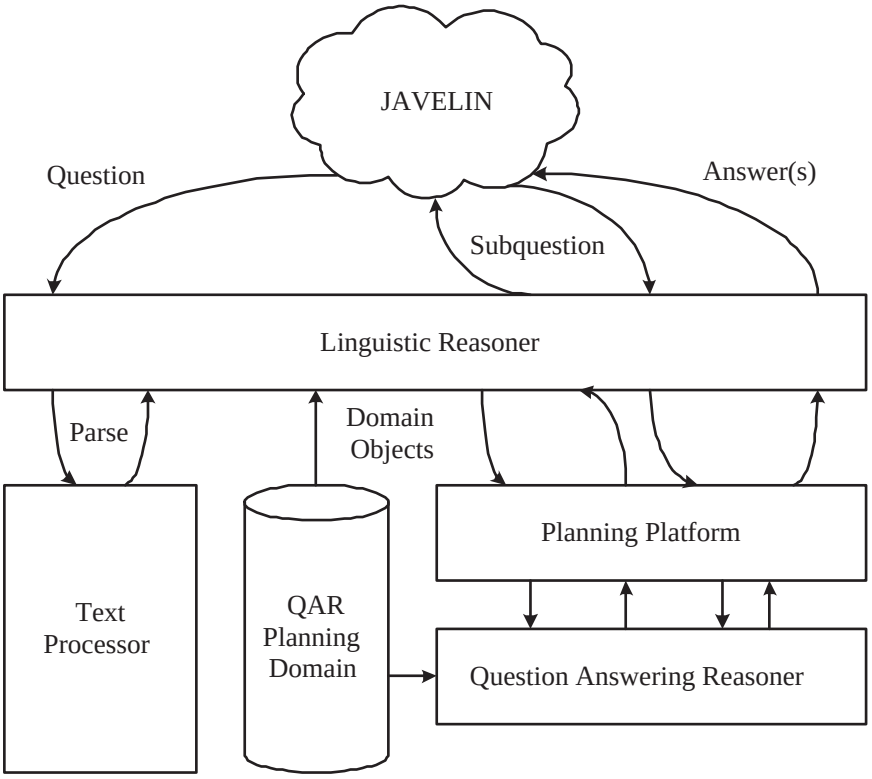
The Linguistic Reasoner is perhaps the least critical of FLOOD's subcomponents. Without it, the Text Processor, Planning Platform, and Question Answering Reasoner all continue to function independently and to produce appropriate results when presented with input. Rather than helping FLOOD to function, the LR instead allows FLOOD to be integrated into an external context: it accepts external input, acts as a pipeline that helps the primary components of FLOOD work together, and provides end results to the external caller.

The LR's basic external interfaces operate in much the same way as the Text Processor's. It can be configured through a simple XML property file, listens on a network socket for incoming requests in a simple XML format, and provides output to the caller as a similar hierarchy of XML tags. Its input and output formats can easily be adapted to consume or produce other data; the only essential input is text for a question and one or more passages, and the basic output is a list of short candidate answer strings. Other than its responsibilities in control flow, the LR is of primary interest in its conversion of semantic analyses into planning domain entities, functionality which is described in more detail below.

Representation When the TP finishes process the text of the question and answer candidate passages, the LR has a variety of data available. Typically, this includes:

- Sentence separation as performed by MXTerminator [54]

Figure 31: Control and data flow for FLOOD as invoked by an external question answering system



- Tokenization and functional parsing as performed by the Link grammar parser [57]
- Part-of-speech tagging as performed by CLAWS [35]
- Basic morphological processing as performed by the morphological analyzer in the RASP package [42]
- Named entity tagging as performed by BBN IdentiFinder [3]
- Semantic role filling as provided by FrameNet [29]

Each individual unit of text (the question or a passage) is represented as a *CLRText* object, which is provided with its Text Processor output (as a hierarchy of Java data elements as described in 3.1.3). These are subclasses as *CLRQuestion* and *CLRPassage* objects, respectively, but at present this is solely for design purposes (see Figure 32 for a graphical representation of these relationships). It is then the responsibility of the *CLRText* object to generate from its TP output (primarily the semantic arguments) a collection of data elements suitable for insertion into the QAR domain.

Translation To populate the basic Question Answering Reasoner domain with the contents of a specific question and passages, the LR constructs a collection of Java objects, each of which corresponds to an object and data members in the target Planning Platform domain. As was discussed in 3.2.2, the Planning Platform represents objects purely as data elements with a name, a class, and a collection of data members. The QAR takes advantage of this to define a basic domain applicable to open domain question answering. This will be discussed in further detail below, but this is essentially a minimal ontology describing people, objects, actions, and other basic entity types which are commonly used to describe question or answer types. The Linguistic Reasoner's Java counterparts can be seen in Figure 33.

Figure 32: LR data UML diagram

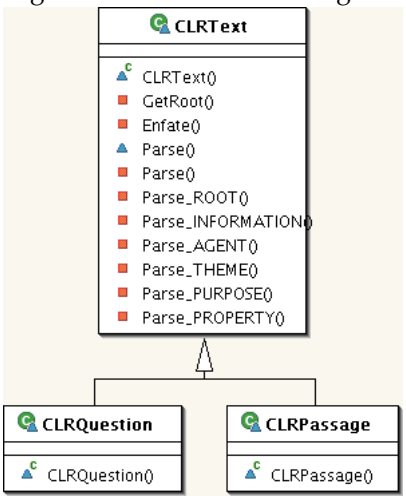
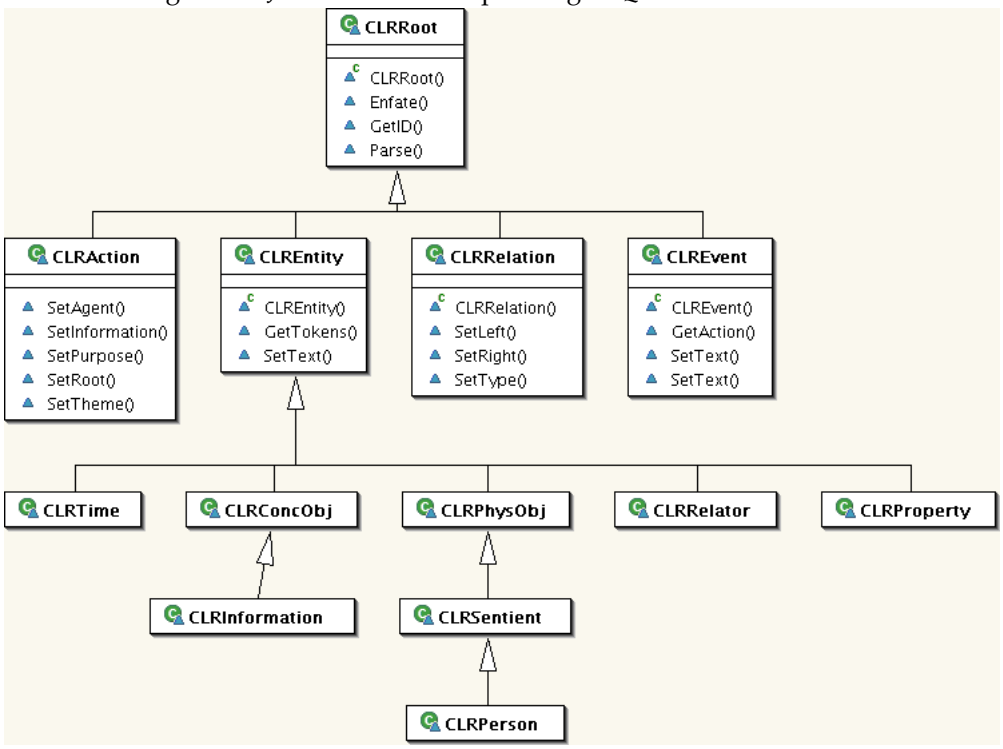


Figure 33: Java classes corresponding to QAR domain classes



To generate a collection of these objects, each *CLRText* first examines its *CArgument* elements (as generated by the TP) on a per-sentence basis. In a manner not unlike the translation of individual parser output into a normalized form by the TP, each *CArgument* typically generates a single entity, action, or property within the QAR domain being populated. The basic algorithm used during this process is presented in Algorithm 3; again as in the TP, customization is performed on a per-argument-type basis as necessary.

Note that in Algorithm 3, types correspond to classes in the LR/QAR domain class hierarchy (as seen in Figure 33). It is permissible for objects to be converted to more specific types (i.e. classes of which their current class is an ancestor); attempts to move a class laterally through the hierarchy produce errors, and attempts to mark an object as a less specific class than it already is (i.e. an ancestor of its current class) are ignored. Each argument type supports a particular set of object types; for example, the first child of an AGENT is always an action, and the second is always a sentient object. These are again encoded on a per-argument-type basis.

Algorithm 3 Translation of argument structure to planning domain objects by the LR

```
Translate( Argument )
    If Argument.Type is ROOT
        Set the text of Objects[ Argument[ 0 ] ] to Argument[ 1 ]
    Else for i ∈ {0,1}
        Let T_i be the most specific object types for Argument.Type
        Let O_i be Objects[ Argument[ i ] ]
        If the type T_i conflicts with any known type for O_i
            Error
        Else
            Set O_i's type to T_i
        If Argument.Type is a member of type T_0
            Set O_0's Argument.Type member to O_1
        Else
            Let P be a property named Argument.Type
            Add P relating O_0 to O_1
```

3.3.2 Question Answering Reasoner

The Question Answering Reasoner performs the actual process of question answering within FLOOD: it accepts a very high-level representations of a question and supporting data (answer candidate passages) and produces similarly high-level representations of answer candidates. This process is performed within the framework of the Planning Platform, and as such is formulated as a planning problem. The QAR provides a planning algorithm, and the question and passages are provided in the form of objects within a planning domain. The key components of this system are:

- The QAR planning algorithm. This is a somewhat modified version of the FLECS algorithm [65] that has been minimally customized to take advantage of the QAR's planning domain.
- A predefined portion of the QAR planning domain. This is a collection of Planning Platform domain entities (actions, classes, goals, etc.) that are universally applicable to any question posed to FLOOD by an external QA system. These are mostly static entities, primarily actions and classes, which provide something of a template which is populated by the LR.
- A question-specific portion of the QAR planning domain. This consists of the objects generated during the LR translation process described above and inserted into the QAR's planning domain through the Planning Platform interfaces.

