Informatics in Education, 2010, Vol. 9, No. 1, 115–132

© 2010 *Institute of Mathematics and Informatics, Vilnius*

Invariant Based Programming in Education – An Analysis of Student Difficulties

Linda MANNILA
Department of Information Technologies, Åbo Akademi University
Turku Centre for Computer Science
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
e-mail: linda.mannila@abo.fi

Received: September 2009

Abstract. In this paper, we analyze the errors novice students make when developing invariant based programs. In addition to presenting the general error types, we also look at what students have difficulty with when it comes to expressing invariants. The results indicate that an introductory course utilizing the invariant based approach is suitable from the very beginning of university studies in CS without being "too advanced". Although inventing the invariant was not found to be trivial, the main difficulty faced by novices when applying a correct-by-construction approach to program development seems to be related to weak skills in translating intuitive and informal statements into a symbolic form using logical notation in general and quantifiers in particular.

Keywords: invariant based programming, programming education, introductory formal methods, student difficulties.

1. Introduction

The concept of loop invariants was introduced by Naur (1966) and Floyd (1967) already in the 1960s. Continuing on their work, Hoare (1969) defined inference rules for correctness proofs based on loop invariants. This development lay the basis for Dijkstra's (1968, 1976) work on outlining a new programming methodology for writing programs with built-in correctness proofs. Half a decade later, Gries (1981) wrote a textbook on using this methodology aimed at education. Dijkstra also discussed the question of how much formality to include in the computer science (CS) curriculum in the paper "On the cruelty of really teaching computer science" (Dijkstra, 1989), a paper which was followed by many responses collected in "A debate on teaching computing science" (Denning, 1989).

Regardless of the rather early developments, loop invariants and formal techniques are commonly underutilized in introductory CS courses. There are many other reasons for this. First, these topics are typically considered difficult, requiring prerequisite knowledge of advanced mathematics and logic, which students simply do not have (Tam, 1992). As a result, different, some more informal, approaches for introducing invariants and program correctness concepts in education have been presented, e.g., (Arnow, 1994; Astrachan, 1991; Evans and Peck, 2006; Ginat, 1995; Tam, 1992). Unfortunately, we have not found any empirical evaluation of these approaches in the literature.

Moreover, most CS curricula treat the teaching of programming and the teaching of more formal courses as separate disciplines, where, for instance, logic is viewed as a separate "add-on" instead of as an integral part of programming. Consequently, students get only little exposure to correctness concepts, precise reasoning and symbol manipulation (Almstrum *et al.*, 2001). When formal techniques are taught as an activity independent from the programming process, students get the impression that formal techniques are only applicable in theoretical courses (McMaster *et al.*, 2007). This in turn gives rise to attitude problems, as students do not see any point in having to take more theoretical courses. Dijkstra described this as "mental resistance" among students (Gries, 1981).

Reed and Sinclair (2004) suggest that another reason for students' resistance may be found in the prevailing culture where students want and are used to getting quick results; a hacker mentality clearly does the job much faster than one aiming at verifying the correctness. Moreover, CS students seem to be driven by external factors, and learning the skills that they believe are relevant and are needed to earn money may make it difficult to motivate the study of other topics. Finally, the challenges can be traced back to the actual teaching situations. Most textbooks on introductory programming, for instance, do not discuss program correctness or concepts such as loop invariants (Astrachan, 1991). In addition, it is common for CS faculty to argue against formal methods (Roychoudhury, 2006).

Although attempts at introducing formal methods in education have been made and the importance of more rigorous formal reasoning in CS has been emphasized in curricula and other recommendations e.g., (Joint Task Force on Computing Curricula, 2001; McGettrick *et al.*, 2004), it seems safe to say that convincing students of the value of formal techniques is everything but easy.

Starting in 2007, we have been evaluating an approach to teaching "practicable formal methods" in a course for CS novices. We refer to this approach as *invariant based programming* (IBP). IBP is a visual program construction and verification methodology, which introduces a minimum of notational overhead and allows students to reason about correctness using mathematical concepts with which they are already familiar (such as set theory and basic logic). Through this course we aim at addressing several of the challenges discussed above: changing the image of formal methods as difficult while at the same time reducing the gap between theory and practice. Issues related to lack of teaching material or teacher attitudes can be addressed by educators and method developers. Overcoming students' mental resistance and changing their cultural preferences are however things that one could believe are not easily accomplished during a single course. Our experience from the course has, however, been positive, indicating that students find it fun and useful. In addition, they seem to appreciate learning about program correctness and seeing the programming activity from another perspective (Back *et al.*, 2007).

For this paper, we have conducted a thorough analysis of student created invariant based programs, focusing on the errors made. The aim is also discovering to what extent IBP requires knowledge and skills falling under the "too advanced" section. The study seeks to investigate three research questions:

1. What kind of errors do novices make when using IBP?
2. Do these errors change as the course progresses, and if so, how?
3. Does the year of study impact student performance?

The paper is organized as follows. Next, we describe the invariant based approach, after which we present the study design. In the following two sections, we put forward and discuss the results. The paper is concluded with some final words and ideas for future work.

2. Invariant Based Programming

IBP is an approach to constructing correct programs, where not only pre- and postconditions, but also loop invariants are to be written before doing any coding. The approach is not new – it was studied already in the 1970s by Reynolds (1978) and Back (1978, 1983). Similar ideas were also proposed by van Emden (1979). In 2004, Back (2005) revisited the topic and has since worked on developing IBP into a practical hands-on method.

In IBP, a program is constructed and verified at the same time. We use the term *situation* for a condition that describes a collection of states. A condition is a predicate on states and can thus be identified with the set of all states satisfying the predicate. Hence, the precondition, postcondition and loop invariants are situations. An invariant based program may have many situations and is not restricted to single-entry, single-exit control structures.

