University of Maryland College Park

Institute for Advanced Computer Studies UMIACS-TR-94-59
Department of Computer Science CS-TR-3275

KASSANDRA: THE AUTOMATIC GRADING SYSTEM*

Urs von Matt†

January, 1994

Abstract. An automatic grading system is presented for grading assignments in scientific computing. A student can interactively use this system to check the correctness of his program assignments. The grade for a correct solution is automatically recorded. This paper also considers the security problems with such an automatic grading system.

Key words. Computerized grading, science education, computer aided instruction.

* This report is available by anonymous ftp from `cs.umd.edu` in the directory `/pub/papers/TRs`. It also appears in SIGCUE Outlook, 22 (1994), pp. 26–40.

† Institute for Scientific Computing, ETH Zürich, CH-8092 Zürich, Switzerland; current address: Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742; e-mail: `na.vonmatt@na-net.ornl.gov`.

KASSANDRA: THE AUTOMATIC GRADING SYSTEM

Urs von Matt*

Abstract. An automatic grading system is presented for grading assignments in scientific computing. A student can interactively use this system to check the correctness of his program assignments. The grade for a correct solution is automatically recorded. This paper also considers the security problems with such an automatic grading system.

Key words. Computerized grading, science education, computer aided instruction.

1. Introduction. At ETH Zürich an undergraduate course in scientific computing is lectured by W. Gander every year. About six teaching assistants have to handle up to 200 students. The grading of assignments represents a major activity for the teaching assistants. Kassandra was designed to alleviate this problem.

Kassandra is based on the observation that numerical assignments can be tested fairly easily. Typically the student has to implement a procedure which operates on given input data and generates some output. If this procedure computes the right answers for different inputs we assume it to be correct, and the student gets credit for it.

We use the software packages Maple [2] and Matlab [5] in the aforementioned course in scientific computing. Consequently, the student must express his solution in terms of these languages. Besides, we also make use of a classical programming language, namely Oberon [9, 14], the successor of Modula-2 [13].

Section 2 of this paper gives a short overview of the history of automatic grading. In section 3 we discuss the functionality of Kassandra from the student's point of view. Section 4 is devoted to the security aspects that such a grading system has to meet. From these requirements we can derive the internal structure of Kassandra in section 5. We discuss in section 6 how assignments are incorporated into Kassandra. Finally, section 7 reports our experiences with Kassandra.

2. History. There are a number of publications concerned with assisting teachers in checking the homework of their students. An approach, which is closely related to Kassandra, was described in 1965 by Forsythe and Wirth [3]. Their system was used for numerical analysis courses at Stanford University. As part of their homework the students had to write subroutines in the BALGOL language, a dialect of ALGOL 58. Their solutions were turned in on punched cards. Then, the teacher ran a grader program which called each of the solutions in turn. Forsythe and Wirth give an example of a grader that tries to evaluate the quality of integration subroutines written by the students.

In 1969, J. Joss also developed a similar system at ETH Zürich. Unfortunately, his work has never been published.

Outside the area of computer science, several papers have been published on the grading of homework assignments. In most cases these systems check the numerical values that have been computed by the students. The data is usually entered into the computer by some sort of reading device.

* Institute for Scientific Computing, ETH Zürich, CH-8092 Zürich, Switzerland; current address: Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742; e-mail: `na.vonmatt@na-net.ornl.gov`.

Taylor and Deever [12] describe a system used in physics and mathematics courses at Otterbein College in Westerville, Ohio. They generate a unique set of data for each student's homework. The students turn in their assignments on porta-punch cards, and their responses are evaluated in a batch mode on a daily basis.

In 1983, Rottmann and Hudson [10] presented a system that supports the teacher in grading multiple choice assignments. They used their system for the instruction of physics courses at the University of Houston, Texas. A mark-sense device serves to input the data into the computer. Then, the computer is used to analyse the results and generate reports that are posted in the classroom.

Myers [6] describes a system for the grading of freshman chemistry laboratory experiments at the West Virginia Institute of Technology at Montgomery, West Virginia. The students turn in their homework on special mark-sense cards. Then, these cards are input into an Apple II computer with the use of a card reader. The computer checks the answers and generates reports for the teacher and the students.

Several papers discuss the automation of multiple-choice tests. In [8], Posteraro, Blackwell and Huddleston present the computer program TECHSCORE which tabulates the responses to multiple choice questions, calculates the percentage of each response and displays the information in tabular form. This system is used to analyse the responses to employee evaluation questionnaires and the results of student examinations at the Texas Tech University School of Medicine in Lubbock, Texas.

Lira, Bronfman and Eyzaguirre [4] describe Multitest II, a system for the generation, correction, and analysis of multiple choice examinations. The tests are graded by means of a user-definable numerical grading system. This system is used at the Catholic University of Chile at Santiago.