Planning algorithm A summary of the FLECS algorithm can be seen in Algorithm 4. This basic planning algorithm was chosen as a basis for the QAR because of its evident suitability for the Planning Platform architecture: when implemented via *IFloodReasoner*, steps 3 through 5 correspond directly to *GetChoice*, step 6 to *GetAction/GetParameters*, and step 7 to *DoPlan*. Most aspects of FLECS were implemented directly as written, but a few proved challenging or were adapted for the purposes of the QAR.

Algorithm 4 The FLECS planning algorithm [65]

1. Initialize
   Fringe goal set $\mathcal{G} = \{$goal statement from domain$\}$
   Current state $\mathcal{C}$ =initial state from domain
   Instantiated operators $\mathcal{O} = \{\}$
   For all goals G, ancestor goal sets $a(G) = \{\}$
   For all operators O, cause sets $c(O) = \{\}$
2. Terminate if goal statement true in $\mathcal{C}$
3. Compute applicable operators $\mathcal{A}$ and pending goals $\mathcal{P}$
   $\mathcal{P} = \{G \in \mathcal{G} | G \notin \mathcal{C} \vee G \in$initial state$\}$
   $\mathcal{A} = \{A \in \mathcal{O} | preconditions(A) \subseteq \mathcal{C}\}$
4. Adjust $\mathcal{P}$ and $\mathcal{A}$ to contain only active members
   $\mathcal{P} = \mathcal{P} - \{P \in \mathcal{P} | \forall S \in a(P). \exists G \in S$ s.t. $G \in \mathcal{C}\}$
   $\mathcal{A} = \mathcal{A} - \{A \in \mathcal{A} | \forall G \in c(A). [(G \in \mathcal{C}) \vee (\forall S \in a(G). \exists G' \in S$ s.t. $G' \in \mathcal{C})]\}$
5. Subgoal or apply
   If $\mathcal{A} = \{\}$ go to step 6
   If $\mathcal{P} = \{\}$ go to step 7
   Choose to apply or to subgoal (*backtrack point*)
      If subgoal and $P \notin \mathcal{C}$, go to step 6
      If apply, go to step 7
6. Choose a goal P from $\mathcal{P}$ not true in $\mathcal{C}$
   Get the set $\mathcal{R} \subseteq \mathcal{O}$ that could achieve P
   If $\mathcal{R} = \{\}$ then
      $\mathcal{P} = \mathcal{P} - \{P\}$
      If $\mathcal{P} = \{\}$ then backtrack
      Otherwise go to step 6
   Choose an operator O from $\mathcal{R}$ (*backtrack point*)
   $\mathcal{O} = \mathcal{O} \cup \{O\}$
   $\mathcal{G} = (\mathcal{G} - \{P\}) \cup preconditions(O)$
   $c(O) = c(O) \cup \{P\}$
   $\forall G \in preconditions(O). a(G) = a(G) \cup \{\{P\} \cup S | S \in a(P)\}$
   Go to step 3
7. Choose an operator A from $\mathcal{A}$ (*backtrack point*)
   Apply A: $\mathcal{C} = (\mathcal{C} \cup additions(A)) - deletions(A)$
   $\mathcal{O} = \mathcal{O} - A$
   $\forall G \in preconditions(A). a(G) = a(G) - \{S \in a(G) | S \cap c(A) \neq \{\}\}$
   $\mathcal{G} = (\mathcal{G} \cup c(A)) - \{G \in preconditions(A) | a(G) = \{\}\}$
   $c(A) = \{\}$
   Go to step 2

The FLECS algorithm requires backtracking between key decision points. Maintaining all of the state necessary for the FLECS algorithm to backtrack correctly, particularly when applied with iterative deepening (as is necessary for proper algorithmic behavior), proved to be nontrivial. At each backtrack point, the current state, fringe goals, pending goals, instantiated actions, applicable operators, cause sets, and ancestor sets must all be stored for future use. This leads to a complex plan history which can be quite memory consuming; optimization allows the storage of deltas and the removal of any backtrack points which are no longer viable, but this can still be difficult to deal with for complex planning problems. One optimization

used by the QAR was to ensure that identical actions and parameter settings are never explored in a given state. This can greatly enhance performance in small domains with a high potential for state recurrence (such as the classic briefcase domain [19]) at the cost of introducing the potential for finding a sub-optimal final plan.

Another, broader concern of the QAR's implementation of FLECS is that the QAR takes advantage of the Planning Platforms provisions for non-binary data, while FLECS is not natively adapted to this type of reasoning. This affects three key aspects of the algorithm:

- Choosing a set of operators $\mathcal{R} \subseteq \mathcal{O}$ that could achieve a current goal P is based on a broader definition of "could achieve" than is assumed by FLECS. Since the PP requires actions to describe objects and expressions that they could potentially create, delete, or modify during execution, the QAR uses the information to determine whether any data elements used as input by P could possibly be affected by some action. This can lead to less restrictive searches than FLECS would in a purely binary planning space, but it also provides great flexibility for domain authors - in this case, the QAR.
- Choosing a particular operator O from $\mathcal{R}$ to use for a given planning step. This suffers the same overgenerality as does the initial choice of $\mathcal{R}$. The QAR attempts to choose an operator with either fully satisfied preconditions or, if one is not available, the most easily satisfied preconditions. This is roughly equivalent to the minimum conspiracy number heuristic recommended in [65].
- Applying an action A. Again, since FLECS assumes a binary planning space, application of an action simply consists of adding and removing predicates from a set of truth values. The Planning Platform allows plan actions to perform essentially arbitrary actions, which means that their execution must be modeled in a more complex manner. This is done by interpreting the contents of their lambda blocks just as would be done in a Lisp or Scheme interpreter; any changes to data are done in a new plan state, leaving the current one untouched, and any existing goals or preconditions which referenced objects in the old state are guaranteed to reference identical objects (if present) in the new state.

Planning domain The planning domain used by the QAR is essentially an expansion of the smaller domain seen in Appendix B. There are four main classes of objects, each with a small set of data members as allowed by the PP:

- Entities. This unfortunately overloaded term here corresponds to physical or conceptual objects as represented in a sentence (and thus in Text Processor output). These are organized into a small ontology as pictured in Figure 33; remember that for the purposes of translation, the LR Java classes are mapped directly onto PP classes within the QAR planning domain.
- Actions and events. Another ambiguous term, these are used to refer to the ways in which verbs - linguistic actions - are mapped into the QAR domain. The raw essentials of a verb are provided in the form of an action: the actor, the agent, any important theta roles for that verb (typically corresponding to the core frame elements in FrameNet), and the basic textual information itself (the plain text and morphological analysis). Events are actions which have been grouped with temporal information: past tense, a specific time, etc. This is intended to facilitate the future integration of more complex temporal reasoning, which is not currently used by the QAR [25].
- Relations (also referred to as properties). While named data members such as "actor" or "agent" allow a finite number of specific relationships to be specifically captured by the QAR, certain relationships will always be outside the scope of its limited ontology. Relations capture this type of information. For example, even the relatively specific *CLRSentient* object still does not have a "wife" member, yet the TP could easily produce arguments of the form "POSSESSION Bill wife" and "PREDICATE wife Hillary" from the phrase, "Bill's wife is Hillary". This would then lead to a relation named "wife" in the QAR leading from the entity "Bill" to the entity "Hillary".

- Questions, slots, and constraints. These are used to describe a question itself based on analysis by the TP. A question is a collection of constraints, slots, and an answer type; the latter indicates which member of a filled slot forms the expected answer. A slot corresponds roughly with a question word in a sentence: if the LR introduces a "who killed..." question into the QAR's domain, a slot will be formed requesting a sentient entity be found which participates as the agent in some event. This event is in turn described by constraints, which would in this example specify that the event must contain the action "kill" and occur in the past. Again, Appendix B contains a more illustrative example of these relationships.

4 Results

During development and evaluation, each of FLOOD's subcomponents underwent its own battery of tests. Those for the QAR are most representative of FLOOD's intended use within an external question answering system, while those for the Text Processor and Planning Platform are more indicative of their individual performance and potential uses as independent modules outside of FLOOD. For each major component, a specific evaluation task was performed:

- The Text Processor was analyzed for linguistic accuracy over a pair of small, broad corpora. This was performed within the limits of its component parsers; the TP is able to normalize data to a consistent form, but it is not intended to correct for errors in its input.
- Using a straightforward implementation of the FLECS algorithm (a simplified version of that used in the QAR), the Planning Platform was required to load and analyze a number of small planning domains. Some of these were based on classical problems, while others represented simpler stress tests.
- The Question Answering Reasoner was presented with a small set of questions and a subcorpus of text containing potential answers. These were analyzed both directly (i.e. by manually constructing the appropriate domain entities) and through the Linguistic Reasoner as they would be presented by an external QA system.

4.1 Text processing

Many of the Text Processor's duties consist of relatively straightforward translation: if a part-of-speech tagger produces an incorrect tag, it's not the responsibility of the TP to correct it. However, the two parsers most important to QAR functionality - the Link grammar parser [57] and the FrameNet lexicon [29] - also possess the most complex normalization processes. So, in addition to basic functional testing over the entire Text Processor, these two parsers were subjected to an evaluation of linguistic accuracy within the constraints of the data provided by their underlying systems (Link and FrameNet).

