Automatic Analysis of Programming Assignments

Christoph Beierle, Marija Kulaš, Manfred Widera
Praktische Informatik VIII - Wissensbasierte Systeme
Fachbereich Informatik, FernUniversität Hagen
58084 Hagen, Germany
{beierle | marija.kulas | manfred.widera}@fernuni-hagen.de

Abstract: In a virtual university, advanced support for all aspects of handling assignments is needed. Homework assignments are particularly in need of help because communication between teachers and learners as well as between learners is not as easy as in presence universities. In this paper, we present an overview of the AT(x) approach (analyze-and-test) for automatically analyzing and testing programs. We describe how AT(x) is used for giving feedback to students working on programming exercises. The AT(x) framework is instantiated to AT(P) and AT(S) aiming at programs written in Prolog and Scheme, respectively.

1 Introduction

Both learning a programming language and giving a programming language course can be tedious tasks. A full programming language is usually a complex subject, so concentrating on some basic aspects first is necessary. A nice thing, however, about learning a programming language is that the student may get quick rewards, namely by seeing one's own program actually being executed by a machine and (hopefully) getting the desired effects upon its execution. However, even writing a simple program and running it is often not so simple for beginners: many different aspects e.g. of the runtime system have to be taken into account, compiler outputs are usually not suited very well for beginners, and unfortunately, also user manuals often aim at the more experienced user.
In distance learning and education, additional difficulties arise. Direct interaction between students and tutors is (most of the time) not possible. While communication via phone, e-mail, or newsgroups helps, there is still need for more direct help in problem solving situations like programming. In this context, intelligent tutoring systems have been proposed to support learning situations as they occur in distance education. A related area is tool support for homework assignments. In this paper, we will present an approach to the automatic revision of homework assignments in programming language courses. In particular, we show how exercises in Prolog and Scheme can be automatically analyzed and tested so that automatically generated feedback can be given to the student.
WebAssign [BHSV99] is a general system for support, evaluation, and management of homework assignments that has been developed at the FernUniversität Hagen for distance learning. Experiences with WebAssign involving hundreds of students over the last few years show that especially for programming language courses (up to now mostly Pascal),

the students using the system scored significantly higher in programming exercises than those not using the system. By now, WebAssign is widely used by many different universities and institutions [Web03].
Whereas WebAssign provides a general framework, specific components for individual courses and types of exercises are needed. For such components we provide an abstract frame in the AT(x) system (analyze-and-test for a language $\underline{x}$) which analyzes programs written by a student, and – via WebAssign – sends comments back to the student. Thereby, AT(x) supports the learning process of our students by interaction that otherwise would not be possible. In this work we especially focus on the AT(x) instances AT(P) and AT(S) analyzing Prolog and Scheme programs, respectively.
Besides their integration into WebAssign AT(P) and AT(S) have also been coupled to VILAB, a virtual electronic laboratory for applied computer science [LGH02]. VILAB is a system that guides students through a number of (potentially larger) exercises and experiments.
The rest of the paper is organized as follows: Section 2 gives an overview over WebAssign and the AT(x) system and their interaction. An example session of AT(P) analyzing a Prolog program is given in Sec. 3. The requirements on an analysis component and the analysis components for Prolog and Scheme programs are described in Sec. 4. Section 5 briefly states the current implementation and use of the system. In Sec. 6 related work is discussed, and conclusions and further work are described in Sec. 7.

2 WebAssign and AT(x)

WebAssign is a system that provides support for assignments and assessment of exercises for courses. As stated in [BHSV99], it provides support with web-based interfaces for all activities occurring in the assignment process, e.g. for the activities of the author of a task, a student solving it, and a corrector correcting and grading the submitted solution. In particular, it enables tasks with automatic test facilities and manual assessment, scoring and annotation. WebAssign is integrated in the Virtual University system of the FernUniversität Hagen [LVU03].
From the students' point of view, WebAssign provides access to the tasks to be solved by the students. A student can work out his solution and submit it to WebAssign. Here, two different submission modes are distinguished. In the so-called *pre-test mode*, the submission is only preliminary. In pre-test mode, automatic analyses or tests are carried out to give feedback to the student. The student can then modify and correct his solution, and he can use the pre-test mode again until he is satisfied with his solution. Eventually, he submits his solution in *final assessment mode* after which the assessment of the submitted solution is done, either manually or automatically, or by a combination of both.
While WebAssign has built-in components for automatic handling of easy-to-correct tasks like multiple-choice questions, this is not the case for more complex tasks like programming exercises. Here, specific correction modules are needed. The AT(x) framework aims to analyze solutions to programming exercises and is such a system that can be used as an automatic correction module for WebAssign. Its main purpose is to serve as an automatic test and analysis facility in pre-test mode.