A related system for multiple choice tests has also been developed by S. Bartoň [1] from the University of Agriculture and Forestry Brno, Czech Republic. However, his work has not yet been published.

Sometimes the grading of assignments is also supported by computer-assisted instruction (CAI) systems. Two such systems are described by Piotrowski [7] and by Schreihofer, Foster, Gleason, Harting, and Hiltz [11].

3. Functionality of Kassandra. The main function of Kassandra consists in checking the correctness of program assignments. A student can interactively call Kassandra and specify the number of the assignment. By that he triggers the following activities:

1. A test program is started which calls the student's procedure with a number of different test data.
2. The results are analysed.
3. In case of errors the test program tries to figure out the causes and reports them to the student. If the solution passes all the tests it is assumed to be correct, and the grade is given (provided the deadline of the assignment has not yet expired).

This mechanism is depicted as Fig. 1.

Kassandra also provides some auxiliary functions. Above all the student can inquire the assignments that have already been credited. Thus he can keep track of his progress.

Finally, the student can also make himself known to Kassandra. This feature is useful, because an assignment is credited to the account from where it has been

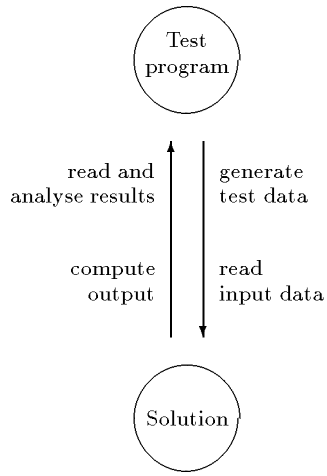

FIG. 1. *Design of Kassandra*

turned in. Such an account is likely to be an (anonymous) numbered course account. Therefore the assistant needs to know which students are actually working on these accounts. This feature is even more important in the case of teamwork where all the members of a group are using the same account.

4. Security. An automatic grading system must meet a few security requirements. The following two problems are of special importance:

1. The test program must be protected against unauthorized access by the student because it contains the reference solution with which the student's solution is compared. Consequently, the student must not read this program, but nevertheless he should be able to execute it in order to check his assignment.
2. All the credits are stored in a file. Obviously this file must not be writable to the student. For reasons of privacy it also should not be readable. Nonetheless this file must be updated if an assignment is turned in successfully.

These two problems are fundamental. There are many half-hearted solutions that rather hide the problem than solve it. All of these approaches are characterized by the fact that they offer a so-called "security by complexity". They try to disguise the structure of the test program to a degree where it can only be deciphered by an inordinate amount of work. The following "solutions" may serve as an illustration:

1. In the case of a conventional programming language the test program is implemented as a main program. Only the object file is available to the student, and at run time the student's solution is linked to the main program.
2. In the case of Maple or Matlab the test program is stored in a file that is readable to the student but whose name is unknown. Additionally this file is located in a directory where the student has no search permission.
3. Under Unix, these "solutions" can be refined even further if so-called set-uid programs are used. When the student executes such a program he gets the same privileges as the assistant who owns the program. In particular he can read and write files that would be inaccessible otherwise.

This security problem can be solved only if a strict separation between assistant and student is adhered to. A good solution consists of *two* test programs exchanging

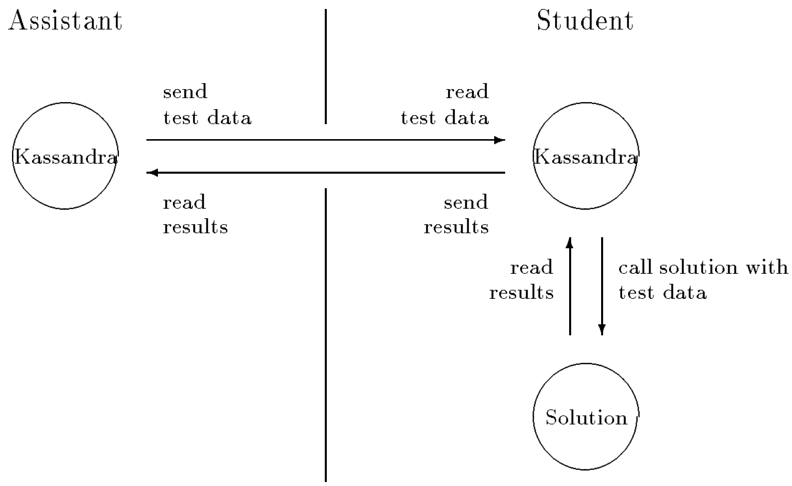

FIG. 2. *Conceptional Design of Kassandra*

data. This is depicted as Fig. 2. Note that the test program on the assistant's side is accessible just to the assistant. It can establish connections to arbitrary many students and test their assignments. Furthermore, it is this test program that decides on the success of the test and credits the assignment.

