Chapter 7

Testing Scheme Programming Assignments Automatically

Manfred Widera[1]

Abstract In distance learning the lack of direct communication between teachers and learners makes it difficult to provide direct assistance to students while they are solving their homework tasks. We address this problem particularly for programming tasks and describe a system for automatically analyzing students' homework tasks, and providing understandable feedback. Our approach is adapted to the special situation in distance learning and is integrated into the virtual university approach at the University of Hagen. It consists of a general framework and instances for individual programming languages. For these instances, one example is presented for the programming language Scheme.

7.1 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. One nice thing, however, about learning to program is that the student may get quick rewards, namely by seeing his 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 very well suited for beginners, and user manuals unfortunately often aim at the more experienced user.

In distance learning and education, direct interaction between students and tutors is particularly difficult. While communication via phone, e-mail, or news-

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

groups 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 describe a framework for analyzing programming homework tasks called AT(x) (analyze-and-test for a language *x*) and show how exercises in Scheme [7] can be analyzed and tested automatically by an instance AT(S) of it. The goal of AT(x) is to provide detailed generated feedback for the student. For the moment, automatic assessment of assignments is not provided by the system and is also only of minor importance for further extensions, compared to refined assistance for the students: while automatic assessment is a goal of interest in every area of teaching, the automatic assistance to the student is a special aim of distance learning and this system.

The destination platform for our AT(x) system is WebAssign [2, 6] which was developed at the University of Hagen for distance learning and is accessible for every teacher. WebAssign is a general system for support, evaluation, and management of homework assignments. Experiences with WebAssign, involving thousands 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. WebAssign is now widely used by many different universities and institutions [12].

Whereas WebAssign provides a general framework, customized components for different types of exercises are needed. For such components AT(x) provides an abstract frame which analyzes programs written by a student and – via WebAssign – sends back comments. In this way, AT(x) supports the learning process of our students by interaction that otherwise would not be possible. Apart from the general design of AT(x) and the benefits of such a generalized approach, in this paper we especially focus on the AT(x) instance AT(S) analyzing Scheme programs as an example for the analysis process on functional programming languages.

While WebAssign is the most important platform for the use of AT(x) in the near future, the system has also been coupled to VILAB, a virtual electronic laboratory for applied computer science [9]. VILAB is a system that guides students through a number of (potentially larger) exercises and experiments. The interface between AT(x) and VILAB is also generic over the different programming languages covered by the AT(x)-instances.

The rest of the paper is organized as follows: Sec. 7.2 gives an overview of WebAssign, the AT(x) system, and their interaction. A sample session of AT(S) analyzing a Scheme program is given in Sec. 7.3. Sec. 7.4 describes the general structure of AT(x) which consists of several components. The general requirements on an analysis component and their realization in the analysis component for Scheme programs are described in Sec. 7.5. Sec. 7.6 briefly states the current implementation and use of the system. In Sec. 7.7 related work is discussed, and conclusions and some further work are described in Sec. 7.8.

7.2 WebAssign AND AT(x)

The AT(x) system described in this paper is specialized to the situation at the FernUniversität in Hagen. For presenting and solving homework assignments online, the WebAssign system is available for use by customized assignment systems. Since WebAssign as the target platform had some influence on several design decisions for AT(x), we offer a brief overview of WebAssign and the way AT(x) is seen from WebAssign's point of view.

WebAssign is a web-based system that provides support for assignments and assessment of homework tasks. As stated in [2], 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 [10].

From the students' point of view, WebAssign provides access to the tasks to be solved by them. 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.

Several standard tasks are achieved by WebAssign and need not be addressed by customized analysis components using it.

- WebAssign provides *only* authenticated access for students and teachers. This can be based on a common authentication database for the whole university or on a database locally administered by WebAssign.

- Persistence of results between sessions and after final submission. In pre-test mode WebAssign stores the last submission for every task and every student in a database and provides it as a starting point to the student in further sessions. For final assessment, the teacher can access the solutions in this database (together with automatically generated comments if available). Comments and assessments from a human corrector or an automatic tool are also stored in this database and are made available to the student via WebAssign.

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) system (and especially the AT(S) instance described here in more detail) aims at analyzing solutions to programming exercises and is 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. (As a side-

effect we can make the output of the system available for the corrector in order to simplify the detection of errors.)

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. 7.1). The division of WebAssign and AT(x) is not only a logical one. While WebAssign is meant to reside on a global university server, the AT(x) components run on local servers that provide full control to individual teachers.

