Lernumgebungen für die Softwareentwicklung

- Report -

Monica Roxana Gavrila

University of Hamburg
Computer Science Department
gavrila@informatik.uni-hamburg.de
Matrikel no: 5425534

Contents

Abstract

This paper presents a web-based system (Lernumgebungen für Softwareentwicklung – LuSe) that was developed at Hamburg University, Computer Science Department during two semesters (summer semester 2003 and winter semester 2003/ 2004). It is part of the Hamburg E-Learning Initiative (ELCH). This system wants to help the student to understand PROLOG and solve PROLOG exercises. In the first part of the paper presented a short overview of Intelligent Tutoring Systems (ITS) is, being also showed the main components of such a system. The second part describes LuSe Project and explains the technologies used.

Introduction

Currently, web-based educational systems are a challenging research and developing area. Among the benefits of web-based education are independence of teaching and learning with respect to time and space. Courses can be installed in one place and may be used by a huge number of users all over the world.

This paper introduces the LuSe Project, a web-based system built for helping students to learn PROLOG by solving several types of problems: database queries and recursion problems.

After an introduction on Intelligent Tutoring Systems, the architecture and the features of the LuSe Project will be presented.

1. Intelligent Tutoring Systems

The notion of intelligent machines for teaching purposes can be traced back to 1926 when Sidney L. Pressey[1] built a machine with multiple choice questions and answers. This machine delivered questions and provided immediate feedback to the user. Educational psychologists have since reported that carefully designed individualized tutoring produces the best learning for most people.

The predecessors of Intelligent Tutoring Systems (ITS) were known as Computed Aided Instruction or Computed Aided Learning Systems (CAIS or CALS). These traditional systems were developed to provide users with instruction in a particular area after which they were tested. Answers to questions, usually multiple choices, were used to direct the course of the study.

Recognizing the deficiencies of the traditional CAIS (These systems are incapable of dynamically generating a response to a particular situation as a human tutor would be able to do.), ITS were subsequently developed which attempted to adapt the speed and level of presentation to that required by a student.

[1] Sidney L. Pressey was a psychologist at Ohio State University and contributed to some of the earliest works of automated-instruction in the 1920s. His work was done during the time that the field of psychology and educational technology was closely related. He created devices that showed that automated-instruction facilitated learning by providing for immediate reinforcement, individual pace setting, and active responding. He commented, "teaching machines are unique among instructional aids, in that the student not merely passively listen, watches, or reads but actively responds." (Teaching Machines and Sidney Pressey, 1964)

1.1. Architecture

The architecture of an Intelligent Tutoring System is made of three components:

1. **The expert module**: This module interfaces with the domain knowledge. Domain knowledge embedded in the system represents an expert's knowledge and problem solving characteristics.
2. **The student module**: The function of this module is to capture a student's understanding or not of the domain knowledge
3. **The tutoring module**: This module contains teaching strategies and essential instructions. Strategies must be tailored by this module to the student's needs without the intervention of a human tutor. The main purpose of this module is to reduce the knowledge differences between the expert and the student to a minimum or to none.

   These modules are described more detailed in the following subsections.

1.1.1.The Expert Module

Early Intelligent Tutoring Systems incorporated an expert system and a number of systems were built around pre-existing expert systems.

Typically an expert system aims to provide expert like solutions to problems in a specific domain. An expert system will often have to deal with uncertain and incomplete information and should be able to explain its decisions and underlying reasoning, as human experts are capable of doing. Figure 1 shown below illustrates a simple expert system architecture.

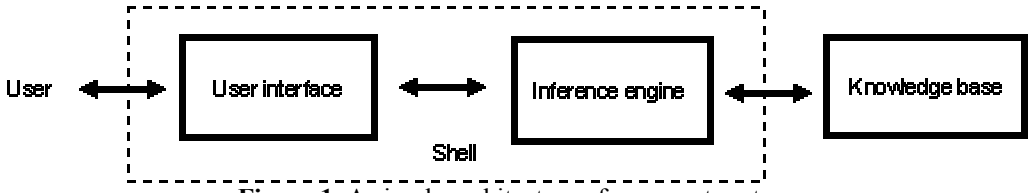


Figure 1. A simple architecture of an expert system

There are three modules within an expert system. These are the *user interface*, which caters for smooth communication between the user and the system. The second module is the *inference engine*, which is an interpreter for the knowledge base. It produces results and explanations for problems presented to it. The *inference engine* and the *user interface* are commonly viewed as a single component known as the *expert system shell*. The heart of the expert system and the final component in figure 1 is the *knowledge base,* which contains the problem solving knowledge of a particular application. The knowledge base itself is isolated from the expert system shell to allow reuse of the shell in other application domains.

A number of strategies for representing knowledge within the knowledge base have been explored:

1. **if-the rules**: If-then rules, often called *production rules*, have been by far the most widely used method of representing domain knowledge in expert systems. A typical production rule will be of the form : **IF *condition(s)* THEN *conclusion(s)***. Two strategies for reasoning with production rules, which will be performed by the inference engine, are *forward chaining* and *backward chaining*. In both cases an

inference path is sought between what is currently known and a goal. When forward chaining is employed the known starting conditions are input into the left hand side of the production rules to generate a set of conclusions. These are checked to see if they match the desired goal, if not, these conclusions are used by subsequent rules to draw further conclusions. When backward chaining is employed rules are selected whose right hand sides will achieve the desired goal. The left had sides are considered as intermediate goals that must be satisfied and rules to achieve these are accessed. This process attempts to find a situation where all the left hand sides are known to be true by the system.

2. **if-then rules with uncertainty measures**: Uncertainty measures can be used in rule based systems to indicate the level of confidence associated with a production rule. For example, suppose an ideal world with everlasting light bulbs and an interruptible power source. A production rule may state: **IF *light switch turned to on* THEN *light comes on***. In this case no uncertainty measures are required due to the unrealistic assumptions made. In the real world the power may be cut off or the bulb may blow and an uncertainty measure associated with this rule as follows: **IF *light switch turned to on* THEN *light comes on* WITH *95%* CONFIDENCE**.
3. **semantic network representations**: A semantic network represents knowledge as a set of nodes connected by labeled arcs. The arcs represent the relationships between the nodes.
4. **frame based representations**: A frame is a collection of attributes and values and can be regarded as an extension of a semantic network. The use of the frame can greatly simplify a semantic network. Frames have also been incorporated into production rule based systems with frames defining the objects that occur in the rules.

1.1.2.The Student Module

The student module forms a framework for identifying a student's current state of understanding of the subject domain. The knowledge that describes the student's current state of mind is stored in a *student model*.

In order to make any learning environment adaptable to individual learners it is essential to implement a student model within the system. The student module should permit the system to store relevant knowledge about the student and to use this stored knowledge to adapt the instructional content of the system to the student's needs.

In order to identify a student's needs a number of student modeling architectures have been devised. The differences or similarities between the expert and the student model are described in terms of misconceptions and missing conceptions. Missing conceptions can be described as some knowledge, which is possessed by the domain expert but not by the student. A misconception is knowledge that the student has but not the expert.