To evaluate these normalization processes, two corpora were produced. Both of these consisted of sentences drawn from the Link sample sentence collection and the English parser evaluation corpus [9] (a subset of the SUSANNE corpus [55] associated with the RASP toolkit). The first corpus was approximately 300 sentences in length and covered the entire range of linguistic functionality understood by the Link parser; each of its approximately 100 link types is represented by between one and a dozen sentences in the corpus. These sentences proved to provide broad enough coverage to evaluate the TP's FrameNet normalization process as well. The functional parses and semantic role assignments produced by these two TP components were then inspected by hand for each sentence in the corpus.

Results of this evaluation are shown in Table 3, organized according to the tool's functionality, the TP's normalization functionality, correct parses ranked highest, correct parses ranked second or third (FrameNet only produces a single parse), and incorrect parses containing partially correct data. Functional parses from the Link grammar parser are, in general, quite accurate; semantic role assignments as provided by

Table 3: Linguistic evaluation of the TP Link grammar and FrameNet lexicon parsers

	Total	**Correct (Tool)**	**Correct (TP)**	**Top 3 (Tool)**	**Top 3 (TP)**	**Partial (Tool)**	**Partial (TP)**
Link	317	280	280	20	20	17	17
FrameNet	317	140	24	-	-	84	134

FrameNet are significantly less so, particularly when compared to the approximately 60% precision/recall baseline discussed for a similar system in [22]. The purpose of the Text Processor is not, however, to provide a single parser for semantic role assignments, but to provide a framework within which such parsers can be integrated and utilized. This performance in semantic role assignment proved adequate for the purpose of experimentation in question answering, and the eventual availability of more sophisticated tools (such as those discussed in [22] and [4]) will, if integrated as parsers in the Text Processing framework, immediately provide a boost to overall system performance.

The second corpus consisted of approximately 500 additional sentences used for functional testing and linguistic change management. After any change in normalization rules which could affect linguistic accuracy, this corpus was analyzed by the appropriate parser(s) and the results compared to those of the previous version. If any sentences showed parse differences which decreased linguistic accuracy (as usual, relative to the accuracy of the input to the Text Processor as provided by its external components), the change in normalization was reverted or modified in such a way as to preserve performance.

4.2 Planning

Like the Text Processor, the FLOOD Planning Platform is intended more as an environment for manipulating data than as a system for generating end results. As such, the majority of its testing was again functional rather than analytical. This included manipulation of planning data through the PP's graphical interface, loading and modifying domains, advancing planning algorithms through problems incrementally, and so forth. In all of these tests, the Planning Platform performed satisfactorily; its key testing, of course, lay in ensuring the functionality of the QAR, to be discussed below.

However, to ensure that the Planning Platform provided an adequate environment for development and execution of planning algorithms and domains, five sample domains were created. Two of these represented classical planning problems, the briefcase problem [19] and the Sussman anomaly [59], while the other two were simple arithmetical domains intended to test the platform's fluent capabilities, search space control, and memory management. More specifically, the five domains and the problems tested within them were:

- Briefcase. An implementation of the briefcase domain as originally described in [19]. In this simple version, only two objects, a pencil and a stapler, are available to put in the briefcase. The goal is to get to work with the briefcase and the stapler.
- Addition #1. This domain contains only a single numerical class, one action which adds two numbers, and two instantiations of the class (representing the numbers 1 and 2). The goal is to use the single action to create new numbers until a number exists whose value equals 8.
- Addition #2. Like Addition #1, save that there are now additional actions to add three numbers, four numbers, or two different variations on adding five numbers. The goal is the create a number whose value equals 9.
- Increment. A similar domain in which there is one numerical class, but there are now two actions, one to increment a number by 1 and one to decrement a number by 1. These modify existing objects rather than creating new ones. The goal is to change the number that originally starts as 1 to a value of 2 and vice versa (change the number that originally starts as 2 to a value of 1).

- Sussman. This is an extremely simple version of the blocks world domain [73] in which the goal is to solve a basic version of the Sussman Anomaly problem [59]. There are actions to pick up or put down a block as well as to stack or unstack blocks. As usual, blocks A and B start on the table and block C starts on block A; the goal is to place block B on block C and block A on block B.

Table 4: Planning Platform results for test domains

	Solution	**Total states**	**Unique states**	**Plan steps**
Briefcase	PutIn(stapler) MoveBriefcase	32	6	164
Addition #1	AddTwoNumbers(2,2) AddTwoNumbers(4,4)	41	41	601
Addition #2	AddFiveNumbers1(2,2,2,2,1)	39	39	23411
Increment	IncrementNumber(1) DecrementNumber(2)	18	8	43
Sussman	PickOffDown(C,A) PickUpOnto(B,C) PickUpOnto(A,B)	197	9	918

The results of this testing using the default implementation of FLECS can be seen in Table 4. Note that this used an initial depth of two levels for iterative deepening and incremented this by one after each level; similar results are obtained when doubling is used instead. For comparison, the results of these analyses using the QAR-enhanced version of FLECS can be seen in Table 5. The savings in search effort are obvious, and for these simple examples, no suboptimal plans are produced.

Table 5: Planning Platform results for test domains (modified FLECS)

	Solution	**Total states**	**Unique states**	**Plan steps**
Briefcase	PutIn(stapler) MoveBriefcase	6	6	152
Addition #1	AddTwoNumbers(2,2) AddTwoNumbers(4,4)	41	41	601
Addition #2	AddFiveNumbers1(2,2,2,2,1)	31	31	18371
Increment	IncrementNumber(1) DecrementNumber(2)	10	8	35
Sussman	PickOffDown(C,A) PickUpOnto(B,C) PickUpOnto(A,B)	12	8	621

4.3 Question answering

Aside from the implicit testing performed on the QAR planning algorithm as described in Section 4.2, two main types of tests were used to evaluate the QAR as a question answerer. The first set of simpler tests used hand-encoded questions and passages to first provide a baseline functional test of the QAR domain's representational and reasoning capabilities. The second passed a small number of simplified questions and answers through the entire pipeline as controlled by the Linguistic Reasoner. These were somewhat contrived in order to avoid conflicts with the Text Processor's marginal theta role filling capabilities; as discussed in Section 4.1, the FrameNet interface is currently somewhat limited in its vocabulary and accuracy. Since the purpose of these tests was to evaluate the QAR and LR, these linguistic issues were avoided whenever possible.

For both sets of tests, text and questions taken from the documents described in Appendix A were used. These represent a subset of the documents available in the Center for Nonproliferation Studies Weapons of Mass Destruction Terrorism database[3]; the documents extracted for use in this evaluation were those dealing with the Al-Qaeda terrorist organization or its leader Osama Bin Laden. The questions were generated as information pertinent to this document collection that might be sought by an intelligence researcher during a background profile of the organization and its high-ranking officials.

4.3.1 Direct encoding

As tests of the QAR's question answering capabilities, three sample questions were encoded into QAR domain objects by hand, each accompanied by several answer candidate passages. The questions and answers used for this evaluation are shown in Table 6, and a snippet of their encoding is shown in Figure 34. These questions all represented basic operations necessary within the QAR domain:

- To determine the relationship between Osama Bin Laden and Al-Qaeda, it is necessary to convert the information that it is led by him into a relationship in which he is its leader.
- To further determine the relationship between the Taliban and Al-Qaeda, the information that the Taliban supports Bin Laden must be combined with this relationship between him and Al-Qaeda. Alternatively, the information that Bin Laden gives the Taliban help or that it gives him safe haven could be converted into a relationship and similarly combined.
- Several options are available for dealing with information about "weapons of mass destruction". For this test, the knowledge that "chemical" or "nuclear" weapons qualified as weapons of mass destruction was specifically added to the domain. This allows a test of QAR reasoning using domain-specific knowledge as an extension to the basic QAR domain, as well as the decomposition of a question into subquestions ("Is <x> a weapon of mass destruction?", "Has Osama Bin Laden sought <x>?", and if so, "What kind of weapon is <x>?")

Table 6: Hand-encoded QAR questions and passages

Questions
What is the relationship between Osama Bin Laden and Al-Qaeda?
What is the relationship between Al-Qaeda and the Taliban?
What types of weapons of mass destruction has Osama Bin Laden sought?

Passages
President Clinton has signed an Executive Order imposing sanctions on the Afghan Taliban for its support of Osama Bin Laden and his terrorist network.
Bin Laden's public statements then ceased under increased pressure from his Taliban hosts.
Bin Laden gets safe haven and in return, he gives the Taliban help in fighting its civil war
... Al-Qaeda, the militant organization led by Osama Bin Laden ...
One official said that Bin Laden has actively sought to acquire chemical weapons.
Bin Laden's group planned terrorist acts and sought to acquire chemical and nuclear weapons.

The encoding spanned several entities, actions, and relations per sentence, as well as interacting with a variety of actions available in the QAR domain: chaining of relationships, processing subquestions, and accumulating multiple answers, to list only some. Since hand-encoding is significantly more detailed than automatic encoding by the LR, performance was a significant issue during these tests. The evaluation details can be seen in Table 7; note that the third question's encoding was slightly simplified for increased

[3] http://cns.miis.edu

efficiency during testing (this represents a difference between a few thousand and a few hundred thousand plan steps; in some ways, it is perhaps fortunate that the FrameNet parser operates with a relatively low recall).

