

Teaching the Construction of Correct Programs Using Invariant Based Programming

Ralph-Johan Back, Johannes Eriksson, and Linda Mannila

Åbo Akademi University, Dept. of Information Technologies
Turku Centre for Computer Science
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
{backrj, joheriks, linda.mannila}@abo.fi

Abstract. In most computer science curricula, formal reasoning about program correctness is taught separately from practical programming, and is thus by most students considered a purely theoretical activity. It has been a challenge to convince students of the practical applicability of formal methods. We present here an effort to apply *Invariant Based Programming* (IBP), a visual and practical program construction and verification methodology, in an introductory formal methods course as part of a pilot study at Åbo Akademi University. The course introduces a minimum of notational overhead, and allows the student to reason about correctness using mathematical concepts with which they are already familiar (such as set theory). We have used a programming environment with theorem prover support (*SOCOS*) to increase student confidence in the correctness of the program components that they construct. We evaluate the course using a mixed method approach, and provide data which show that IBP is well suited for teaching introductory formal methods.

1 Introduction

In 1989, Edsger Dijkstra called for giving formal methods a higher profile in the computer science (CS) curriculum [18]. His proposal was the starting shot for an extensive debate on CS education and the role of formal methods in it. Some scientists agreed with Dijkstra's suggestion, whereas others disagreed [16]. Two years later, David Gries followed with basically the same message, stating that undergraduates should learn formal methods as a fundamental topic [24]. Ever since, CS academics have debated the importance of encouraging formal practices in CS education.

In this paper, we present a practical invariant based approach to introducing correctness in undergraduate CS courses. The approach is highlighted by a diagrammatic notation and emphasizes formal reasoning. Introducing correctness early in the CS curriculum and the particular approach we have used naturally raise some basic questions:

- How do students experience learning formal methods using this approach?
- How applicable is the use of tool support in the course?
- What difficulties do students encounter when learning formal methods using this approach?

The contribution of this paper lies in addressing these questions as well as in describing the invariant based approach and presenting a model for how it can be used in education. We begin by discussing the role of formal methods in education in section 2, after which section 3 describes the invariant based approach. In section 4, we present the educational setting. The study is presented in section 5, and the results are put forward in section 6. After discussing the findings in section 7, we conclude the paper with some final observations and suggestions for future work in section 8.

Although this study takes place in the context of CS education, we believe that formal methods play an important role for software engineers as well. In our opinion, the mathematical foundations of programming and knowledge about how mathematics can be used to improve reliability and robustness are essential for anyone designing and creating software, regardless of whether they have a degree in CS or software engineering.

2 Formal Methods in Education

Many attempts to introduce program correctness to novice CS students have been made (e.g. [1, 15, 26, 33, 37, 38]), but convincing students of the value of formal methods is a challenge. Formal techniques are commonly perceived as difficult and requiring mathematical sophistication. Moreover, '[t]he computing education community has adopted a curriculum strategy of dividing curricula elements into areas of "theory" and "practice". This causes both faculty and students to view the theory of computing as separate and distinct from the practice of computing.' [1, p. 79] As an effect of this separation students get only little exposure to correctness concepts.

When formal verification is taught as an activity independent from the programming process [27], the students get the impression that the formal approach is merely applicable in theoretical courses. Students are more likely to be motivated by gaining skills that they know are relevant, bring immediate benefits and are valued in industry. These preferences are also used by CS faculty as arguments against teaching formal methods [31]. If such teachers do teach something related to the topic, they will most likely not be enthusiastic or show a true interest in what they are teaching. And a "I don't really believe in this, I just have to teach it" mentality hardly goes far in having a positive impact on students' attitudes to or experiences of the topic at hand.

The nature of software construction may also reduce the experienced need for formal methods. It is completely possible to break design rules when constructing software and still end up with a working program, and it has become more or less the norm in industry to release buggy software. When well-known companies can get away with not writing correct programs, it is not easy to convince novice CS students that they need to do it.

As a result of the general lack of interest in formal methods, it is common that students do not apply what they have learnt in the theoretical courses when doing actual programming. Instead, novices learning to program go about it in a manner resembling a "trial and error" activity, resorting to "endless debugging" with the approach being "try it and see what happens" [12, p. 63]. Although testing and debugging certainly have their place when learning to program, it is essential that CS students learn that

these approaches can never prove that a program is correct, and that other methods are available for that purpose. In the following, we outline one such approach.

3 Invariant Based Programming

Invariant based programming (IBP) is an approach to constructing correct programs, where not only pre- and postconditions, but also loop invariants are written before the actual code. IBP is not new — it was studied already in the 1970s by one of the authors [4, 5] and similar ideas were proposed by for instance Reynolds [29] and van Emden [35]. In 2004, Back [6] revisited the topic and has since then worked on developing IBP into a more practical hands-on method.