In essence, IBP provides a visual representation of a program. A variety of graphical programming/pseudocode formats have been proposed in the literature (Blackwell *et al.*, 2001; Roy, 2006), and all of these have one common goal: "to provide a clear picture of the structure and semantics of the program through a combination of graphical constructions and some additional textual annotations" (Roy, 2006; p.3). To our knowledge, these have, however, focused on representing control flow and data flow. IBP, on the other hand, describes programs from another perspective as it emphasizes the invariant properties of the program data structures, and thus makes it possible to reason about the correctness of the constructed program in a rather straightforward manner. This is accomplished without sacrificing either clarity or expressiveness of the diagrams.

2.1. *An Illustrating Example*

We will here exemplify the work flow for developing invariant based programs by constructing a program that implements the *selection sort* algorithm. We use a cursor to traverse an array from left to right, and for each position we find the smallest element to the right of the cursor and swap that element with the one pointed at by the cursor. After each swap the cursor is advanced, and the array is sorted when the entire array has been traversed.

We formulate the problem more precisely by drawing figures capturing the basic data structures involved and how their values change during execution of the algorithm. This

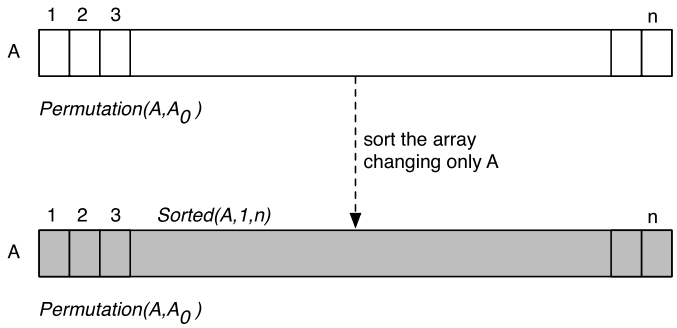

Fig. 1. Visualization of the specification.

is an essential step of the IBP work flow as the figures describe the algorithm at work and thus help the programmer get a feeling for the behavior of the algorithm. As this example illustrates, the figures also aid in identifying the situations of the program.

The first figure (Fig. 1) illustrates the specification (the pre- and postcondition), which helps us identify the initial and final situations. Here, A_0 refers to the initial array, i.e., the array where the elements are still in the original order. $Permutation(A, A_0)$ means that the elements in array A form a permutation of the elements in the initial array A_0.

To describe the final situation, we need a way to express that an array, or a part of it, is sorted. We assume that $Sorted(A, i, j)$ means that the elements in array A are non-decreasing in the interval $[i, j]$. As situations are sets of states, the final situation is a subset of the initial one where an additional constraint, $Sorted(A, 1, n)$, is satisfied. Shading is used to indicate which elements have already been sorted. Initially, no elements have been sorted (no shading), whereas in the final situation all elements have been sorted (entire array is shaded).

We use a *nested invariant diagram* to represent the program and the strengthening of situations. Our first diagram is shown in Fig. 2. Because situations are contained in each other (as sets), they are drawn in a nested manner. An constraint in an outer set needs therefore not be repeated in the inner ones (for instance, n: $integer$ holds in both the initial and the final situation). Dashed arrows are used to indicate the computation that we want to define and are labeled with a potential guard (written in square brackets, see Figs. 3–7) and the variables that may be changed in the computation.

In the same manner as the final situation was identified as a subset of the initial one, we introduce new situations by adding new constraints to the ones present in the more general situations. We further develop the figure of the algorithm at work by introducing the intermediate situation (Fig. 3). Here only the elements in the beginning of the array ($A[1]...A[i-1]$) have been sorted (illustrated by the shaded region). In addition, we know that all elements in the unsorted part are larger than or equal to any element in the sorted part. To express this, we define the predicate $Partitioned(A, i)$ to indicate that every element in array A below index i is smaller or equal to any element in A at index i or higher.

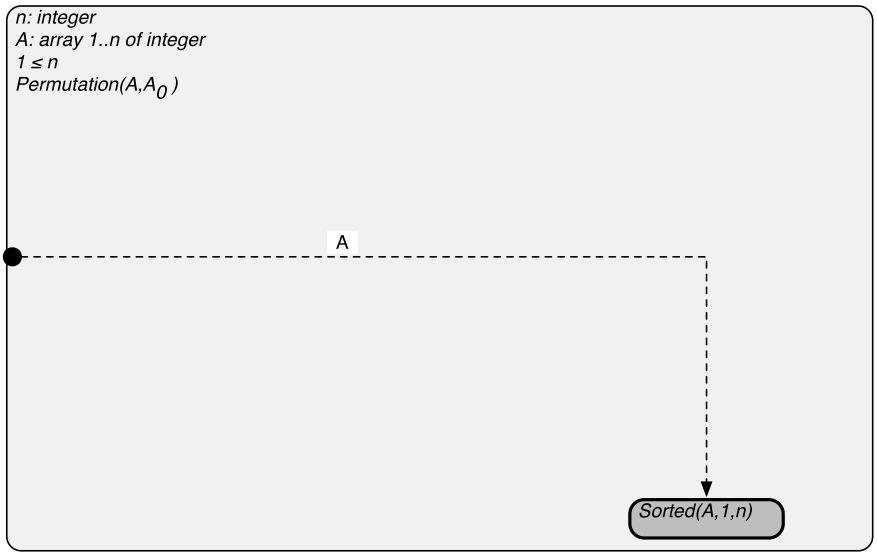

Fig. 2. Invariant diagram with the initial and final situations.