Figure 34: Snippet of hand-encoded test questions and passages

```
(object *taliban* (SENTIENT)
   (text "Taliban"))

(object *osama_bin_laden* (PERSON)
   (text "Osama Bin Laden")
   (first_name "Osama")
   (last_name "Bin Laden"))

(object *support* (ACTION)
   (root "support")
   (agent *taliban*)
   (benefactor *osama_bin_laden*))

(object *terrorist* (PROPERTY)
   (text "terrorist"))

(object *terrorist_network* (RELATION)
   (left *network*)
   (right *terrorist*)
   (type *PROPERTY*))

(object *his_network* (RELATION)
   (left *osama_bin_laden*)
   (right *network*)
   (type *POSSESSION*))
```

Table 7: Results of QAR test question analysis

Question	Answer	Total states	Unique states	Plan steps
Bin Laden/Al-Qaeda	Bin Laden leads Al-Qaeda	1,079	1,079	9,206
Al-Qaeda/Taliban	The Taliban supports Bin Laden Bin Laden leads Al-Qaeda	3,284	3,284	38,367
WMD	Chemical and nuclear	333	322	1,676

4.3.2 Linguistic Reasoner

To present an evaluation of the process of operating the entire FLOOD system, it is helpful to trace the progress of a single question/answer pair through the system. One of the shortest yet most relevant sentences in the Al-Qaeda subcorpus is, "They say evidence exists Al-Qaeda sought nuclear material." This pairs nicely with the question of interest, "What types of weapons of mass destruction has Al-Qaeda sought?" For the purposes of this evaluation, we can rewrite these as

They say evidence that Al-Qaeda sought nuclear material exists.

to simplify processing by the Text Processor (and to minimize the resultant output). These are passed to the LR through the XML format discussed in Section 3.3.1, which in turn invokes the TP to produce appropriate linguistic data. The functions and argument structure formed at this point can be seen in Tables 8 and 9. In the parses for the question, "what" has been marked as a question word; this indicates that its syntactic and semantic position corresponds to the desired information in the QAR domain.

Table 8: Functions provided to the LR by the TP

They say evidence that Al-Qaeda sought nuclear material exists.	What weapons has Al-Qaeda sought?
SUBJ say they	DET weapons what
COMPL say exists	OBJ sought weapons
SUBJ exists evidence	AUX sought has
COMPL evidence sought (that)	SUBJ sought Al-Qaeda
SUBJ sought Al-Qaeda	
OBJ sought material	
MOD material nuclear	

Table 9: Semantic roles provided to the LR by the TP

They say evidence that Al-Qaeda sought nuclear material exists.	What weapons has Al-Qaeda sought?
INFORMATION say exists	PROPERTY weapons what
AGENT say they	AGENT sought Al-Qaeda
THEME exists evidence	PURPOSE sought weapons
PROPERTY evidence sought	
AGENT sought Al-Qaeda	
PURPOSE sought material	
PROPERTY material nuclear	

As discussed in 3.3.1, the LR takes this linguistic information and populates the QAR domain with converted objects. A sample of the objects generated in this case can be seen in Figure 35 in their Lisp-serialized form; note that the generated objects are not given names, since they reference each other directly.

5 Discussion

Since the FLOOD system largely represents an implementation rather than a theoretical problem, most of its interesting aspects have been discussed in Section 3. It remains here to address the strengths and weaknesses of the system, as well as to discuss potential future directions for development.

5.1 Drawbacks

The two largest and clearest issues encountered during work with FLOOD were efficiency and fragility. Countless factors combine to make the overall execution of the FLOOD system significantly slower than necessary. In the Text Processor, each interface between the TP and a wrapped parser binary incurs a time penalty, as does each interconversion between XML and Java objects; XML may be convenient for human readability, but it is quite inefficient as a protocol for rapid data exchange. The Planning Platform's implementation is itself far less rapid than it could be due to its entrenchment in Java and complex class

Figure 35: Sample of LR generated QAR domain objects

```
(*no-name-1* (EVENT)
   (text "say")
   (what *no-name-2*))

(*no-name-2* (ACTION)
   (root "say")
   (information *no-name-3*)
   (agent *no-name-4*))

(*no-name-3* (EVENT)
   (text "exists")
   (what *no-name-5*))

(*no-name-5* (ACTION)
   (root "exist")
   (theme *no-name-6*))
```

organization; this helps to support the platform's goal of ease of development, but at a relatively high cost in runtime performance. The flexibility which it offers to domain and reasoner developers also incurs a penalty in speed, since complex features like fluent data and dynamic object creation greatly reduce the efficiency with which planning can take place. Several options are available to deal with this (the blocks world example discussed in Section 4.2, for example, uses goal marking in its domain to force fluent values to be treated in a binary manner), but even the best case performance is far worse than that of a more directed planner. Similarly, the FLECS implementation used in the QAR is itself less efficient than it might be, and the whole algorithm could easily be replaced by a more efficient planning paradigm or one better suited to the tasks encountered by the QAR.

While fragility may be a less pervasive and more intermittent problem than performance, it is significantly more difficult to overcome. Beginning with the Text Processor, each layer of the hierarchy shown in Figure 5 builds on the data in its ancestors. An early error virtually guarantees that later processing will be inaccurate. In most cases, this is not a problem; the lower-level parsers are almost by definition simpler and more reliable because of this simplicity. By the time the level of functional parsing is reached, however, this becomes an issue. Semantic role filling is difficult to begin with, and any errors introduced by the Link parser only make the task more challenging. Several options are available to address this issue. The TP already does extensive normalization on the outputs of both Link and FrameNet parsing, and in many cases additional postprocessing can remove errors. Similarly, input normalization for FrameNet processing using WordNet synsets could further reduce error. Another route would be to integrate more specific, accurate semantic parsers such as [22] as they become available.

In the Planning Platform, fragility is mainly introduced in the encoding of domains and the complexity of planning algorithm implementation. FLECS, for example, may look simple as described in Algorithm 4, but its implementation requires strict history maintenance for backtracking and careful manipulation of data to avoid excessive memory consumption. The PP's debugging facilities are currently somewhat limited, making it unfortunately easy for small obstacles in domain development to become significant annoyances later during the planning process. Both of these problems are easily overcome with time, but reducing their occurrence in the future will require more extensive enhancements to make the PP more worthy of the name "platform".

Unsurprisingly, the most noticeable point of failure in the QAR is the translation process introduced in the LR. When used in isolation, the QAR tends to encounter no more serious difficulties than those introduced by the inefficiency or fragility of the PP itself. The Linguistic Reasoner's translation, however, is essentially a process of generating code (albeit very simple code) from natural language, a task inevitably fraught with

peril. This problem is evident in even the simple examples described above, and its solution likely a holistic process involving every component of the FLOOD system. Increased robustness in the TP and PP would decrease the likelihood of translation error and reduce its severity, respectively. Further development of the LR itself could easily transform it into a normalization layer much like a TP parser, reading in "external" data and performing a variety of tasks on it (rather than straightforward translation) to produce a more palatable "internal" version for the QAR.

5.1.1 Text Processing

The main disadvantage of the Text Processor as a separate tool is the uniform data model which it imposes on its component processors. While the runtime restrictions of the TP are fairly light, its data model makes several broad assumptions about the form of linguistic data: syntactic parses will be labeled trees, functional and semantic parses will be binary relations between tokens, and so forth as described in Section 3.1.3. While none of the restrictions are so heavy as to make any obvious types of analysis impossible, they can make some processes simpler or more complex than others. It would be conceivable, for example, to include a generic "token span" data type which could be used by syntactic parsers, named entity taggers, chunkers, and other processors operating on a level between single tokens and full sentences. Such processors can currently operate using property annotations, but this loses a level of detail which might be available in a different data model.

Less intrusive, but also less easily fixable, is the TP's broadness of purpose: it simultaneously masquerades as a uniform parser with no internal components and as an empty shell consisting of nothing but internal components. Even worse, these components can easily develop interdependencies between each other where, given the uniform data model, none should exist; in principle, any syntactic parser should work equally well (at least functionally) with any functional or semantic parser, but in practice this will rarely be the case. This is not so much a flaw in the Text Processor, however, as it is an unavoidable statement about any modular software system. It is significant enough to warrant an explicit statement here, but aside from simple applications of diligence and testing, it is difficult to remedy in practice.

5.1.2 Question Answering

The efficiency issues discussed above are most likely the greatest drawback of the Planning Platform. Again, the FLOOD PP is most suited to lengthy, research-oriented tasks such as question answering because the domain and runtime models encourage generality and flexibility in favor of performance. The intent of this focus is to allow domain knowledge to guide planning where even extremely efficient, restricted planners would fail. When searching a large body of documents or a semantic space, for example, the domain knowledge and task specificity which the FLOOD PP allows will hopefully provide better overall performance (and understandability) than a more efficient planner which must still search exhaustively.

5.2 Advantages

FLOOD's components each tend to focus on two key features: modularity and flexibility. While these properties are the cause of the system's primary drawbacks, they are also perhaps its greatest strengths. The Text Processor and Planning Platform both represent autonomous elements able to be used productively in contexts outside of FLOOD or, indeed, question answering. The Text Processor pipeline represents a valuable combination of features for arbitrary parsing problems. Its data hierarchy provides a useful, normalized representation for a variety of data, similar to but more general than that discussed in [8]. Similarly, its control structure's focus on pluggability allows a wide variety of tools to be used as parsers (other annotators alongside of the current BBN IdentiFinder tagger, for example) and for new parsers to be added with minimal effort.

