Chapter 1

Testing Scheme Programming Assignments Automatically

Manfred Widera[1]

1.1 INTRODUCTION

Both learning a programming language and giving a course in computer programming 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 to program 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 Scheme can be automatically analyzed and tested so that automatically generated feedback can be given to the student.

WebAssign [BHSV99, HS97] is a general system for support, evaluation, and management of homework assignments that has been developed at the FernUni-

[1]Praktische Informatik VIII - Wissensbasierte Systeme, Fachbereich Informatik, FernUniversität Hagen, 58084 Hagen, Germany; Email: `manfred.widera@fernuni-hagen.de`

versität 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 *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) instance AT(S) analyzing Scheme programs.

Besides its integration into WebAssign AT(S) has 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 1.2 gives an overview over WebAssign and the AT(x) system and their interaction. An example session of AT(S) analyzing a Scheme program is given in Sec. 1.3. Section 1.4 describes the general structure of AT(x) which consists of several components. The requirements on an analysis component and the analysis component for Scheme programs are described in Sec. 1.5. Section 1.6 briefly states the current implementation and use of the system. In Sec. 1.7 related work is discussed, and conclusions and further work are described in Sec. 1.8.

1.2 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(S) system (as well as other AT(x) instances) 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. 1.1).

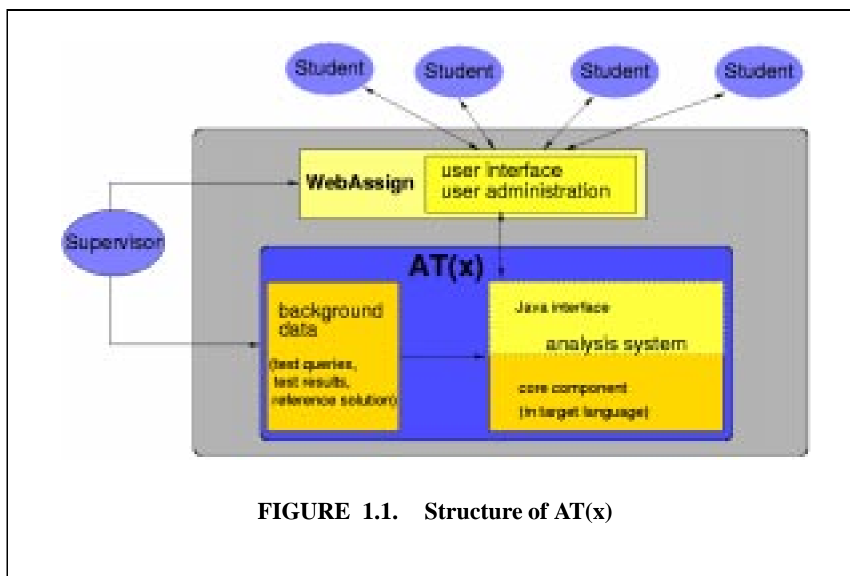

FIGURE 1.1. Structure of AT(x)

Due to the learning situation in which we want to apply the analysis of Scheme programs, we did not make any restrictions with respect to the language constructs allowed in the students' solutions. AT(S) is rather able to handle the full standards of the Scheme programming language as it is implemented by MzScheme [Fla03].

1.3 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 Scheme homework task. The task is described as follows:

> Define a function `fac` that expects an integer n as input and returns the factorial of n if $n \geq 0$, and the atom `negative` otherwise.

Let us assume that the following program is submitted:

```
(define (fac i)
  (if (= i 0) 1
      (+ i (fac (- i 1)))))
```

This program contains two errors:

1. The test for negative numbers is missing.
2. The first operator in the last line must be $*$ instead of $+$.

Then the system's output is the following:

```
The following errors where detected in your program:
-------------------------------------------------
The following test was aborted to enforce
termination:
(fac -1)
The function called when the abortion took place
was ``fac''.
A threshold of 10000 recursive calls was exceeded.
Please check whether your program contains an
infinite loop!
-------------------------------------------------
The following test generated a wrong result:
(fac 5)
The result generated was 16 instead of the
expected result 120.
-------------------------------------------------
The following test generated a wrong result:
(fac 6)
The result generated was 22 instead of the
expected result 720.
-------------------------------------------------
The following test generated a wrong result:
(fac 1)
The result generated was 2 instead of the
expected result 1.
-------------------------------------------------
The following test generated a wrong result:
(fac 10)
The result generated was 56 instead of the
expected result 3628800.
-------------------------------------------------
```

```
The following test generated a wrong result:
(fac 8)
The result generated was 37 instead of the
expected result 40320.
-------------------------------------------------
```

One interesting aspect of the AT(S) system is the following: the system is designed to perform a large number of tests. In the generated report, however, it can filter some of the detected errors for presentation. Several different filters generating reports of different precision and length are available. The example above, shows all detected errors at once.

1.4 STRUCTURE OF THE AT(X) FRAMEWORK

The AT(x) framework combines different tools. Interfaces to different user groups (especially students and supervisors) have to be provided via WebAssign. The design decisions caused by this situation are described in this section.

