Automatic Student Coaching and Monitoring Thanks to AUTOMATON: The Case of Writing a Compiler

Linden, Isabelle; Toussaint, Hubert; Classen, Andreas; Schobbens, Pierre-Yves

Published in:
ECEL 2008: Proceedings of the 7th European Conference on e-Learning, Cyprus 6-7 November 2008

Publication date:
2008

Document Version
Early version, also known as pre-print

Automatic Student Coaching and Monitoring Thanks to AUTOMATON.
The Case of Writing a Compiler.

Isabelle Linden, Hubert Toussaint, Andreas Classen, Pierre-Yves Schobbens
University of Namur, Namur, Belgium
ili@info.fundp.ac.be
hto@info.fundp.ac.be
acs@info.fundp.ac.be
pys@info.fundp.ac.be

Abstract:
Evaluating programs written by students is a repetitive and time-consuming task. Too often, this evaluation is done once the development is finished, leaving the students no time to change and enhance their program. In an effort to solve this problem, we developed AUTOMATON, a tool capable of automatically analysing the functional capabilities of pieces of code written by students

The current version of AUTOMATON has been built as a *complete*, *flexible* and *customisable* tool. It provides two kinds of functionalities. On the one hand, AUTOMATON *manages* the student's assignments, their *evaluation* and the *reporting* of the results. On the other hand, AUTOMATON provides support for *coaching* and *monitoring* an exercise, possibly supervised by a human tutor.

Reachable through a web-interface, available 7 days/week and 24 hours/day, AUTOMATON is a precious virtual tutor for the students. Indeed, AUTOMATON provides them feedback within a couple of minutes. Using this feedback, the students can increase the quality of their work several times until the deadline. For the tutor, discharged of the technical manipulations, continuous monitoring of the students' progress becomes possible. Furthermore, AUTOMATON provides useful pieces of information that allow coaching the students more efficiently by focusing on relevant points.

In this paper, we examine various pedagogical scenarios and show how AUTOMATON can be customised to support them. We explain the main functionalities of AUTOMATON and show how they can be used to the benefit of the students and the tutor alike.

AUTOMATON has been in service at our university for ten years, mainly for a compiler writing exercise, an important part of the *syntax and semantics* course, and integral part of the curriculum. We describe in detail how we use AUTOMATON as a continuous incremental evaluator in this context. The description focuses on both technical and pedagogical issues.

Keywords: Automatic evaluation, e-coaching, e-monitoring

1. Context

In the first stages of a computer science curriculum, a student writing his first lines of code examines the question: *does my code do what it is expected to do?* This question carries, in fact, two questions: *(i) what is the program expected to do?* and *(ii) does the behaviour of the program do exactly that?* Reformulated this way it becomes clear that the programming task is part of a more complex elaboration schema. Actually, the programming phase is, at least, preceded by a *specification phase,* that defines the expected behaviour of the program, and followed by a *verification phase*, that checks if the behaviour of the program matches those specifications. These two tasks are integral parts of the software engineering process, being themselves subject of complete research areas and courses.

Most basic programming courses assume that a specification is available and focus on writing code and documentation. Thus students do not only learn how to write code, but also how to read and understand a program specification. Indeed, the fulfilment of the specification is the key quality of their work, and therefore understanding the specification is as important as writing correct code.

The teaching of those capabilities starts with small programming assignments. Their code requires about a hundred lines, a workable size for a human reader. Thus the tutor can provide direct feedback to students. When more ambitious projects are considered, this is not possible any more. Indeed, the code contains several thousands of lines, requiring a deep analysis by the tutor to verify the

fulfilment of the specification. The correction then requires substantial testing efforts and is generally only done once, for the final version of the program handed in by the students. The problem with this approach is, however, that the testing of final code often reveals multiple errors that might stem from the very first version, that the students have no more opportunity to correct.