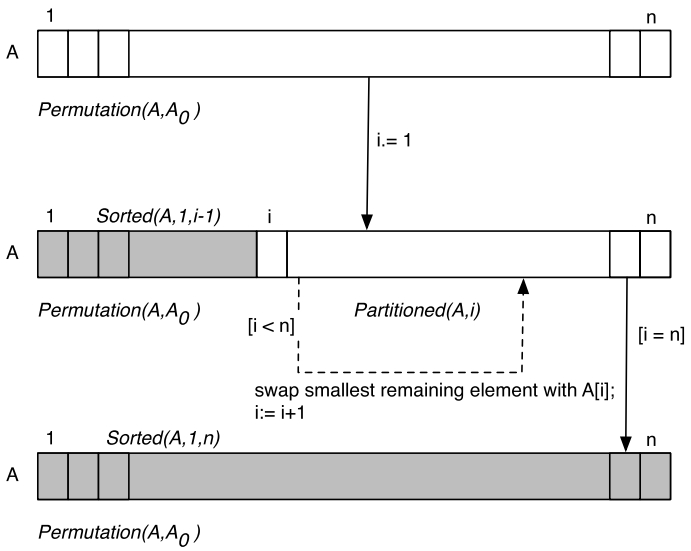

Fig. 3. Sorting program with invariant.

As is shown in the corresponding diagram (Fig. 4), this newly inserted situation is a subset (i.e., a constrained version) of the initial situation. Whereas dashed arrows illustrate what we want to accomplish, we use solid ones to indicate computations that we have already planned and defined. We call these solid arrows *transitions*. Each transition is labeled with a potential guard and the program statements executed when the transition is carried out.

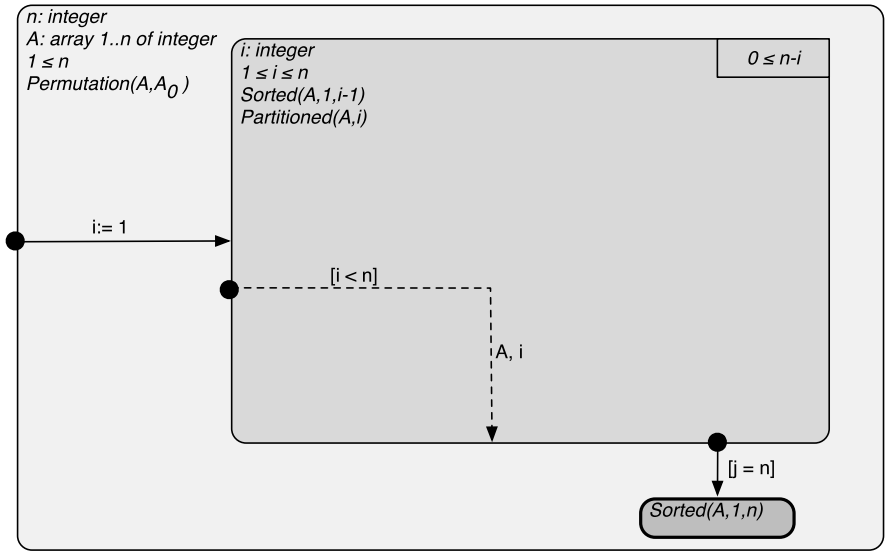

Fig. 4. Invariant diagram with the intermediate situation inserted.

We still need one more loop to find the smallest element in the remainder of the array, since this requires that we scan through all remaining elements. Again, we use figures as a tool to help us get an idea of how the algorithm works. We add yet another situation, where part of the unsorted elements have been scanned for the smallest element (indicated by lighter shading). The new situation is shown in Fig. 5 and the corresponding invariant diagram in Fig. 6.

To finish the program, we need to define the final transition, i.e., how the smallest element is to be found. Fig. 7 shows the complete invariant diagram for the selection sort program.

When all situations and transitions have been added to the diagram, we still have to check that no infinite loops exist, i.e., that the program *terminates*. We introduce a termination function (variant) for each intermediate situation. A termination function is an integer function that is bounded from below and whose value is decreased before re-entering the situation. The termination functions are written in the right upper hand corner of the respective invariants (Fig. 7).

Finally, we must check that the program is *live*, i.e., that termination only occurs in final situations. In practice, this means that we must make sure that for all situations, except for final ones, there is at least one enabled transition.

An invariant based program is correct if it satisfies the three criteria above, i.e., 1) is consistent, 2) terminates and 3) is live. For a more in-depth presentation of IBP as a method, see the articles by Back (2005, 2006, 2009).

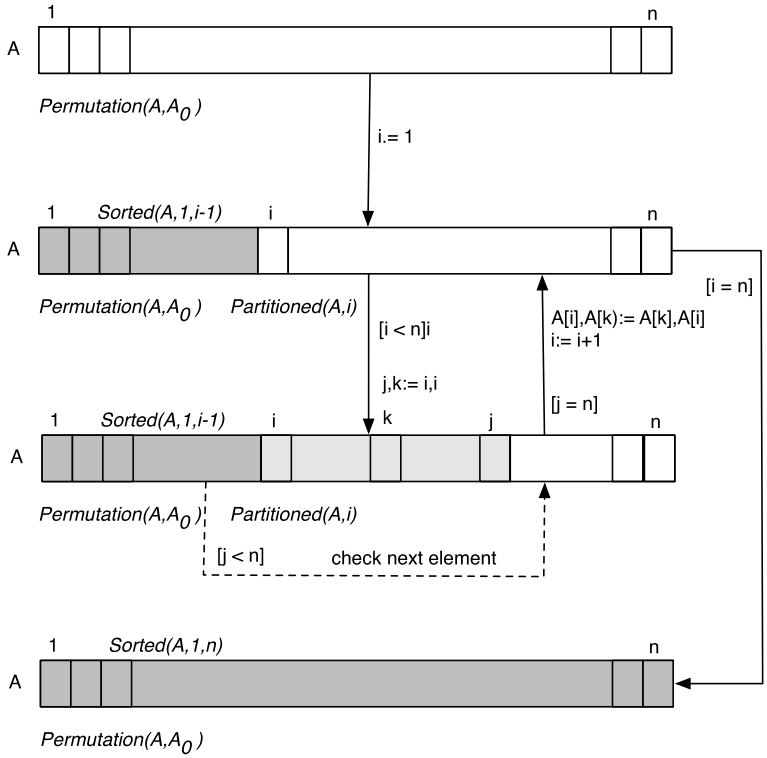

Fig. 5. Sorting program with two invariants.