1.4.1 Components of the AT(x) System

AT(x) is divided into two main components: the main work is done by the analysis component. Especially in functional (and also in logic) programming, the used language is well suited for handling programs as data. The analysis component of AT(S) is therefore implemented in the target language Scheme.

A further component implemented in Java serves as an interface between this analysis component and WebAssign. The reason for using such an interface component is its reusability and its easy implementation in Java. The WebAssign interface is based on Corba communication. A framework for WebAssign clients implementing an analysis component is given by an abstract Java class. Instead of implementing an appropriate Corba client for each of the AT(x) instances in the individual target languages independently, the presented approach contains a reusable interface component implemented in Java (that makes use of the existing abstract class) and a very simple interface to the analysis component (cf. Fig. 1.1).

1.4.2 The Analysis Component

The individual analysis component is the main parts of an AT(x) instance. It performs a number of tests on the students' programs and generate appropriate error messages. The performed tests and the detectable error types of AT(S) are discussed in detail in Sec. 1.5. Here, we concentrate on the (quite simple) interface of this component.

The analysis component of each AT(x) instance expects to read an exercise identifier (used to access the corresponding information on the task to solve) and a student's program from the standard input stream. It returns its messages, each as a line of text, at the component's standard output stream. These lines of text

contain an error number and some data fields containing additional error descriptions separated by a unique identifier. The number and types of the additional data fields is fixed for each error number.

An example for such an error line is the following:

```
###4###(fac 5)###16###120###
```

Such a line consists of a fixed number of entries (four in this case) which are separated by `###`. This delimiter also starts and ends the line. The first entry contains the error number (in this case 4 for a wrong result). The remaining entries depend on the error number. In this case, the second contains the test `(fac 5)` causing the error, the third contains the wrong result 16, and the fourth the expected result 120.

1.4.3 Function and Implementation of the Interface Component

WebAssign provides a communication interface based on Corba to the analysis components. In contrast, the analysis components used in AT(x) use a simple interface with textual communication via the stdin and stdout streams of the analysis process. We therefore use an interface program connecting an analysis component of AT(x) and WebAssign which performs the following tasks:

- Starting the analysis system and providing an exercise identifier and the student's program.
- Reading the error messages from the analysis component.
- Selecting some of the messages for presentation.
- Preparing the selected messages for presentation.

The interface component starts the analysis system (via the Java class *Runtime*) and writes the needed information into its standard input stream (which is available by the Java process via standard classes). Afterwards, it reads the message lines from the standard output stream of the analysis system, parses the individual messages and stores them into an internal representation.

During the implementation of the system it turned out that some language interpreters (especially SICStus Prolog used for the analysis component of AT(P) [BKW03]) generate a number of messages at the stderr stream, e.g. when loading modules. These messages can block the Prolog process when the stream buffer is not cleared. Our Java interface component is therefore able to consume the data from the stderr stream of the controlled process without actually using them.

For presenting errors to the student, each error number is connected to a text template that gives a description of this kind of error. An error message is generated by instantiating the template of an error with the data fields provided by the analysis component together with the error number. The resulting text parts for the individual errors are concatenated and transferred to WebAssign as one piece of HTML text. An example of a message generated by the analysis component

can be found in Subsec. 1.4.2. The example session in Sec. 1.3 shows how this message is presented to the student.

For using this system in education it turns out that presenting all detected errors at once is not the best action in every case.

Example 1.1. Consider the example session described in Sec. 1.3. Having error messages for all detected errors available, a student can write the following program that just consists of a case distinction and easily outfoxes the system.

```
(define (fac n)
   (cond
     ((= n -1) 'negative)
     ((= n 5) 120)
     ((= n 6) 720)
     ((= n 1) 1)
     ((= n 10) 3628800)
     ((= n 8) 40320)))
```

To avoid this kind of programs that are fine tuned to the set of tests performed by the analysis component, the interface component has the capability of selecting certain messages for output according to one of the following strategies:

- Only one error is presented. This is especially useful in beginners courses, since a beginner in programming should not get confused and demotivated by a large number of error messages. He can rather concentrate on one message and may receive further messages when restarting the analysis with the corrected program.
- For every type of error occurring in the list of errors only one example is selected for output. This strategy provides more information at once to experienced users. A better overview over the pathological program behaviour is given, because all different error types are described, each with one representative. This may result in fewer iterations of the cycle consisting of program correction and analysis. The strategy, however, still hides the full set of all test cases from the student and therefore prevents fine tuning a program according to the performed tests.
- All detected errors are presented at once. This provides the complete overview over the program errors and is especially useful when the program correction is done offline. In order to prevent fine tuning of a program according to the performed tests, students should be aware that in final assessment mode additional tests not present in the pre-test mode will be applied.

1.5 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 a component for analyzing programs in Scheme instantiating AT(x) to AT(S) in more detail.