The test program on the student's side, however, is readable and executable to the student. If a student calls Kassandra he essentially starts this test program on his side. Its sole purpose is to establish the connection to the other test program and to guarantee the data transfer.

The security of this approach is based on the fact that only data is exchanged between the assistant and the student. The assistant asks a number of questions to the student (i.e. he transmits test data), and the student tries to answer these questions (i.e. he send his results back). It is irrelevant how the student gets at his answers. He is only judged by his results. There is no other way to influence the test program on the assistant's side.

5. Internal Structure of Kassandra. Assignments in scientific computing include the use of the software packages Maple and Matlab. Therefore we must provide a means to incorporate them into Kassandra. In general such packages provide no direct support for interprocess communication; however, we can assume that all these packages support input and output on files. Under Unix this feature can be used to contrive a simple interprocess communication scheme.

Unix supports so-called named pipes. These are buffers (first-in-first-out queues) between a producer and a consumer process. Thus we can accomplish the necessary synchronization between two communicating processes. Such a pipe has also an entry in the file system and can therefore be handled like an ordinary file. This means that from the point of view of the software package there is no difference between an ordinary file and a named pipe, except that reading from an empty pipe or writing on a full pipe will block the current process.

For the data transfer between student and assistant, named pipes are not the best choice. Kassandra works with sockets (bidirectional links between two independent processes). In this way the solutions of different students can be tested simultaneously. Furthermore, it is possible for the assistant's processes to run on a different computer.

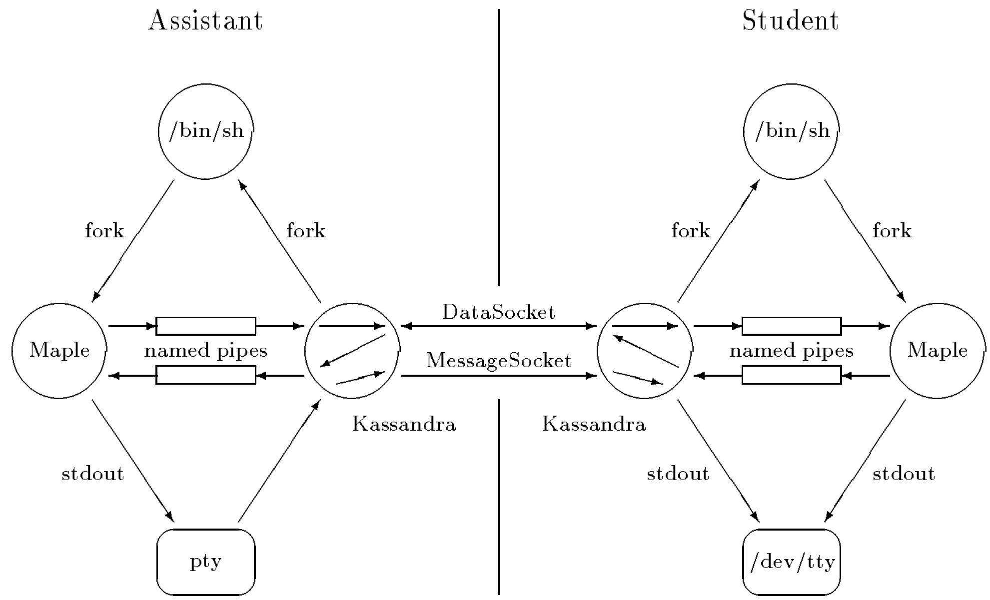

Fig. 3. *Detailed Design of Kassandra*

This represents the ultimate separation between assistant and student. The final scheme is depicted as Fig. 3.

The following remarks will illuminate this figure:

1. Circles represent Unix processes. Rounded rectangles stand for terminals.
2. The process "Kassandra" on the assistant's side is running permanently as a daemon. It is waiting for a student to establish a connection and check an assignment.
3. As soon as the student starts Kassandra the sockets between the two processes are set up. If the student wants to check an assignment the named pipes are created in the filesystem. Then on both sides a Bourne shell is started executing a script file. These scriptfiles contain the proper test programs (e.g. Maple code).
4. After these initializations the two Kassandra processes only ensure the data transfer between student and assistant. For practical reasons we not only use a single socket to transfer the test data, but also another one to forward messages from the assistant's test program to the student.
5. We use a pseudo terminal (pty) to forward messages from the assistant's test program to the student. In this way the test program (e.g. Maple) behaves interactively. Otherwise the output would be buffered and transferred only when a certain block size has been reached.
6. It is possible to check the syntax of the data flowing in the DataSocket. This is especially important for software packages like Maple that directly execute their input from a file.

6. Extensibility. A main feature of Kassandra is its extensibility. It is easy enough to add new assignments to the system. We achieve this by a division of Kassandra into a kernel and a collection of test programs. The kernel remains the same for all the assignments.