In IBP, a program is constructed and verified at the same time. The notion of an invariant is generalized to a *situation*. Each situation is a collection of constraints and describes the set of states that satisfy these constraints. Thus, a loop invariant is a situation, as well as a precondition or a postcondition. An invariant based program may have many different 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 [13, 30], 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.” [30, p.3] To our knowledge, all of 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 all accomplished without sacrificing either clarity or expressiveness of the diagrams.

3.1 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 start the process by drawing figures illustrating the basic data structures involved and how they will change during execution of the algorithm. Drawing the figures 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 (invariants) of the program.

The first figure (Fig. 1) illustrates the specification (the pre- and postcondition), which helps us identify the initial and final situations. As situations are considered sets of states, the final situation is a subset of the initial situation where an additional

constraint, $Sorted(A, 1, n)$, is satisfied. We use a Venn-like diagram, a *nested invariant diagram*, to represent the program and the strengthening of situations. Our first diagram is shown in Fig. 2. Due to the nesting, all constraints in an outer set implicitly also hold in all of its subsets and need therefore not be repeated (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 and the variables that may be changed in the computation.

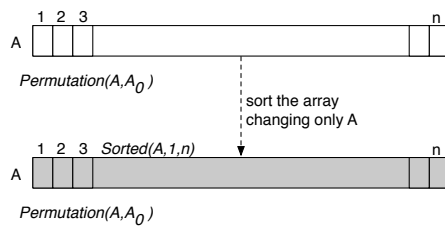


Fig. 1. Visualization of the specification

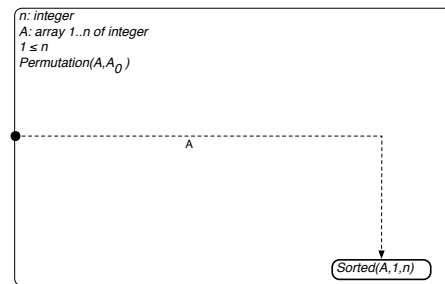


Fig. 2. Corresponding invariant diagram illustrating the initial and final situations

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). 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.

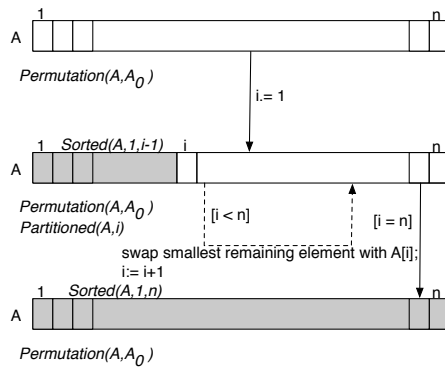


Fig. 3. Sorting program with one loop

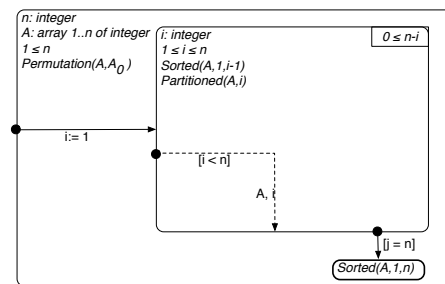


Fig. 4. Invariant diagram with the intermediate situation inserted

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. We have to check that each transition preserves the situations as follows: assume that we initiate execution in the source situation of a transition and that all the constraints hold for the starting state. Also assume that we reach some target situation after executing the statements for the transition (there may be more than one possible target situation). Then all the constraints of the target situation must hold for the final state. We say that a program is *consistent* if each transition preserves all situations. Consistency is checked for each new transition that we add to the diagram, i.e. we make sure that the newly added transition preserves all situations.

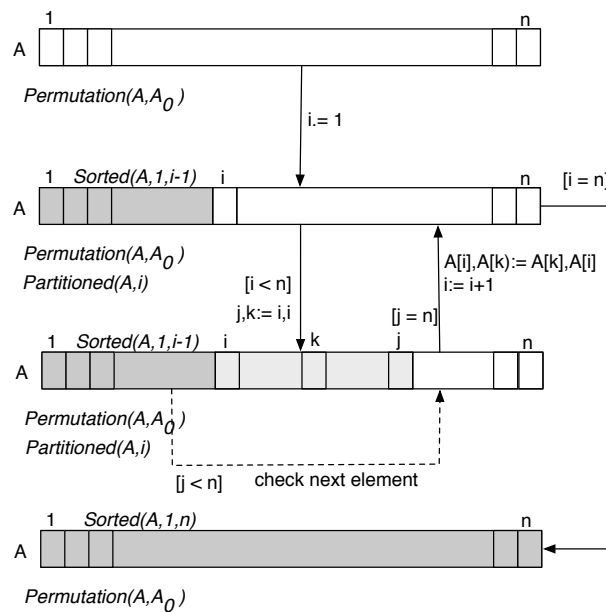


Fig. 5. Complete algorithm at work

We still need one more loop to find the smallest element in the remainder of the array. Again, we use figures as a tool to help us get an idea of how the algorithm works (Fig. 5). The corresponding invariant diagram (Fig. 6) represents our final program.