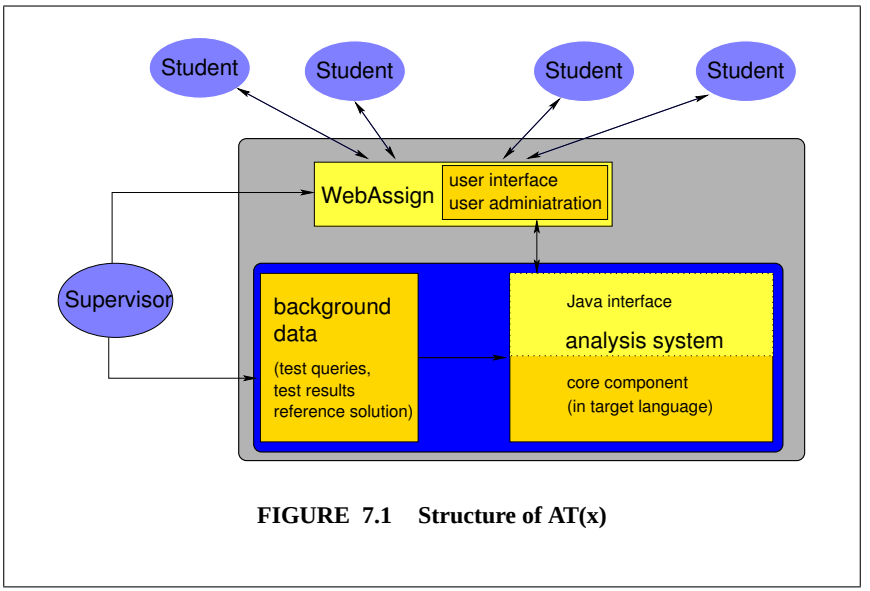

FIGURE 7.1 Structure of AT(x)

Owing 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 able to handle the full standards of the Scheme programming language as it is implemented by MzScheme [4].

7.3 A SAMPLE SESSION

Before we go into the description of the individual components of the AT(x) system, we give an example of the execution of a homework task.

Solving a homework task includes the following subtasks: after logging into the WebAssign system the student chooses a task to solve in a web interface. The

task is presented as a web page containing forms for the solution. After filling in a solution (or correcting a previously supplied solution which is preserved between sessions), the student clicks a *submit* button. A few seconds later the system answers his submission with a new web page containing the analysis results. The submitted version replaces the previously preserved version of a solution.

Usually, several individual tasks are combined into an exercise. When the student is satisfied with all tasks in the exercise, he can close it, and the manual correction and assessment can start.

The following example is based on the AT(x) instance AT(S) for Scheme programs. 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. After authentication has been performed by WebAssign, this is the only input the student has to pass to the system in order to solve the task.

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

In this program the test for negative numbers is missing, and the first operator in the last line must be $*$ instead of $+$.
The system's output, identifying these two errors, 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 was aborted to enforce termination:
(fac -42)
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 10)
The result generated was 56 instead of the
expected result 3628800.
-----------------------------------------------------
```

One important 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 (for a rather small test set) at once.

7.4 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.

7.4.1 Components of the AT(x) System

AT(x) is divided into two main components: the main work is done by the analysis component (lower part of the analysis system in Fig. 7.1). 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 (upper part of the analysis system in Fig. 7.1). As shown in the figure, this interface completely performs the interaction between AT(x) and the WebAssign server. 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 independently for each of the AT(x) instances in the individual target languages, 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.

The background data in Fig. 7.1 consists of text templates for error messages used by the interface and different static inputs to the core analysis component as described in Sec. 7.5.1.

7.4.2 Communication Interface of the Analysis Component

The individual analysis component is the main part of an AT(x) instance. It performs tests on the students' programs and generates appropriate error messages. The performed tests and the detectable error types of AT(S) are discussed in detail in Sec. 7.5. Here, we concentrate on the 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 the types of the additional data fields are fixed for each error number.

An example of 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 entry contains the test `(fac 5)` causing the error, the third one contains the wrong result 16, and the fourth one the expected result 120.

The presentation of the messages in a readable form is done by the Java interface component. An example of such a presentation is given in Sec. 7.3.

7.4.3 Function and Implementation of the Interface Component

WebAssign provides a communication interface based on Corba to the analysis components. In contrast, the analysis components of AT(x) use a simple interface with textual communication via the stdin and stdout streams of the analysis process, which avoids the need to re-implement a Corba client in the language used for the analysis component. We therefore use an interface process connecting an analysis component of AT(x) to 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 AT(P)-instance [1]) generate a number of messages at the stderr stream, e.g. when loading modules. These messages can block the analysis process when the stderr 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. With a minor change to the Java interface component the messages from stderr can, of course, be accessed. From our experience (using SICStus Prolog and MzScheme) it is, however, preferable to catch errors by custom error handlers inside the analysis components, providing appropriate messages via the standard interface of the analysis component. This keeps the interface between the two components uniform and avoids the need for parsing messages from the compiler that are usually not designed with automatic parsing in mind.

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. 7.4.2. The sample session in Sec. 7.3 shows how this message is presented to the student.

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

Example 7.1. Consider the example session described in Sec. 7.3. Having error messages for all detected errors available, a student could write the following program that only consists of a case distinction and easily outfoxes the system.

