Introducing Formal Methods via Program Derivation

Dipak L. Chaudhari
Department of Computer Science and Engg.
Indian Institute of Technology, Bombay
Mumbai, India, 400076
dipakc@cse.iitb.ac.in

Om Damani
Department of Computer Science and Engg.
Indian Institute of Technology, Bombay
Mumbai, India, 400076
damani@cse.iitb.ac.in

ABSTRACT

Existing attempts towards including formal methods in introductory programming courses focus on introducing program verification tools. When using the verification tools, there is no structured help available to the students in the actual task of implementing the program, except for the hints provided by the failed proof obligations. In contrast, in the correct-by-construction programming methodology, programs are incrementally derived from their specifications.

By restricting our attention to program derivation, we have identifed a small core of the formal method concepts that can easily be taught in the first two years of a computing curricula. Based on our learning from multiple years of paper and pencil based teaching, we have developed a programming assistant tool that addresses several of the issues faced by the students in the manual program derivation. The tool ensures that the most common students' error of performing incorrect proofs does not happen.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science EducationComputer science education; D.2.4 [**Software Engineering**]: Software/Program Verification-Correctness Proofs, Formal Methods, Programming by Contract

General Terms

Algorithms, Verification, Human Factors

Keywords

Correct by Construction; Calculational Style; Teaching Formal Methods

1. INTRODUCTION

In its final report [1], the ITiCSE 2000 Working Group on Formal Methods Education aspired *to see the concepts of formal methods integrated seamlessly into the computing curriculum.* Fifteen years later that aspiration still remains an aspiration. In our opinion, the major reason for this is the fact that the points of integration identified in the report, in Appendices C and E, come much later in the curriculum. By that time, the students are already used to the informal ways of developing programs and software and the old habits die hard. Ideally formal methods should be introduced as early as possible, particularly when students are just learning how to design programs [5].

Existing attempts in this direction focus on employing formal verification for teaching program correctness [13, 3, 7, 12]. The Implement-and-Verify program development methodology involves an implementation phase followed by a separate verification phase. Although the failed proof obligations provide some hint, there is no structured help available to the students in the actual task of implementing the programs. Students often rely on ad-hoc use cases and informal reasoning to guess the program constructs.

In contrast with this, in the *Calculational Style of Programming* (CSoP) [6, 10], programs are incrementally derived from their specifications. At every step in the derivation process, a partial program is transformed into a more refined form, by following certain transformation rules. The resulting programs are correct-by-construction since the correctness is implicit in the program transformations employed during the derivation. Since the students see the program transformation strategy that led to the introduction of a particular programming construct, they understand why a particular programming construct was introduced at a particular point in a program.

Based on the four offerings of the program derivation elective course to sophomores, we have identified a small core of the formal method concepts which is sufficient for teaching derivations of a large number of programs. We have also identified several difficulties faced by the students in using the method effectively (discussed in Section 4).

To address these difficulties, we have developed a tool called CAPS [4], for deriving sequential programs from their formal specifications. To the best of our knowledge, no comparable tool exists. As discussed before, existing tools [13, 3, 7] only ensure the correctness of the already implemented programs. Besides providing counter-examples, these tools provide limited help to the students in learning the program design techniques.

In CAPS, we have kept the derivation style and notation as close to the pen-and-paper style of derivation as possible. Our main emphasis has been on the usability in the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ITiCSE'15, July 6–8, 2015, Vilnius, Lithuania.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3440-2/15/07 ...$15.00.
http://dx.doi.org/10.1145/2729094.2742628.

class; in particular on being able to model and replay the ad-hoc interactions and iterations that usually occur during the manual program derivation by the students.

The organization of the rest of the paper is as follows. In Section 2, we discuss the core formal method concepts employed by us for teaching CSoP. In Section 3, we illustrate the methodology with a detailed example. In Section 4, we discuss our experiences from the years of teaching CSoP to sophomore students. In Section 5, the CAPS tool and the students experience with it is presented. Section 6 concludes the paper.