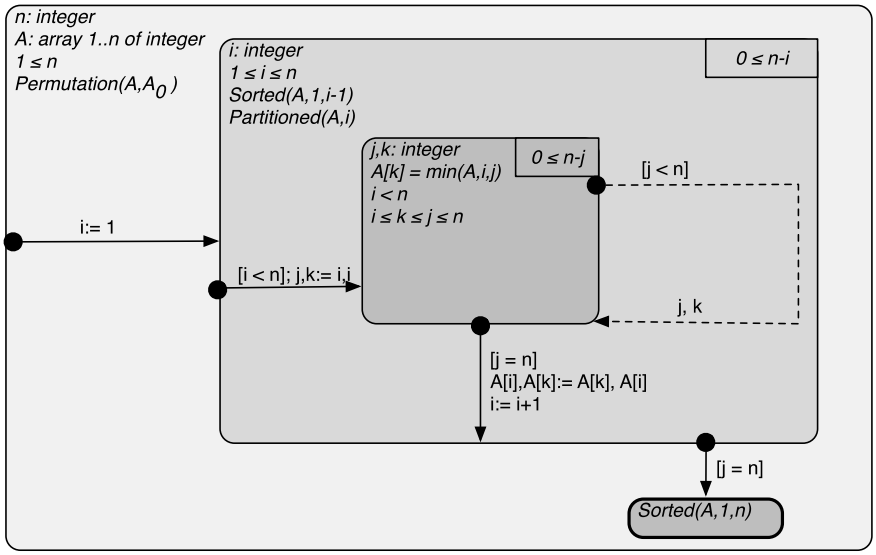

Fig. 6. Invariant diagram with the newly inserted situation.

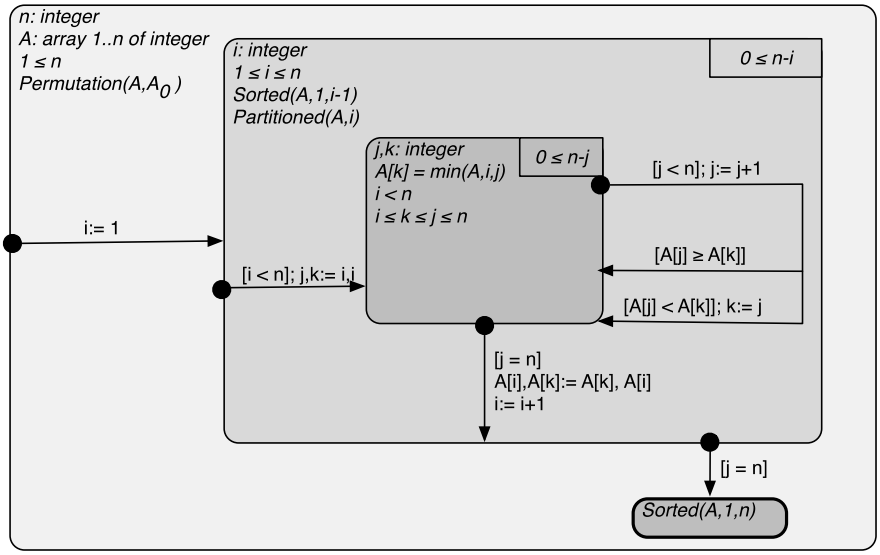

Fig. 7. Complete invariant diagram.

3. Design of Our Error Study

The study presented in this paper is part of a larger developmental research project (Richey and Klein, 2005; van den Akker, 1999), in which we empirically evaluate the use of three methods for teaching introductory CS courses. The empirical evaluations act as a feedback mechanism for making improvements to the methods or changes to the ways in which the methods are used in education.

3.1. *Data Collection*

A course covering IBP was introduced at the Department of Information Technologies at Åbo Akademi University in Turku, Finland in spring 2007. The data for the study reported on in this paper have been collected during 2007–2008, when 23 students completed the course.[1] Half of the students were on their first or second study year, whereas the other half had studied for three years or more. Most students were CS or software engineering majors.

The course includes 34 hours in class and has been given in an interactive lab lecture format. Approximately a third of the time has been used for hands-on training. During these exercise sessions, students solve assignments which are later handed in for grading and feedback. At the end of the course, all participants take a final exam. The exercise sessions and the exam include different types of assignments:

[1] In this context, completing the course stands for "handing in at least 50 % of all assignments and participating in the final exam".

- "Look at this invariant diagram and explain what it accomplishes" (*reading*).
- "Modify this invariant diagram so that the program does X instead of Y. Prove that the resulting program is correct" (*reading and modification*).
- "Find the corresponding imperative program for this invariant based one" (*reading*).
- "Construct an invariant based program that does X. Prove that the program is correct" (*construction*).

In this study, we focus on evaluating students' difficulties when constructing programs, and will thus only consider the construction part of the final type of assignment. All in all, we have analyzed solutions to six assignments for 23 students. Two of the solutions were solved at the beginning of the course (Exercise set 1), two towards the end (Exercise set 2) and the final two on the exam (Exam). Nine solutions were missing, giving us a total of 129 analyzed solutions (out of 138).

In the following, we briefly describe and exemplify the types of assignments included in the exercise sets and the exam. The assignments were of increasing difficulty level, i.e., those included in the first exercise set were the easiest, while the ones in the second set and on the exam were more difficult.

Exercise set 1: In these assignments, students were provided with readily defined predicates and there was no need to use quantified expressions. The following is an example of this type of exercise.