Instances of the AT(x) framework have a task database that contains an entry for each task. When a student submits a solution, AT(x) gets an assignment number identifying the task to be solved and a submitted program written to solve the task via WebAssign's communication components. Further information identifying the submitting student is also available, but its use is not discussed here. Taking the above data as input, AT(x) analyzes the submitted program. Again via WebAssign, the results of its analysis are sent as feedback to the student (cf. Fig. 1).
AT(x) itself is divided into two main components: the main work is done by the analysis component. Especially in functional and logic programming, the used languages are well suited for handling programs as data. The analysis components of AT(P) and AT(S) are therefore implemented in the target language (i.e. the language the programs to be tested are written in).
A further component implemented in Java serves as an interface between this analysis component and WebAssign. It starts the analysis system, reads the error messages from the analysis component, selects some of the messages for presentation according to some pedagogical strategy, and prepares the selected messages for presentation.

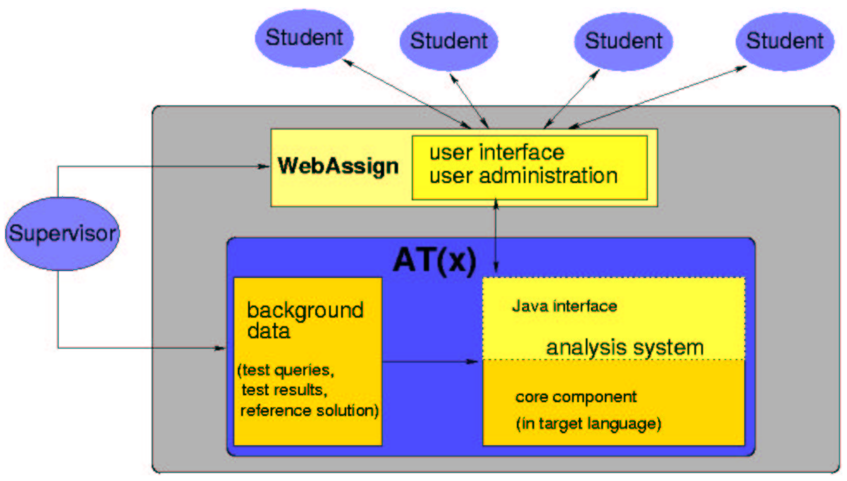

Figure 1: Structure of AT(x)

Due to the learning situation in which we want to apply the analysis of Prolog and Scheme programs, we did not make any restrictions with respect to the language constructs allowed in the students' solutions. AT(P) and AT(S) are rather able to handle the full standards of the Prolog and Scheme programming languages, respectively.

3 An Example Session

Before we go into the description of the individual components of the AT(x) system, we want to show an example execution for a Prolog homework task. The task is described as follows: *Let N and M be natural numbers with $N \leq M$. Define a predicate between/3 such that a query* `between(X, N, M)` *is true if* `X` *is a number between* `N` *and* `M`*.*
Let us assume that the following program is submitted:

```
between(X, N, M) :- var(X), integer(N),
                     integer(M), N =< M,
                     gen_list(N, M, X).

gen_list(N, N, [N|[]]) :- !.
gen_list(N, M, [N|R]) :- L is N + 1, gen_list(L, M, R).
```

Then the system's output is the following:

```
The following query failed, though it should succeed:
between(10,10,20)
-------------------------------------------------
Wrong solutions where generated for the following query:
between(A,100,102)

The wrong solutions for this query are listed below:
between([100,101,102],100,102)
-------------------------------------------------
Solutions were overlooked for the following query:
between(A,100,102)

The overlooked solutions for this query are listed below:
between(100,100,102)
between(101,100,102)
between(102,100,102)
```

One interesting aspect of the AT(x) framework is the following: the system is designed to perform a large number of tests. In the generated report, however, it filters some of the detected errors for presentation. Several different filters generating reports of different precision and length are available. In the example above, a single representative for each kind of detected error was selected.

4 The Core Analysis Components

The heart of the AT(x) system is given by the individual analysis components for the different programming languages. In this section we give an overview over the general requirements on these analysis components and describe components for analyzing programs in the languages Prolog and Scheme instantiating AT(x) to AT(P) and AT(S) in more detail.