When all situations and transitions have been added to the diagram, we still need to check that no infinite execution loops exist, i.e. that the program *terminates*. We introduce a termination function (one for each loop), usually an integer function that is bounded from below and whose value is decreased before re-entering the situation. Moreover, the termination functions must be chosen so that no inner loop modifies the

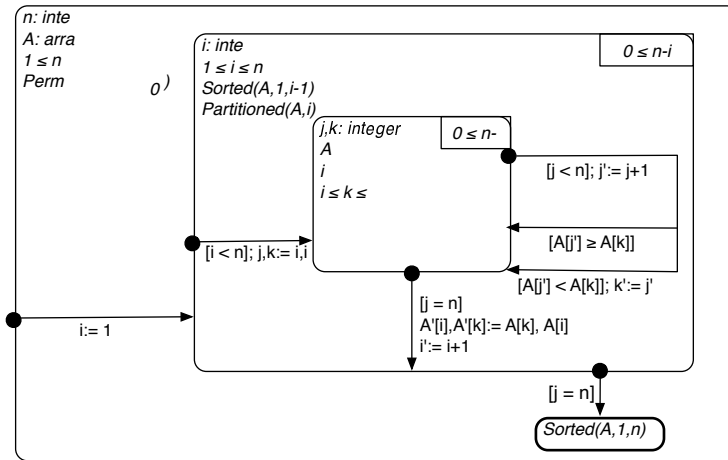


Fig. 6. Final invariant diagram

value of the termination function of an outer loop. The termination functions are written in the right upper hand corner of the respective invariant (Fig. 6).

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 in the diagram (except for the final ones) the available out-going transitions cover all possibilities.

An invariant based program is correct if it satisfies the three criteria above, i.e. it 1) is consistent, 2) terminates and 3) is live. For a more in-depth presentation of IBP as a method, see [6–8].

3.2 Tool Support for IBP

Invariant based programs can be constructed using only pen and paper, and in many cases this is the best way for initially drafting a program. However, even small programs generate a large number of verification conditions, many of which are rather trivial and can be automatically proved or greatly simplified by state of the art theorem provers. Additionally, the (considerable) risk of human error in manual proofs and specifications can be mitigated with proper tool support. Finally, we want to be able to execute the diagrammatic representation directly, without first having to hand translate it into some existing programming language.

SOCOS [9] is a graphical programming environment for the construction and verification of invariant based programs (Fig. 7). It analyzes invariant diagrams semantically, and generates correctness conditions which are sent to external proof tools (currently Simplify [17] and PVS [32] are supported). SOCOS also compiles invariant diagrams to executable Python code.

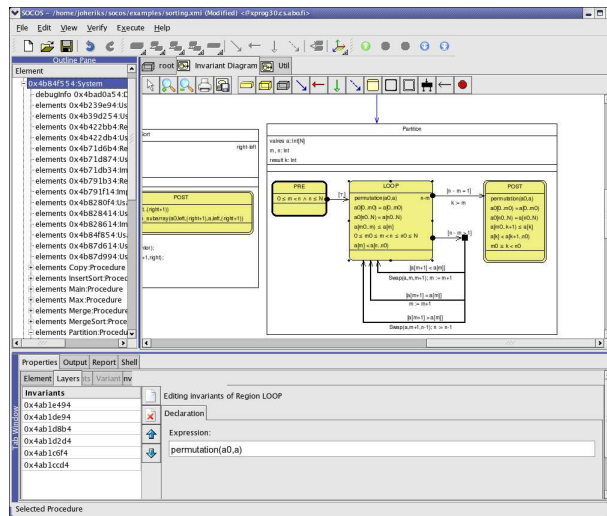


Fig. 7. The SOCOS IBP Environment.

4 IBP in Education

4.1 Motivation

The invariant property and the benefits from using it were presented in quite a natural and easily understandable way already in the original articles by Floyd [21] and Naur [28] on proving the correctness of computer programs. Introducing invariants early in the CS curriculum has been studied previously [2, 3, 20, 23], and the main message in all of these studies is that program correctness and loop invariants can be introduced at an early stage of CS education provided that the way in which it is done is adapted to the level of the students.

Starting in 2004, the development of IBP has been intertwined with informal experiments on teaching the method to see how it could be made more applicable in education. We have organized and observed 14 sessions with, in most cases, two CS students or academics having no prior experience of IBP. Each session started with an introduction to the approach, after which the participants were to solve a given problem using IBP on the blackboard. In spring 2005, a course on IBP was given to CS PhD students. These experiments have provided us with valuable feedback on the approach (positive experiences, difficulties, places for improvements etc) and two years later, the approach was deemed to be mature enough to be introduced at undergraduate level for the first time.