> *Construct a correct invariant based program*[2] *that checks if an integer ($n > 0$, given) is odd and returns the result as a boolean value (True if the integer is odd, False otherwise). You are only allowed to use addition. To simplify your work you can define the predicate $isOdd(x) = x \bmod 2 \neq 0$ to describe what it means that an integer is odd.*

Exercise set 2: In these assignments, students needed to define predicates themselves and also use quantified expressions, for instance as in the following:

> *Construct a program that substitutes the number one* (1) *for all odd numbers in an array. The even numbers should be kept unchanged. Use the mod operator ($x \bmod n$) to decide if a number is odd or even.*

Exam: In these assignments, students needed to define predicates and use quantified expressions.

> *Construct a program that calculates the sum of every m-th integer between 0 and n. Assume that n and m are given integers ($n \geqslant 0$, $m > 0$), i.e., if $m = 3$ and $n = 10$, the following integers should be summed up:* 0 1 **2** 3 4 **5** 6 7 **8** 9 10 *($sum = 15$).*

One of these assignments also included nested loops:

> *A polynomial is an expression of the format $a_0 + a_1x^1 + a_2x^2 + ... + a_{n-1}x^{n-1} + a_nx^n$. Construct a program that calculates the value of the polynomial given a non-negative integer n, the value of x and an array a containing the coefficients a_0, a_1, ..., a_n ($a[0] = a_0$, $a[1] = a_1$, ..., $a[n] = a_n$).*

[2] In the following assignment specifications the word "program" refers to a correct invariant based program.

If we had the precondition $n = 3 \wedge x = 4 \wedge a = [2, 3, 0, 5]$, the results of the calculation should be 334 ($2+3\cdot4^1+0\cdot4^2+5\cdot4^3$). Note! You do not have access to the exponential function in the calculations, so you will need to come up with another way to calculate the powers (e.g., in a nested loop).

3.2. *Method*

When analyzing student solutions, we are dealing with rich data. In order to be able to interpret such data, it first needs to be reduced. In this study, a content-analytical approach was chosen for this purpose.

The basic idea of content analysis is to take textual material and analyze, reduce and summarize it using emergent themes. These themes can then be quantified, and as such, content analysis is suitable for transforming rich data into a form which can be statistically handled and analyzed (Cohen *et al.*, 2007).

The content analysis was done in two phases. First, a subset of the student programs were analyzed one at a time by comparing these to the problem specification. Each solution could contain no, one or several error types. During this initial analysis, all errors were listed, resulting in 14 different errors types. In order to further organize and reduce the data, detailed types were combined into higher level error categories. This resulted in eight (8) categories.

Based on these categories, a coding scheme was created. This coding scheme was then used in the second phase to analyze all assignments, resulting in an overview of the errors each student made in the different assignments.

The results of this second round analysis indicated a need for making a more thorough analysis of the problems related to the invariant. Therefore, all solutions, in which an invariant related error had been found, were revisited. A process similar to the one described above was initiated to find the different types of errors made by the students when attempting to find and express the invariant. This resulted in a new scheme containing nine (9) error types for invariants.

Whereas questions looking to describe a phenomenon are best answered using a qualitative approach, quantitative methods are better at addressing more factual questions (Cohen *et al.*, 2007). Hence, when the initial (quite qualitative) analysis had been completed, a quantitative approach was taken in order to present and statistically analyze the data. We also wanted to find out whether there was any difference between the exam performance of novices (students on their first or second study year) and students having studied for a longer period of time. The *Kolmogorov–Smirnov* test showed that the exam data were not normally distributed, and hence the non-parametric *Mann–Whitney U* test was used in the analysis.

4. Results

4.1. *General Error Types*

In this subsection, we address the first two research questions: ”What kind of errors do novices make when using IBP?” and ”Do these change as the course progresses, and in that case how?”

The analysis revealed eight main error types, which are presented and exemplified in the following.

- **Updates**: Missing or redundant update. The student may, for instance, have forgotten to update the variable containing the result when finishing execution and moving to the final situation. Redundant updates do not make the program erroneous, but were still considered an error as they are unnecessary.
- **Guards**: Missing guard or incorrect bounds.
- **IBP notation:** Issues with the IBP syntax. The student may, for instance, have left out some brackets surrounding guards, set boundaries or a transition arrow. Another version of this error was the use of a *Java*-like syntax for expressing transition actions, i.e., using = as the assignment operator instead of :=.
- **Logical notation:** The student had problems expressing situations correctly using logical notation. Note that this error type does not include quantifier related errors found in the invariant (these are included in the "invariant" error).
- **Invariant:** The analysis revealed the invariant related error type to be a multi-faceted one; hence a more thorough analysis of these errors was conducted. The results from this analysis are discussed in Section 4.2 below.
- **Postcondition:** Erroneous or imprecise. The student may have used an incorrect postcondition or expressed it carelessly.
- **Terminating function:** Missing or erroneous.
- **Algorithm:** The student may have constructed a program solving another problem than the one intended or taken shortcuts in the algorithm (e.g., not storing the result of a calculation in a variable, but instead assuming it was kept in memory).

The frequency of error types in the two exercise sets and on the exam is illustrated in Fig. 8. As the assignments were different, it is naturally not possible to make an absolute comparison of the error frequencies in them. The error frequencies do show the trend of how the errors made by the students developed as the course progressed and the assignments became more difficult. Evidently, the frequency of all error types decreased from the beginning to the end.