Student models have been devised that record either misconceptions, missing conceptions or a combination of both:

- **Overlay student models**: The *overlay* student model has been used in a number of Intelligent Tutoring Systems. It is particularly appropriate when the teaching material can be represented as a prerequisite hierarchy. Within this model the student's knowledge is assumed to be a subset of the expert's knowledge and the goal of tutoring is to enlarge this subset as shown in figure 2:

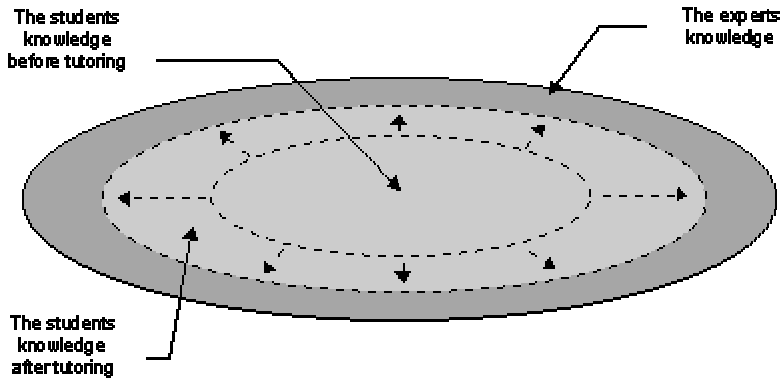


Figure 2. A representation of an overlay model showing the effects of tutoring.

This model assumes that the student will not learn anything that the expert does not know. Specifically it does not cater for misconceptions or *bugs* that the student may have or acquire during tutoring. A second issue with the overlay model is that there is no mechanism to differentiate between knowledge the student has not yet grasped and knowledge the student has not yet been exposed to which have implications for the tutoring strategy. The differential student model addresses this point

- **Differential student models**: The *differential* student model is an extension of the overlay model. Knowledge is separated into that which the student has been exposed to and that which the student has not. A differential student model is depicted in figure 3.

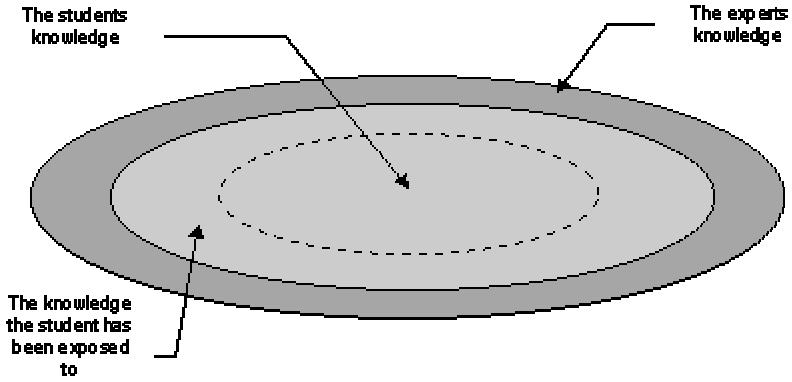


Figure 3. A representation of a differential student model.

The differential student model partitions the domain knowledge into already presented knowledge and that, which has not yet been presented to the student. An overlay model as described above is applied to that knowledge exposed to the student. As with the overlay student model, the differential model does cater for student misconceptions or bugs.

- **Perturbation student models**: The *perturbation* or *buggy* student model is depicted in figure 4. This model caters for knowledge possessed by the student that is not present in the expert domain knowledge.

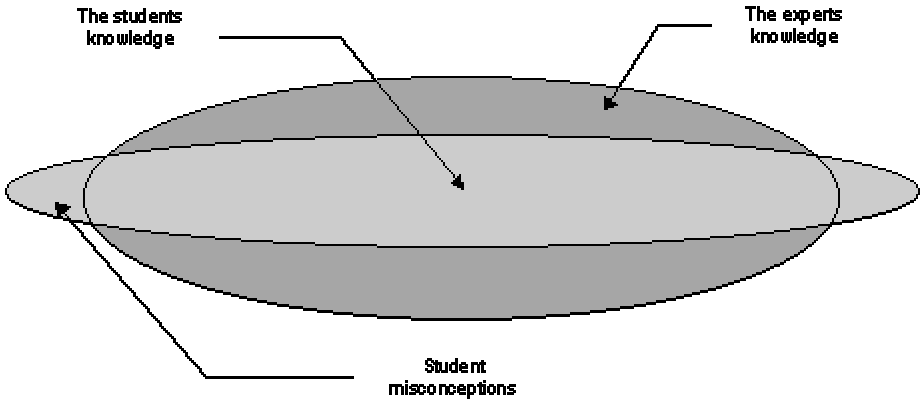


Figure 4. A representation of the perturbation or buggy student model.

The perturbation student model extends the experts knowledge with the addition of a *bug library*. The process to create a bug library can be *enumerative* or *generative*. The enumerative process lists all possible bugs usually via an analysis of the problem domain and the errors that students make. The generative approach attempts to generate bugs from an underlying cognitive theory.

1.1.3.The Tutoring Module

The function of the curriculum and instruction or *tutoring* module is three fold. Firstly it must control the presentation, ordering and selection of material most appropriate for the student. Secondly it must be able to answer questions from the student and thirdly it must determine which type of help should be given to students.

Any global tutoring strategy should be based on sound educational and psychological principals with reference to needs identified within the student model. These needs are identified by the student diagnosis process and can result in guidance or remedial instruction being feedback to the student.

Beyond the presentation of material inline with the overall tutoring strategy the system may provide feedback to the student in a number of forms:

- A facility may exist to allow the student to request help when stuck with a problem. In this situation the system should provide suitable and relevant hints.
- Assistance may be provided in the problem solving process allowing the student to concentrate on specific issues while gaining an appreciation of other relevant issues.
- The students may be given the facility to review their own decision-making processes during or after a problem-solving episode. If this facility is given during problem solving a back tracking facility should allow the errors to be corrected.
- The tutoring system may provide reactive feedback by challenging a student's decisions and forcing a justification for a decision.
- The system may provide model answers so that the student can see the expert at work.

The tutoring module may provide over the shoulder coaching by monitoring porgies and providing relevant advice. This could take the form of remedial advice, further information, and encouragement for a correct solution or a warning that the student's current solution strategy is not optimal.

Student diagnosis is the process to evolving the student model. In order to evolve the student model interactions between the student and the Intelligent Tutoring System need to be analyzed. This analysis is typically performed by checking answers to questions posed by the system or analyzing the steps taken during a problem solving session. Other factors such as requests for assistance or the analysis of browsing patterns within hypertext-based systems can also be used.

One simple approach to student diagnosis is known as *performance measuring*. Performance measuring checks a student's domain related knowledge by looking at solutions to problems. Counters within the student model can be set to indicate what has and has not yet been learned. Although this is a straight forward method of measuring a student's performance it gives reasonable clues about what type and how much information the learner will need.

Model tracing systems analyze problem solving episodes and maintain a model of problem solving which is traced against the student's activities. At a given state of problem solving, rules, which model the problem solving activities within the domain, are selected and fired to predict the next state. Expert and bug rules are fired with the goal of matching the student's new state. The student is assumed to be using the rule that predicted the new state and the student model updated to reflect this. Any deviation between the student's solution and the systems can be acted upon and suitable action taken.

A popular approach to student diagnosis has been to use an expert system. The expert system is employed to analyze answers or conclusions drawn by the student and the conclusions of the expert system are used to maintain the student model.

1.2. Conclusions

The "Traditional Trinity" (the three modules) has formed the basis for a number of Intelligent Tutoring Systems. However, in the literature it is suggested that recent trends in intelligent tutoring research do not map well to it.

These trends have seen tutoring strategies moving away from fact based tutoring towards the tutoring of problem solving strategies and the analysis of these strategies. This *meta-level* knowledge is less easy to represent within the domain knowledge base.