AUTOMATON was initially conceived to deal with exactly this problem. Without any intermediate feedback, the students would have no means to realise that their understanding of the specification was incorrect. However, providing the tests directly would have exempted the students from reading and understanding the specification, and, for this reason, would not have fit the teaching goals. Therefore, the first task of AUTOMATON was to offer the students a distant access to the testing scripts of their tutor. This solution helps the students while still fitting the teaching goals. It provides the students with a score indicating their degree of fulfilment of the specification (continuous feedback), and the only way to improve the score is to improve the understanding of the specification (teaching goal).

This is still the core functionality of AUTOMATON. Yet, it has grown well beyond that, and multiple extensions were proposed and implemented in the 10 years of its existence. The complete evolution of AUTOMATON has been (and still is) motivated by the concern for providing to teachers a more flexible and customisable tool that fits their pedagogical requirements.

The rest of the paper is structured as follows. In section 2, we draw a quick overview of features provided by existing assessment management tools and argue for the need for generic tools. In section 3, we describe AUTOMATON and its main functionalities for the student as well as for the tutor. The flexibility and customisability of AUTOMATON are developed and illustrated in the next sections. In section 4, we present the particular use case of writing a compiler and how AUTOMATON has been customised in order to capture the particular teaching requirements of this exercise. Then, section 5 suggests how AUTOMATON can be used in various teaching scenarios. Finally, we conclude the paper in section 6 by summarising the spectrum of customisable functionalities of AUTOMATON.

2. Assessment management tools

2.1 Existing tools

Douce (2005a) provides a wide historical overview of existing tools developed in view of managing student submissions in different ways. The authors identify 3 generations of tools according to the technologies supporting them. Addressing each of them in detail would be beyond the scope of this paper. Let us focus here on the main functionalities provided by tools of the third generation: the web oriented systems.

Curator (Virginia Tech 2004) (previously called Grader) admits submissions of files of any type. In case of source code, it supports correctness evaluation based on textual comparison of generated outputs with expected ones.

The BOSS system (Heng 2005) is more flexible: it allows preliminary checking of submitted files and supports several programming languages. In particular, the testing of Java programs is not limited to textual comparisons but can be integrated with JUnit.

RoboProf (Daly 1999) is limited to the evaluation of small programs, to be typed in a textbox. Its particularity is that it manages gradual difficulty levels. It deals with Java only.

The ASAP tool (Douce 2005b) not only checks correctness based on the textual analyses of the outputs but also provides mechanisms for testing individual functions.

The Web-based automated Programs Assignments aSsessment System PASS (Choy 2005) allows two kinds of submissions. On the one hand, testing submissions are automatically evaluated based on a textual comparison and the results are directly returned to the students. On the other hand, grading submissions are reported to the tutor, who can then add detailed feedback.

Those tools offer adequate functionalities but allow little customisability and no introduction of new mechanisms. Teachers and researchers are continuously imagining new scenarios and the adaptation of those tools might prove difficult. The requirement of a generic tool has been taken into account in more recent tools.

CourseMarker (Higgins 2003), (previously CourseMaster, built on Ceilidh) manages a wide set of languages. It provides correctness evaluation (based on textual comparison) and quality evaluation (typographic qualities, plagiarism detection,…) and a very refined and customisable marking process.

Oto (Tremblay 2007) is a generic and extensible tool for marking program assignments. In particular, correctness evaluation is extended to the use of JUnit and marking involves numerous customisable parameters.

Those last two tools provide interesting customisation features. However, they have a strong marking (summative) orientation. In our view, the main pedagogical use of assignment management tools is continuous formative (self-)evaluation. In these cases, level definitions, qualitative feedback are more crucial aspects of the tool.

2.2 Functionalities panel

As shown by the survey of existing tools, assessment management tools are not only in charge of administration but can also be used in various pedagogical scenarios. Let us complete the overview by drawing a panel of the functionalities that a tool can offer to teachers. Tremblay (2007) suggests a classification of assessment management tools along three high-level functionalities: management of assignments, evaluation of correctness and evaluation of quality. We suggest a slightly refined classification.