2. CORE IDEAS

The advantage of teaching formal methods via CSoP is that the students can quickly write programs to solve non-trivial problems after being exposed to a small set of concepts. After starting with the *sum of an array* and the *maximum element of an array*, we quickly move to the *binary search, fast exponentiation*, and the *maximum segment sum*. After that we cover various other optimal array segment and search problems, and other similar problems like *decomposing a number in a sum of two squares*. Then we move on to array rearrangement problems such as *array partitioning* and *sorting*. In one offering, we were able to cover even more advanced problems such as the *area of the largest square under a histogram*. We wish to emphasize that all of it can be done using a small set of formal method concepts. Besides propositional logic, and the concepts of assertions and loop invariants, we only need a formal concept of quantified expressions and the rules for manipulating them.

We employ the *Eindhoven* notation ($OP\, i : R.i : T.i$) [10] for representing quantified expressions. Outside the formal methods community, this notation is typically used only for quantified terms in arithmetic ($\sum$, $\prod$). Here OP (say, $\sum$ or MAX) is the quantified version of a symmetric, associative binary operator op (say, $+$ or max), i is a list of dummy/quantified variables, $R.i$ is the *Range* - a boolean expression restricting the possible values that the dummies can take, and $T.i$ is the *Term* - over which the underlying binary operator is repeatedly applied. For example, just the way $(\sum i : 0 \leq i < N \wedge A[i]\%2 = 0 : A[i] * A[i])$ represents the sum of the square of the even elements of the array A, $(MAXi : 0 \leq i < N \wedge A[i]\%2 = 1 : A[i] * A[i])$ represents the maximum of the square of the odd elements of the array A. Since all the quantified operators (including the logical operators $\forall$ and $\exists$) are represented using the same notation, we can use generalized calculational rules [10].

While a large number of quantified expression manipulation rules are known [11, 2, 10], we find that for our purpose only three rules suffice: Range Split, Empty Range and, One Point Rule. The Range Split rule is most commonly used to form an inductive hypothesis: to show that the loop body maintains the loop invariant. The Empty Range and the One Point rules are used to evaluate an expression when either zero or exactly one dummy satisfies the range condition. The entire expression is evaluated in an inductive fashion by applying the Range Split, and the Empty Range or the One Point rule. Due to the lack of space, we do not present a detailed discussion of these rules but only illustrate them with the help of an example.

Beside these rules, the only non-trivial concept from propositional logic that we use is that of Distributivity and its adaptation for the Quantifier Calculus. Just the way $*$ distributes over $+$: $x * (y + z) = x * y + x * z$, similarly $\wedge$ (logical *and*) and $\vee$ (logical *or*) distribute over each other: $(P \wedge (Q \vee R) = (P \wedge Q) \vee (Q \wedge R))$, and $+$ distributes over max: $x + (y\, max\, z) = (x + y) max (x + z)$. With this small set of core manipulation rules, we can teach derivation of a large number of problems.

Just the way ITiCSE Working Group on Formal Methods [1] viewed *formal methods as the "calculus" of software engineering*, we view rules for manipulation of Quantified Operators as the "calculus" of program derivation.

3. AN EXAMPLE DERIVATION

We now present a calculational derivation of the well-known Maximum Segment Sum problem. This derivation highlights the typical steps that are involved in a program derivation session. The natural language specification for this problem is:

Let $A[0..N)$ be an array of integers. Compute the maximum sum of the elements of all segments of A.

This problem is formally specified in a natural fashion as shown in Figure 1(a), where S is the required program with the desired postcondition R. To do inductive computation, we introduce a fresh variable n and rewrite postcondition R as $P_0 \wedge P_1 \wedge n = N$ where P_0 and P_1 are given in Figure 1(b).