```
(define (fac n)
   (cond
     ((= n -1) 'negative)
     ((= n -42) 'negative)
     ((= n 5) 120)
     ((= n 6) 720)
     ((= n 10) 3628800)))
```

To avoid the 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 instead 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 ex-

perienced users. A better overview of 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. Compared to returning just one message, this filter becomes more useful the more different errors can be distinguished.

- 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.

Hiding some of the error messages and test cases from the student is, however, not a safe way to avoid fine tuned programs. Iterated testing with programs tuned towards all tests which are known so far eventually yields the whole set of test cases. Since the system is designed to support the students (and since e.g. a randomized test case generation needs special care to cover all special cases and is therefore quite complex), this weakness can be accepted for the purpose of AT(S).

7.4.4 Global Security Issues

Security is an issue that is common to all instances of AT(x). It should therefore be addressed by the framework rather than in every individual instance. Security includes the following topics:

- **Authentication:** access to WebAssign (apart from some introductory web pages) is only possible by authenticated users. User identificators are available with every submission. Since AT(x) is only accessible via WebAssign (using a Corba interface), and since WebAssign has proven its reliability during several years with thousands of students, further authentication is not necessary by AT(x).

- **Denial of service:** AT(x) is only accessed via WebAssign. The AT(x) system can therefore be protected by a firewall that can only be passed by the WebAssign server.

- **Malicious code from students:** without restricting the considered programming language, students' programs can access the machine running an AT(x) instance directly. Mechanisms preventing problems for the service include:

  – The analysis component can rule out malicious code. Here it is problematic to detect every malicious program without rejecting correct programs.

  – Several UNIX mechanisms can be employed to provide some relative form of security. It is possible to protect the machine and AT(x) itself, but a malicious program might still interfere with an analysis of another student's

program. This approach is implemented at the moment in AT(x) and was sufficient so far for programming exercises that do not need access to hardware.

– For system programming or other areas with extended need for security, a sandbox approach is necessary. In such an approach the interface component could start each instance of the analysis component in a new sandbox simulating the machines behaviour. Adapting or implementing such a sandbox is an area of future work in our implementation.

7.5 THE CORE ANALYSIS COMPONENT

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 of the general requirements for these analysis components and describe a component for analyzing programs in Scheme instantiating AT(x) to AT(S) in more detail.

7.5.1 Requirements on the Analysis Components

The intended use in testing homework assignments rather than arbitrary programs implies some important requirements and 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.

Though the requirements formulated here carry over to an extension towards automatic *assessment* (comparable to e.g. [5]) we especially focus on the goal of quick, reliable and understandable feedback given to the students.

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 mainly for Prolog programs (cf. [1]). They are, however, also available for AT(S).)

This part of input is called the *static input* to the analysis component because it usually remains unchanged between the individual test sessions. Each 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.

We now 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 suspected), 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. Especially in checking generated results for correctness, special care has to be taken that all correct alternative solutions are considered correct.

Runtime errors of every kind must be caught without affecting the whole system. 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 [4]. 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 can be detected: a threshold for the maximal number of function calls (counted independently for each function) 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 be tested in a supervised manner. The different approaches for supervising recursion include 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 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.

7.5.2 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 does not indicate an error.

A problem inherent to functional programs is the potentially complex structure

[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.

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 7.2. 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, $1 \leq l \leq n$.

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 only differ in the order of the 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. Instead, we 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 the submission is corrected and graded by a human tutor. In order not to confuse the student it is, however, critical not to report correct results as erroneous, merely because they differ from the expected result.

Example 7.3. For the task in Example 7.2 the equality check could be

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

with an appropriate sort function `sort`.

A more complex test can e.g. consist of comparing functions from numbers to numbers. Such a test can return true after comparing the results of both functions for n (for some appropriate number n) well-chosen test inputs for equality. If an assignment is expected to return more complex functions, it is even possible to consider the returned function as new homework and to call the analysis component recursively, provided that the specimen program is given as a result for a new task.

Termination analysis of Scheme programs is done by applying a program transformation to the student program. We have implemented a function that counts the number of function calls for different lambda expressions independently and that aborts the evaluation via an exception if the number of calls exceeds a threshold for one of the lambda expressions. To perform the counting, 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 count function containing a unique identifier of the lambda expression and `tester::tmp` is just a dummy variable whose value is not used.