4.2 Undergraduate IBP Course

In spring 2007, an elective course covering the basics of IBP was developed and given to CS students at Åbo Akademi University (Turku, Finland). Our main motivation was

to address the common issues discussed in section 2 aiming at 1) changing the image of formal methods as being difficult (requiring highly advanced mathematics), uninteresting and of no use in practice and 2) showing that formal reasoning about program correctness can in fact be done in a practical manner with only fundamental logic skills. The goal of the course was for students to develop their programming and algorithmic thinking skills at the same time as learning about program correctness and formal reasoning. Another main design criteria was to make the course interesting and accessible so that it would inspire and motivate students to learn about correctness concepts.

The course is part of a project at our department aiming at designing a three course “course bundle” that would give all students a solid foundation in both the theory and practice of programming already during their first study year. The bundle includes, in the following order, a course covering “traditional” practical programming using a “simple” language (in our case, Python), a course on logic, and one that covers the mathematical foundations of programming (the IBP course). Together with the introductory course in mathematics (which introduces set theory that is needed for understanding the diagrammatic notation), the two former courses provide the students with all the background knowledge they need in order to successfully complete the IBP course.

The course on logic introduces *structured derivations*, which is a calculational proof format developed by Ralph-Johan Back and Joakim von Wright [10, 11]. They have extended the derivational style approach to proofs as presented by Dijkstra [19] and van Gasteren [36] by adding nested derivations (subderivations), allowing inferences to be presented at different levels of detail. The approach provides simple yet precise rules for how to write mathematical derivations and proofs that are easy to read as each step in the proof is clearly motivated. The goal of the course on logic is for students to learn 1) a clear format for writing well structured proofs that they know how to apply in practice and 2) basic propositional and first-order logic. Structured derivations are well suited for constructing proofs for invariant based programs, and using the same format in the IBP course was a natural choice.

Course Syllabus The course was given in an interactive format, emphasizing student activity throughout the course. All in all, the course included 17 sessions (90 minutes each) out of which 11 were used for lectures, and six for practical exercise. The main part of the course was taught without tool support; the SOCOS environment was only used in the four final sessions.

During each practice session, the students were to solve a set of IBP-related tasks, such as constructing a program, proving a certain transition or checking the correctness of a given program. Three of the assignments were reviewed collectively in class, whereas three were handed in and checked by the teacher who then gave detailed individual feedback for all tasks.

Integrating SOCOS in the Course Although the main part of the course was given without tool support, we felt a need to include SOCOS as the burden of organizing proofs quickly becomes noticeable even for relatively simple programs. Also, CS students are accustomed to using specialized software (e.g. compilers, interpreters, editors) in course work, and may regard programs and proofs constructed with pen and paper as

mere academic exercises. Actually being able execute the program may give the student some additional sense of accomplishment and thus act as a motivating factor.

Incorporating SOCOS in the course also made it possible for us to evaluate it in the context of teaching introductory formal methods, as well as to identify potential issues. The students were not expected to be familiar with PVS (mechanical theorem proving requires a separate course), so only the automatic prover was used in the course. In situations where the prover failed, students were required to complement the solution with a manual proof. The goal was to reduce the busy-work of proving simple, repetitive conditions, so that we were able to give more complex programs as exercises.

Examination The course examination consisted of active class participation, passed assignments and a final exam. The exam included programming problems similar to the ones in the assignments as well as questions that tested the students' understanding of invariant based programs. The students did not have access to the SOCOS environment on the exam.

5 The Study

Methodology The aforementioned studies on loop invariants in education ([2, 3, 20, 23]) include no evaluation of the approaches presented. Our goal was not only to present a new approach but also to evaluate its use and applicability in practice. We conducted a descriptive case study aiming at addressing the research questions presented in section 1, at the same time gaining insight into whether, and in that case how, the course and the method should be improved.

The study follows the principles of action research [14]. In action research, practitioners in a field improve practice by doing or changing something and reflecting on the results. The improvement can be in three areas: "improving a practice; improving the understanding of a practice [...] and improving the situation in which the practice takes place" [14, p.106]. The main purpose is to collect data and experience that help in gaining a better understanding of the practice.

Settings The undergraduate course was elective but still attracted 16 active participants (students that handed in at least one assignment). Nearly half were first or second year students with no background in formal methods. One of the students was absent for over half of the course due to medical treatment. Ten students participated in the SOCOS part of the course.

Data Collection Data were collected using pre- and post course questionnaires, observations, hand-in assignments and a final exam. Moreover, eight students were selected for semi-structured interviews one month after the exam. In this paper we will focus the analysis on the post course questionnaire, the assignments, the exam and the interviews. This mixed method approach with triangulation [25] was used to arrive at a multifaceted picture of the students' opinions and attitudes about the course in general

as well as the applicability of SOCOS. The use of different research instruments also increases the trustworthiness of the results, as it allows the researcher to look at the same phenomenon from several perspectives and thus arrive at a more complete account.