We can take $P_0 \wedge P_1$ as the loop invariant and $\neg\,(n = N)$ as the loop condition. We observe that the assignment $r, n := 0, 0$ establishes the invariants P_0 and P_1 initially. Now, we arrive at the program shown in Figure 1(c).

We explore the inductive step $n := n+1$. Now, if we want the loop invariant P_0 to be true after the assignment, then we need the assertion $P_0(n := n+1)$ before the assignment, where $P_0(n := n + 1)$ represents the formula obtained by replacing n with $n+1$ in the body of P_0. That is, if we want $\{r = (MAXp, q : 0 \leq p \leq q \leq n : Sum.p.q)\}$ to be true after $n := n + 1$ is executed then $\{r = (MAXp, q : 0 \leq p \leq q \leq n + 1 : Sum.p.q)\}$ must be true before the assignment is executed. This is called *necessary* (or *weakest*) *precondition* $np.(n := n + 1).P_0$ w.r.t. the assignment.

Now we expect to modify r to some r' before incrementing n, where r' is a metavariable. A metavariable is not a program variable - it just represents an unknown expression. Then, $np.(r := r').(np.(n := n + 1).P_0)$ is the necessary precondition for the assignment $r := r'$. To calculate r', we assume P_0, P_1, and $n \neq N$ and simplify the necessary precondition as shown in Figure 1(d). In calculational style, every step in a calculation is associated with a hint justifying the step.

In step 15 in Figure 1(d), the quantified expression is not easily computable or expressible in terms of the existing program variables. This motivates the introduction of a new variable s and the loop invariant $P_2 : s = (MAXp : 0 \leq p \leq n : Sum.p.n)$. P_2 can be established initially by $s := 0$, since the summation over an empty range equals 0. We now arrive at the program shown in Figure 1(e).

To establish $P_2(n := n+1)$ as a precondition of the assignment to r, we introduce an unknown program fragment S_1. To develop S_1, we introduce the assignment $s := s'$. Similar to the calculation of r', we calculate the value of s' to be $(s + A[n])\, max\, 0$. In computing s', once again we use the Empty Range rule that the summation over an empty range returns 0. The final program is presented in Figure 1(f). This completes our derivation. Note the small number of concepts that were needed to derive an elegant solution to a

con N: int $\{N \geq 0\}$; **var** A: array $[0..N)$ of int;
var r: int;
S
$R : \{r = (MAXp, q : 0 \leq p \leq q \leq N : Sum.p.q)\}$
$Sum.p.q : (\sum i : p \leq i < q : A[i])$

(a)

$P_0 : \{r = (MAXp, q : 0 \leq p \leq q \leq n : Sum.p.q)\}$
$P_1 : \ 0 \leq n \leq N$
$P_2 : s = (MAXp : 0 \leq p \leq n : Sum.p.n)$

(b)

1		$np.(r := r').np.(n := n+1).(P_0)$
2	$\equiv$	{ definition of P_0 }
3		$np.(r := r').np.(n := n+1).(r = (MAXp, q : 0 \leq p \leq q \leq n : Sum.p.q))$
4	$\equiv$	{ definition of np }
5		$np.(r := r').(r = (MAXp, q : 0 \leq p \leq q \leq n+1 : Sum.p.q))$
6	$\equiv$	{ definition of np }
7		$r' = (MAXp, q : 0 \leq p \leq q \leq n+1 : Sum.p.q)$
8	$\equiv$	{ $q \leq n+1 \equiv q \leq n \vee q = n+1$ }
9		$r' = (MAXp, q : (0 \leq p \leq q \leq n) \vee (0 \leq p \leq q = n+1) : Sum.p.q)$
10	$\equiv$	{ Range Split }
11		$r' = \left(\begin{array}{l} (MAXp, q : 0 \leq p \leq q \leq n : Sum.p.q) \\ max\ (MAXp, q : 0 \leq p \leq q = n+1 : Sum.p.q) \end{array} \right)$
12	$\equiv$	{definition of P_0 }
13		$r' = r\ max\ (MAXp, q : 0 \leq p \leq q = n+1 : Sum.p.q)$
14	$\equiv$	{$q = n+1$ }
15		$r' = r\ max\ (MAXp : 0 \leq p \leq n+1 : Sum.p.(n+1))$
16	$\equiv$	{let us introduce P_2; assume $P_2(n := n+1)$}
17		$r' = r\ max\ s$