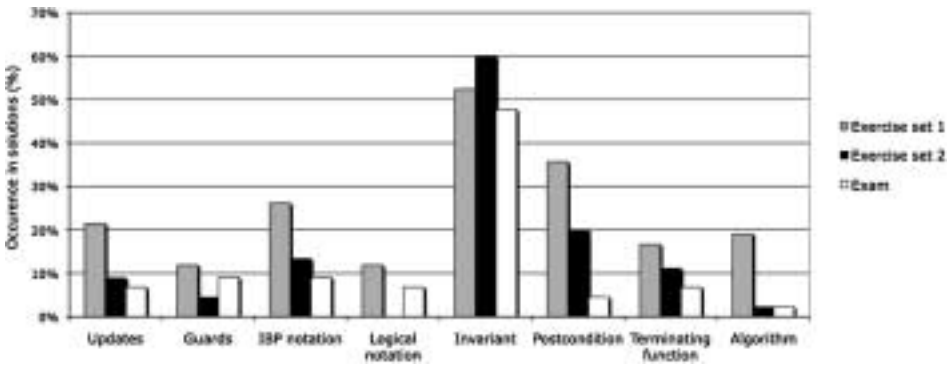

Fig. 8. Occurrence of general error types in the two exercise sets and on the exam.

4.2. *Invariant Related Error Types*

As the diagram in Fig. 8 shows, invariant related errors were most common throughout the course, and as mentioned above, this called for a deeper analysis in order to investigate where the main difficulties lie.

The results from this analysis revealed that the invariant related errors were indeed quite multifaceted, as nine error types were found.

- **Quantifier syntax:** Malformed quantified expression.
- **Quantifier bounds:** Incorrect bounds in a quantified expression.
- **Variable bounds:** Missing, incorrect ("off-by-one") or incomplete (e.g., lacking upper bound) bounds for a declared variable.
- **Variable declaration:** Use of an undeclared variable.
- **Logical notation:** Imprecise logical notation not related to quantifiers.
- **Incomplete:** An essential part of the invariant is missing.
- **Missing relationship:** The invariant includes all essential information, but lacks the relationship between a variable and a describing predicate. For instance, stating only $sorted(A, 0, k)$ instead of $isSorted = sorted(A, 0, k)$.
- **Strong:** A part of the invariant is too strong, for instance, stating that a variable is a constant (e.g., $isSorted = False$) instead of expressing it in terms of a predicate.
- **Erroneous:** Where the student had written a program solving the wrong problem, the invariant naturally also was incorrect. A variation of this error was demonstrated in solutions where the student apparently had not been able to come up with a sensible invariant.

As in the previous subsection, we illustrate the occurrence of these error types for the exercise sets and the exam separately using a diagram (Fig. 9).

These nine error types can further be classified as *severe* or *less severe*. For instance, one could argue that using an incorrect syntax in a quantified expression is not a severe error, as long as it is possible for the human reader to interpret what the student has meant and this underlying meaning is sound. Using a 100% correct syntax when using the quantifier is not necessary to accomplish this. Similarly, one could claim that variable

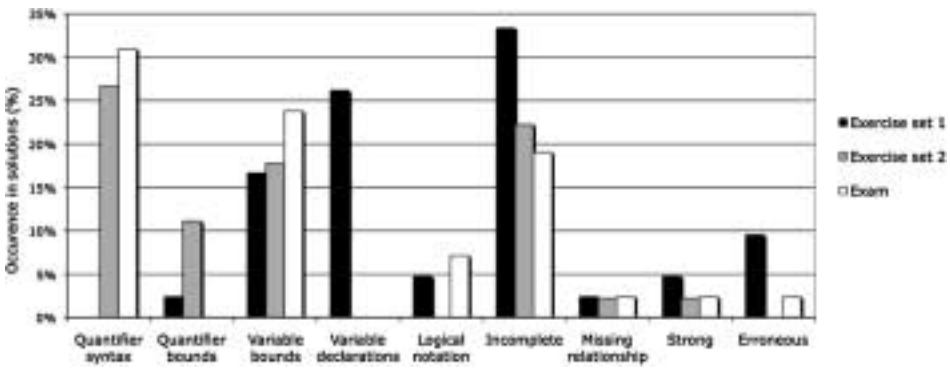

Fig. 9. Occurrence of invariant related error types in the two exercise sets and on the exam.

declarations are easily overlooked when writing programs by hand. This type of errors would not necessarily get caught when writing formal proofs by hand.

Incomplete, too strong or erroneous invariants on the other hand can be classified as severe, as these are directly related to the student's ability to find a suitable invariant. Likewise, bound errors indicate that the student has not done a good job at verifying the program, because such misses would easily have been found during the verification process. Based on this distinction, we get the following classification:

- **Severe errors:** related to the invariant (incomplete, strong, erroneous, quantifier bounds, variable bounds)
- **Less severe errors:** notational (quantifier syntax, logical notation, missing relationship) or careless (missing variable declarations)

As is shown in the diagram, the frequency of most severe errors decreased as the course progressed. The only exception is for errors related to variable bounds, which showed an increasing trend.[3] A similar development can be seen for the less severe "notational" errors, where the increase in errors related to quantifier syntax is especially eye-catching. No careless errors (missing variable declarations) were found after the first exercise set.

Of the 23 students, 12 demonstrated a decrease in invariant related errors when comparing their solutions in the first exercise set to those written on the exam. For seven (7) students, the number of invariant related errors increased, whereas it remained unchanged for the remaining four (4). Unfortunately, one of the assignments in Exercise set 2 contained a subtlety resulting in many students' invariants being incomplete. In order to get a sufficiently strong invariant, an unchanged property that was not easy to "see" needed to be stated. If these errors had not been taken into account, the percentage of incomplete invariants for the second exercise set would have been 9% instead of 22%. On the exam, all occurrences of incomplete invariants except one were found in an assignment involving nested loops; apparently, either invariant tends to become sloppily expressed in such programs.

Finally, we also wanted to examine how the proportion of completely correct invariants would change if only considering the severe errors. If looking at all errors (both the severe and the less severe ones), totally correct invariants were found in 31% of student programs in the first exercise set, with a slight increase in the second set and on the exam. If, however, only considering the severe errors, correct invariants were found in 45%, 53% and 60% for the exercise sets and the exam respectively.

4.3. *Exam Performance*

Our final research question aimed at investigating whether "older" students performed any differently in the course compared to the novices (students on their first or second study year).