There has also been a move towards multiple representations of the domain knowledge in order to cater for specific situations and contexts. It has been recognized that the expert and learner work in different ways. The student will go through several phases while attaining expert problem solving skills. Alternative domain representations may be required as these phases are established and progressed through. Multiple domain representation does not map well to the overlay and bug rule based student models.

The emergence of alternative learning environments such as those that allow for experimentation and simulation and an increased range of interaction are not well catered for either by the "Traditional Trinity".

One interaction style that has recently been gaining in popularity is hypertext. There is significant advantage associated with hypertext and a number of tutoring systems have been devised that incorporate a hypertext component.

Machine learning techniques have also been used in a number of recent intelligent tutoring systems.

A student can learn from an Intelligent Tutoring System by solving problems. The system selects a problem and compares its solution with that of the student and then it performs a diagnosis based on the differences. After giving feedback, the system reassesses and updates the student skills model and the entire cycle is repeated. As the system is assessing what the student knows, it is also considering what the student needs to know, which part of the curriculum is to be taught next, and how to present the material. It then selects the problems accordingly. In the following figure, the structure of such a system is presented:

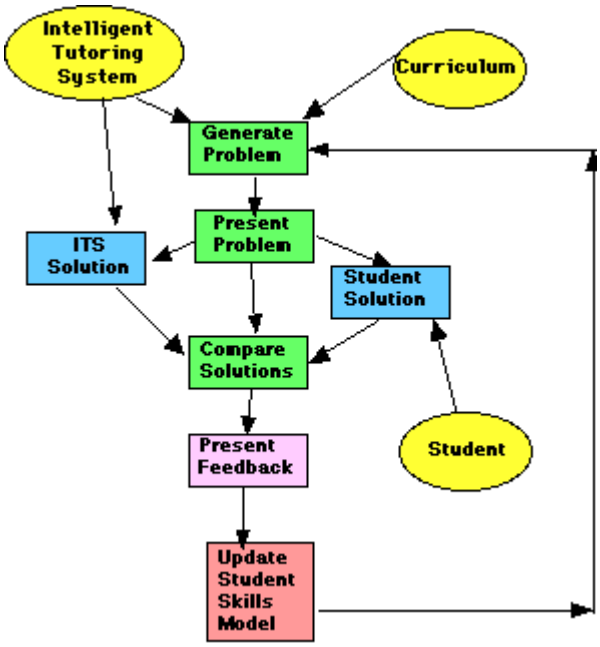


Figure 5. The structure of an Intelligent Tutoring System

1.3. Examples

In this section two examples will be shortly presented: a LISP Tutor and a web-based PROLOG Tutor (PLOT).

1.3.1.The LISP TUTOR

The LISP TUTOR is an Intelligent Tutoring System developed to teach the basic principles of programming in LISP.

In the LISP TUTOR the expert model was created as a series of correct production rules for creating LISP programs and a learner model was built as a subset of these correct production rules along with common incorrect production rules. LISP TUTOR is based on the principle of "learning by doing", where the learner discovers the productions while working through problems. The tutor acts as a problem-solving guide, but never states the productions to be learned.

LISP TUTOR is an application of Andersons ACT* theory. ACT* theory is one of the earliest attempts to establish a complete theory of human cognition. It combines declarative knowledge in the form of semantic nets with procedural knowledge in the form of production

rules. In ACT* learning is accomplished by forming new procedures through the combination of existing production rules. The main principles of the ACT theory are:

- Cognitive functions can be represented as a set of production rules. The use of a production depends on the state of the system and the current goals.
- Knowledge is learned declaratively through instructions. The learner must carry out the process of knowledge compilation if the productions are to be properly understood and integrated into their existing knowledge and later recalled and used.

Anderson and his team used GRAPES (Goal Restricted Production System Architecture) to represent the knowledge in LISP TUTOR as approximately 325 production rules. The system also embodies around 425 buggy production rules that represent misconceptions that any novice programmer can easily have. LISP TUTOR employs model tracing to provide a learner with detailed feedback. The learner is given a problem and the tutor monitors the learner's input character by character. The tutor generates all the possible next characters using both correct and buggy production rules.

- If the character is predicted by the correct rule the learner is allowed to continue.
- If the character is predicted by a buggy production rule, then remedial instructions are given.
- If the character is not predicted the tutor says that it cannot understand and asks the learner to try again. After several tries the tutor explains the next step.

This method has the advantages of early diagnosis of learner misconceptions and of giving immediate feedback to the learner. The learner never strays far from a correct solution. However, this can be viewed as unnecessarily restrictive and counter productive as the student is never allowed to explore incorrect behavior. More information about this project can be found in (Etienne Wegner 1987, chapter 13).

1.3.2.PLOT

PLOT is a web-based intelligent educational System for PROLOG developed at University of Osnabrück.

Among the goals of this project we can find:

- The tutoring system is addressed to a large number of users who learn PROLOG;
- The Possibility for students to use the system from different places;
- Enabling the user to work with a familiar working space
- Privacy of user data
- Quick and efficient system administration

It is a client-server application that used a JAVA client as user front-end. The system compares the student solution with the example solution. Error analysis is accomplished in several steps by different modules (parser analyzer, syntax checker, comparing with the example solution). For more details on how the system works and what it can offer, see (Christoph Peylo et al, 2000).

2. Lernumgebungen für die Softwareentwicklung (LuSe)

Lernumgebungen für Softwareentwicklung (LuSe) is a project developed during two semesters at Hamburg University, Computer Science department (for more details see Annex 3). The LuSe Project is a web-based system built for helping students to learn PROLOG by solving several types of problems: database queries and recursion problems.

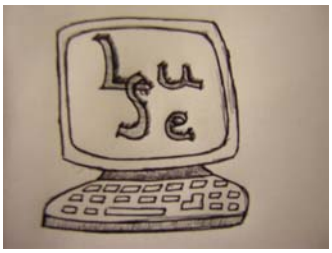


Figure 6. LuSe Logo

The LuSe Project is implemented in Prolog, and has an interface based on PLHT[2] and HTML, and it uses the XPCE server.

2.1. Descriptions of the Project

Many Computer Science students have problems with learning programming, especially logic programming. With the project LuSe, it is tried to attain the following aims:

- find out the typical difficulties that a novice programmer normally has to meet.
- build a web-based environment that reads the approach to a specified problem of the student and analyze his/her solution.
- generate the appropriate feedbacks that correspond to the concept of the programming. With the feedbacks the student can improve his/her solution successively and override the blockades of learning programming in the early phase.

2.2. The LuSe Architecture

The set of problems considered in LuSe is divided into two categories: database query problems, and recursion problems. In the following figure, the flow for recursion exercises can be seen.

The program flows for database query exercises and recursion exercises are somehow similar, the only differences being:

1. In the database case if there are syntax errors, the feedback is generated without doing other operations;
2. In the database case an instantiation checking and a simplification of unification equations are done.

[2] PLHT is the abbreviation for Prolog Hypertext Processor and it was created by Yannick Versley (student at Hamburg University, Computer Science Department).