(d)

$r, n := 0, 0;$
$\{\ loop\,inv : P_0 \wedge P_1\ \}$
do $n \neq N \rightarrow$
$\quad S_0$
od

(c)

$r, n, s := 0, 0, 0;$
$\{\ loop\,inv :\ P_0 \wedge P_1 \wedge P_2\ \}$
do $n \neq N \rightarrow$
$\quad \{ P_2 \}\, S_1\, \{ P_2(n := n+1) \};$
$\quad r := r\ max\ s;$
$\quad \{ P_0(n := n+1) \}$
$\quad n := n+1;$
od

(e)

$r, n, s := 0, 0, 0;$
$\{\ loop\,inv :\ P_0 \wedge P_1 \wedge P_2\ \}$
do $n \neq N \rightarrow$
$\quad s := (s + A[n])\ max\ 0;$
$\quad \{ P_2(n := n+1) \}$
$\quad r := r\ max\ s;$
$\quad \{ P_0(n := n+1) \}$
$\quad n := n+1;$
od

(f)

Figure 1: Selected stages in the derivation of the *Maximum Segment Sum* problem.

non-trivial problem. We next discuss our experience teaching this methodology.

4. COURSE FEEDBACK AND TOOL SUPPORT FOR CSOP

The students' interest in the methodology is reflected in the course feedback where we received 87% score in the last offering of the course. Following two comments exemplify the students' excitement: "*A quite different approach to programming, very innovating and interesting too. Some really great insights.*" and "*We learned many good things. I never thought that program could be derived. The experience was enriching.*" Despite the mostly positive feedback, we also realized that students were facing a number of difficulties in manually (without using any tool support) deriving the programs:

Common Difficulties:

(CD0) Difficulty in understanding formal logic: Used to informal reasoning, students make several mistakes in understanding and applying inference rules.
(CD1) Not checking transformation applicability conditions: Many of the program transformation rules have prerequisites that need to be checked. For example, + distributes over quantified MAX only if the range is non-empty. Students often forget to check such conditions.
(CD2) Long derivations: Compared to the guess and test approach, the calculational derivations are longer even for simple programs. Students get restless if the derivation runs too long, leading to more errors.
(CD3) Mistakes made during guessing: Manual derivations often involve small jumps where the unknown program expressions are simple enough to be guessed easily. Students often inadvertently take big steps during guessing, resulting in incorrect program expressions. For example, for program S_1 in Figure 1(e), many students make a jump and guess the value of s' to be $s + A[n]$.
(CD4) Forgetting to add bounds to the introduced variables: It is a general guideline to add bounds for a newly introduce variable, such as the bounds for n in the maximum segment sum problem. Students often forget to add such bounds, and later in the derivation, when the bound constraints are needed, they have to backtrack and take the corrective actions.
(CD5) Forgetting to prove proof obligations: With their focus on unraveling the unknown program fragments, students many times forget to prove some of the proof obligations.