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 several error messages, which can stem from failed equality checks, the termination control and the runtime system. These include wrong results generated by the student's program, aborted executions due to suspected infinite loops, syntax errors, undefined identifiers, and several other kinds of runtime errors detected by the system. A generic error code can be used by the system to give detailed descriptions on failed tests for certain program properties, e.g. factorial can be checked always to return a non-negative integer.

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 is then presented to the student. When implementing a new instance of AT(x) an appropriate set of codes needs to be defined, and text templates for these codes have to be provided to the interface component by instantiating an abstract Java class.

7.6 IMPLEMENTATION AND EXPERIENCES

The AT(x) framework with its instance AT(S) (and a further instance for Prolog) is fully implemented and operational. The analysis component runs under the Solaris 7 operating system and, via its Java interface component, serves as a client for WebAssign.

Owing 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 an additional two weeks were necessary.

At the moment the system with the instances AT(P) and AT(S) for Prolog and Scheme goes through its first application in a programming course. It is available only for selected homework tasks. Although using the system means sending in homeworks in two different ways (WebAssign for the selected available tasks, plain paper sent in by mail for the remaining tasks) two thirds of the active students used the system. Feedback from the students was positive in general, mentioning both a better motivation to solve the tasks and better insight in the new programming paradigm.

7.7 RELATED WORK

In the context of teaching Scheme, the most popular system is DrScheme [11]. The system contains several tools for easily writing and debugging Scheme programs by students. 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. [13]).

A system very similar to our approach is presented in [5] for Ceilidh. While our approach is focused on quick and understandable feedback to the students, Ceilidh is used for automatic assessment of homework assignments. Since the WebAssign system offers automatic assessment, it might be possible to extend the scope of our system in this direction. Because of the undecidability of program equivalence and program correctness, however, we decided to run some tests with hand correction of assignments first, using the corresponding pre-correction outputs to simplify the manual final correction.

Other testing approaches to functional programming (e.g. QuickCheck [3]) 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 be found in our approach as well. The randomized generation of test cases used in QuickCheck is a possible extension of our system. We must, however, make sure that tests for special cases are contained in every test set.

A further topic related to our approach is the area of intelligent tutoring systems (ITS) (see e.g. an overview in [8]). Our approach does not aim at the goals of an ITS, but is just a testing tool to be integrated in the distance learning context of the FernUniversität in Hagen. Even when thinking of an "intelligent" testing tool, finding and understanding the errors in the student solutions is a first necessary step, so that our tool can be of use in constructing an ITS in future.

An automatic tool for testing programming assignments in WebAssign already exists for the programming language Pascal [12]. 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 possible analysis tools aiming at 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 read-evaluate-print-loop of the language, and to implement the analysis component completely in the target language.

Putting the differences together, the AT(x) approach is novel in providing a framework that is highly generic over both the chosen programming language (with a focus on high-level languages providing a REP-loop) and the communi-

cation platform (up to now mostly WebAssign, but also VILAB). It is completely focused on aiding the student in solving programming tasks in a distance learning framework.

7.8 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 self-tests and homework assignments.

In this paper we have presented 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, which is available at the FernUniversität Hagen, and 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 components, instances of AT(x) for different programming languages are generated. This was presented for the instance AT(S), 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.

Future work on AT(S) can address the following topics. While the current system aids the students in preparing their homework assignments, an automatic assessment stage comparable to [5] can reduce the effort required by the teacher to correct them. This, however, makes it necessary to understand errors not only in terms of the I/O-behaviour, but in terms of the source code. The precise assessment can be calculated as the similarity of the student's solution to a specimen program according to some appropriate distance function. Understanding errors in terms of the source code is also necessary in order to extend AT(S) towards an ITS. Furthermore, an ITS needs a model of the student's programming skills and possible misunderstandings, in order to find reasons for certain errors and to provide more specialized help. In all these extensions we believe that useful online assistance to the students should always be one of the most important aims (or even the most important aim) in distance learning.

REFERENCES

[1] C. Beierle, M. Kulaš, and M. Widera. Automatic analysis of programming assignments. In *Proceedings of the 1. Fachtagung "e-Learning" der Gesellschaft für Informatik (DeLFI 2003)*. Köllen Verlag, Bonn, 2003.

[2] 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.

[3] K. Claessen and J. 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., Sept. 18–21 2000. ACM Press.

[4] M. Flatt. *PLT MzScheme: Language Manual*, Aug. 2003.

[5] S. Foubister, G. Michaelson, and N. Tomes. Automatic assessment of elementary standard ml programs using ceilidh. *Journal of Computer Assisted Learning*, 1996.

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

[7] R. Kelsey, W. Clinger, and J. R. (Editors). Revised^5 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9):26–76, Sept. 1998.

[8] R. Lelouche. Intelligent tutoring systems from birth to now. *Künstliche Intelligenz*, 13(4):5–11, Nov. 1999.

[9] R. Lütticke, C. Gnörlich, and H. 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.

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

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

[12] H. WebAssign. `http://www-pi3.fernuni-hagen.de/WebAssign/`. 2003.

[13] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.