The post course questionnaire included both multiple choice questions and open ended ones asking the students about their opinions about the course in general as well as about IBP and SOCOS. In the multiple choice questions, Likert-type scales were used. Solutions to homework assignments were sent directly to the teacher by e-mail. Grading for the SOCOS assignments was based on the correctness of the solution (amount of verification conditions proved) and how precisely the pre- and postconditions were expressed.

The results reported on in this paper are based on the analysis of 12 post course questionnaires, 8 interviews and 13 sets of assignments and exams.

6 Results

6.1 Questionnaire Data

The answers to each open-ended question were first read and categorized as either positive or negative. In cases where the answer included both positive and negative aspects, the answer was divided into two parts accordingly. Next, all answers were reread and classified according to common themes representing the overall views of the twelve students (S1 - S12). The categories found with regard to what the students had experienced as 1) beneficial and useful, and 2) difficult in the course are listed below. Each category is exemplified with excerpts from the answers. The citations have been freely translated from Swedish by one of the authors.

Experienced Benefits

1. Introduction to program correctness and formal verification

Knowledge about proofs and correctness will hopefully lead to better programs (S2)

To learn a method for verifying programs formally (S7)

A good introduction to formal verification and how tools can be used in that context (S9)

Helps remove errors in the algorithm that could lead to bugs (S7)

2. A practical method for introducing program correctness

IBP seems to be a more practical verification approach than other methods I have seen (S4)

IBP summarizes the proof conditions in a good way (S4)

IBP is intuitive (S8)

3. Introduction to a more abstract view of programming

The course is about program design. You get a specification and design a correct program based on that (S3)

Learning to think about how a program works in general, without resorting to a given programming language (S3)

- Learned to think about a program as states and transitions instead of merely as transitions as is usually the case (S10)*
4. **More tangible overview of a program's structure**
Learning to draw a program makes it easier to see its structure (S12)
Makes it easy to keep track of the various parts [pre- and postconditions, invariants] of a program (S4)
 5. **The course arrangements**
Good teaching material, methods and lectures (S9)
The assignments helped me learn (S11)
All topics were thoroughly covered (S5)
 6. **New and useful contents**
I learned something new (S8)
The things I learned in the course will be useful in the future, especially in further studies (S9)

Experienced Difficulties

1. **Syntax and notation**
It is difficult to formulate one's programs according to the standard (S8)
Since I have programmed previously, e.g. the Java way of expressing things is quite ingrained (S3)
Formulating the conditions in a way that makes it easier to prove the program (S4)
2. **Proofs**
Proving programs by hand is very work intense (S4)
Proving complex programs (S1)
Proving programs 'honestly', i.e. to realize that one has made a mistake and correct it instead of trying to merely come up with explanations (S9)
The formal proof conditions should have been introduced earlier in the course (S1)
3. **Finding the appropriate conditions**
Finding the correct postcondition is most difficult. The difficulty of finding the invariant depends on how difficult it is to find the postcondition (S6)
Finding the invariant in complex programs (S7)

The quantitative data gathered in the questionnaire supported these qualitative findings. For instance, the course was found useful, interesting, somewhat fun and of medium difficulty level. On average, the data suggested that students found IBP rather easy to learn and useful in practice. The difficulties in constructing proofs and finding the invariant for more complex programs were also pointed out in the multiple choice questions. All students but one stated that they had enough mathematical skills for taking—and passing—the course.

Ten students attended the SOCOS part of the course and answered the related questionnaire. In line with our expectations the students preferred SOCOS over pen and paper, as the automation increases productivity. One student commented that it was “rather straightforward to understand the idea of the tool and how to apply it.” On the question whether IBP could be a practicable method in realistic software construction

the answers were scattered but still predominantly positive. Finally, the idea of supporting a formal method with a practical tool in the same course was very well received. The survey also indicates that unfamiliarity with the SOCOS syntax was the main cause of difficulty. Unfortunately, SOCOS lacks a good reference manual so teaching was mainly example-driven, and due to time constraints the students did not really achieve fluency in the SOCOS syntax.

The SOCOS related answers to the open ended questions supports the above mentioned findings, indicating that the tool was found useful, but somewhat difficult to use due to lack of time and an incomplete manual.

6.2 Assignments and Exam

The max score for the pen and paper assignments was 40, and the average was 25.5 (stdev 11.2). Seven students scored more than 30 points. Most errors were related to syntax (e.g. using Java like syntax) or the proofs not following the given format. The most common error related to program correctness was incomplete invariants, e.g. in the form of a missing lower or upper bound for a variable. A couple of students had problems with the algorithm, e.g. not updating variables to arrive at the result or writing a correct program that, however, was not the program asked for in the assignment. Some cases of using undeclared variables were also found. Merely one student seemed to have problems with the diagrammatic notation, writing the statements inside the situations instead of adjacent to the transition arrows. Only one “off by one” error was found.