One aspect of this not apparent from the Text Processor's implementation is the trade-off which it allows between automatic and knowledge-based parsing. Several of the currently implemented parsers (such as

CLAWS, for example), are largely statistically based and involve little to no postprocessing by the TP; the entire process of generating and incorporating their output into the TP's data hierarchy is performed in an unsupervised manner. Others, however, particularly the complex Link and FrameNet parsers, involve dozens or hundreds of knowledge-based rules to properly generate and interpret their results. The Link grammar parser, for example, has a lexicon of rules which is manipulable by hand, and its TP wrapper uses hand-developed rules for normalization of its output. This is, of course, far more tedious than unsupervised parsing, but it also helps support the extremely high level of accuracy seen in the Text Processor's functional output. It is this type trade-off which the Text Processor seeks to allow: a simple baseline can provide adequate performance, but additional effort is possible and produces additional rewards.

In a similar manner, the Planning Platform provides an environment which is a compromise between flexibility, performance, and complexity. On one hand, it provides a relatively minimal planning environment free from the developmental constraints of more advanced planning systems; on the other, it provides a great deal of generality in representation and is as agnostic regarding specific planning algorithms as possible. While its incremental nature makes implementation of continuous or real-time planning systems impractical, it allows for any number of discrete reasoners like FLECS to be implemented, inspected, and compared in otherwise identical contexts. This, of course, is key to the QAR's application of the Planning Platform to question answering. A vital aspect of future QAR development is the ability to quickly and easily interchange planning algorithms, strategies, and domains for question answering - all of which is possible through the PP.

One aspect of the Planning Platform that exemplifies this trade-off is its ability to incorporate arbitrary Java code in the form of client-defined operators. This is not unlike the PRODIGY system's incorporation of Lisp code into its operators, and it incurs similar penalties. Using this capability, FLOOD can communicate with an external question answering system such as JAVELIN to perform subquestions - but such operations are completely opaque to the planner, and its search space is greatly impacted by their presence. In the future, this will provide an interesting opportunity to explore planning strategies in a QA environment, which, unlike many planning environments of interest, is completely discrete, but can still involve missing information, opaque operators, and extremely broad search spaces.

Of course, the optimistic view of this complexity is that it allows the introduction of extremely powerful operations into the traditionally less intricate QA task. To address scenario-based question answering, domain-specific information can be encoded directly into the planning domain used by the QAR (as was the information regarding weapons of mass destruction in Section 4.3.1). The types of inference allowed by this environment greatly simplify certain general information extraction problems as well. In the sentence, "John hits the ball," finding the relationship between "John" and "the ball" is quite difficult without linguistic knowledge; for the QAR, it is relatively easy to transform this action into a "hitter" relationship (as was done regarding Osama Bin Laden's leadership in Section 4.3.1). Like the more limited theorem proving rules which have proven so successful for [43], it is our hope that the additional power and linguistic information available to the QAR in this environment will continue to enhance question answering performance in the future.

5.3 Future work

Although many of them have already been alluded to in previous sections, there are countless points still open for future development in FLOOD. These are broken down most conveniently into implementation issues (such as the performance problems discussed in Section 5.1), theoretical issues, and experimentation. All of these represents enhancements to or explorations with FLOOD that simply require more time, and their answers or resolutions will undoubtedly continue to fuel further interesting work with the system after they have been addressed.

5.3.1 Implementation

The clearest issue which must be corrected is the lack of performance present in essentially all of FLOOD's subsystems. The bodies of the Text Processor and Planning Platform could be reimplemented in a more

efficient programming language. In many cases, the interfaces of TP wrappers with their respective external parsers could be streamlined. The type and amount of data manipulation in the TP could likely be reduced (e.g. to avoid direct handling of XML where possible), and the class interactions in the PP could certainly be simplified without losing their generality.

The accuracy of semantic role filling in the Text Processor needs to be addressed. This could be done by hand-coding further rules for normalization of FrameNet data, adding further similar resources to the system (such as PropBank [31] or VerbNet [32]), by incorporating more sophisticated external semantic parsers (such as [22]), or by a combination of these methods.

The Planning Platform needs to be given more robust error handling and debugging mechanisms in order to serve more properly as a platform. This includes better error reporting, more functionality offered through the GUI, and more sophisticated interaction between the platform and component domains and reasoners (e.g. an error in a planning algorithm or domain should not adversely affect the PP).

The Linguistic Reasoner needs to be expanded to address the fragility issues raised in Section 5.1. This can be approached from two sides: by making the LR more capable of producing output appropriate for the QAR and by adapting the QAR to be more accepting of errors and omissions by the LR. Of course, every improvement in semantic role filling makes this problem both less critical and less difficult as well.

5.3.2 Theory

The normalization scheme used by the Text Processor should be more fully explored. This includes primarily the functional and semantic roles discussed in Section 3.1.3, but also several areas that are currently underdeveloped; part-of-speech tags and constituent parses are minimally normalized, for example. Ensuring that the TP provides a robust data representation methodology will greatly aid incorporation of more complex component processors in the future.

Care should be taken to represent appropriate planning features in the Planning Platform. While the choice of features exemplified by the FLOOD name - fluents, object-orientation, and dynamism - provide the basics necessary for complex planning, they are neither comprehensive nor optimally represented in the current system implementation. Further work could easily make the PP a more appropriate environment for general planning education and experimentation with reference implementations of complex planners.

The tiny ontology represented in the Question Answering Reasoner's planning domain is obviously incomplete. With increased performance from the PP, this could be expanded into a full upper ontology. This process would need to be balanced against development in the LR, however, since any increased complexity in the target representation would likely lead to increased fragility in the translation process.

5.3.3 Experimentation

The FLOOD system must be evaluated in a more integrated environment. This includes further testing through the LR as well as integration with a full external question answering system such as JAVELIN.

In a similar vein, assembly of a larger linguistic test corpus for maintenance and development of the Text Processor would be both useful for development and educational as an evaluation of its representational power.

Implementation and evaluation of other planning algorithms for question answering will eventually be key to FLOOD's usefulness in a question answering environment. This includes changes to the planning algorithm (currently FLECS), the small question answering ontology represented in the QAR's domain, and the variety of operators available to the QAR during planning for question answering. This would also provide an opportunity to investigate the effects of more careful operator cost analysis (e.g. answering a subquestion is extremely expensive, while generating a relationship from an action is not).

5.4 Conclusions

The FLOOD (Fluent, Linguistic, Object-Oriented, and Dynamic) system represents an integrated framework for experimentation with complex question answering. It provides a Text Processor for linguistic analysis, consisting of a modular assembly of parsers capable of transforming raw text into a high-level, normalized semantic representation. FLOOD also provides a Planning Platform, which represents an environment abstracting away the low-level components of planner development (domain handling, state space maintenance, etc.) and providing a convenient way to implement arbitrary and highly customizable planning algorithms. In FLOOD, these are represented by the Question Answering Reasoner, a combination of a customized planning algorithm and domain made to accept data from the Text Processor (as translated by the Linguistic Reasoner) and perform analysis and inference to produce answers to complex questions.

A Appendix A: The Al-Qaeda Subcorpus

Using the document identifiers from the CNS corpus, the Al-Qaeda subcorpus consists of the following documents:

29277 31186 31195 31350 31417 31430 31508 31520 31521 31529 31535 31536 31541 31542 31543 31546 31551 31553 31554
31570 31576 31577 31579 31592 31595 31600 31606 31628 31640 31642 31644 31645 31651 32358 32678 32679 32681 32682

Sample questions relevant to this domain include:

1. How has Al-Qaeda conducted its efforts to acquire a weapons of mass destruction (WMD) capability, and what are the likely results of this endeavor?
2. What is Al-Qaeda?
3. What organizations are associated with or related to Al-Qaeda?
4. What are the natures of these associations?
5. What is the relationship between Al-Qaeda and the Taliban?
6. What is Osama Bin Laden's primary organization?
7. What has Al-Qaeda done in order to acquire WMD?
8. What types of WMD has it sought?
9. From what organizations has Al-Qaeda sought chemical, biological, or radiological agents?
10. How did it find these sources?
11. How has it acquired these agents?
12. What is the difference between acquiring a chemical, biological, or radiological agent and acquiring a weapon, and which has Al-Qaeda already acquired?
13. What steps are required to create a biological weapon using anthrax?
14. Under what circumstances would Al-Qaeda use a weapon of mass destruction?
15. What type of WMD attack by Al-Qaeda is most likely?