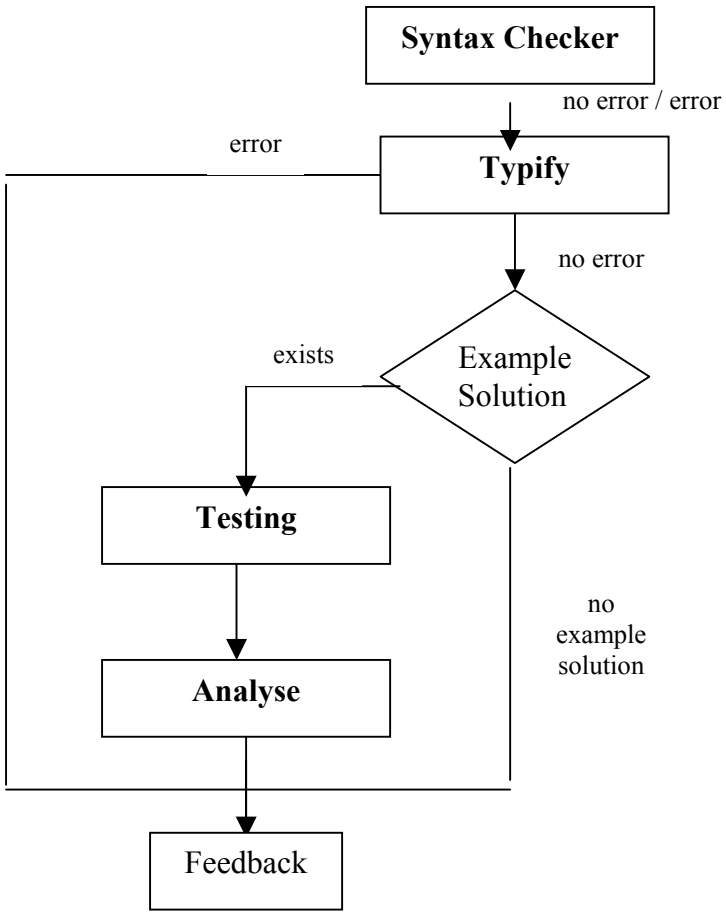


Figure 7. The general flow for recursion problems

The architecture of LuSe can be divided into three layers: front-end, backend and resource layer (see Annex 1).

The front-end layer is responsible for reading solutions to a described exercise and representing the feedbacks. Front-end works on the HTTP server of SWI-Prolog and is supported by the PLHT library written by Yannick Versley. PLHT is an approach similar to Java Server Pages that tries to make separation of presentation and application logic possible for applications written in Prolog.

The backend layer analyses student's solution and generates diagnose by comparing this solution with a sample solution. At present we just have one sample solution for one exercise. The diagnose process goes through following components: ‘normalizer’, Syntax Checker, type checker, and analyzer. The ‘normalizer’ brings the student solution back to the normal form. The Syntax Checker verifies the student's input if it is syntax conform. The semantic checking begins with type checker that examines the types used in the input. Semantic errors will be found by the analyze components as Auswertung, Idiom, Test engine. The backend layer uses a base library of session management, iterative deepening search and feedback generation.

The resource layer contains exercise descriptions and sample solutions that are saved in form of XML files and PROLOG database.

These three parts will be described in the following subsections.

2.2.1.The Front-end Layer

The LuSe Front-end is formed of the **.plht** files and of the XPCE server.

PLHT is web based user interface that can be used with any browser, without the need of any installation on the side of the user. “PLHT tries to make this separation of presentation and application logic possible for applications written in PROLOG using the HTTP support library and a compiler which transforms XML input into PROLOG predicates…” (Yannick Versley, 2003).

In PLHT there are tags for looping, if-then conditions, writing atoms and terms, calling a PROLOG predicate, handling form values, session management, and for including other files. For more details see (Yannick Versley, 2003).

XPCE is a toolkit for developing graphical applications in PROLOG and other interactive and dynamically typed languages. XPCE follows a rather unique approach of for developing GUI applications, which is summarized using the points below.

- **Add object layer to Prolog:** XPCE's kernel is an object-oriented engine that allows for the definition of methods in multiple languages. The built-in graphics are defined in C for speed as well as to define the platform-independence layer. Applications, as well as some application-oriented libraries are defined as XPCE-classes with their methods defined in Prolog. As of XPCE-5, Prolog-defined methods can receive arguments in native PROLOG data, native PROLOG data may be associated with XPCE instance-variables and XPCE errors are (selectively) mapped to PROLOG exceptions. These features make XPCE a **natural extension** to your PROLOG program.
- **High level of abstraction:** XPCE's graphical layer provides a high abstraction level, hiding details on event-handling, redraw-management and layout management from the application programmer, while still providing access to the primitives to deal with exceptional cases.
- **Exploit rapid PROLOG development cycle:** Your XPCE classes are defined in PROLOG and the methods run naturally in Prolog. This implies you can easily cross the border between your application and the GUI-code inside the tracer. It also implies you can modify source-code and recompile while your application is running.
- **Platform independent programs:** XPCE/Prolog code is fully platform-independent, making it feasible to develop on your platform of choice and deliver on the platform of choice of your users. As SWI-Prolog saved-states are machine-independent, applications can be delivered as a saved-state. Such states can be executed transparently using the development-environment to facilitate debugging or the runtime emulator for better speed and space-efficiency.

LuSe **Graphical Interface** is generated using a mixture of PLHT and HTML. In Figure 8 we can see the LuSe homepage.

The Graphical Interface is very simple and perhaps it does not offer all the support needed by the student. There are no many graphical elements (buttons, check boxes, etc.) and all the student has to do is type in the solution of the exercise into a text area and press a button. He will receive in a table the list of errors and possible solutions to them.

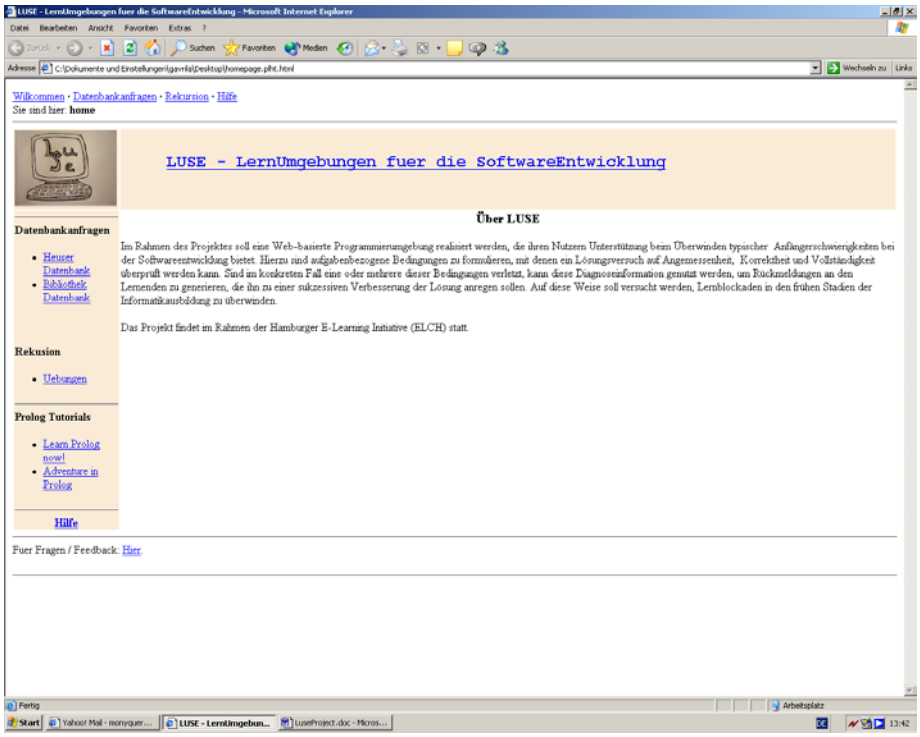


Figure 8. Snapshot with the LuSe homepage

2.2.2.The Backend Layer

The backend layer part is formed of several modules:

- LuSe Diagnose Module
- LuSe Normalize Module
- LuSe Syntax Checker
- LuSe Type Checker
- LuSe Evaluation Module
- LuSe Idiom Detector
- LuSe Testing Module
- LuSe Session Manager
- LuSe Iterative Deepening Search
- LuSe Structure
- LuSe Feedback Generator

Below are presented some of LuSe modules:

LuSe Diagnose Module

The diagnose module differs for the two types of problems (database query and recursion problems). The predicates used are:

- For database query: ***diagnose_db(+Id, +Anfrage, -ErrorList, -CorrectedTerm)***. It gathers the diagnosis information from syntax checker, type checker, and instantiation

checker. The parameters are the following: Id is the id of the exercise, Anfrage is the student solution, Errorlist contains the errors the system found in the student solution, and CorrectedTerm is the possible correction the system provides.