4.1 Requirements on the Analysis Components

The intended use in testing homework assignments rather than arbitrary programs implies some important properties of the analysis components discussed here: it can rely on the availability of a detailed specification of the homework tasks, it must be robust against non terminating input programs and runtime errors, and it must generate reliable output understandable for beginners.
The description for each homework task consists of the following parts:

- A textual description of the task. (This is essentially used in preparation of the homework assignment, but not in the testing task itself.)
- A set of test cases for the task.
- Specifications of program properties and of the generated solutions. (This applies especially for declarative languages like Prolog.)
- A reference solution. (This is a program which is assumed to be a correct solution to the homework task and which can be used to judge the correctness of the students' solutions.)

This part of input is called the *static input* to the analysis component, because it usually remains unchanged between the individual test sessions. A call to the analysis system contains an additional *dynamic input* which consists of a unique identifier for the homework task (used to access the appropriate set of static input) and a program to be tested.
Now we want to discuss the requirements on the behaviour of the analysis system in more detail. Concretizing the requirement of reliable output we want our analysis component to return an error only if such an error really exists. Where this is not possible (especially when non termination is assumed), the restricted confidence should clearly be communicated to the student, e.g. by marking the returned message as a *warning* instead of an *error*. For warnings the system should describe an additional task to be performed by the student in order to discriminate errors from false messages.
Runtime errors of every kind must be caught without affecting the whole system. For instance, if executing the student's program causes a runtime error, this should not corrupt the behaviour of the other components. Towards this end, our AT(P) and AT(S) implementations exploit the hooks of user-defined error handlers provided by SICStus Prolog and MzScheme, respectively. An occurring runtime error is reported to the student, and no further testing is done, because the system's state is no longer reliable.
For ensuring termination of the testing process, infinite loops in the tested program must also be detected and interrupted. As the question whether an arbitrary program terminates is undecidable in general, we chose an approximation that is easy to implement and guarantees every infinite loop to be detected: a threshold for the maximal number of function calls (either counted independently for each function or accumulated over all functions in the program) is introduced and the program execution is aborted whenever this threshold is exceeded.[1] As homework assignments are usually small tasks, it is possible to estimate

[1] In the context of the Prolog and Scheme programs considered here, every iteration is implemented by recursion and therefore supervising the number of function calls suffices. In the presence of further looping constructs, a refined termination control is necessary.

the maximal number of needed function calls and to choose the threshold sufficiently. The report to the student must, however, clearly state the restricted confidence on the detected non termination.
Counting the number of function calls is only possible when executing the program to test in a supervised manner. Different approaches of supervision are described in the following subsections for the individual analysis components.

4.2 Analysis of Prolog Programs

Due to the declarative character of Prolog, a variety of tests can be performed on Prolog programs. Certain main properties of a program can be tested by annotations. Our system AT(P) is based on the TSP-approach of H. Neumann [Neu98] where several kinds of Prolog annotations are proposed, together with an algorithm for their validation with respect to a given student's program and a reference program. One general kind of annotation is a positive/negative annotation. Such an annotation consists of a test query, a flag whether this query should succeed or fail and a description of the property that is violated if the query does not behave as expected. This description is reported to the student together with the query and the intended result.
Another kind of annotation, unique to logic programming, is a mode annotation: since each argument of a predicate may be input, output, or both, a mode annotation specifies which case is intended. From a number of ground atoms and a specification of all input/output modes a program should be usable with, further test queries are generated. This automatic query generation ensures all intended modes to be checked.
A second part of the test procedure consists of the application of the student program to test queries and checking the resulting substitutions with the reference program. The following steps of comparison are performed:

- Solutions of a student's program are reported as wrong solutions if they are falsified by the reference program (i.e. if the query given by the solution fails in the reference program).
- A solution of the reference program is reported as overlooked, if it is not subsumed by some solution of the student's program. (It is not sufficient for the student's program to *accept* every solution generated by the reference program. The student's program must rather be able to *generate* all these solutions.)
- For some of the test queries the number of expected solutions can be given (*completeness annotation*), and the number of solutions generated by the student's program is compared with this specification.

In case of an infinite number of solutions just a prefix is generated and tested for correctness. The system is still applicable to those tasks with infinite solution set if a natural order on the solutions exists and therefore the generated solutions of the student's program and the reference program match. An example of a problem with natural order is the generation of all prime numbers. In contrast, there is no single natural order for generating all words over the alphabet $\Sigma = \{\texttt{@}, \texttt{\#}, \texttt{\$}\}$.

The central part of the Prolog analysis component is a meta-interpreter that evaluates test queries with respect to the student's program and with respect to the reference program, respectively, and collects the individual results in a list. Both programs are held in memory in parallel using different modules. Queries matching clauses from the tested module are decomposed and interpreted; queries matching imported predicates are passed to the runtime system using *call/1*. The interpreter has full control over the history of a call and can especially count the number of open calls in the current situation. Upon exceeding of the threshold for the number of open calls, the current computation is aborted and evidence for an infinite loop is reported.