The base functionality is the *administrative management*:

- Providing the specifications of the assignments
- Receiving submissions: format and deadline verification,…
- Filing submissions

The core of the system consists of *submission analysis*:

- Correctness evaluation (tests through textual comparison, JUnit or any other)
- Quality evaluation (quantitative and qualitative evaluation)
- Marking: based on the two previous evaluations, a marking process can be proposed

Related to these analysis tasks, the capability of defining levels and providing feedback to students are important points.

Finally, *monitoring* functionalities are useful for learning guidance:

- Managing access to analysis results for students and tutor
- Generating reports and views on the elements of the submissions and the results.

In order to be complete, one should also consider other categories such as user management. Indeed, they have a strong impact strong on the usability of the tool for teacher and tutor. However, they do not modify the quality of pedagogic support.

The development of AUTOMATON aims at providing customisable functions in each of these categories.

3. Introducing AUTOMATON

All along the specification and the development of the second (and current) version of AUTOMATON, the leading idea was to provide a flexible tool. Let us introduce shortly the vocabulary and the use of AUTOMATON.

The complete work required from students is called an *exercise*. According to the aim of providing early feedback to the students, a complete exercise can be split into several sub-tasks or intermediate tasks called *problems.* Each of these problems is verified by a set of *tests.* Participation in an exercise is allowed for students or groups of students. A *submission* consists in providing files to

AUTOMATON and requiring their evaluation according to a given problem. When it receives a submission, AUTOMATON applies four treatments to the received files as represented on figure 1. Firstly, the *initialisation* applies a pre-treatment to the submission. This initialisation can be specific for each problem. If the initialisation succeeds, every test associated to the evaluated problem is run according to its own script. When every test has been performed, an *analysis* is done on the generated files. This analysis can, for example, involve the generation of a report in a file of the system, its sending to the tutor and/or to the submitter. Finally, a *cleaning* phase removes all the useless files from the system. After the run of the submission, the students and tutors can access several pieces of information through a web interface.

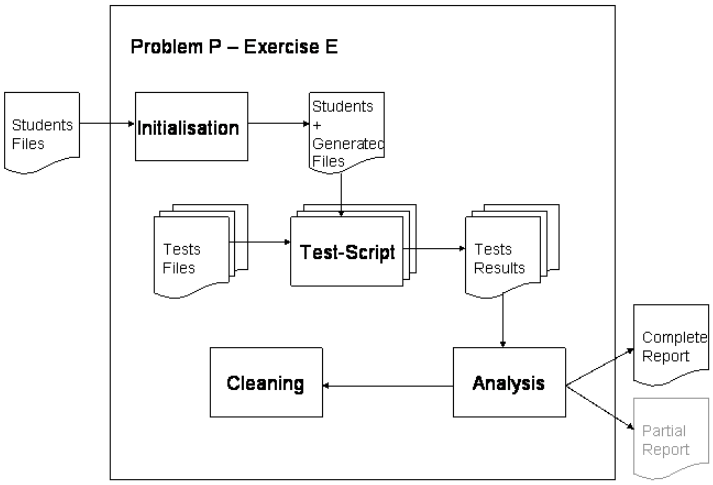

Figure 1: A Submission Computation

As this short description of the functioning of AUTOMATON indicates, introducing a new exercise in the system requires the definition of several parameters. However, for most of them, standard scenarios are already defined. A more precise description of those parameters will be provided in section 4 through the example of a compiler writing assignment.

3.1 The functionalities offered to the students

Within the availability period defined by the author of the exercise, all the functionalities of AUTOMATON are accessible for registered students 24 hours a day, 7 days a week through a secure web interface.

First of all, students can upload sets of files to the system and require their evaluation according to a given problem. The evaluation is then run in an asynchronous way. Indeed, in case of demanding tests, the evaluation might take up to a couple of minutes.

At any time, a student can consult the list of the previous submissions of his group and for each of them, its date and its status (uploaded, running, evaluated). Moreover, for any submission, the files submitted and the analysis generated by AUTOMATON can be consulted. Files can be downloaded and their versions be compared with an integrated diff tool.