1.5.1 Requirements on the Analysis Components

The intended use in testing homework assignments rather than arbitrary programs implies some important properties of the analysis component 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 not directly needed for analyzing students' programs. For the teacher it is, however, convenient in preparing the tasks to have the task description available together with the other data items described here.)
- A set of test cases for the task.
- 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.)
- Specifications of program properties and of the generated solutions. (This is not a necessary part of the input. In our implementation we use abstract specifications for Prolog programs (cf. [BKW03]), but not yet for Scheme.)

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(S) implementation exploits the hooks of user-defined error handlers provided by MzScheme [Fla03]. 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 aborted. 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 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. The different approaches for supervising recursion contain the implementation of an own interpreter for the target language and the instrumentation of each function definition during a preprocessing step such that it calls to a counter function at the beginning of every execution of the function. The second approach was chosen for AT(S) and is described in more detail in the following subsection.

1.5.2 Analysis of Scheme Programs

The aim of the AT(S) analysis component is the evaluation of tests in a given student's program and to check the correctness of the results. A result is considered correct, if comparing it with the result of the reference solution on the same test succeeds.

A problem inherent to functional programs is the potentially complex structure of the results. 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.

Example 1.2. Consider the following homework task:

> Implement a function `words` that expects a positive integer n and returns a list of all words over the alphabet $\Sigma = \{0,1\}$ with length l fulfilling $1 \leq l \leq n$.

For the test expression `(words 3)` there are (among others) the valid solutions

```
(0 1 00 01 10 11 000 001 010 011 100 101 110 111)
(1 0 11 10 01 00 111 110 101 100 011 010 001 000)
(111 110 101 100 011 010 001 000 11 10 01 00 1 0)
```

which just differ in the order of the same words. Since no order has been specified in the task description, all these results must be considered correct.

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

[1] In the context of the 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.

safe since in the usual final assessment situation the submission is corrected and graded by a human tutor.

Example 1.3. For the task in Ex. 1.2 the equality check could be

```
(define (check l1 l2)
  (equal? (sort l1) (sort l2)))
```

with an appropriate sort function `sort`.

The test for comparing e.g. 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.

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.[3] 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 are searched element by element. Functions contained in a structure and especially local function definitions occurring in a functions body are replaced by a transformed version.

After performing the transformation on the student's program, the individual tests are evaluated in the transformed program and in the reference solution. The results from both programs are compared, and messages are generated when errors are detected. Runtime errors generated by the student's program are caught, and an explaining error message is sent to the interface component of AT(S).

In detail the analysis component of AT(S) is able to distinguish the following error messages, which can stem from failed equality checks, the termination control and the runtime system:

---

[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.

[3] Precisely, the count function distinguishes lambda closures that are generated from different source code. Different lambda closures, that are generated by evaluating the same piece of code in different environments, are identified.

1. An internal error of the runtime system occurred during evaluating the test. (This error should not occur.)
2. An unidentified error occurred during evaluating the test. (This error message covers every kind of error that is not important and frequent enough to get an own error number.)
3. The termination control aborted the execution of a test in order to guarantee termination of the analysis component.
4. The student's program generated a wrong result for a test.
5. Syntax error in the student's program.
6. Read error generated by the student's program
7. An uninstantiated variable was accessed.
8. A function was called with a wrong argument type.
9. A function was called with the wrong number of arguments.
10. I/O error in accessing a file.

For each of these errors the interface component of AT(S) contains a text template that is instantiated with the details of the error and then presented to the student.

1.6 IMPLEMENTATION

AT(S) is fully implemented and operational. The analysis component runs under the Solaris 7 operating system and, via its Java interface component, serves as clients for WebAssign. 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.

1.7 RELATED WORK

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]). Other testing approaches to functional programming (e.g. [CH00]) do not focus on testing programming assignments and are therefore not

designed to use a reference solution for judging the correctness of computation results. The approach of abstractly describing properties of the intended results can, however, be incorporated into our approach if necessary.

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. For instantiating AT(x) to another functional programming language, it is, however, advisable to use the languages read-evaluate-print-loop, and to implement the analysis component completely in the target language.

1.8 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, developed at the FernUniversität Hagen, which provides a general framework for all activities occurring in the assignment process. 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 component, the instance AT(S) has been generated, which performs the analysis task for Scheme programs. This analysis component is robust against programs causing infinite loops and runtime errors, and is able to generate appropriate messages in these cases. 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, AT(S) will be applied in courses at the FernUniversität Hagen and its benefit for Scheme programming courses in distance learning will be evaluated.

REFERENCES

[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.

[BKW03] Christoph Beierle, Marija Kulaš, and Manfred Widera. Automatic analysis of programming assignments. In *Proceedings of the 1. Fachtagung "e-Learning" der Gesellschaft für Informatik (DeLFI 2003)*, 2003. (to appear).

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 268–279, N.Y., September 18–21 2000. ACM Press.

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

[Fla03] Matthew Flatt. *PLT MzScheme: Language Manual*, August 2003.

[HS97] A. Homrighausen and H.-W. Six. Online assignments with automatic testing and correction facilities (abstract). In *Proc. Online EDUCA*, Berlin, Germany, October 1997.

[LGH02] Rainer Lütticke, Carsten Gnörlich, 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.

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

[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.