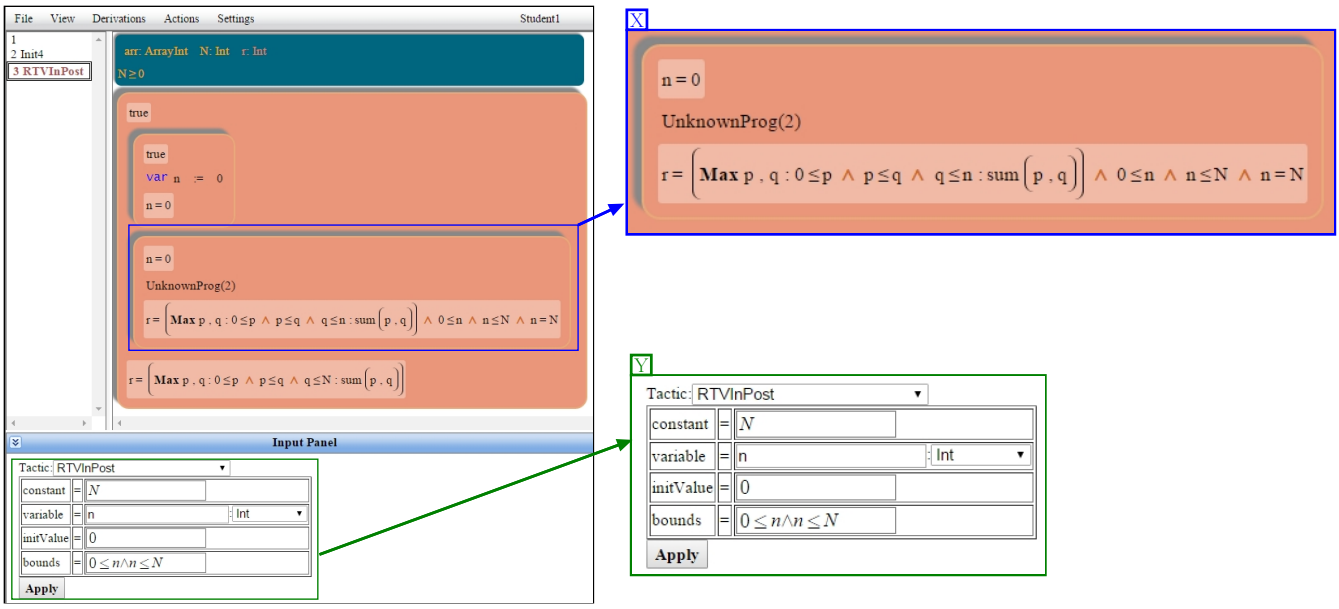

Figure 2: Graphical User Interface of the CAPS tool. The left panel is the *Tactics Panel*, the center panel is the *Content Panel* and the bottom panel is the *Input Panel*. Magnified area X shows an annotated unknown program along with its pre- and post- conditions whereas the magnified area Y highlights the form for entering the inputs of the "Replace term by variable"(*RTVInPost*) tactic.

(CD6) Problem with organizing derivation: The derivation process is not always linear; it involves multiple iterations involving failed derivation attempts. Students often fail organize the derivation in cases where they need to go back and make some corrective changes. Unorganized derivation often leads to some missing proofs of correctness.

Based on the errors experienced during the multiple course offerings, we decided to develop tool support for teaching this methodology. We next outline the required functionality for the desired tool support. First and foremost, we must ensure correctness of all the steps involved in the derivation. The manual derivation occasionally employs informal reasoning. For example, the Step 8 in the Figure 1(d) implicitly uses the rule that $\wedge$ distributes over $\vee$. To ensure correctness, we need to have a unified framework to manage both the program and the formula transformations. We must have a mechanism for dealing with the long derivations. In addition to automating the tasks involved, having the ability to organize the long proofs is vital. We also need to maintain history to provide backtracking and branching functionality. Finally, the user interface should allow seamless navigation across the derivation history.