The full-time and distant availability of AUTOMATON provides to students with an almost immediate feedback on their work. Moreover, the splitting of an exercise into problems allows a step-by-step continuous (and gradual) evaluation of their progress.

3.2 The functionalities offered to the tutor

The fact that AUTOMATON takes over an important part of the evaluation process does not mean that tutor will become obsolete. The goal of AUTOMATON is not to replace the judgement and didactic capabilities of the tutor, but only the repetitive and technical tasks of his job. In addition, AUTOMATON helps the tutor to follow the progress of groups so that problems can be detected early on. Let us

present the functionalities that AUTOMATON offers all along the realisation of an exercise. The next section details how they can be used.

Before the beginning of the exercise, the tutor has to introduce the users and/or groups allowed to make submissions for the exercise into the system.

At any moment, the tutor can obtain a *class activity report* for a given date. This report involves for any group, the list of the submissions realised before the specified date. The submissions are ordered by problem and for each problem by date. For each submission, the report mentions the date, the submitter and indicates how many tests of the problems terminated with success, how many terminated but provided an unexpected result, how many took too much time and how many terminated with failure. Fully successful submissions are emphasised. Various shorter versions can also be generated.

The set of submissions can be browsed through a web interface. The proposed views are the following ones.

- For each problem, the state (success/failure/no submission) of each group
- For each group, the list of all its submissions (date – submitter – results)
- For each submission, the submitted files, the list of the result of each test, the returned analysis and for each test, the generated files and tests files.

4. AUTOMATON as a support tool for the development of a compiler

Roughly speaking, a compiler can be seen as a translator from a given *source language* into a *target language*. As an illustration of the *Syntax and Semantics* course at the University of Namur, students have to realise a compiler that can translate source code written in a Pascal-like language into PCode (a machine-like language defined first by (Wirth 1975); we use the version of (Wilhem & Maurer 1995)).

Writing a compiler is quite a challenging exercise for students. It requires the use of various techniques as well as a good understanding of the specifications of the languages. A compiler is also a very complex program to evaluate. Firstly, it is made of multiple files and its compilation requires several manipulations. Secondly, its functionalities are of two types. On the one hand the compiler has to detect if the provided source code is valid or not. On the other hand, it has to generate a computable version for a valid source code. The evaluation of the compiler thus consists in testing whether or not valid and invalid source codes are correctly distinguished, and whether the computation of the generated program does what it is supposed to.

4.1 Definition of the exercise in AUTOMATON

The definition of the language is provided to the students in form of a denotational semantics. The careful reading and understanding of this specification is an important step of the work. In order to force the students to pay the required attention to this reading, the feedback on failing tests is very terse. All they have access to are the result of the test (success, failure,...), the output and the generated error messages. They do not receive information about the test itself (no input files). At first, this might seem not to be helpful for the students. However, as the complete exercise is split into several problems, the students can incrementally work through the specification. This avoids the tendency of students to develop by counterexamples, modifying the code just to pass one test.

The exercise is divided into 12 intermediate problems separated into two categories: (1) syntactic (2) semantic. Each category consists of 6 sublanguages (i.e. constituting a part of the complete specification), each one being an extension of the previous one. This means that problems 11, 21, …, 61 check the syntactic verification and problems 12, 22, …, 62 check the validity of the generated PCode. The precedence relation is described by figure 2.

Figure 2: Precedence Relation among Problems of the Compiler Exercise

The introduction of the problems requires the definition of the initialisation, analysis and cleaning scripts, and the tests in each problem are associated with test-scripts.

The initialisation consists in the usual building of a compiler: (i) produce the code for the lexical analyser from the lexical description (in this case with flex (Nicol 1993)), (ii) produce the code for the syntactic analyser from the grammar (in this case with bison (Donnelly 2003)), (iii) compile the compiler (in this case with gcc (Stallman 2003)) and (iv) retrieve the compiler in an executable file.