[3] Errors related to quantifier bounds did increase from exercise set 1 to exercise set 2, but did not occur at all in the exam at the end of the course.

The maximum score on the exam was 30, out of which 15 points were required to pass the exam. The average score was 21.2 (std dev = 4.82), while one student failed (13 points). The Mann–Whitney U test indicated no difference in exam performance ($U = 50$, $Z = -0.992$, $p < 0.05$). Similarly, the content analysis showed no difference in error frequencies or error types between the two groups.

5. Discussion

The results indicated a decreasing trend for all general error types comparing the situation at the beginning of the course to that on the exam. Although the invariant related errors seemed to be a problem throughout the course, the more detailed analysis of these resulted in some interesting findings. When dividing the invariant related errors into two groups (severe and less severe), we found that the number of most severe errors decreased as the course progressed. The decrease in invariant related errors implies that students became more proficient at finding the invariant, even if the problems to solve became more difficult. This is an encouraging result, as one of our initial suspicions when introducing the approach in education was that the largest difficulties would be related to identifying the invariant.

With this general trend for the severe errors, the increase in errors related to the variable bounds seems somewhat counterintuitive. Why would students become better at more difficult things (finding the invariant) while at the same time start doing worse on aspects that are quite easily checked? Given the way in which the programs were written, one could argue that the bound errors should be seen as careless errors. After all, there was no interpreter or compiler checking if a variable had been declared or if it had been given the correct bounds. Under pressure, these are "smaller details" that may be easily overlooked. This is especially true for the exam, where the students did not have the time to do proper proofs.

Looking at how the number of solutions with correct invariants would change if only considering the severe errors, it became clear that the less severe errors made up a substantial part of the errors. Although these are not as serious as the other errors, their occurrence accentuates a need to further stress the importance of going through and proving each transition separately, if not by writing a formal proof, at least informally by checking that all properties hold. Without explicit checks, formal or informal, the invariant approach is no more "safe" than the traditional "trial-and-error" approach to program development.

In addition, the results indicated that many students are weak at formalizing informal statements using logic. To be able to, for instance, describe situations precisely and correctly, students need to be able to move confidently between informal (e.g., situations illustrated in the figures) and formal representations (the corresponding logical expressions). Apparently, predicate logic and quantifiers are particularly difficult to students. Similar indications can be found in the literature (Selden and Selden, 1995). This finding also serves as a partial explanation to why students find proofs difficult (Back *et al.*,

2007). How could a student construct a correct proof if he or she does not know how to express and manipulate the verification conditions formally and precisely? Consequently, it seems crucial that CS students were given more training in translating informal statements into symbolic form.

A tool could help in addressing both the issue of careless errors and that of logical notation and syntax. *SOCOS*[4] (Back *et al.*, 2006) is a graphical programming environment developed within our research group at Åbo Akademi University for the construction and verification of invariant based programs. It analyzes invariant diagrams semantically, and generates correctness conditions which are sent to external proof tools. Using *SOCOS*, trivial verification conditions can thus be proved or simplified automatically. In addition, it also compiles invariant diagrams to executable Python code.

A beta version of *SOCOS* was used in the course, and based on the initial teaching experiences and student feedback, the tool is currently being further developed to better suit the needs and skill levels of novice CS students. Nevertheless, we still believe that it is essential to introduce the approach using pen and paper only. Learning to build invariant based programs and constructing the corresponding proofs by hand is important in order for students to become familiar with the approach. The hands-on experience also makes explicit the link between mathematics and programming and shows that proving programs manually is a tedious and time demanding process; knowing the effort that goes into these activities, students can appreciate tool support to a larger extent later on.

The results also indicated that the IBP notation does not pose a problem to students, as only few such errors were found. Most of the notational errors found were due to students using a *Java* like syntax when expressing transitions and invariants. Given that the students already "knew" how to program, they already had an "ingrained" syntax. Thus, it is understandable that such students may face initial difficulties in abandoning their old approach (Denman, *et al.*, 1994). Choosing to use a programming language like notation may also be closely related to the problems with using logical notation discussed above; the programming language syntax may be the only formal notation they feel confident in using.

Finally, the findings presented in the previous section indicate that IBP is just as suitable for novices as for students who have studied for a longer period of time.

6. Concluding Remarks

In this paper, we have investigated the difficulties that students face when learning IBP. Although inventing the invariant was not found to be trivial, the main difficulty seems to be related to a lack of skills in formalizing expressions and interpreting logical notation. Problems related to constructing program correctness proofs also appear, at least to some extent, explainable by the lack of these very same skills. In order to successfully use the invariant based approach, more attention thus needs to be put on appropriate training aimed at developing these skills – as early as possible.

[4] `http://mde.abo.fi/confluence/display/SOCOS`.

Apart from the problems with logical notation, we have not found any indication that IBP would be "too advanced" for CS students during their first or second study year. Although the invariant can be tricky to come up with, our experience shows that this is not beyond the capability levels of novices. Taken together with the positive attitudes among students towards the approach presented in (Back *et al.*, 2007), IBP seems to be a worthwhile, alternative way to introduce a more formal approach to program development early in CS education.

References

Almstrum, V.L., Dean, C.N., Goelman, D., Hilburn, T.B., Smith, J. (2001). Support for teaching formal methods. *SIGCSE Bull.*, 33(2), 71–88.

Arnow, D. (1994). Teaching programming to liberal arts students: using loop invariants. *SIGCSE Bull.*, 26(1), 141–144.

Astrachan, O. (1991). Pictures as invariants. In: *Proc. of the 22nd SIGCSE Symposium*. ACM Press, New York, USA, pp. 112–118.

Back, R.-J. (1978). Program construction by situation analysis. *Research Report*, 6. Computing Centre, University of Helsinki, Helsinki, Finland.