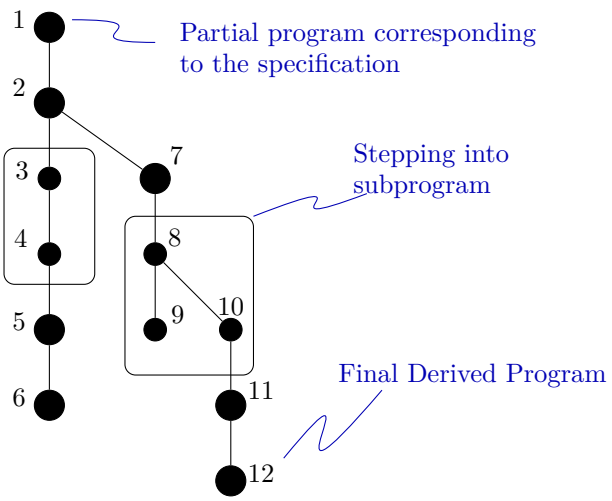

Figure 3: Schematic Derivation Tree.

5. CAPS TOOL

We now discuss the CAPS tool for the CSoP methodology. Its core components are implemented in the Scala language. The graphical user interface is implemented as a web application. The CAPS tool uses the Why3 [9] verification platform as an interface to various backend theorem provers. But the students need not concern themselves at all with how the theorem prover works. The implementation details of the tool have been published in [4] and here we concern ourselves only with the use of the tool in classroom teaching.

5.1 Program Derivation Methodology

Students incrementally transform a formal specification into a fully derived program by applying predefined transformation rules called *Derivation Tactics*. For example, two of the tactics that we employed in derivation in Figure 1 are *Replace constant by a variable*, and *Range Split*. To apply a tactic, one needs to select a tactic from a list and provide the required input parameters, and the tool automatically performs the corresponding formula manipulations. By forcing students to enter the required parameters, errors such as

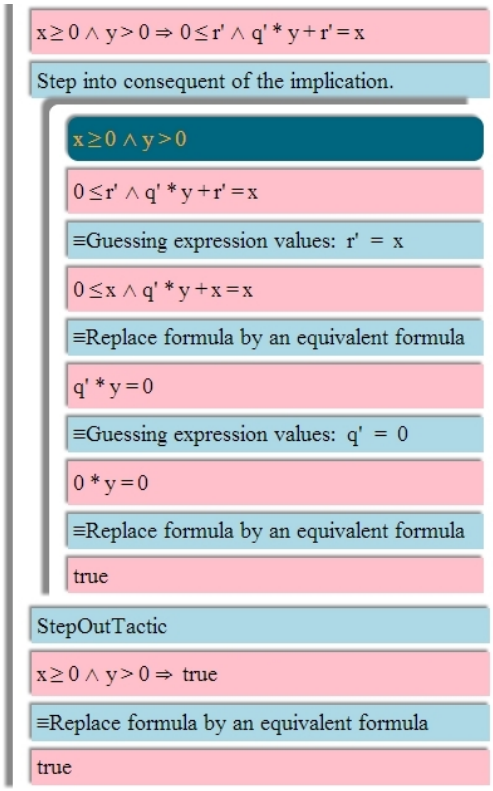

Figure 4: Calculation of initial assignment $(q, r := 0, x)$ to establish invariant $0 \leq r \wedge q * y + r = x$ while deriving *Integer Division* program(Set q, r to the quotient & remainder of the division of x by y).

CD4 are prevented. The tool ensures correctness after application of every tactic. The GUI of the tool is shown in Figure 2.

5.2 Derivation History and Backtracking

The CAPS tool maintains the entire derivation history in the form a derivation tree. The user can also branch off from any point in the derivation to explore different derivation strategies. This helps take care of the errors resulting from *CD4* and *CD6*.

Figure 3 shows a schematic representation of a derivation tree. Node 1 is the starting node representing the specification and node 12 represents the final derived program. Node 6 and node 9 are the nodes where the user faces some difficulties with the derivation and decides not to carry out the derivation further and prefers to backtrack and branch out. The backtracking mechanism makes it easier for the user to try out different alternatives with least amount of rework. The user interface also makes it easy to navigate across different solutions.