The analysis is basic and consists in reporting how many tests succeeded, how many failed, how many were aborted because they took too much time/memory, and how many crashed. The cleaning consists in removing meaningless temporary files (generated code of the lexical analyser and generated code of the syntactic analyser). Produced output and error messages can provide useful information for the tutor.

For syntactic tests (serie 1), the script structure is shown on figure 3. The compiler is launched on a file containing the source code that is to be tested. If the computation terminates (within the time and memory limits, and without crash), it produces an output indicating whether the compiler considers the syntax as correct or not. This output is then compared to the expected result.

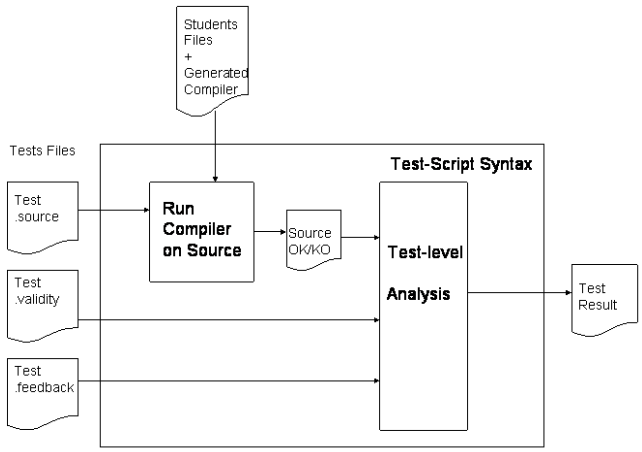

Figure 3: Test-script for syntactic tests

For semantic tests, the script is described in figure 4. As in the syntactic case, it begins by launching the compiler with the valid source code of the test as input. If the program is identified as invalid by the compiler, the analysis is done as in the previous case. Otherwise, the output produced by the compiler consists of the PCode translation of the source file. Unlike in the previous case, the expected output cannot be be compared to the actual output since the same source code might be compiled in various ways. Therefore, the PCode is evaluated by executing it on a set of predefined inputs using our P-machine, called GPMachine (Bontemps 2002). The source language of this exercise is always deterministic, so that there is exactly only one correct output of this computation. The actual output is hence compared with the expected output and the test and returns the result (possibly complemented by indications on what went wrong in case of failure).

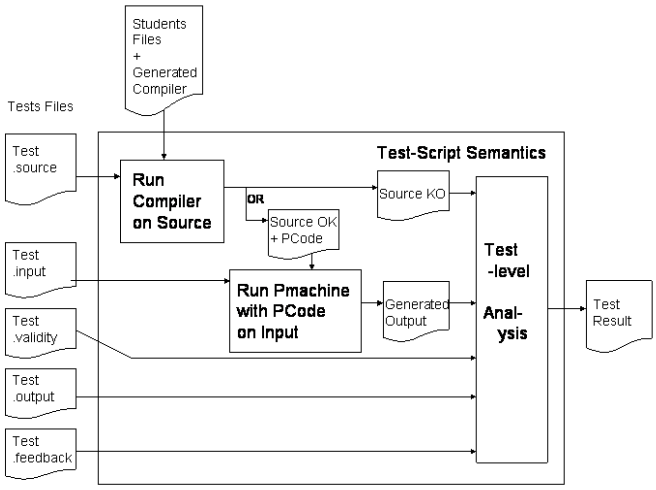

Figure 4: Test-script for semantic tests

4.2 Students monitoring along the compiler exercise

Intermediate deadlines are fixed for tests 22, 42, 62 and a final deadline for the delivery of the final version (together with a written report by the students). Except for those benchmarks, the students can follow their own organisation. Given the size of the exercise, however, supervision by a human tutor is often required. This supervision can be very discrete and flexible thanks to the use of AUTOMATON.

At the beginning of the exercise, the tutor observes if every group is submitting something within the first days of the work. If not, he can contact those groups and see what is hampering their progress: Do they have problems understanding the specification? Didn't they start working yet? Are there problems with the usage of AUTOMATON?