- For recursion: ***diagnose_rec(+Id, +Program, -Errs***). It gathers the diagnosis information from syntax checker, type checker, and the testing module and offers the user the list of errors and possible solutions.

LuSe Normalize Module

In LuSe there are several way of bringing a clause in a so-called 'normal form', and these forms are described below.

Form 1: first the head and the body of a clause are separated using the predicate ***normalize(+Klausel, -NormalisierteKlausel)*** and then, using **normalizBody(+ClauseBody, -BodyList, [])**, the body is transformed as follows:

If at the beginning the ClauseBody is of the form,
(append(Rest1,[Element|Rest2],List),length(Rest1,Pos),append(Rest1,Rest2,Rest))

it becomes a list of the form:

[append(_G989, [_G986|_G987], _G991), length(_G989, _G994), append(_G989, _G987, _G998)]

Form 2: It follows the rule. no equation of the form Var=Term occurs in the body. This is done by the predicate ***simplifyUni(What, TermsIn, TermsOut)*** that replaces the subgoals of the form Var=Expr in TermsIn by substituting Var with Expr. The result is TermsOut, and the list of simplified equations is put in What.

Form 3: The head of a clause contains only variables as an argument and the body of a clause contains equations with a variable on the left side and a functor with only variables as arguments on the other and predicate calls with only variables as arguments

LuSe Syntax Checker

LuSe Syntax Checker is based on the Backus-Naur Form (BNF)[3] definition of PROLOG syntax.

```
program ::= sentence { "." sentences}
sentence ::= head ":-" body "." | head "."
body ::= goal
goal ::= compound_term | term " " "is" " " expr
goal ::= term infix_operator2 term | term infix_operator3 term
goals ::= goal | goal {"," goals}
compound_term ::= atom "(" terms ")" | list
compound_terms ::= compound_term | compound_term {"," compound_terms}
expr ::= simpleExpr | simple_expr infix_operator simple_expr
simple_expr ::= term | "(" expr ")"
term ::= constant | variable | "_" | compound_term
terms ::= term | term {"," terms}
infix_operator ::= "+" "-" "*" "/" "mod"
infix_operator2 ::= "==" |"=" |"=\="
infix_operator3 ::= "=<"|"<" \|">"| ">=" |"@="|"=:="
```

[3] The **Backus-Naur form (BNF)** (also known as **Backus normal form**) is a metasyntax used to express context-free grammars.

list ::= "[" "]" | "[" terms "]" | "[" terms "|" term "]"
atom ::= lowercaseChar | nameChars
constant ::= integer | atom
integer ::= digit | digits
digit ::= "0" ...| "9"
variable ::= uppercaseChar | uppercaseChar nameChars
lowercaseChar ::= "a" ...| "z"
uppercaseChar ::= "A" ...| "Z"

The Syntax Checker provides following functionalities:

- ***check_program_syntax(+Program, -Clauses, -ErrorList)*** reads a PROLOG program that has several clauses and returns a list of corrected clauses and a list of errors. The error list has this form: [1-[error([0], missing())), error([1, 0], missing(]))]]. The number 1 followed by the list of errors indicates that the errors belong to the first clause.
- ***check_clause_syntax(+String, -Term, -Errors)*** reads just one clause as input and returns a corrected term with a list of errors. In this error list we will not have the number in front of error list because in this case we just have one clause.
- ***check_query_syntax(+String, -Term ,-Errors)*** does the same as *check_clause_syntax* but just be applied for a query.

LuSe Type Checker

The Module verifies if the parameters in the solution respect the type constraints. This is done by the predicate ***typify_query(Query, Errors, VarTypes)***, which uses the predicate ***get_types_from_exprs(Exprs)***, for which Expr is a list of terms from a PROLOG clause.

LuSe Evaluation Module

The evaluation is done by the predicate ***auswertung(+Loesung, +AufgabenId, -Fehler)***, which has as input the student solution and the id of the problem and as output the errors. It uses the Iterative Deepening Module for looking for the errors.

LuSe Idiom Detector

This module tries to find idioms inside a student solution and in sample solution and then compare the found idioms. If there are differences, a list of errors is given as output. It uses the predicate ***idiom_detector(Pred1, Pred2, Errs)***.

LuSe Testing Module

This module can be find in the file *test_recursive.pl*, and the main predicate is ***test_with_goal(Clauses1, Clauses2, Goal, Errors)*** that compares the solution sets of two predicates and has among the parameters Clauses1: the clauses of the sample predicates and Clauses2: the clauses of the user predicates. These two have to be lists of clauses in the form Head-[BodyGoals]. This predicate uses the predicate ***dprove(Goal, Vars, Clauses, MaxDepth, Result),*** which is a depth-limited prover whose arguments are: Goal: the goal we want to test (this can be a subgoal), Vars: what it is wanted to have in the goal, Clauses: the clauses of the predicate in Head-[Bodylist] form, MaxDepth: the depth that is left, and the result of the proving.

LuSe Session Manager

The session management is done by PLHT, which offers a limited facility for the persistence of data throughout the browsing session of the user. One can get the contents of a session variable with the PROLOG predicates ***session_getValue(+Key, -Var)*** and ***session_setValue(+Key, +Var)***. The ***session_getValue*** predicate fails if there is no given value for the given key. It can also be implemented user specifiable defaults by using the PLHT ***pl:sessionparam*** tag.

LuSe Iterative Deepening Search

Iterative deepening search is a graph search algorithm. When searching for a path through a graph from a given initial node to a solution node with some desired property, a depth-first search may never find a solution if it enters a cycle in the graph. We can either add an explicit check for cycles so that we never extend a path with a node it already contains or we can use iterative deepening where we explore all paths up to length (or "depth") N, starting from N=0 and increasing N until a solution is found.

In LuSe there are several predicates named *ids_search*, built one on top of the other. The one used in the evaluation modules is *ids_search/7´*. The arguments of the predicate ***ids_search(Query, Pos0, State0, State1, MinPenalty, MaxPenalty, Errors)*** are Query the description of the search space, Pos0 the initial position, State0 the beginning state of the object to be operated, State1 the last state, MinPenalty a minimum bound, set at 0, MaxPenalty the maximum depth bound (set at 100), and Errors the solution path of error terms.

LuSe Feedback Generator

The Feedback Generator has a database of possible feedbacks. If any diagnose component of the Backend layer finds an error in the student solution, the Feedback Generator will produce the appropriate feedback. The Feedback Generator does the following if it is called with a list of errors as input: It rewrites the student's input and locates the error in this input by emphasizing it.