In order to install a new assignment in Kassandra the assistant only needs to write the two test programs and update a configuration file. In this configuration file he can specify a deadline for each assignment.

Things become a bit more complicated if one wants to grade programs written in another software package. Basically such a package must meet the requirements discussed in section 5. If the data stream in Fig. 3 need not be checked, only minor modifications to Kassandra are necessary. Otherwise one needs to specify the exact syntax of the data stream and implement a corresponding parser. In the case of a sufficiently simple syntax this can be done with a moderate amount of work.

7. Experiences with Kassandra. The undergraduate course in scientific computing is lectured every year, and it is attended by up to 200 students. There are six teaching assistants giving classes. During this course, about 50 assignments are handed out to the students. Consequently, up to 10^4 positive grading decisions are made by Kassandra each year. We do not record the number of unsuccessful trials by the students, but we may assume this number to be substantially higher than the number of accepted assignments.

The main design goal of Kassandra was to alleviate the work load on the teaching assistants. Since up to 10^4 grading decisions are now performed automatically each year, Kassandra has been a great success in this respect.

In the winter term of 1992/93, the students' opinions about the course in scientific computing were gathered via a questionnaire. A total of 74 students have answered the questions, and we now give their answers concerning the two questions on Kassandra. The first question relates to the acceptance of Kassandra:

How do you feel about the use of Kassandra?	
15 %	I am against Kassandra. The assignments should be checked by the teaching assistants.
36 %	It does not matter whether the assignments are checked by Kassandra or by the teaching assistants.
49 %	I prefer it if my assignments are checked by Kassandra.

The use of Kassandra is endorsed by the majority of the students. As a matter of fact there are also a number of advantages to the students that contribute to the good reception of the system. For instance, the student is free to decide when to solve his homework. Kassandra is available to check his assignments at any time. In particular he can also work at home and access the system by a modem. It is possible to test the assignments even after the deadline, but without getting a credit, of course.

The second question is concerned with the quality of the automatic grading:

How well does Kassandra check your assignments?	
46 %	A teaching assistant would check my assignments better than Kassandra does.
26 %	Kassandra checks my assignments as well as a teaching assistant.
28 %	Kassandra checks my assignments better than a teaching assistant does.

Many students, who are not satisfied by the grading quality of Kassandra, complain on the error messages they get. They wish that Kassandra would also recognize and comment on fundamental misconceptions in their solutions. Many students also struggle because their solutions have to comply with the strict syntactic and semantic requirements of such an automatic grading system.

The work to develop good test programs should not be underestimated. First, the assignments must be designed in such a way that they are well suited for automatic checking. This includes a very concise description of the problem and its intended solution. Then the test programs must be devised such that they not only generate test data for the average case of the problems but also for the special cases.

If sophisticated test programs are available, however, Kassandra can often judge an assignment better than an assistant could. For a human it is easy to miss some subtle details if one merely analyses the program itself. Only the execution of such a program will reveal these hidden bugs.

We should also warn the reader against exaggerated expectations about such an automatic grading system. For instance, it is no longer possible to judge the programming style of the student. Even complicated or inefficient solutions are accepted. It has also become easier for the student to copy assignments. But in our opinion it cannot be the aim of such a grading system to detect plagiarism, and we count on the self-responsibility of the student in this respect.

8. Conclusions. The automatic grading system Kassandra has been introduced in the course on scientific computing at ETH Zürich in order to alleviate the work load of the teaching assistants. The students can interactively check their homework assignments on the computer, and correct solutions are graded automatically. The teaching assistants no longer need to undergo the tedious chore of grading assignments. Rather, they can now invest their time in the much more attractive job of developing sophisticated test programs. As soon as the test programs have been implemented this initial effort quickly pays off when the same course is offered repeatedly. Thus, Kassandra has been very effective in reducing the work load on the teaching assistants.

Kassandra has proved its considerable flexibility, as it is capable of checking assignments implemented in such diverse languages as Fortran, Maple, Matlab, and Oberon. It would not be difficult to include even further software packages.

Kassandra is also received well by the students. Such an automatic grading system is much more impartial in grading than a human, and it is infinitely patient with lazy students.

Since Kassandra is used on a regular basis, appropriate security precautions must be taken in order to avoid unauthorized access to the system. We have proposed a safe solution to this problem, which consists in separating the test programs into two parts. The test program on the assistant's side contains the reference solution, and it generates the data for the test runs. On the student's side we have the other part of the test program which is in charge of calling the student's solution. Since these two parts can only communicate by exchanging data, we can guarantee the integrity of the system.

9. Appendix. In this appendix we present more details related to the actual implementation of the test programs. In section 9.1 we discuss the two test programs on the student's and the assistant's side for a sample Maple assignment. These comments, however, also apply to assignments in Matlab and in conventional compiled languages, as we will see in sections 9.2 and 9.3.