All along the exercise, the tutor can see the results of every group. In particular, if one group makes a lot of submissions for the same problem and always obtains the same results, he can, thanks to AUTOMATON, browse the submission files, consult the results files, and understand the exact reason of the failure and turn the attention of the students to that particular point of the specification. If the tutor considers it necessary, he can even provide a few lines of the test source code in order to help them fix their problem.

If a group requiring help consults the tutor, he can consult the set of their submissions and see for how long the students have been "fighting" with a problem and how much attempts they have done. This can help him to assess how desperate the situation is and what amount of help is required. He can then browse the submissions, find their exact problem and give them the appropriate hints.

At every intermediate deadline, and at the end of the exercise, the tutor generates a global report with indications about dates, numbers and success of the submissions. This often provides interesting information about the way the different groups work: Do they adopt a try and test strategy or are they systematic in solving problems? Do they work as a team? Which member of the group makes the most submissions? Who makes the successful submissions? Who works at night? Which group only starts submitting the day before the deadline? and so on.

4.3 Evaluation of this use case

The **quality** of student code increased sharply with the introduction of AUTOMATON. In the years in which the tool was not available, most of the students lacked an understanding of the subtleties of the specification. Because their own testing strategies suffered from the same problem were not aware of the problem. With the continuous feedback of AUTOMATON, however, most of the students were able to fulfil the complete functional specification.

A new important part of the tutors' job is now the definition of the exercise in AUTOMATON. It might appear that this is an additional task. This is, however, a short-sighted conclusion, since even without AUTOMATON, the tutor would need to define and execute tests. On the contrary, the structure of the tests imposed by AUTOMATON will be of help in defining these tests. Furthermore, AUTOMATON requires the tutor to define his tests before providing the exercise to the students. This will prevent him from having to write the tests as the exercise progresses, a time he should devote to the support of the students. This also ensures that all details of the semantics are well described, since the tests can unveil special cases.

It is also important to point out that through the continuous monitoring capabilities of AUTOMATON, the tutor is able to provide support that is more relevant to actual problems of the students. Moreover, he can focus his help on students who need it. Indeed, on the one hand (anxious) students doing a good job receive immediate confirmation of the quality of their work, and stop polling the tutor. On the other hand, students having more difficulties but who do not ask for help can be identified by consulting the submission statistics.

5. Further uses

AUTOMATON has been developed in order to be as flexible and customisable as possible. In particular, as we did for the GPMachine, it can be integrated with several pieces of software and the variety of possible application domains is consequently unlimited. Let us focus here on three uses considered in our programming languages team at the University of Namur.

In a first scenario, AUTOMATON could be used as a simple report receiver. Indeed, it suffices to define a problem involving no test with a simple initialisation verifying the number and types of the submitted files. Of course any other more sophisticated initialisation and analysis could be imaginable. For example, checking the size of the files, the number of words and so on.

A second scenario consists in using AUTOMATON in a competition perspective. In particular, we consider its use in managing programming contests. The exercise is made of a set of independent problems, the number of submissions allowed for every group is limited (checked by the initialisation script). The first group that succeeds solving a problem wins the associated points.

In the previous scenario, the students submit pieces of code that are checked against test input/output pairs. A dual exercise would consist in submitting pairs of input/output files in order to detect which of the programs are correct and which are not. This teaching goal of such an exercise would be to strengthen the student's capabilities in reading specifications and writing tests.

6. Conclusion

We showed how the introduction of AUTOMATON lead to an increased pedagogical quality in our department and what its benefits are for the supervision of large programming assignments. Let us conclude by a final focus on the main qualities that, to the best of our knowledge, AUTOMATON is the single tool to address simultaneously.

AUTOMATON has been built as a *complete*, *flexible* and *customisable* tool that provides two kinds of functionalities. On the one hand, AUTOMATON *manages* the student's assignments, their *evaluation* and the *reporting* of the results. On the other hand, AUTOMATON provides support for *coaching* and *monitoring* an exercise, possibly supervised by a human tutor.