Back, R.-J. (1983). Invariant based programs and their correctness. In: Biermann, W., Guiho, G., Kodratoff, Y. (Eds.), *Automatic Program Construction Techniques*, No. 223–242. MacMillan Publishing Company.

Back, R.-J. (2005). Invariant based programming revisited. *Tech. Rep.*, 661. TUCS – Turku Centre for Computer Science, Turku, Finland.

Back, R.-J. (2006). Invariant based programming. In: Donatelli, S., Thiagarajan, P.S. (Eds.), *Petri Nets and Other Models of Concurrency – ICATPN*. pp. 1–18.

Back, R.-J. (2009). Invariant based programming: basic approach and teaching experiences. *Form. Asp. Comput.*, 21(3), 227–244.

Back, R.-J., Eriksson, J., Mannila, L. (2007). Teaching the construction of correct programs using invariant based programming. In: *Proc. of the 3rd South-East European Workshop on Formal Methods*. Thessaloniki, Greece.

Back, R.-J., Eriksson, J., Myreen, M. (2006). Verifying invariant based programs in the socos environment. In: *BCS-FACS Workshop on Teaching Formal Methods: Practice and Learning Experience*. London, UK.

Blackwell, A.F., Whitley, K.N., Good, J., Petre, M. (2001). Cognitive factors in programming with diagrams. *Artificial Intelligence Review*, 15, 95–114.

Cohen, L., Manion, L., Morrison, K. (2007). *Research Methods in Education*, 6th Edition. Routledge, New York.

Denman, R., Naumann, D.A., Potter, W., Richter, G. (1994). Derivation of programs for freshmen. In: *Proc. of the 25th SIGCSE Symposium*. ACM Press, New York, USA, pp. 116–120.

Denning, P.J. (1989). A debate on teaching computing science. *Commun. ACM*, 32(12), 1397–1414.

Dijkstra, E.W. (1968). A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3), 174–186.

Dijkstra, E.W. (1976). *A Discipline of Programming*. Prentice-Hall.

Dijkstra, E.W. (1989). On the cruelty of really teaching computer science. *Communications of the ACM*, 32(12), 1398–1404.

Evans, D., Peck, M. (2006). Inculcating invariants in introductory courses. In: *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*. ACM Press, New York, USA, pp. 673–678.

Floyd, R. (1967). Assigning meanings to programs. In: *Symposium on Applied Mathematics*, Vol. 19. American Mathematical Society, pp. 19–32.

Ginat, D. (1995). Loop invariants and mathematical games. *SIGCSE Bulletin*, 27(1), 263–267.

Gries, D. (1981). *The Science of Programming*. Springer-Verlag.

Hoare, C. A.R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10), 576–580.

Joint Task Force on Computing Curricula (2001). *Computing Curricula, Computer Science*.

McGettrick, A., Boyle, R., Ibbett, R., Loyd, J., Lovegrove, G., Mander, K. (2004). Grand challenges in computing education. *Tech. rep.*, BCS – The British Computer Society.

McMaster, K., Anderson, N., Rague, B. (2007). Discrete math with programming: better together. In: *Proc. of the 38th SIGCSE Symposium*. ACM Press, pp. 100–104.

Naur, P. (1966). Proof of Algorithms by General Snapshots. *BIT Numerical Mathematics*, 6(4), 310–316.

Reed, J.N., Sinclair, J.E. (2004). Motivating study of formal methods in the classroom. In: *TFM 2004*. Springer-Verlag Berlin Heidelberg, pp. 32–46.

Reynolds, J.C. (1978). Programming with transition diagrams. In: Gries, D. (Ed.), *Programming Methodology*. Springer Verlag, Berlin.

Richey, R.C., Klein, J.D. (2005). Developmental research methods: Creating knowledge from instructional design and development practice. *Journal of Computing in Higher Education*, 16(2), 23–38.

Roy, G.G. (2006). Designing and explaining programs with a literate pseudocode. *J. Educ. Resour. Comput.*, 6(1), 1.

Roychoudhury, A. (2006). Introducing model checking to undergraduates. In: *Formal Methods Education Workshop*. pp. 9–15.

Selden, J., Selden, A. (1995). Unpacking the logic of mathematical statements. *Educational Studies in Mathematics*, 29(2), 123–151.

Tam, W.C. (1992). Teaching loop invariants to beginners by examples. In: *Proc. of the 23rd SIGCSE Symposium*. ACM Press, New York, USA, pp. 92–96.

van den Akker, J. (1999). *Design Approaches and Tools in Education and Training*. Kluwer Academic Publishers, Ch. Chapter I: Principles and Methods of Development Research.

van Emden, M.H. (1979). Programming with verification conditions. *IEEE Transactions on Software Engineering*, SE-5(2), 148–159.

L. Mannila received her PhD in computer science in 2009 from the Department of Information Technologies at Åbo Akademi University, Finland, where she now works as a researcher. Her main research interests are related to computer science education in general, and teaching programming and mathematics in particular.

Invariantais grįstas programavimo mokymas – mokiniams iškilusių sunkumų analizė

Linda MANNILA

Šiame straipsnyje analizuojamos pradedančiųjų studentų klaidos aiškinant invariantais grįstas programas. Pateikiant bendruosius klaidų tipus, atsižvelgiama į tai, dėl ko studentams kyla sunkumų išreiškiant invariantus. Rezultatai rodo, kad įvadinis kursas, naudojant invariantu grįstą metodą, yra tinkamas ir pradiniuose universitetų informatikos kursuose (kai studentai turi dar mažai programavimo žinių). Nors invarianto kūrimas pasirodė esanti sunki užduotis, tačiau pagrindiniai sunkumai su kuriais susidūrė pradedantieji, buvo susiję su įgūdžių stoka. Pradedantieji programos kūrimui taikė taisyklingą konstravimo būdą, naudodami loginius žymenis ir kvantorius, tačiau jiems stygo įgūdžių užrašant intuityvius ir neformalius sakinius į simbolinę formą.