The quality of Kassandra stands and falls with the quality of the test programs. Ideally, the test data should comprise "typical" problems along with a number of special cases. To illustrate the point we assume that the student must write a procedure to solve the linear system of equations $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$. Here, the test program would first generate a couple of well-conditioned matrices as test data. Afterwards, it analyses the accuracy of the solver with the help of a few ill-conditioned matrices. Matrices of size 1 and 2 make sure that these special cases are handled correctly. Finally, the test program would set up linear systems that can only be solved with a proper pivot strategy.

Our sample test programs in sections 9.1 and 9.3 have been reduced to the bare essentials in order to fit into the limited space. A real-life test program would run more test cases and analyse and comment the results in more detail. First a few general remarks are in order:

1. There are a number of Unix environment variables available in order to parameterize the test programs. This allows to easily install Kassandra in another directory, and to change the number of an assignment. The following variables are used:
   - `KASSANDRA_HOME` Kassandra's home directory.
   - `KASSANDRA_USER` The username of the student who is checking an assignment.
   - `KASSANDRA_EX` The number of the assignment that is being checked.
   - `KASSANDRA_PID` The process identification of Kassandra.
2. It proves to be practical to effect tests both in the assistant's and the student's test program. On the student's side we check that the desired result is computed and that the type of the result meets the requirement. Only then the assistant's test program makes sure that the numerical values agree with the exact values.
3. When the student passes the test successfully he gets his credit by means of a program called "Update". Here we check that the deadline of the assignment has not yet expired. If the assistant calls Update interactively he can also credit an assignment after the deadline.

9.1. Sample Assignment in Maple. Let us now present the implementation of the test programs with the help of a sample Maple assignment. For the sake of simplicity we assume that the student only has to implement a Maple procedure for the evaluation of the Chebyshev polynomials $T_n(x)$. They can be defined by the recurrence relationship

$$\begin{aligned} T_0(x) &:= 1, \\ T_1(x) &:= x, \\ T_n(x) &:= 2xT_{n-1}(x) - T_{n-2}(x), \qquad n > 1. \end{aligned}$$

A possible solution would be the following Maple code which the student is asked to store in the file `Chebyshev.maple`:

```
Chebyshev := proc (n, x);
  if n = 0 then
    RETURN (1)
  elif n = 1 then
    RETURN (x)
  else
    RETURN (collect (2*x*Chebyshev (n-1, x) - Chebyshev (n-2, x), x))
  fi;
end:
```

Now, let us first have a look at the test program on the student's side:

```
#!/bin/sh
# Initializations
. $KASSANDRA_HOME/src2/maple.sh

#################### Begin of Preamble ####################

echo "Test of Assignment $KASSANDRA_EX"
echo

if test ! -f Chebyshev.maple
then
  echo "File Chebyshev.maple not found!"
  exit
fi

###################### End of Preamble #####################

echo "starting maple ..."
maple -q << EOF_maple

fifo_out_filename := \`$FIFO_OUT\`:
fifo_in_filename := \`$FIFO_IN\`:
read (\`$KASSANDRA_HOME/src2/Kassandra.maple\`):

################### Begin of Testprogram ###################

read (\`Chebyshev.maple\`):

if not assigned (Chebyshev) then
  lprint (\`The variable Chebyshev is not defined!\`):
  stop:
fi:

if not type ([Chebyshev], [procedure]) then
  lprint (\`Chebyshev is not a procedure!\`):
  stop:
fi:

n := ReceiveInteger ():
x := ReceiveFloat ():
result := Chebyshev (n, x):
if not type ([result], [float]) then
  lprint (\`Chebyshev does not compute a floating-point number:\`):
  print (result):
  stop:
fi:
SendFloat (result):

#################### End of Testprogram ###################

# synchronization with other Maple
request := ReceiveInteger ():
quit:
EOF_maple
```

This test program on the student's side consists of a Bourne shell script. First it makes sure that the file `Chebyshev.maple` exists. Then the Maple program on the student's side is started. For the sake of convenience the Maple code is also contained within the shell script. This enables us to perform certain preprocessing on the test program. For instance, the names of the pipes, that are necessary for the interprocess communication (cf. Fig. 3), are stored in the global variables `fifo_out_filename` and `fifo_in_filename`. Furthermore, the procedures for sending and receiving data (`SendInteger`, `ReceiveInteger`, `SendFloat`, and `ReceiveFloat`) are defined by the first `read`-statement.

After these initializations the file `Chebyshev.maple` is read, and it is checked whether the variable `Chebyshev` has been assigned a procedure. If all has gone well up to now the actual test data is received. In our case only a single test is performed. Of course, a real test program would carry out a number of tests with carefully selected test data.

We call the student's procedure with the test data. If no floating-point number is returned an error is signalled. Otherwise, the result is sent to the assistant's test program for further analysis. At the end we have a short piece of synchronization code to make sure that both test programs terminate at the same time.