B Appendix B: A Simple QA Domain

```
; What is the occupation of Bill Clinton's wife?
; Bill Clinton's wife is Hillary Clinton.
; Hillary Clinton is a senator.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Classes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(class ENTITY ()
     (text (string)))
     (class PERSON (ENTITY)
     (first_name (string))
     (last_name (string)))

(class RELATION ()
     (left (ENTITY))
     (right (ENTITY))
     (name (string)))

(class CONSTRAINT ()
     (type (string))
     (value (fobject)))

(class SLOT ()
     (name (string))
     (type (fclass)))

(class QUESTION ()
     (constraints (list))
     (slots (list))
     (answers (list))
     (atype (string)))

(class SUBQUERY (QUESTION)
     (id (number)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Actions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(action FillSlot
     (params (<o> fobject) (<q> QUESTION))
     (conditions
          (forall (<c> CONSTRAINT) [constraints <q>]
               (SatisfiesConstraint <o> <c>))
          (:type "fixed" or (isnull [atype <q>])
               (exists (<s> SLOT) [slots <q>]
                    (ContainsSlot <o> <s> [atype <q>]))))
     (modifies [answers <q>])
     (body
          (ExtractSlots <o> <q>)))

(action ProcessSubquery
```

```
     (params (<c> CONSTRAINT))
     (conditions
          (:type "fixed" equal (cvalue [value <c>]) ~SUBQUERY)
          (not (isnull [answers [value <c>]])))
     (modifies [value <c>])
     (body
          (set [value <c>] (first [answers [value <c>]]))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Procedures
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(procedure SatisfiesConstraint
     (params (<o> fobject) (<c> CONSTRAINT))
     (body
          (and (not (equal (cvalue [value <c>]) ~SUBQUERY))
               (mhas <o> [type <c>])
               (equal (mbyname [type <c>] <o>) [value <c>]))))

(procedure ContainsMember
     (params (<o> fobject) (<s> string))
     (body
          (exists (<m> fmember) (fmembers <o>)
               (equal (mname <m>) <s>))))

(procedure ContainsSlot
     (params (<o> fobject) (<s> SLOT) (<c> string))
     (body
          (and (ContainsMember <o> [name <s>])
               (equal (cvalue (mbyname [name <s>] <o>)) [type <s>])
               (ContainsMember (mbyname [name <s>] <o>) <c>))))

(procedure ExtractSlots
     (params (<o> fobject) (<q> QUESTION))
     (body
          (forall (<s> SLOT) [slots <q>]
               (and (or (isnull [atype <q>])
                    (ContainsSlot <o> <s> [atype <q>]))
               (push (mbyname [name <s>] <o>) [answers <q>])))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Problem
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Passage
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(object *bill* (PERSON)
     (text "Bill Clinton")
     (first_name "Bill")
     (last_name "Clinton"))

(object *hillary* (PERSON)
```

```
     (text "Hillary Clinton")
     (first_name "Hillary")
     (last_name "Clinton"))

(object *senator* (ENTITY)
     (text "senator"))

(object *wife_relation* (RELATION)
     (left *bill*)
     (right *hillary*)
     (name "wife"))

(object *senator_relation* (RELATION)
     (left *hillary*)
     (right *senator*)
     (name "occupation"))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Question
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(object *sq_constraint_01* (CONSTRAINT)
     (type "left")
     (value *bill*))

(object *sq_constraint_02* (CONSTRAINT)
     (type "name")
     (value "wife"))

(object *sq_slot_01* (SLOT)
     (name "right")
     (type ~PERSON))

(object *subquery_01* (SUBQUERY)
     (id 1)
     (constraints (*sq_constraint_01* *sq_constraint_02*))
     (slots (*sq_slot_01*)))

(object *constraint_01* (CONSTRAINT)
     (type "left")
     (value *subquery_01*))

(object *constraint_02* (CONSTRAINT)
     (type "name")
     (value "occupation"))

(object *slot_01* (SLOT)
     (name "right")
     (type ~ENTITY))

(object *question* (QUESTION)
     (constraints (*constraint_01* *constraint_02*))
     (slots (*slot_01*))
     (atype "text"))
```

```
(goals
     (not (isnull [answers *question*])))
```

C Appendix C: Sample TP Output

Requesting

- Sentence separation via MXTerminator [54]
- Tokenization and functional parsing via the Link grammar parser [57]
- Part-of-speech tagging via CLAWS [35]
- Morphological analysis via the RASP toolkit's morphology processor [42]
- Named entity tagging via BBN IdentiFinder [3]
- Semantic role filling via FrameNet [29]

for the sentence

Mortimer eats macaroni.

produces the following output from the Text Processor:

```
<text id="1" source="external">
 Mortimer eats macaroni.
 <passage id="2" source="mxterminator">
  <sentence id="3">
   <property name="offset">0</property>
   <property name="ne" source="bbn">
    <property name="1">
     <property name="type">PERSON</property>
     <property name="offset">0</property>
     <property name="text">Mortimer</property>
    </property>
   </property>
   <text>Mortimer eats macaroni.</text>
   <tokens id="11" source="link">
    <token id="12">
     <left>0</left>
     <right>1</right>
     <poss>
      <pos id="16" source="claws">
       <property name="prob">1</property>
       NP1
      </pos>
     </poss>
     <text>Mortimer</text>
    </token>
    <token id="19">
     <property name="morphology" source="morpha">eat+s</property>
     <left>1</left>
     <right>2</right>
     <poss>
      <pos id="26" source="claws">
       <property name="prob">1</property>
```

```
    VVZ
   </pos>
  </poss>
  <text>eats</text>
</token>
<token id="29">
  <left>2</left>
  <right>3</right>
  <poss>
   <pos id="33" source="claws">
    <property name="prob">1</property>
    NN1
   </pos>
  </poss>
  <text>macaroni</text>
</token>
<token id="36">
  <left>3</left>
  <right>4</right>
  <poss>
   <pos id="40" source="claws">
    <property name="prob">1</property>
    .
   </pos>
  </poss>
  <text>.</text>
</token>
<syntax id="43">
  <functions id="44">
   <function id="45">
    <name>SUBJ</name>
    <token id="19"/>
    <token id="12"/>
   </function>
   <function id="49">
    <name>OBJ</name>
    <token id="19"/>
    <token id="29"/>
   </function>
   <arguments id="53" source="framenet">
    <argument id="54">
     <predicate>THEME</predicate>
     <entity id="56"/>
     <entity id="57"/>
    </argument>
    <argument id="58">
     <predicate>AGENT</predicate>
     <entity id="56"/>
     <entity id="61"/>
    </argument>
    <argument id="62">
     <predicate>ROOT</predicate>
     <entity id="56"/>
     <token id="19"/>
```

```
          </argument>
          <argument id="66">
            <predicate>ROOT</predicate>
            <entity id="57"/>
            <token id="29"/>
          </argument>
          <argument id="70">
            <predicate>ROOT</predicate>
            <entity id="61"/>
            <token id="12"/>
          </argument>
        </arguments>
      </functions>
    </syntax>
  </tokens>
 </sentence>
</passage>
</text>
```

D Appendix D: An End-to-End Example

For the sake of brevity (even in an appendix, pages upon pages of TP analysis would be a bit excessive), we will address here a simple question/answer pair as might be processed by the FLOOD Text Processor.

D.1 The Question

Eric has been hired by the Pittsburgh police to investigate a death. It seems that his hapless coworker, Bob, was walking home one evening when he was buried under a pile of Sony Vaios which suddenly collapsed as he passed underneath. Initial investigation has revealed that this mishap was no coincidence: someone has murdered Bob. Eric immediately takes the opportunity to test the JAVELIN question answerer, which has just been augmented by the addition of the FLOOD system. So, from the scene of the crime, Eric fires up his trusty Dell laptop and enters the question:

```
Who killed Bob?
```

Fortunately, the JAVELIN Retrieval Strategist has been provided with an extensive library of documentation regarding the case. This consists solely of a police memo containing the following text:

```
Michael Dell killed Bob.
```

The existing JAVELIN pipeline is stumped by this hopelessly intricate situation and, after much consideration, decides to send this information to FLOOD for further processing.

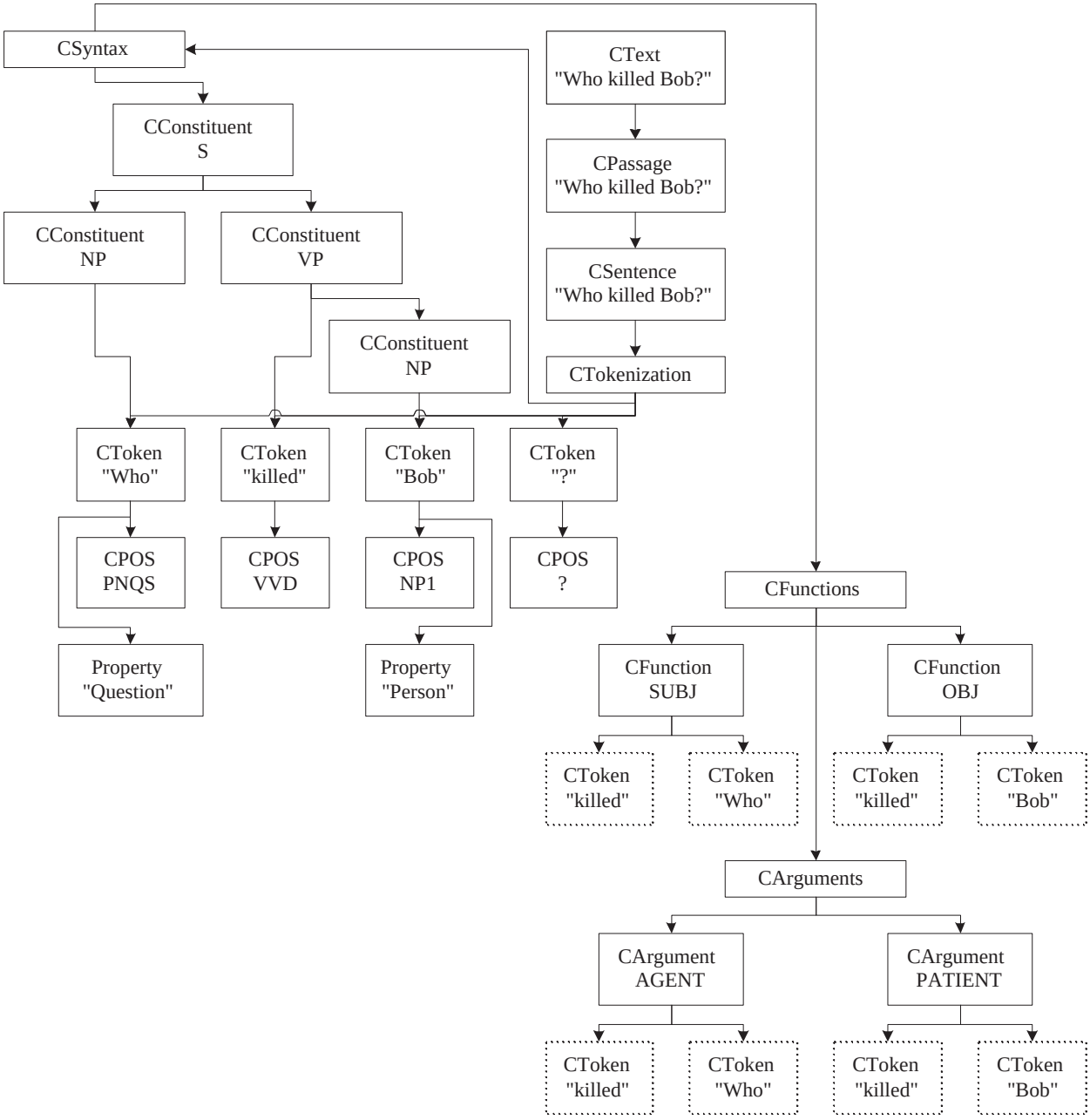
D.2 Text Processing

In preparation for analysis by FLOOD and the QAR, the text of the question and all potentially relevant passages must first be processed by the TP. Using the same hierarchical notation described in Section 3.1.3, this results in the element tree seen in Figure 36 (represented interchangeably as XML or Java objects while inside the JAVELIN/FLOOD pipeline). Note that this tree only details the results for, "Who killed Bob?" The results for, "Michael Dell killed Bob," will be nearly identical, save for an additional token (caused by the multiple word entity "Michael Dell") and the loss of the "Question" marker on "Who". The additional token will be combined into a single entity at the functional and argument levels.