5.3 Focusing on Subcomponents

At every stage in the derivation process, there exists a correct-by-construction program containing multiple unknown subprograms. The user may want to focus her attention on the development of one of these subprograms. Hence it is desirable that all the context information relevant for the derivation of the subprogram is extracted and presented. Similarly, while transforming formulas, the user may want to focus on a subformula while ignoring the rest of the formula.

To focus on a subcomponent, the *StepIn* tactic is applied. Application of the tactic brings the context of the subcomponent under consideration in focus and hides the rest of the program. (The details of the context extraction process are described in [4].) After transforming the subcomponent to a desired form, users can apply the *StepOut* tactic to bring the focus back to the whole program. The *StepIn* tactic application can be nested any level deep. In Figure 3, the entry to and exit from the rectangles correspond to application of *StepIn* and *StepOut* tactics. Whenever user steps into a subcomponent, a new Frame is created to store the appropriate contextual information. The contextual information is then available for use during the transformations of the subcomponent.

Example: Figure 4 shows a calculation that uses the "Focusing on subformula" functionality. Contextual information (assumptions) for the inner frame are displayed at the top of the frame.

5.3.1 Automating Formula Transformations

In the manual calculations, all the steps are kept small enough to be manually verified by the user. This is the main reason why the program derivations are long even for simple problems, and formal methods are hated by several students. With a tool support, however, we can afford to take large steps, as long as the readability is maintained. In general, small steps are good for readability. However, there are situations where certain calculation is not important from the derivation point of view. We would like to automate such calculations. We employ a backend theorem prover to perform required proofs. This makes the program calculations flexible and reduces the derivation length. This helps with the observed errors *CD2* and *CD5*.

Calculations not involving any metavariables should be automated to the extent possible. For example, in Fig. 1, we skipped the proof of preservation of the invariant $P_1 : 0 \leq n \leq N$. As no metavariable is involved this proof obligation, it is uninteresting from the derivation viewpoint. Students resent doing such proofs. We, however, still need to discharge them to ensure correctness. The proof obligation for P_1 can be directly transformed to $true$ by applying a VerifiedTransformation tactic that uses the backend theorem prover. The introduction of this tactic takes care of the errors *CD1* and *CD3*.

In case a proof obligation is not automatically discharged by the theorem provers, we have to carry out the detailed step-by-step proof. A failed calculational proof often provides clues about how to proceed further with the derivation.

Another example is the calculations involved in verifying the applicability conditions for some tactics. Consider the "Empty Range" tactic for the summation that transforms the formula $(\sum i : false : T.i)$ to 0. If a student wants to apply this tactic directly to $(\sum i : R.i : T.i)$, she first needs to show that $R.i \equiv false$ (which may take several steps) and then apply the tactic. Other way is to directly apply the Empty Range Tactic to $(\sum i : R.i : T.i)$ and the tool ensures that the $R.i$ is unsatisfiable.

Automated formula transformations take care of the most of the common logic related errors (*CD0*).

5.4 Evaluation

The tool became ready to be used by the students only towards the end of the last offering of our program derivation course. It received very enthusiastic response from the students. We did an anonymous survey to get specific feedback about the tool. There were a total of fourteen responses. Ten students felt that the use of the tool increased their confidence in the correctness of the derived program, while three did not feel so, and one student was unsure. Same pattern was observed for the question whether the tool simplifies or complicates the task of the derivation. To the question of how would they like to derive the programs in future, five said using the tool alone, six said that they would like to use the tool along with paper and pencil, and three students commented that they would not use the tool. Eight out of the fourteen students also felt that the tool should have been introduced right from the beginning of the semester, while three suggested introduction around the middle of the semester, and three students felt that the tool should not be introduced at all but they did not write any comments. Due to the anonymity of the survey, we are unable to determine why three students did not like the tool at all.