We are now ready to present the corresponding test program on the assistant's side:

```
#!/bin/sh
# Initializations
. $KASSANDRA_HOME/src1/maple.sh

maple -q << EOF_maple

# Initializations
fifo_out_filename := \`$FIFO_OUT\`:
fifo_in_filename := \`$FIFO_IN\`:
read (\`$KASSANDRA_HOME/src1/Kassandra.maple\`):

#################### Begin of Testprogram ###################

Chebyshev := proc (n, x)
  local T2, T1, T;

  if n = 0 then
    RETURN (1)
  elif n = 1 then
    RETURN (x)
  else
    T2 := 1;
    T1 := x;
    for k from 2 to n do
      T := collect (2*x*T1 - T2, x);
      T2 := T1;  T1 := T;
    od;
    RETURN (T)
  fi;
end:

# Initialization of the random number generator
_seed := `date +%m%d%H%M%S`:  # date is evaluated by the shell

n := 2 + rand (10)();
x := (rand (4*10^10)() - 2*10^10) / Float (1, 10);
SendInteger (n):
SendFloat (x):
```

```
ref := Chebyshev (n, x):
result := ReceiveFloat ():

if abs (result - ref) <= Float (1, -7) * abs (ref) then
  lprint (\`Your procedure Chebyshev is working all right.\`):
  system (\`Update -user $KASSANDRA_USER -ex $KASSANDRA_EX\`):
else
  lprint (\`Your procedure Chebyshev is still erroneous.\`):
fi:

#################### End of Testprogram ###################

# synchronization with other Maple
SendInteger (0):
quit:
EOF_maple
```

This test program also consists of a Bourne shell script. As soon as Maple is started and the necessary initializations have been performed, we define the reference solution. Notice that this procedure `Chebyshev` is coded non-recursively in contrast to the student's implementation. But since we are assessing programs by executing them this does not cause any problems.

In order to generate random numbers for n and x that differ from one test run to the next we initialize the `_seed` of Maple's random number generator by the current time of day. The command `date` is a program under Unix which is called by the Bourne shell prior to the execution of Maple.

Afterwards, the actual test data is generated and transferred to the student's test program. While the student is computing his answer the reference solution is calculated by a call of the procedure `Chebyshev`. The student's result is received and analysed. If the answer is accurate enough the assignment is graded by a call of the program `Update`. Otherwise an error message is issued. Finally, the test program terminates after executing the synchronization code which corresponds to the synchronization code in the student's test program.

For this assignment a sample Kassandra session appears as follows:

```
$ Kassandra -ex 991
Test of Assignment 991

starting maple ...
                                        n := 7

                                   x := -.636596708

Your procedure Chebyshev is working all right.
Assignment 991 has been accepted.
```

As soon as the student has started Kassandra, the test data for this run is displayed. When his solution has passed this test the assignment is immediately graded. By executing the Unix command `Kassandra -ex` the student can also obtain a list of all his assignments that have already been credited.

9.2. Sample Assignment in Matlab. Matlab assignments can be checked in very much the same way as Maple assignments. Since Matlab has only a single data type, the matrix, we only need the two procedures `SendMatrix` and `ReceiveMatrix` to transfer data back and forth between the assistant's and the student's test program.

9.3. Assignments in Compiled Languages. Kassandra is also well suited to test programs written in a conventional programming language like Modula-2, C, or Fortran. In this case we want to implement the test programs also in the same programming language. Still we can use the process structure of Fig. 3.

The student has to implement his solution as a procedure stored in a designated file. The shell script on the student's side contains the Unix commands to compile and link the student's procedure to a main program. This main program is provided by the assistant, and it is responsible for the data transfer and the calling of the student's procedure.

On the assistant's side we also need to write a test program in the same programming language. For this task a set of data transfer procedures is available. The corresponding shell script does little more than call the precompiled test program.

These comments apply equally to programming languages like C, Fortran, Pascal, or Modula-2. In the case of Oberon, however, a few modifications to Kassandra become necessary. In Oberon it is not possible to write stand-alone programs that are executed by a Unix shell. Rather the entire Oberon system consists of a single process, as far as Unix is concerned. Furthermore, Oberon comes with its own user interface, and all the procedures have to be activated from within this interface by interactive commands. As a consequence of this design philosophy we have implemented the student's side of Kassandra in Oberon as a set of modules. This means that the process structure on the student's side in Fig. 3 is now realized in Oberon by means of different modules. Thus the module `Kassandra` represents the interface to the assistant. Similarly, the test programs on the student's side are also implemented as modules.

On the other hand the test programs on the assistant's side are written in Modula-2, since they have to run as independent processes under Unix. Consequently, Fig. 3 remains unchanged on the assistant's side.