4.3 Analysis of Scheme Programs

In Scheme programs there is less declarative freedom than in Prolog programs. Especially, there are no different modes a function can be used with and no function call can return several solutions. Therefore, the Scheme analysis component focuses on evaluating function calls in a student's program and in the reference program and on comparing the results.
A problem inherent to functional programs is the potentially complex structure of the result. Not only can several results to a question be composed into a structure, but it is furthermore possible to generate functions (and thereby e.g. infinite output structures) as results. For comparing such structures, a simple equality check is not appropriate. We rather provide an interface for the teacher to implement an equality function that is adapted to the expected output structures and that returns true if the properties of the two compared structures are similar enough for assuming correctness in the context of pre-testing. Using such an approximation of the full equality is safe since in the usual final assessment situation the submission is corrected and graded by a human tutor.
E.g. the test for comparing functions from numbers to numbers can return true after comparing the results of both functions for n (for some appropriate number n) well-chosen test inputs for equality. When considering the alternative case of several results being collected in a list, it is important to compare just the *set* of list elements without considering the order given by the list. The standard test applicable to many kinds of result values is the predefined Scheme function *equal?* implementing a deep equality check on structures.
Termination analysis of Scheme programs is done by performing a program transformation to the student program. We have implemented a function *count*[2] that counts the number of calls to a function for different functions independently, and that aborts the evaluation via an exception if the number of calls exceeds a threshold for one of the functions. The counting of function calls is done by transforming every lambda expression as follows: Each lambda expression of the form `(lambda (args) body)` is transformed into `(lambda (args) (let ((tester::tmp tc)) body))` where `tc` is an expression sending a message to the function *count* containing a unique identifier of the lambda expression and `tester::tmp` is just a dummy variable whose value is not used. Functions given in MIT style are transformed in an analogous manner. Structures

[2]The real name of the function in our implementation is chosen in a way that makes name conflicts with the tested program very unlikely. Furthermore, it can be changed easily.

are searched element by element. Functions contained in a structure are replaced by a transformed version. After this transformation the evaluation can be passed to the runtime system using *eval*.

5 Implementation

Both AT(P) and AT(S) are fully implemented and operational. The analysis components run under the Solaris 7 operating system and, via their respective Java interface components, serve as clients for WebAssign. During the summer semester 2003, AT(P) is currently being used in the framework of a course on deduction and inference systems at the FernUniversität Hagen, and both AT(P) and AT(S) will be used for a course on logic and functional programming starting October 2003 (cf. [FBI03]).
Due to the modular design of our system the implementation of new analysis components can concentrate on the analysis tasks. The implementation of the analysis component of AT(S) took approximately three person months. For the adaption of the starting procedure and the specific error codes inside the interface component additional two weeks were necessary.

6 Related work

In the area of testing and analysis of Prolog programs there have been many proposals, ranging from theorem proving to interactive debugging. Here we shall only refer to proposals with a strong declarative bias, i. e. using logical assertions (or annotations) for representing and/or validating program properties. The proposals differ along several axes: static or run-time analysis, restricting the target language or not, expressiveness of the annotation language.
Some authors restrict the target language by throwing out “impure” predicates. LPTP [Stä98] is an interactive theorem prover for a pure (no `cut`, `var` or `assert`) subset of Prolog. LPTP’s language is a first-order logic, enriched with connectives for success, failure and universal termination. GUPU [Neu97] is a teaching environment for a pure subset of Prolog. Before a student is allowed to feed in some clauses for a predicate, a partial specification (as a set of annotations) must be supplied. GUPU’s annotation language can express examples, counter-examples and termination statements.
Yet other authors take the challenge of the “full” or standard Prolog language. The first approach to validation of full Prolog seems to be the ADVICE package [O’K84]. It was followed by a theoretical work [DM88] on static validation. Some current practical approaches, performing run-time validation, include the language of annotations of the CIAO-Prolog [HPB99] and the NOPE system of annotations [Kul00]. The TSP system [Neu98], at the heart of our AT(P), performs run-time validation of full Prolog, and its annotation language can express examples, counter-examples, modi and numerical constraints on the number of computed answers. These constraints, together with the TSP algorithm for checking them without having to compute all answers, appear to be entirely novel.