A feedback is chosen which tells position where, what the error is and how it should be solved.

There are three groups of feedbacks. The first group covers the possible syntax errors (priority 1-20). The second group handles the semantic errors (priority 21-80), and the last one includes the errors that can be identified by the LuSe system (priority greater than 81).

The feedback has the following structure: ***feedback(-Priority, +ErrorTerm, -Where, -What, -How)***, where the parameters are the following: Priority of the feedback for this error, ErrorTerm has the structure error([ErrorPosition], ErrorDescription) (e.g. error(_,missing('('))), Where is the generic name of the error (this parameter is sometimes not specified), What is the short description of the error, and How a manner to handle this error. Errors, which can be identified and are solvable, have the higher priority.

Example:

```
feedback(15,
error(_,misspelled_predicate(Misspelled,Corrected)),
Misspelled,
'Dieses Prädikat gibt es nicht.',
 ['Meintest du <b>',Corrected,'</b>?']).
```

Having such a structure, the feedback messages can be easily extended. The feedback messages are ordered according to the priority and a message is shown once, even the error appears several times.

The Feedback Generator provides two functionalities:

locate_error_in_term(+ErrorList ,+Term): This reads a list of errors as input and a normalized input, and then writes into the output stream (in this case HTML) a term with emphasized positions which are errors.

return_the_feedbacklist(+ErrorList, -FeedbackList): This reads a list of errors and finds appropriate feedbacks, then returns them to Front-end.

2.2.3.The Resource Layer

In LuSe architecture there are used three types of files:

- .xml files,
- database files, and
- data types files.

.xml files are used for introducing exercises, either for database queries, or for recursion problems.

The structure of the files used for both types of exercises (database query and recursion) is somehow similar:

1. Database query

```
<! ELEMENT domain (description, aufgabe*)>
<! ATTLIST domain name CDATA #REQUIRED>
<! ATTLIST domain db CDATA #REQUIRED>
<! ATTLIST domain types CDATA #REQUIRED>
<! ELEMENT description #PCDATA>
<! ELEMENT aufgabe (text, loesung)>
<! ELEMENT text (#PCDATA, match*) *>
<! ELEMENT loesung #PCDATA>
<! ELEMENT match #PCDATA>
<! ATTLIST match var CDATA #OPTIONAL>
<! ATTLIST match constant CDATA #OPTIONAL>
<! ATTLIST match functor CDATA #OPTIONAL>
```

where:
<domain> is the root tag that has 3 attributes: name (generic name of the database), db (the database file), and types (the type file).
<description> contains a short overview on the database, so that the student can have a glimpse in LuSe interface.
<aufgabe> contains the information about the exercises.
<text> contains the text of the exercise
<loesung> contains the example solution for the corresponding exercise
<match> is used for generating feedback (the text between this tag will appear bold to the student so that he/she knows where to look in the exercise in order to make the corrections)

Example of an entry:

```
<aufgabe>
```

```
<text>
In <match var="Strasse">welchen Strassen</match> gibt es
<match constant="mfh">Mehrfamilien<match functor="obj">häuser</match>
</match>?
</text>
<loesung>
obj(_,mfh,Strasse,_,_)
</loesung>
</aufgabe>
```

2. Recursion

```
<! ELEMENT rekursion (description, aufgabe*)>
<! ATTLIST rekursion name CDATA #REQUIRED>
<! ELEMENT description #PCDATA>
<! ELEMENT aufgabe (titel, text, loesung,  testfall)>
<! ELEMENT text #PCDATA>
<! ELEMENT loesung #PCDATA>
<! ELEMENT testfall #PCDATA>
```

where:

<rekursion> is the root tag; it has an attribute name (a generic name).
<description> is a short description of the exercise types.
<aufgabe> contains information about the exercise.
<text> is the exercise (at this point there is no match tag for feedback).
<loesung> contains the example solution.
<testfall> contains a test case for the corresponding exercise, which will be used by the testing module

Example of an entry:

```
<aufgabe>
<titel>Entfernen eines Elements</titel>
<text>
Schreiben Sie ein Predikat das ein Element von eine Liste entfernt:
entfernen(Liste, Element, Liste_ohne_Element)
</text>
<loesung>
entfernen([],X,[]).
entfernen([X|Xs],X,Ys):-entfernen(Xs,X,Ys).
entfernen([X|Xs],Z,[X|Ys]):-X \= Z, entfernen(Xs,Z,Ys).
</loesung>
<testfall>
entfernen([1,2,3],1,X).
</testfall>
<testfall>
entfernen([1,2,3,2,3,2,1],2,X).
</testfall>
<testfall>
entfernen([],1,X).
</testfall>
</aufgabe>
```

The database files used at this moment are: *haeuser.xml* and *test.xml*, and the recursion file used is *recursion.xml*.

The database query exercises are taken from the P1 home-works (Hamburg University, Computer Science department). The exercises for recursion are limited at this moment to list recursion and they are formulated in such a way that only one solution is possible (in the exercise there is given the order and types of the predicate arguments). This fact restricts very much the user's solution.

The XML files are read with consumeTree predicate***: consumeTree(+XMLTags)***.

.

The database files are normal .pl files in which the information is given. The database files used at this moment are: bibliothek.pl and haeuser1.pl. Below there is an example taken from *bibliothek.pl*.

```
%leser(LeserNr,Name,Vorname,Adresse,Geburtsjahr)
leser(2264,hiller,michael,modderkamp_13,1957).
%buch(Signatur,Autor,Titel,Standort)
buch(a3325,mueller,vom_sinn_und_unsinn,regal117).
%ausleihe(LeserNr,Signatur,Rueckgabe,Verlaengert,Mahnung).
ausleihe(2264,a3325,d(22,11,2000),0,nein).
%vorbestellung(LeserNr,Signatur,Datum).
vorbestellung(2264,c5674,d(15,10,2000)).
```

In **data types files** are declared types that appear in the databases or in the recursion problems: *type_decl*, *enum_decl*, *range_decl*. These are defined *in infer_types.pl*. The data types files used at this moment are: *bibliothek_typen.pl*, *haeuser_typen.pl*, *recursive_typen.pl*. Below it is an example taken from *bibliotek_typen.pl*:

```
type_decl(buch(signatur,autor,titel,standort)).
type_decl(ausleihe(leserNr,signatur,datum,anzahl,janein)).
enum_decl(leserNr, [2264, 3167, 4238]).
enum_decl(datum, [d(tag, monat, jahr)]).
range_decl(tag, 1, 31).
range_decl(monat, 1, 12).
```

2.3. The LuSe Evaluation

Evaluation 'in a broader sense, is a process aiming at the investigation of any quality of an object'. According to the classification from (W. von Hahn, L. Tessiore 2000) it can be classified in at least five classes:

- **Evaluation (narrow sense)**: it can be seen as establishing the utility of an object with respect to a specified user
- **Verification**: it means verifying the resemblance of an object to a stated requirement
- **Validation**: it verifies the compatibility of a system by exposing to users
- **Test**: it means discovering whether an object supports a specific feature or not
- **Assessment**: it can show that a specific task runs correctly

Concerning the methods given above, LuSe was evaluated by testing and assessment. The participants at the projected tested with concrete examples the program.

The system was also tested by Chr. Walters and the results can be seen in (Chr. Walters, 2004). The conclusion of this evaluation is that "The LuSe system is capable of aiding the user in solving simple programming tasks in a restricted range of possible mistakes".