To illustrate this scheme we assume that the student has to implement the error function

$$\operatorname{erf}(x) := \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

around the origin by means of the taylor series

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)\cdot k!}.$$

The student is asked to implement his solution as a module `Erf`, which could look as follows:

```
MODULE Erf;

  IMPORT Aufgabe993, Math;

  PROCEDURE* Erf (x: REAL): REAL;
    VAR k               : INTEGER;
        term, sum, sum1: REAL;
  BEGIN
    sum := x;  term := x;  k := 0;
    REPEAT
      INC (k);
      term := -term*x*x / k;
```

```
    sum1 := sum;
    sum := sum + term / (2*k + 1)
  UNTIL sum = sum1;
  RETURN 2*sum / Math.sqrt (Math.pi)
END Erf;

PROCEDURE Test993*;
BEGIN
  Aufgabe993.Test (Erf)
END Test993;

END Erf.

System.Free Erf ~
Erf.Test993
```

The student is provided with a skeleton of this module, and all he has to do is to fill in the code for the procedure `Erf`. In order to check his solution the student only needs to compile his module and to activate the command `Erf.Test993` by a simple mouse-click. Note that the module `Erf` imports the module `Aufgabe993` which represents the test program on the student's side. This module looks as follows:

```
MODULE Aufgabe993;  (* Student *)

  IMPORT Kassandra;

  TYPE ErfProc* = PROCEDURE (x: REAL): REAL;

  PROCEDURE Test* (proc: ErfProc);
    VAR ok     : BOOLEAN;
        x      : REAL;
        request: LONGINT;
  BEGIN
    Kassandra.StartTestExercise (993, ok);
    IF ok THEN
      Kassandra.ReceiveReal (x);
      Kassandra.SendReal (proc (x));
      (* synchronization with Assistant *)
      Kassandra.ReceiveInteger (request);
      Kassandra.SendInteger (0);
      Kassandra.EndTestExercise
    END
  END Test;

END Aufgabe993.  (* Student *)
```

The module `Kassandra`, which is imported by `Aufgabe993`, provides the procedures to initiate and terminate a test run. Furthermore, it implements the mechanism for the data exchange between the test program on the student's side and the assistant's side. It should also be pointed out that the student has to pass his solution as a procedural parameter (parameter `proc` in `Test`). It is checked by the compiler whether the student's procedure matches its declaration in the module `Aufgabe993`.

Finally, we can present the test program on the assistant's side:

```
MODULE Aufgabe993;  (* Assistant *)

  IMPORT Kassandra, MathLib, RealIO, SimpleIO;

  VAR x, ref, result: REAL;
      receipt       : INTEGER;
```

```
  PROCEDURE Erf (x: REAL): REAL;
    CONST Pi = 3.14159265358979323846264338328;
    VAR k              : INTEGER;
        term, sum, sum1: REAL;
  BEGIN
    sum := x;  term := x;  k := 0;
    REPEAT
      INC (k);
      term := -term*x*x / FLOAT (k);
      sum1 := sum;
      sum := sum + term / FLOAT (2*k + 1)
    UNTIL sum = sum1;
    RETURN 2.0*sum / MathLib.Sqrt (Pi)
  END Erf;

BEGIN  (* Aufgabe993, Assistant *)
  x := Kassandra.Random () * 2.0 - 1.0;
  SimpleIO.WriteString ("x = ");
  RealIO.WriteReal (x, 9, 6);
  SimpleIO.WriteLn;

  Kassandra.SendReal (x);
  ref := Erf (x);
  Kassandra.ReceiveReal (result);

  SimpleIO.WriteString ("Your result: ");
  RealIO.WriteReal (result, 9, 6);
  SimpleIO.WriteString ("    correct value: ");
  RealIO.WriteReal (ref, 9, 6);
  SimpleIO.WriteLn;

  IF ABS (result - ref) <= 1.0E-5 THEN
    SimpleIO.WriteString ("ok.");
    SimpleIO.WriteLn;
    Kassandra.AcceptExercise
  ELSE
    SimpleIO.WriteString ("You're out of luck.");
    SimpleIO.WriteLn
  END;

  (* synchronization with Student *)
  Kassandra.SendInteger (0);
  Kassandra.ReceiveInteger (receipt)
END Aufgabe993.  (* Assistant *)
```

This test program is written in Modula-2. An auxiliary module `Kassandra` provides the necessary procedures for transferring data and accepting an assignment. A snapshot of Oberon-Kassandra is presented as Fig. 4.

REFERENCES

[1] S. Bartoň, *LEARN and SIFLEARN*, University of Agriculture and Forestry, Brno, Czech Republic, 1991.

[2] B. Char, K. Geddes, G. Gonnet, B. Leong, M. Monagan and S. Watt, *Maple V Language Reference Manual*, Springer-Verlag, New York, 1991.