In the context of teaching Scheme, the most popular system is DrScheme [PLT03]. The system contains several tools for easily writing and debugging Scheme programs. For testing a program, test suites can be generated. Our AT(S) system differs from that approach primarily in providing a test suite that is hidden from the student and that is generated without a certain student's program in mind, but following the approach called *specification based testing* in testing literature (cf. e.g. [ZHM97]).
An automatic tool for testing programming assignments in WebAssign already exists for the programming language Pascal [Web03]. In contrast to our approach here, several different programs have to be called in sequence, namely a compiler for Pascal programs, and the result of the compilation process. The same holds for other compiled programming languages like e.g. C and Java. To keep a uniform interface, it is advisable to write an analysis component that compiles a program, calls it for several inputs, and analyzes the results. This component can then be coupled to our interface component instead of rewriting the interface for every compiled language.

7 Conclusions and Further Work

We addressed the situation of students in programming lessons during distance learning studies. The problem here is the usually missing or insufficient direct communication between learners and teachers and between learners. This makes it more difficult to get around problems during performing self-tests and homework assignments.
In this paper, we have presented a brief overview on the AT(x) approach which is capable of automatically analyzing programs with respect to given tests and a reference solution. In the framework of small homework assignments with precisely describable tasks, the AT(x) instances are able to find many of the errors usually made by students and to communicate them in a manner understandable for beginners in programming (in contrast to the error messages of most compilers.)
The AT(x) framework is designed to be used in combination with WebAssign [BHSV99], which provides a general framework for all activities occurring in the assignment process, or with VILAB [LGH02], a virtual computer science laboratory. This causes AT(x) to be constructed from two main components, an analysis component (often written in the target language) and a uniform interface component written in Java.
By implementing the necessary analysis components, two instances AT(P) and AT(S) have been generated, which perform the analysis task for Prolog and Scheme programs. The general interface to WebAssign makes it easy to implement further instances of AT(x), for which the required main properties are also given in this paper. During the next semesters, both AT(P) and AT(S) will be applied in courses at the FernUniversität Hagen and their benefit for programming courses in distance learning will be evaluated.

Acknowledgements: The basis for analysis component of AT(P) is taken from the TSP system implemented by Holger Neumann [Neu98]. TSP offers a variety of different tests and turned out to be extremely stable. Many of the ideas taken from there also were very helpful in implementing the analysis component of AT(S).

References

[AIAI88] Edinburgh University A. I. Applications Institute. Edinburgh Prolog Tools. ftp://-sunsite.doc.ic.ac.uk/packages/prolog-pd-software/tools.tar.Z, 1988.

[BHSV99] J. Brunsmann, A. Homrighausen, H.-W. Six, and J. Voss. Assignments in a Virtual University – The WebAssign-System. In *Proc. 19th World Conference on Open Learning and Distance Education*, Vienna, Austria, June 1999.

[DM88] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.

[FBI03] Fachbereich Informatik, FernUniversität Hagen, `http://www.informatik.fernuni-hagen.de/studium+lehre/uebersicht.html`. 2003.

[HPB99] M. Hermenegildo, G. Puebla, and F. Bueno. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In K. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*. Springer-Verlag, 1999.

[Kul00] M. Kulaš. Annotations for Prolog – A Concept and Runtime Handling. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation. Selected Papers of the 9th Int. Workshop (LOPSTR'99), Venezia*, volume 1817 of *LNCS*, pages 234–254. Springer-Verlag, 2000.

[LGH02] Rainer Ltticke, Carsten Gnrlich, and Hermann Helbig. VILAB - A Virtual Electronic Laboratory for Applied Computer Science. In *Proceedings of the Conference Networked Learning in a Global Environment*. ICSC Academic Press, Canada/The Netherlands, 2002.

[LVU03] Homepage LVU, FernUniversität Hagen, `http://www.fernuni-hagen.de/LVU/`. 2003.

[Neu97] U. Neumerkel. A programming course for declarative programming with Prolog. http://www.complang.tuwien.ac.at/ulrich/gupu/material/1997-gupu.ps.gz, 1997.

[Neu98] H. Neumann. Automatisierung des Testens von Zusicherungen für Prolog-Programme. Diplomarbeit, FernUniversität Hagen, 1998.

[O'K84] Richard A. O'Keefe. *advice.pl*. In DEC-10 Prolog Library [AIAI88]. Interlisp-like advice package.

[PLT03] *PLT DrScheme: Programming Environment Manual*, May 2003. version204.

[Stä98] Robert F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *J. of Logic Programming*, 36(3):241–269, 1998. Source distribution http://www.inf.-ethz.ch/˜staerk/lptp.html. Release 1.05.

[Web03] Homepage WebAssign. `http://www-pi3.fernuni-hagen.de/WebAssign/`. 2003.

[ZHM97] H. Zhu, P. Hall, and J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.