Overall, we are quite happy with the use of tool in the course. The biggest advantage was that the students could not submit incorrect derivation. They could only submit either correct or partially correct answers; since programs were correct-by-construction at all stages (although they may have been incomplete). We could look at the derivation history of partial submissions and identify the problems because of which they were stuck at a particular point. Students were happy about the fact that they knew that their solution was correct before making the submission. Note that this adds a completely new dimension to the concept of automatic grading of assignments [8]. We plan to use the tool right from the beginning of the next offering to understand its shortcomings in detail.

One unexpected downside of the introduction of the tool was the increase in ad-hocism in some of the derivations. In the class, we teach various derivation heuristics and the associated proof obligations, and students are supposed to follow them in the manual derivation. However, with the tool trying to automatically discharge proof obligations, some students make wild guesses about the required program constructs, resulting in very inelegant programs. For example, rather than deriving the value of s' in Figure 1(f), many students introduce several *if* statements enumerating different cases involving positive and negative values of s and $A[n]$. In comparison, *max* operator in our derivation can be implemented using a single *if*. Note that these programs were inelegant compared to what is possible with the derivation methodology, and not compared to what is achieved in the standard guess and test methodology. Essentially, these students use the tool as a program verification system and not as a program derivation system. This in some sense buttress the argument we made in the introduction section as to why the program derivation and not the program verification should be used to introduce formal methods.

6. CONCLUSION

Instead of program verification, we have been using program derivation as the vehicle for introducing formal methods in the introductory programming classes. This has been done employing only a small core of formal method concepts. Based on our experience in teaching this method to several batches, we have identified a list of common errors made by the students while deriving the programs manually, and have developed a programming assistant to take care of these problems. The preliminary student response to the tool has been very positive. Based on the learnings from the first offering of the tool, we plan to further enhance the tool and deploy it right from the beginning of the next offering of the course.

Acknowledgements.

The work of the first author was supported by the Tata Consultancy Services (TCS) Research Fellowship and a grant from the Ministry of Human Resource Development, Government of India.

7. REFERENCES

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

[2] R. Backhouse. *Program construction: calculating implementations from specifications*. Wiley, 2003.

[3] G. Caso, D. Garbervetsky, and D. Gorín. Integrated program verification tools in education. *Software: Practice and Experience*, 2012.

[4] D. Chaudhari and O. Damani. Automated theorem prover assisted program calculations. In *Proc. of the 11th International Conference on Integrated Formal Methods, iFM*, 2014.

[5] A. J. Cowling. Stages in teaching formal methods. In *23rd IEEE Conference on Software Engineering Education and Training*, 2010.

[6] E. W. Dijkstra and W. H. Feijen. *A Method of Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[7] I. Dony and B. Le Charlier. A tool for helping teach a programming method. In *Proc. of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 2006.

[8] J. English and T. Rosenthal. Evaluating students' programs using automated assessment - a case study. In *Proc. of the Conference on Integrating Technology into Computer Science Education, ITiCSE*, 2009.

[9] J.-C. Filliâtre and A. Paskevich. Why3 – Where Programs Meet Provers. In *ESOP'13 22nd European Symposium on Programming*, volume 7792 of *LNCS*, Rome, Italie, Mar. 2013. Springer.

[10] A. Kaldewaij. *Programming: the derivation of algorithms*. Prentice-Hall, Inc., NJ, USA, 1990.

[11] D. G. Kourie and B. W. Watson. *Correctness-by-Construction Approach to Programming*. Springer, 2012.

[12] K.-K. Lau. A beginner's course on reasoning about imperative programs. In C. Dean and R. Boute, editors, *Teaching Formal Methods*, volume 3294 of *LNCS*. Springer Berlin Heidelberg, 2004.

[13] M. Sitaraman and B. Weide. Special session: "hands-on" tutorial: Teaching software correctness with resolve. In *SIGCSE 2014 - Proc. of the 45th ACM Technical Symposium on Computer Science Education*, 2014.