D.3 Preparation for planning

While interesting, this linguistic melange is not directly useful to the planning internals of FLOOD. Instead, the LR and its translation layer sits between FLOOD and the rest of the JAVELIN universe, (as well as the TP). This layer accepts TP output and produces basic entities for insertion into the QAR planning domain. This relies primarily on the theta role argument structure produced by the TP; as mentioned previously, actions become events, nouns become entities, etc. We can use the sample domain from Appendix B as a stand-in for the full QAR domain. In this case, the passage, "Michael Dell killed Bob," is processed into the following elements:

Figure 36: Example TP results for “Who killed Bob?”



```
(object *michael_dell* (PERSON)
(text "Michael Dell")
(first_name "Michael")
(last_name "Dell"))

(object *bob* (PERSON)
(text "Bob")
(first_name "Bob"))

(object *kill_event* (EVENT)
(text "killed")
(agent *michael_dell*)
(patient *bob*)
(root "kill")
(time "past"))
```

Similarly, the question, "Who killed Bob?" becomes the problem description:

```
(object *constraint_01* (CONSTRAINT)
(type "patient")
(value *bob*))

(object *constraint_02* (CONSTRAINT)
(type "root")
(value "kill"))

(object *constraint_03* (CONSTRAINT)
(type "time")
(value "past"))

(object *slot_01* (SLOT)
(name "agent")
(type ~PERSON))

(object *question* (QUESTION)
(constraints (*constraint_01* *constraint_02* *constraint_03*))
(slots (*slot_01*))
(atype "text"))
```

These could easily be replaced by more complex representations; the "kill" action and "past" time, for example, can be represented as more complex objects in the system, the former as an "action" object containing information about "kill" (synonyms, etc.) and the latter as a more complex time calculus marker[25].

After instantiation of these objects for insertion into the QA domain, it is the responsibility of the LR to invoke the PP and prepare it for operation of the QAR.

D.4 Planning

The Planning Platform's role in question processing is, after all is said and done, somewhat peripheral. Even the LR translation layer knows more about the question and its answer than the PP does; all that's left for it to do is load up some domain (about which it knows nothing), some reasoner (again with no knowledge), hook the two up, and let the reasoner run. This is orchestrated by the LR, which:

- Instantiates the PP.

- Loads the baseline QAR domain (here represented by the domain in Appendix B).
- Inserts question-specific objects and the problem statement into the domain (described above).
- Loads the Question Answering Reasoner.
- Invokes the PP runtime.

All other PP operation during QA is carried out in the context of the QAR.

D.5 The Question Answering Reasoner

The basic operation of the QAR can be inferred from the actions seen in Appendix B and the question-specific objects described above. As a planning problem, a single goal must be achieved (producing at least one answer to the question) without violating any of the constraints. In the question, "Who killed Bob?", the implied constraints are:

- A human agent was involved.
- The action in question is a "killing".
- The event occurred in the past.
- The patient being "killed" was "Bob".

The first constraint is a consequence of the question specifier, "Who", and is thus translated by the LR (and, indirectly, the TP) into the answer type (or slot) for the question. The other three constraints appear explicitly as properties of the event for which the QAR must search. In this case, none of the subquery actions outlined in the toy domain are necessary; the inherent dependency satisfaction mechanism in the planning algorithm is able to see that of all available objects (populated from the available passages), only the `*kill_event*` satisfies these requirements and possesses an `agent` capable of filling the question's slot. Thus, the `*question*` object is modified to contain `*michael_dell*` in its answer list and processing stops.

D.6 The Culprit

At this point, all that remains to be done is some clean-up work by the LR communications layer. Once processing stops in the QAR (and thus the PP), the LR inspects the QAR domain. If one or more objects have been inserted into the question object's answer list, the LR extracts them, retrieves their text ("Michael Dell" in this case), and returns it to JAVELIN through normal XML and socket-based communications channels.

After several minutes of chugging, Eric is awarded with this answer, which he relays back to the police. They award him with the Picksburghers Yinz Done Good medal of honor and dispatch a posse to Texas to round up the suspect. Eric returns to work with Bob, who stages a miraculous recovery and picks up one of the Vaios as a souvenir - it was about time for a new one anyhow.

E Appendix E: XML Property List Format

The DTD for XML property lists as specified by Apple Computer is as follows:

```
<!ENTITY % plistObject "(array | data | date | dict | real | integer |
   string | true | false )" >
<!ELEMENT plist %plistObject;>
<!ATTLIST plist version CDATA "1.0" >
<!ELEMENT array (%plistObject;)*>
<!ELEMENT dict (key, %plistObject;)*>
<!ELEMENT key (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT data (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT true EMPTY>
<!ELEMENT false EMPTY>
<!ELEMENT real (#PCDATA)>
<!ELEMENT integer (#PCDATA)>
```

References

[1] ATTARDI, G., CISTERNINO, A., FORMICA, F., SIMI, M., AND TOMMASI, A. PiQASso: Pisa question answering system. In *Proceedings of TREC-2001* (2001).

[2] BACCHUS, F., AND KABANZA, F. Planning for temporally extended goals. In *Proceedings of the National Conference on Artificial Intelligence* (1996), pp. 1215–1222.

[3] BBN. BBN IdentiFinder. *http://www.bbn.com/speech/identifinder.html*.

[4] BLAHETA, D., AND CHARNIAK, E. Assigning function tags to parsed text. In *Proceedings of the 1st Annual Meeting of the North American Chapter of the ACL (NAACL)* (Seattle, WA, 2000).

[5] BLUM, A., AND FURST, M. Fast planning through planning graph analysis. *Artificial Intelligence 90* (1997), 281–300.

[6] BORTHWICK, A., STERLING, J., AGICHTEIN, E., AND GRISHMAN, R. Exploiting diverse knowledge sources via maximum entropy in named entity recognition. In *Proceedings of the Sixth Workshop on Very Large Corpora* (New Brunswick, New Jersey, 1998).

[7] BRILL, E. Some advances in transformation-based part of speech tagging. In *National Conference on Artificial Intelligence* (1994), pp. 722–727.

[8] CARROLL, J. High precision extraction of grammatical relations. In *Proceedings of the 19th International Conference on Computational Linguistics* (Taipei, Taiwan, 2002).

[9] CARROLL, J., MINNEN, G., AND BRISCOE, E. *Treebanks: Building and Using Syntactically Annotated Corpora*. Dordrecht: Kluwer, 2002, ch. Parser evaluation using a grammatical relation annotation scheme.

[10] CHANOD, J.-P., AND TAPANAINEN, P. A non-deterministic tokeniser for finite-state parsing. In *ECAI Workshop on Extended Finite State Models* (Budapest, 1996).

[11] CHARNIAK, E. *Statistical Language Learning*. MIT Press, 1993.

[12] CLIFTON, C., GRIFFITH, J., AND HOLLAND, R. GeoNode: An end-to-end system from research components. In *ICDE Demo Sessions* (2001), pp. 12–14.

[13] COLLINS, M. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics* (1996).

[14] CURRIE, K., AND TATE, A. O-Plan: The open planning architecture. *Artificial Intelligence 52* (1991), 49–86.

[15] DORR, B. J. The use of lexical semantics in interlingual machine translation. *Journal of Machine Translation 7*, 3 (1992), 135–193.

[16] DURME, B. V., HUANG, Y., KUPSC, A., AND NYBERG, E. Towards light semantic processing for question answering. In *Proceedings of the HLT-NAACL 2003 Workshop on Text Meaning* (Edmonton, Canada, May 2003).

[17] EDELKAMP, S., AND REFFEL, F. Deterministic state space planning with BDDs. In *Proceedings of the 5th European Conference on Planning* (Durham, United Kingdom, September 1999), Springer-Verlag, pp. 81–92.

[18] FELLBAUM, C., Ed. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[19] FIKES, R., AND NILSON, N. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2* (1971), 189–208.

[20] GAIZAUSKAS, R., WAKAO, T., HUMPHREYS, K., CUNNINGHAM, H., AND WILKS, Y. Description of the LaSIE system as used for MUC-6. In *Proceedings of the Sixth Message Understanding Conference* (1995), Morgan Kaufmann, pp. 207–220.

[21] GATES, D., LAVIE, A., LEVIN, L. S., WAIBEL, A., GAVALDA, M., MAYFIELD, L., WOSZCZYNA, M., AND ZHAN, P. End-to-end evaluation in JANUS: A speech-to-speech translation system. In *ECAI Workshop on Dialogue Processing in Spoken Language Systems* (1996), pp. 195–206.