2.4. Present Limitations of the Project and Future Developments

After analyzing the way LuSe was evaluated and comparing to the initial goal of the project the following limitations and possible future developments can be extracted

Limitations

- The project has no real student model that would be very helpful for guidance, and choosing exercises. The only observations that were taken into consideration were the ones offered by the coordinators of the project (see Annex 3) that have experience in working with students and in teaching PROLOG and by Yannick Versley, who is tutor for students that are studying Prolog.
- The project has a very simple interaction model and a very simple graphical user interface.
- There is no interface for the creation of new exercises or for modification of the types used in exercises. If somebody wants to add exercises, he/she should be familiar with the structure of the files used in LuSe.
- The actual feedback is quite simple (it indicates the user the place were the error is located and possible solution, but there is no real connection with the theoretical part – 'why' a exercise should be solved in that certain way).
- At this moment there is no natural-language feedback for recursive programs.
- PLHT is not really ready for production use (XPCE web server has single threading, PLHT is not thread safe, so it cannot be used with the threading web server).
- Sessions never time out.
- There are several syntax analysis problems (it sometimes fails because the search space grows too big, several problems when it is missing a parenthesis and it can be inserted in multiple places, etc.
- It can appear confusion of variables and atoms (X -> x). The rule used might be too general.
- The following predicates were not taken into consideration: not/1, !/0.
- The user solution have to exactly match the sample solution, and this restricts the user: the needed variables in the student solution should be in the same place in a predicate and have the same meaning with the ones in the sample solution.

Future developments:

- To integrate syntax analysis with the type checker so that it can choose one possibility from several ones (as in *ausleihe(2264,a1234,d(1,1,2003,0,nein)*)
- It can be implemented a new way of testing: graft a (partial) trace from the sample solution and see if this makes the trace correct
- To extend it by trying to construct invariants (e.g. peano_add)
- New methods for improving the speed of the ids_search (possible using heuristics)
- To find a way to integrate LuSe into the swi emacs editor
- Possible extensions to other languages (Sql, Pl-Sql, Scheme)

3. Conclusions

The paper presents a short introduction in Intelligent Tutoring Systems and then the accent is put in describing the Lernumgebungen für Softwareentwicklung (LuSe) project, developed at Hamburg University, Computer Science department.

After analyzing all the things said above, it can be said that the project is not a full Intelligent Tutoring System, even though it follows the scheme presented in Figure 5, because it does not offer complete explanations to the students. The student sees his/her errors, but no theoretical background is shown to him/her and no explanations are given. There is no response to the question: "*Why these things are wrong?*" and it cannot be said that using LuSe a student can learn Prolog, without the guidance of a human tutor. It also misses a documented student model.

Using this program a student can get help for solving some types of PROLOG problems (database queries and some recursion problems).

References

LUSE HOMAPEGE, http://nats-informatik.uni-hamburg.de/view/LUSE/WebHome

W. VON HAHN, L. TESSIORE (2000). *Functional validation of a Machine Interpretation System: Verbmobil*, p.611-631, "Verbmobil: Foundations of Speech-to-Speech Translation", Springer

CHRISTOPH PEYLO ET AL. (2000). *A Web-Based Intelligent Educational System for PROLOG*, Institute for Semantic Information Processing, University of Osnabrück

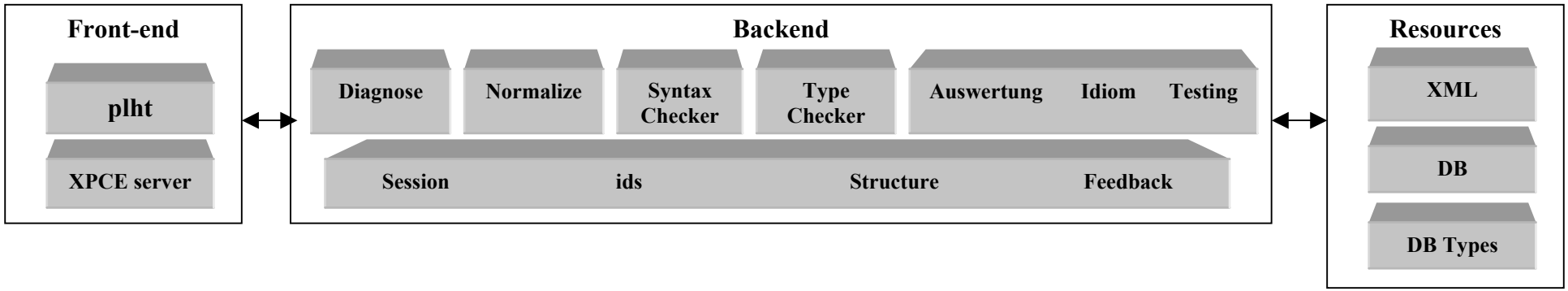
MARK URBAN-LURAIN. *Intelligent Tutoring Systems: An Historic Review in the Context of the Development of Artificial Intelligence and Educational Psychology*, http://www.cse.msu.edu/rgroups/cse101/ITS/its.htm

YANNICK VERSLEY (2003). *PLHT: The PROLOG Hypertext Processor*, manuscript

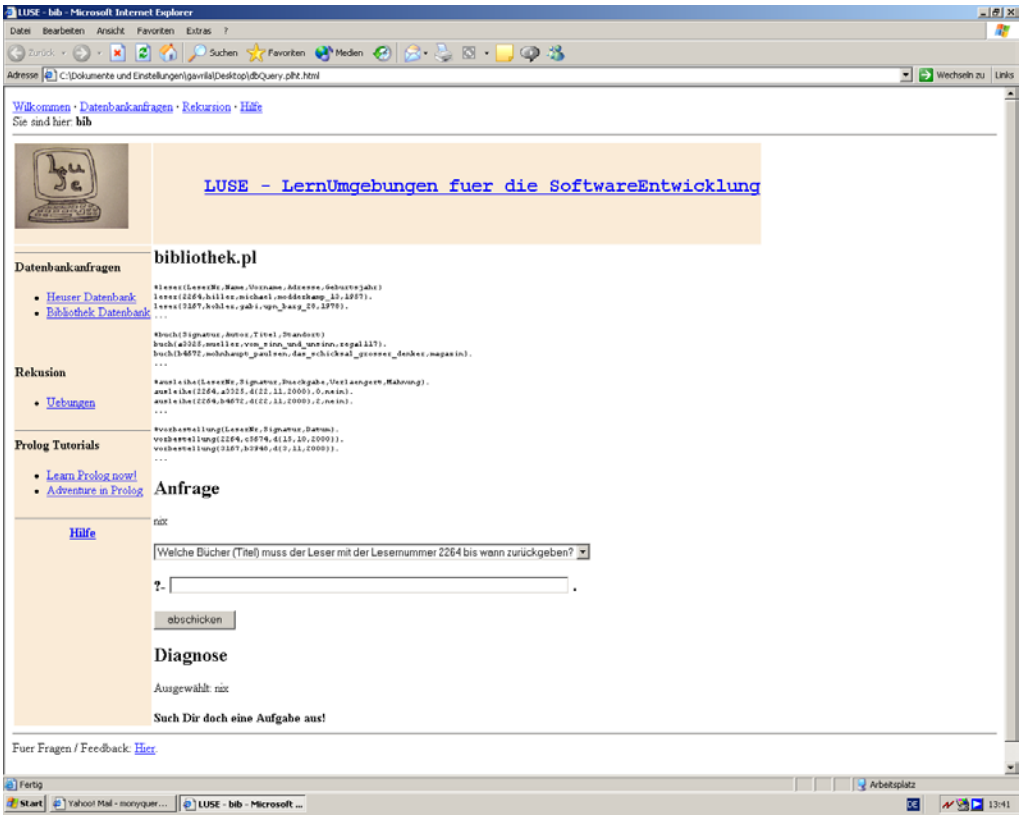
CHR. WALTERS (2004). *Evaluation of the LUSE* system, www.nats-informatik.uni-hamburg.de