Students who handed in solutions to the SOCOS assignments performed well. The highest scoring student achieved 20/20 points, while the average score was 14/20 points. Two students failed the exercise as a result of not handing in solutions—in one of these cases the student had been absent from the introductory sessions and subsequently lacked basic knowledge of the tool.

So far, 13 students have taken the exam,¹ out of which 11 have passed the course (four students with the highest grade). One of the two students who failed was the one who was absent for over half the course. As the goal of this paper is to describe how we have used IBP in education and report on the overall results, it does not contain any in-depth analysis of the students’ assignment and exam answers.

6.3 Interviews

Eight students (I1-I8) were interviewed by the lecturer one month after the exam. The interviewees were selected based on their course results in order for the interview data to be as representative as possible of the entire student group.

We chose not to conduct the interviews directly at the end of course as we wanted to have time to go through the other data first in order to construct interview questions based on the difficulties and other interesting points found in the other data. The process resulted in 12 broad questions that made it easy to ask follow-up questions when needed. The students were, for instance, asked about what they had learned and what

¹ At Åbo Akademi University, students have several alternative dates for taking the exam, and are thus not obliged to take part in the first ones that are arranged.

they had found difficult. They were also to describe the process of how they solve a problem using IBP and discuss how confident they are that the final program is correct.

The semi-structured interviews were transcribed and analyzed manually. All in all, the interview data strengthened the results found in the questionnaires. Students generally considered the IBP course a practical theory course quite different from other courses they had taken previously. The interactive nature of the class sessions was appreciated and the course considered suitable for first year university students. The interviewees were to describe how they typically solve a problem using IBP, and most of the descriptions followed the work flow presented in the course. Most students also said that they formally prove their programs after they have completed the diagram, whereas they rely on informal reasoning while while constructing the diagram.

Although the students found the invariant based approach per se useful, clear and simple, they did point out some difficulties, similar to those that were mentioned in the questionnaires. SOCOS was considered a helpful tool that, however, needs better user manuals and support. The students still pointed out the need for learning the fundamentals of IBP using only pen and paper.

7 Discussion

7.1 The Course and IBP in General

The feedback on both the course and IBP was in general quite positive. Students felt that IBP was easy to learn and the diagrammatic notation easy to understand. We were pleased to find that many students had recognized our original motivation for developing this course, that is, to present a practical method for introducing formal reasoning when constructing programs. Moreover, students also found that the approach made the general structure of the program more comprehensible.

We acknowledge that success in assignments and on exams is not a direct indicator of student learning, but we do feel that the programs written by the students on the exam and in the assignments show that they had understood the idea behind IBP and were able to construct and prove simple invariant based programs. These are the same students out of which many were not even able to explain basic concepts like “precondition” and “invariant” prior to the course.

The students clearly appreciated the diagrammatic notation of IBP. Studies [22, 34] estimate that between 75% and 83% of all students are visual learners, and because of their highly textual nature, the use of traditional programming languages or pseudocode is not necessarily the single best way for expressing algorithms to the majority of our first year students. As one of the IBP-students said in the post-course interview: *“Nice to see how a program really works. You saw it for yourself. And then you also understand the algorithm better when you see it in front of you. It’s more difficult to see what a program does directly from code.”*

Another benefit of using the invariant based approach is that it provides good support for finding bugs during the program construction (instead of after). This was also pointed out by the students in the interview. For example, we only found one single off by one error in the assignment solutions. Some of the other errors were related to the

use of undeclared variables. One could assume that writing out the type for a variable might easily be overlooked when writing the programs by hand as there is no compiler to check that the programs are correct (the SOCOS tool would naturally point out such errors). Thus, the students might simply have “forgotten” the declaration part when introducing a new variable in the program.

We had expected the students to find identifying the invariants the most difficult task, but this was not the case. Although some students mentioned the invariants, writing proofs by hand still seems to have been most problematic as they required much time and effort. The manual proofs do become rather long, for instance as all assumptions are written in each step of the derivation, but it is still interesting to see that students rate the difficulty of a given task based on how much time or effort it requires. Whether that is a reasonable indicator for the difficulty level of the task can be questioned. The format for the structured derivations has, however, been revised, and the modifications will automatically make the manual proofs less repetitive.

The questionnaire data pointed out the need for a clear standardized syntax. Students reported on sometimes finding it difficult to know how to express conditions and when to write their own definitions. This was to some extent expected, as the students had very little, if any, prior training in building their own domain theory. More practice and information about how to define predicates and reason about common data structures will therefore be included in the course from now on.

When designing the course, we thought it would be good to start by reasoning only informally about the correctness of the programs, before going further to formal mathematical proofs. This did, however, not turn out to be the case; instead, the students would have wanted the formal proof obligations to be introduced earlier. One explanation could be that students who are not mathematically mature do not know how to reason “informally” but first need to learn a formal approach with a set of rules.