[22] GILDEA, D., AND JURAFSKY, D. Automatic labeling of semantic roles. *Computational Linguistics 28*, 3 (2002), 245–288.

[23] GREEN, B., WOLF, A., CHOMSKY, C., AND LAUGHERTY, K. BASEBALL: an automatic question answerer. In *Proceedings of the Western Joint Computer Conference* (1961), pp. 219–224.

[24] GRISHMAN, R., AND SUNDHEIM, B. Message understanding conference-6: A brief history. In *Proceedings of the 16th International Conference on Computational Linguistics* (Copenhagen, Denmark, 1996).

[25] HAN, B., AND KOHLHASE, M. A time calculus for natural language. In *The 4th Workshop on Inference in Computational Semantics* (Nancy, France, September 2003).

[26] HARABAGIU, S. M., MOLDOVAN, D. I., PASCA, M., MIHALCEA, R., SURDEANU, M., BUNESCU, R. C., GIRJU, R., RUS, V., AND MORARESCU, P. The role of lexico-semantic feedback in open-domain textual question-answering. In *Meeting of the Association for Computational Linguistics* (2001), pp. 274–281.

[27] HUANG, Y., AND MITAMURA, T. Utility of question analysis in JAVELIN question answering system. Unpublished report, 2002.

[28] ITTYCHERIAH, A., AND ROUKOS, S. IBM's statistical question answering system - TREC-11. Unpublished report, 2002.

[29] JOHNSON, C. R., FILLMORE, C. J., PETRUCK, M. R. L., BAKER, C. F., ELLSWORTH, M., RUPPENHOFER, J., AND WOOD, E. J. *FrameNet: Theory and Practice*, 1 ed. International Computer Science Institute, September 2002.

[30] KAPLAN, J. Designing a portable natural language database query system. *ACM Transactions on Database Systems 9*, 1 (March 1984), 1–19.

[31] KINGSBURY, P., PALMER, M., AND MARCUS, M. Adding semantic annotation to the Penn TreeBank. In *Proceedings of the Human Language Technology Conference* (San Diego, CA, 2002).

[32] KIPPER, K., DANG, H. T., AND PALMER, M. Class-based construction of a verb lexicon. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence* (Austin, TX, 2000).

[33] KOSKENNIEMI, K. *Two-level Morphology: A General Computational Model for Word-form Recognition and Production*. PhD thesis, University of Helsinki, 1983.

[34] LAVIE, A. *GLR*: A Robust Grammar-Focused Parser for Spontaneously Spoken Language*. PhD thesis, Carnegie Mellon University, 1996.

[35] LEECH, G., GARSIDE, R., AND SAMPSON, G., Eds. *The Computational Analysis of English: A Corpus-Based Approach*. Longman, 1987.

[36] LEHNERT, W., CARDIE, C., FISHER, D., RILOFF, E., AND WILLIAMS, R. MUC-3 test results and analysis. In *Proceedings of MUC-3* (May 1991), Morgan Kaufmann, pp. 116–119.

[37] LEHNERT, W., FISHER, D., MCCARTHY, J., RILOFF, E., AND SODERLAND, S. MUC-4 test results and analysis. In *Proceedings of MUC-4* (June 1992), Morgan Kaufmann, pp. 151–158.

[38] LEHNERT, W. G. A conceptual theory of question answering. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence* (Cambridge, MA, August 1977), R. Reddy, Ed., William Kaufmann, pp. 158–164.

[39] LITOWSKI, K. C. Question-answering using semantic relation triples. In *Proceedings of TREC-8* (1999).

[40] MCDERMOTT, D. The planning domain definition language. In *Proceedings of the Artificial Intelligence Planning Systems Competition* (1998).

[41] MIKHEEV, A. Tagging sentence boundaries. In *Proceedings of the 6th Applied Natural Language Processing Conference* (Seattle, Washington, April 2000).

[42] MINNEN, G., CARROLL, J., AND PEARCE, D. Applied morphological processing of English. *Natural Language Engineering 7*, 3 (2001), 201–223.

[43] MOLDOVAN, D., HARABAGIU, S., GIRJU, R., MORARESCU, P., LACATUSU, F., NOVISCHI, A., BADULESCU, A., AND BOLOHAN, O. LCC tools for question answering. Unpublished report, 2002.

[44] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *The First Text REtrieval Conference (TREC-1)* (1992).

[45] NAU, D., CAO, Y., LOTEM, A., AND NOZ AVILA, H. M. SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the International Joint Conferences on Artificial Intelligence* (1999), pp. 968–973.

[46] NEWELL, A., SHAW, J. C., AND SIMON, H. A. Report on a general problem-solving program. In *Proceedings of the International Conference on Infomnation Processing* (1959).

[47] NYBERG, E., MITAMURA, T., CALLAN, J., CARBONELL, J., FREDERKING, R., COLLINS-THOMPSON, K., HIYAKUMOTO, L., HUANG, Y., HUTTENHOWER, C., JUDY, S., KO, J., KUPSC, A., LITA, L. V., PEDRO, V., SVOBODA, D., AND DURME, B. V. The JAVELIN question-answering system at TREC 2003: A multi-strategy approach with dynamic planning. In *Proceedings of TREC-2003* (2003).

[48] NYBERG, E., MITAMURA, T., CARBONELL, J., CALLAN, J., COLLINS-THOMPSON, K., CZUBA, K., DUGGAN, M., HIYAKUMOTO, L., HU, N., HUANG, Y., KO, J., LITA, L., MURTAGH, S., PEDRO, V., AND SVOBODA, D. The JAVELIN question-answering system at TREC 2002. In *Proceedings of TREC-2002* (2002).

[49] PALMER, D. D., AND HEARST, M. A. Adaptive sentence boundary disambiguation. In *Proceedings of the 1994 Conference on Applied Natural Language Processing* (Stuttgart, Germany, October 1994).

[50] PEDNAULT, E. P. D. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning* (Toronto, Canada, May 1989), R. J. Brachman, H. J. Levesque, and R. Reiter, Eds., Morgan Kaufmann, pp. 324–332.

[51] PENBERTHY, J. S., AND WELD, D. S. UCPOP: A sound, complete, partial oder planner for ADL. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, B. Nebel, C. Rich, and W. Swartout, Eds. Morgan Kaufmann, San Mateo, California, 1992, pp. 103–114.

[52] PORTER, M. F. An algorithm for suffix stripping. *Program 14*, 3 (1980), 130–137.

[53] RAYNER, M. Linguistic domain theories: Natural-language database interfacing from first principles. In *Proceedings of the 2nd Symposium on Logical Formalizations of Commonsense Reasoning* (Austin, TX, 1993).

[54] REYNAR, J. C., AND RATNAPARKHI, A. A maximum entropy approach to identifying sentence boundaries. In *Proceedings of the Fifth Conference on Applied Natural Language Processing* (University of Pennsylvania, May 1997).

[55] SAMPSON, G. *English for the Computer: The SUSANNE Corpus and Analytic Scheme*. Clarendon Press, 1995.

[56] SIMMONS, R. F. Answering English questions by computer: a survey. *Communications of the ACM 8*, 1 (1965), 53–70.

[57] SLEATOR, D., AND TEMPERLEY, D. Parsing English with a link grammar. Tech. Rep. CMU-CS-91-196, Carnegie Mellon University, October 1991.

[58] SUNDHEIM, B. Overview of results of the MUC-6 evaluation. In *Proceedings of the Sixth Message Understanding Conference* (Columbia, MD, 1995), Morgan Kaufmann.

[59] SUSSMAN, G. *A computer model of skill acquision*. American Elsevier, New York, NY, 1975.

[60] TATE, A. I-X and ¡I-N-C-A¿: An architecture and related ontology for mixed-initiative synthesis tasks, 2001.

[61] THE XTAG RESEARCH GROUP. A lexicalized tree adjoining grammar for English. Tech. rep., University of Pennsylvania, 2002.

[62] THOMPSON, C., MOONEY, R., AND TANG, L. Learning to parse natural language database queries into logical form. In *Workshop on Automata Induction, Grammatical Inference and Language Acquisition* (1997).

[63] TOLLE, K. M., AND CHEN, H. Comparing noun phrasing techniques for use with medical digital library tools. *Journal of the American Society for Information Science 51*, 4 (2000), 352–370.

[64] VELOSO, M., CARBONELL, J., PÉREZ, A., BORRAJO, D., AND FINK, E. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence 7*, 1 (1995).

[65] VELOSO, M. M., AND STONE, P. FLECS: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research 3* (1995), 25–52.

[66] VOORHEES, E. The TREC-8 question answering track report. In *Proceedings of TREC-8* (1999).

[67] VOORHEES, E. M. Overview of the TREC-9 question answering track. In *Proceedings of TREC-9* (2000).

[68] VOORHEES, E. M. Overview of the TREC 2001 question answering track. In *Proceedings of the Text REtrieval Conference* (2001).

[69] VOORHEES, E. M. Overview of the TREC 2002 question answering track. In *Proceedings of the Text REtrieval Conference* (2002).

[70] VULCAN INC. HALO pilot - call for proposals, August 2002.

[71] WELD, D. S. An introduction to least commitment planning. *AI Magazine 15*, 4 (1994), 27–61.

[72] WELD, D. S., ANDERSON, C. R., AND SMITH, D. E. Extending Graphplan to handle uncertainty and sensing actions. In *Proceedings of AAAI* (1998).

[73] WINOGRAD, T. *Understanding Natural Language*. Academic Press, 1972.

[74] WOODS, W. A. Progress in natural language understanding: An application to lunar geology. In *Proceedings of AFIPS National Computer Conference* (1973), pp. 441–450.

[75] YANG, H., AND CHUA, T. The integration of lexical knowledge and external resources for question answering. In *Proceedings of TREC-2002* (2002).