ETIENNE WEGNER (1987). *Artificial Intelligence and Tutoring Systems. Computational and Cognitive Approaches to the Communication of Knowledge*, Morgan Kaufmann Publishers, California

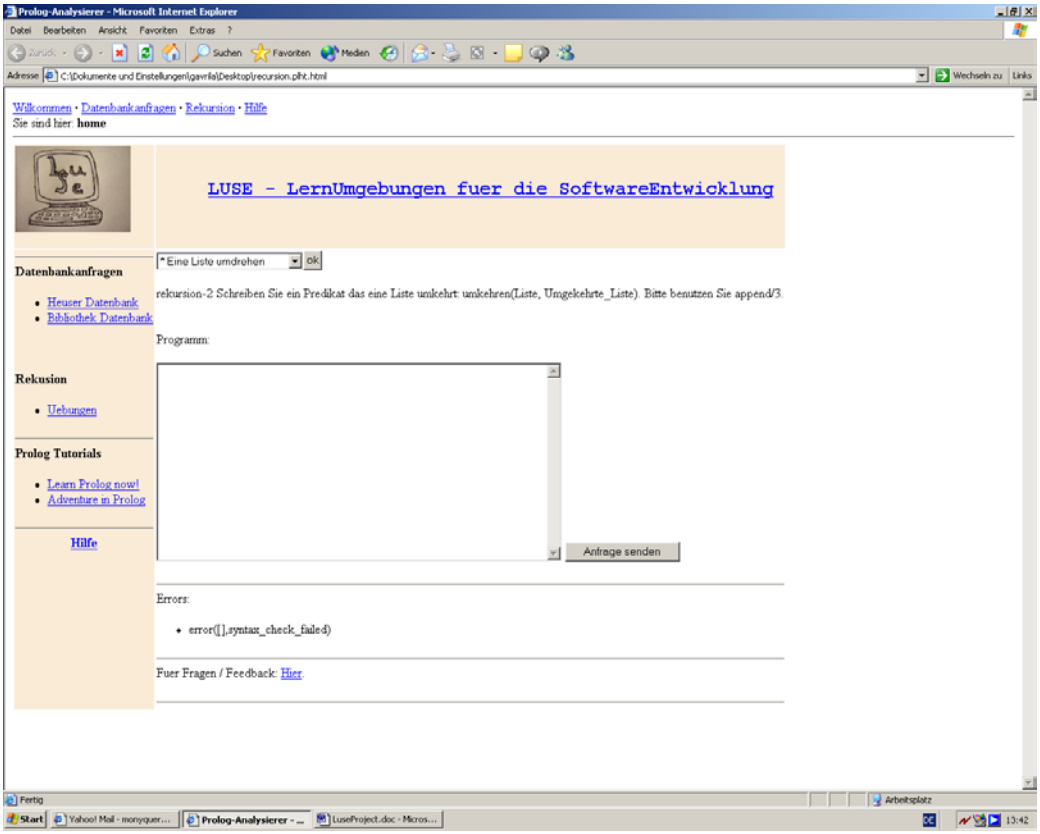
Annex 1. LuSe General Architecture



Annex 2. Some Snapshots of the LuSe Graphical Interface



Snapshot with the database query module



Snapshot with the recursion module

Annex 3. Several Testing Results

Tests on databases query (database used: haueser.pl)

Question: In welche Strassen gibt es Mehrfamilienhauser?

Test 1: obj(_, mfh, _,_,_)
Result:
Diagnose
Ausgewählt: haeuser-1

Deine Anfrage passt nicht genau auf die gestellte Aufgabe. Achte besonders auf die fett gedruckten Teile:
In **welchen Strassen** gibt es Mehrfamilienhäuser ?
[obj(_39, mfh, **_41**, _43, _44)]

Fehler	Erklärung	Hilfe
(_41)	War diese Information in der Aufgabe gegeben?	An dieser Stelle sollte es eine Variable sein.
_41	An dieser Stelle sollte es eine Konstante sein.	Finde diese Angabe in der Aufgabestellung.

Test 2: obj(_,mfh,,_,_)
Diagnose
Ausgewählt: haeuser-1

[obj(_143, mfh, _145, _146)]

Fehler	Erklärung	Hilfe

Test 3: obj(_,mfh,X,_,_
Diagnose
Ausgewählt: haeuser-1

[obj(_463, mfh, X, _489, _490)]

Fehler	Erklärung	Hilfe
)	In deiner Eingabe fehlt dieses Zeichen.	Das verursacht Syntaxfehler.

Test 4: obj(_,mfh,X,_,_)
Diagnose
Ausgewählt: haeuser-1

Du bist perfekt. !!!
Probieren wir die Anfrage doch mal aus:
X= bahnhofsstr

Question: Welche Hauser(Objectnummer) sind vor 1970 erbaut worden

Test: obj(_,mfh,X,_,_)
Diagnose
Ausgewählt: haeuser-2

Deine Anfrage passt nicht genau auf die gestellte Aufgabe. Achte besonders auf die fett gedruckten Teile:
Welche Häuser (**Objektnummer**) sind **vor 1970** erbaut worden?
[obj(_1691, mfh, X, _1693, _1694)]

Fehler	Erklärung	Hilfe
1. Teilziel	Dies reicht noch nicht.	Bitte prüfe, wir müssen mehr Teilziele haben.
(_1691)	War diese Information in der Aufgabe gegeben?	An dieser Stelle sollte es eine Variable sein.
(_1694)	War diese Information in der Aufgabe gegeben?	An dieser Stelle sollte es eine Variable sein.
X	An dieser Stelle sollte es eine Konstante sein.	Finde diese Angabe in der Aufgabestellung.
_1691	An dieser Stelle sollte es eine Konstante sein.	Finde diese Angabe in der Aufgabestellung.
_1694	An dieser Stelle sollte es eine Konstante sein.	Finde diese Angabe in der Aufgabestellung.
Mfh	Diese Information brauchst Du nicht, oder?	Du kannst an dieser Stelle eine anonyme Variable haben.

Tests on recursion:
Errors are presented as follows:
* _1773 - [error([],missing(.)),error([0],missing([])),error([0],missing([]))]
* error([2],atom_as_var([]))
* error([1],atom_as_var(entfernen))

There is no natural language feedback and there are still problems.

Annex 4. Several Data about the LuSe Project

The project took place at Hamburg University during the summer semester 2003 and winter semester 2003 / 2004 and the participants were the following:

COORDINATORS:
Prof. Dr.-Ing. Wolfgang Menzel
Prof. Dr. Leonie Dreschler-Fischer

STUDENTS:
Yannick Versley (both semesters)
Anne Schick (1st semester)
Christoph Walters (1st semester)
Jens Wolfhagen (1st semester)
Manuchehr Sadeghian (1st semester)
Ralf Brandhorst (1st semester)
Le Nguyen Thinh (2nd semester)
Monica Roxana Gavrila (2nd semester)