7.2 Use of SOCOS

Incorporating computer aided verification in an introductory course was not an entirely uncontroversial choice. We were aware of the risk that students could apply the automatic prover as a magical tool and resort to a test-and-modify cycle (i.e., guess an invariant, guess transitions, run the automatic theorem prover, modify if the proof fails, ad nauseam). However, this risk was not manifested. Apparently the theoretical part of the course had given the students sufficient insight into the difficulty of the verification problem to realize that such testing would not converge into a correct program.

In line with Wing’s study [38], the students clearly appreciated the theory being complemented by a tool such as SOCOS. Surprisingly, most students learned to use the tool sufficiently well to solve the (non-trivial) exercises in the limited time available. Syntactical issues were the main cause of difficulty, largely due lack of documentation and occasional “rough edges” in the user interface. These usability issues are understandable and expected since the tool is experimental, and in our opinion do not indicate a fundamental flaw in the work flow.

Based on the open ended feedback, we have realized that there is a definite need for more extensive support in the form of documentation and manuals as well as personal guidance. Also, two weeks is far too little time to introduce and familiarize a verification

tool, which by nature contains considerable complexity. Both of these issues will be considered and rectified in sequel courses.

8 Conclusions and Future Work

Our initial experience from teaching IBP is positive, and we feel that our study has addressed the questions mentioned in section 1. The students appreciated learning about program correctness and seeing the programming activity from another perspective. IBP helps students further develop their programming skills at the same time as they learn how to reason formally about their programs. As opposed to the traditional separation between theoretical and practical courses that contrast formal and informal approaches, IBP integrates mathematics with the presentation of software design. Teaching IBP implies teaching all core topics in software design rather than a specific topic for which a dedicated formalism or tool exists. Moreover, the material is presented with minimal notational burden and builds upon students' previous knowledge (e.g. set theory). The use of SOCOS in the course was also appropriate; by automating trivial tasks, tool support enables the student to focus on difficult and interesting parts of the problem at hand. Furthermore, provided that the basics of invariant based programming are well understood, the exacting rigor of machine checked proofs considerably increases confidence in the correctness of the solution.

The study also shed light on some issues that need to be addressed. The syllabus will be modified according to the findings presented in this paper, for instance by including the formal proof conditions early. Moreover, a small and simple domain theory for array manipulation will be developed. In order to facilitate the construction of manual proofs, the preceding course on logic will be redesigned to better support the IBP course in providing the students with the skills needed to reason logically and write proofs using structured derivations. The format for the structured derivations has also been modified, and the changes will automatically shorten the proofs as all assumptions do not have to be written in every step of the proof. Additionally, the usability of SOCOS will be improved, the user manual will be further developed in order to become more comprehensive, and the proportion of the SOCOS part of the course will be considered and adjusted accordingly. The assignments and exam answers will also be thoroughly analyzed in order to find any indications of difficulties or problems that need to be considered.

Encouraged by the results from this pilot study, a course covering the basics of IBP will be offered annually to CS students at our department starting in the upcoming academic year (2007/2008), complementing the preceding courses on practical programming and logic. Our aim is thus not to substitute IBP for traditional programming courses. Quite on the contrary, we acknowledge the need for "traditional" programming, testing and debugging. We do not, however, see any reason for why these approaches could—or should—not coexist. We recognize that one single course is not enough for bridging the gap between theory and practice in the CS curriculum. If the students are to truly benefit from the skills acquired in the IBP course, these should be built upon in upcoming courses. A discussion with other CS faculty is thus essential in order to guarantee that there is a joint agenda on this point. We are also planning on developing

followup courses on this topic, for instance covering mechanical verification and application domain theory. By doing so, we aim at introducing a continuum in students' exposure to formal reasoning throughout their education.