[3] G. E. Forsythe and N. Wirth, *Automatic Grading Programs*, Comm. ACM, 8 (1965), pp. 275–278.

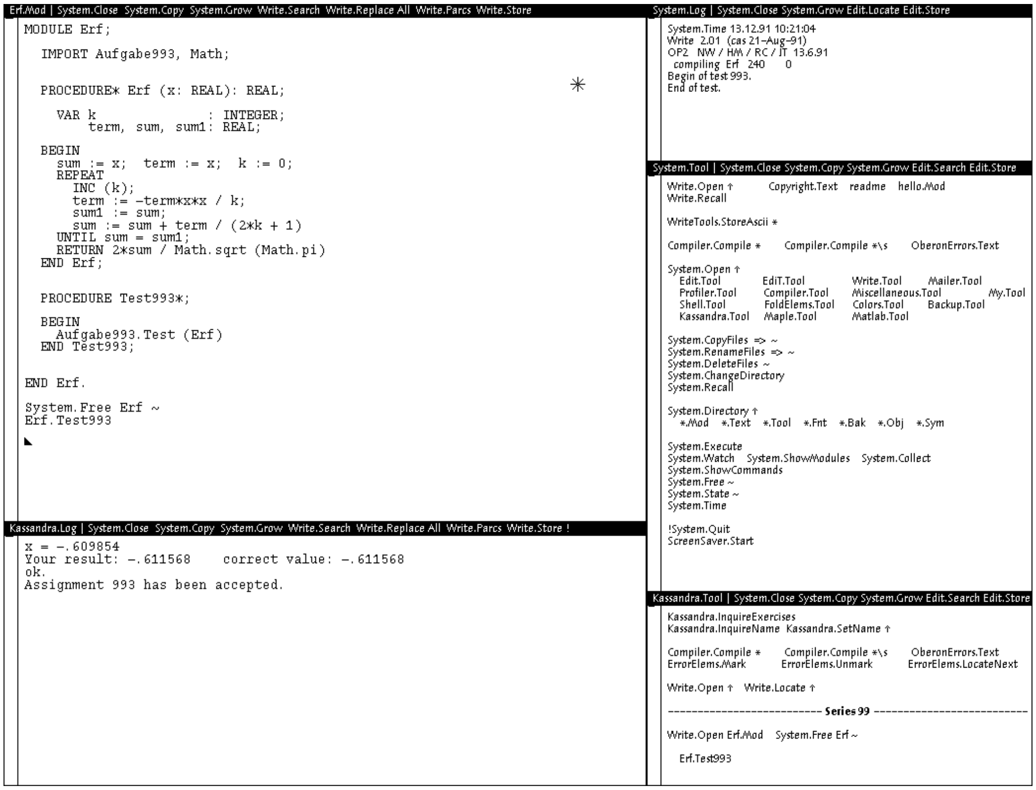

FIG. 4. *Oberon-Kassandra*

[4] P. LIRA, M. BRONFMAN AND J. EYZAGUIRRE, *MULTITEST II: a program for the generation, correction, and analysis of multiple choice tests*, IEEE Transactions on Education, 33 (1990), pp. 320–325.

[5] C. MOLER ET AL., *MATLAB User's Guide*, The MathWorks Inc., South Natick, 1990.

[6] R. MYERS, *Computerized Grading of Freshman Chemistry Laboratory Experiments*, Journal of Chemical Education, 63 (1986), pp. 507–509.

[7] J. PIOTROWSKI, *The small computer assisted lecturing system*, SIGCSE Bull., 20 (1988), pp. 8–12.

[8] R. POSTERARO, D. BLACKWELL AND A. HUDDLESTON, *Techscore: A program for tabulating the results of multiple choice questions and correcting multiple choice examinations*, Comput. Biol. Med., 16 (1986), pp. 259–265.

[9] M. REISER, *The Oberon System, User Guide and Programmer's Manual*, ACM Press, New York, 1991.

[10] R. M. ROTTMANN AND H. T. HUDSON, *Computer Grading As an Instructional Tool*, Journal of College Science Teaching, 12 (1983), pp. 152–156.

[11] E. SCHREIHOFER, J. FOSTER, B. GLEASON, H. HARTING AND S. HILTZ, *Software tools for a virtual classroom*, in Proc. of NECC '88, ed. W. Ryan, Int. Council Comput. Educ., Eugene, 1988, pp. 230–236.

[12] J. TAYLOR AND D. DEEVER, *Constructed-Response, Computer-Graded Homework*, American Journal of Physics, 44 (1976), pp. 598–599.

[13] N. WIRTH, *Programming in Modula-2*, Springer-Verlag, Berlin, 1985.

[14] N. WIRTH AND J. GUTKNECHT, *The Oberon System*, Software—Practice and Experience, 19 (1989), pp. 857–893.