There can be customisations at every level of the evaluation process: availability, treatment and feedback can be specified by the author of an exercise. The provided files can be analysed in many different ways, with possibilities ranging from the illustrated "compile-run-test" scenario, to the inclusion of the files within a wider application, running quality evaluation tools or running plagiarism detection tools. The analysis of a problem can build on the result of the different tests, but also on their crashes, their running time, the memory used, and so on. In the usual case of a "compile-run-test" scenario, the compilation can be done in any language (Pascal, Java, C, C++,...) and the testing

can be as various as comparing a generated output file with the expected one, verifying its matching with a given grammar, using it as a library for another program and so on.

The specification of the level of detail of the feedback provided to the students is an important pedagogical tool. A very detailed feedback leads the students to learn the thinking of their evaluator rather than to understand the problem, whereas a terse feedback leads to a more precise reading of the specification. Turning the feedback to (almost) nothing, AUTOMATON can be used to manage a contest or a final evaluation.

Discharged of running evaluations himself, the tutor will have more time for providing explanations and feedback to the students. Moreover, AUTOMATON, keeps him updated on the progress of the students, and he can use the information gathered by AUTOMATON to better focus his support and enhance the students' supervision in general.

To conclude let us point out that AUTOMATON will soon be available under the GPL licence, and hopefully see a growing community of users contributing to its development.

7. Acknowledgments

AUTOMATON is the brainchild of Professor Baudouin Le Charlier and its first proof-of-concept implementation was written by Vincent Letocart. Since then, several members of our Faculty have provided ideas or contributed to the project. Among them, let us mention Séastien Verbois, Stéphane Bonfitto, Yves Bontemps and Jean-Marc Zeippen. To all of them, we would like to express our sincere gratitude.

References

Bontemps, Y. (2002) *GPMachine: a virtual machine interpreting P-Code* [online]. [Accessed 9th July 2008]. Available from World Wide Web: < http://www.info.fundp.ac.be/~gpm/ >

Choy, M. Nazir, U. Poon , C.K. Yu, Y.T. (2005) "Experiences in Unsing an Automated System for Improving Students'Learning of Computer Programming", *LNCS*, Vol. 3583, pp. 267-272

Daly, C. (1999) "RoboProf and an introductory computer programming course", *ACM SIGCSE Bulletin,* Vol. 31, No. 3, September, pp 155-158.

Donelly, C. and Stallman, R. (2003) *Bison manual : using the YACC compatible parser generator,* The Free Software Foundation.

Douce, D. Livingstone, D. and Orwell, J. (2005a) "Automatic test-based assessment of programming: A review", *ACM Journal on Educational Resources in Computing,* Vol. 5, No. 3, September, Article 4.

Douce, D. and al. (2005b) *A Technical Perspective on ASAP – Automated System for Assessment of Programming,* Proceedings of the 9th CAA Conference, Loughborough: Loughborough University

Heng, P. Joy, M. Boyatt, R. and Griffiths, N. (2005) "Evaluation of the BOSS Online Submission and Assessment System", Research report CS-RR-415 at The University of Warwick department of computer science.

Higgins, C. Hegazy, T. Symeonidis, P. Trintsifas, A (2003) "The CourseMarker CBA System: Improvements over Ceilidh", *Education and Information Technologies,* Vol. 8, No. 3, September, pp287-304.

Nicol, G.T. (1993) *Flex: The Lexical Scanner Generator.* The Free Software Foundation. ISBN 1882114213

Tremblay, G., Guérin, F., Pons, A. and Salah, A. (2008) "Oto, a generic and extensible tool for marking programming assignments", *Software Practice and Experience*, Vol. 38, No. 3, March, pp 307-333.

Virginia Politechnic Institute and State University (2004) *Curator: an Electronic Submission Management Environment* [online]. [Accessed 20th September 2008]. Available from World Wide Web: <http://courses.cs.vt.edu/curator/>

Wilhelm, R. and Maurer, D. (1995) *Compiler Design.* Wokingham, England, Reading, MA : Addison-Wesley.

Wirth, N. (1975) *Algorithms + Data Structures = Programs*. Prentice-Hall.