References

1. Vicki L. Almstrum, C. Neville Dean, Don Goelman, Thomas B. Hilburn, and Jan Smith. Support for teaching formal methods. *SIGCSE Bull.*, 33(2):71–88, 2001.
2. David Arnow. Teaching programming to liberal arts students: using loop invariants. *SIGCSE Bull.*, 26(1):141–144, 1994.
3. Owen Astrachan. Pictures as invariants. In *Proc. of the 22nd SIGCSE symposium*, pages 112–118, New York, NY, USA, 1991. ACM Press.
4. Ralph-Johan Back. Program construction by situation analysis. Research Report 6, Computing Centre, University of Helsinki, Helsinki, Finland, 1978.
5. Ralph-Johan Back. Invariant based programs and their correctness. In W. Biermann, G Guiho, and Y Kodratoff, editors, *Automatic Program Construction Techniques*, number 223-242. MacMillan Publishing Company, 1983.
6. Ralph-Johan Back. Invariant based programming revisited. Technical Report 661, TUCS - Turku Centre for Computer Science, Turku, Finland, 2005.
7. Ralph-Johan Back. Invariant based programming. In *ICATPN*, pages 1–18, 2006.
8. Ralph-Johan Back. Invariant Based Programming: Basic Approach and Teaching Experiences, 2007. Accepted for publication in *Formal Aspects of Computing Science*.
9. Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. Verifying invariant based programs in the SOCOS environment. In *Teaching Formal Methods: Practice and Experience (BCS Electronic Workshops in Computing)*. BCS-FACS, Dec 2006.
10. Ralph-Johan Back, Jim Grundy, and Joakim von Wright. Structured calculation proof. *Formal Aspects of Computing*, 9:469–483, 1997.
11. Ralph-Johan Back and Joakim von Wright. A method for teaching rigorous mathematical reasoning. In *Proceedings of Int. Conference on Technology of Mathematics*, University of Plymouth, UK, Aug 1999.
12. Mordechai Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.
13. Alan F. Blackwell, Kirsten N. Whitley, Judith Good, and Marian Petre. Cognitive factors in programming with diagrams. *Artificial Intelligence Review*, (15):95–114, 2001.
14. Tony Clear. Critical enquiry in computer science education. In S. Fincher and M. Petre, editors, *Computer Science Education Research*, pages 101–125. Taylor and Francis Group, 2004.
15. Richard Denman, David A. Naumann, Walter Potter, and Gary Richter. Derivation of programs for freshmen. In *Proc. of the 25th SIGCSE symposium*, pages 116–120, New York, NY, USA, 1994. ACM Press.
16. Peter J. Denning. A debate on teaching computing science. *Commun. ACM*, 32(12):1397–1414, 1989.
17. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
18. Edsger W. Dijkstra. On the cruelty of really teaching computer science. *Communications of the ACM*, 32(12):1398–1404, Dec 1989.
19. Edsger W. Dijkstra and Carel S. Scholten. Predicate Calculus and Program Semantics. *Texts and Monographs in Computer Science*, pages 21–29, 1990.

20. David Evans and Michael Peck. Inculcating invariants in introductory courses. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 673–678, New York, NY, USA, 2006. ACM Press.
21. Robert Floyd. Assigning meanings to programs. In *Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.
22. Lynne Fowler, Maurice Allen, Jocelyn Armarego, and Judith Mackenzie. Learning Styles and CASE Tools in Software Engineering. February 2000.
23. David Ginat. Loop invariants and mathematical games. *SIGCSE Bulletin*, 27(1):263–267, 1995.
24. David Gries. Teaching calculation and discrimination: a more effective curriculum. *Commun. ACM*, 34(3):44–55, 1991.
25. Andrew P. Johnson. *A short guide to action research*. Allyn & Bacon, Boston, 3 edition, 2008.
26. Henry McLoughlin and Kevin Hely. Teaching formal programming to first year computer science students. In *Proc. of the SIGCSE symposium*, pages 155–159, 1996.
27. Kirby McMaster, Nicole Anderson, and Brian Rague. Discrete math with programming: better together. In *Proc. of the 38th SIGCSE symposium*, pages 100–104. ACM Press, 2007.
28. Peter Naur. Proof of Algorithms by General Snapshots. *BIT Numerical Mathematics*, 6(4):310–316, July 1966.
29. J. C. Reynolds. Programming with transition diagrams. In D. Gries, editor, *Programming Methodology*. Springer Verlag, Berlin, 1978.
30. Geoffrey G Roy. Designing and explaining programs with a literate pseudocode. *J. Educ. Resour. Comput.*, 6(1):1, 2006.
31. Abhik Roychoudhury. Introducing model checking to undergraduates. In *Formal Methods Education Workshop*, pages 9–15, 2006.
32. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 411–414, New Brunswick, NJ, USA, July/August 1996. Springer-Verlag.
33. Ann E. Kelley Sobel. Empirical results of a software engineering curriculum incorporating formal methods. In *Proc. of the 31st SIGCSE symposium*, pages 157–161, New York, NY, USA, 2000. ACM Press.
34. Lynda Thomas, Mark Ratcliffe, John Woodbury, and Emma Jarman. Learning styles and performance in the introductory programming sequence. In *Proc. of the 33rd SIGCSE symposium*, pages 33–37, New York, NY, USA, 2002. ACM Press.
35. M. H. van Emden. Programming with verification conditions. *IEEE Transactions on Software Engineering*, SE-5(2):148–159, 1979.
36. Antonetta J.M. van Gasteren. On the Shape of Mathematical Arguments. *Lecture Notes in Computer Science*, pages 90–120, 1990.
37. J. Stanley Warford. An experience teaching formal methods in discrete mathematics. *SIGCSE Bull.*, 27(3):60–64, 1995.
38. Jeannette M. Wing. Weaving formal methods into the undergraduate curriculum. In *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*, pages 2–7, May 2000.