(In-)Formal Methods: The Lost Art
A Users' Manual

Carroll Morgan[1,2(✉)]

[1] School of Computer Science and Engineering,
University of New South Wales, Sydney, Australia
[2] Data 61 (formerly NICTA), Sydney, Australia
carroll.morgan@unsw.edu.au

Abstract. This article describes an experimental course in "(In-)Formal Methods", taught for three years at the University of New South Wales to fourth-year undergraduate Computer-Science students (http://www.cse.unsw.edu.au/~cs6721/). An adapted version was then taught (disguised as "Software Engineering") to second year undergraduate students (http://webapps.cse.unsw.edu.au/webcms2/course/index.php?cid=2332).

Fourth-year CS students at UNSW are typically very-good-to-excellent programmers. Second-year students are on their way to the same standard: but many of them have not yet realised how hard it will be actually to get there.

Either way, whether good or on the way to good, few of these students have *even heard* of static reasoning, assertions, invariants, variants, let alone have learned how to use them. . . None of the simple, yet profoundly important intellectual programming tools first identified and brought to prominence (more than 40 years ago) has become part of their programming toolkit.

Why did this happen? How can it be changed?

What will happen if we do change it?

Below we address some of those questions, using as examples actual material from the two related courses mentioned above; they were given in the years 2010–4. As an appendix, we present feedback from some of the students who took one course or the other.

At the same time, some suggestions are made about whether, when and how courses like this one could possibly be taught elsewhere.

1 Part I - Generalities

1.1 Background, Genesis and Aims

Here is an excerpt from *βeta*, the fortnightly magazine of UNSW's Computer Science and Engineering student society. It appeared the year after the first *Informal Methods* course:[1]

[1] In 2011: http://beta.csesoc.unsw.edu.au/2011/05/getting-max-right/.

Z. Liu and Z. Zhang (Eds.): SETSS 2014, LNCS 9506, pp. 1–79, 2016.
DOI: 10.1007/978-3-319-29628-9_1

Practical applications of formal methods
for Computer Scientists
Last year one of my lecturers posed a very simple question to a class containing about fifteen 3rd and 4th year students: "Write a function that, given an array `A` *containing* `n` *integers, computes (and returns) the maximum value contained in the array". Essentially, write the "max" function. Naturally, I thought at the time that this was obvious and simple. I proceeded to write...*

⋮ (actual code omitted; but it had some problems...) ⋮

By using this simple example, the lecturer made a beautiful point: formal methods can be practical!

You can use formal methods in two ways. You can study formal methods from a theoretical perspective. You can rigorously and thoroughly apply formal methods to algorithms or systems in an isolated, academic exercise. Nice results can be obtained from this, such as the work done on [mechanically proved correctness of critical software]. This process, however, is too long and time-consuming for most developers working in industry.

Alternately, you can learn the techniques of formal methods and apply them in practical ways. This means you don't have to prove every statement and every property to the n^{th} degree. Instead, by using simple reasoning during the coding process [...] you can help reduce the number of bugs introduced into a system. Additionally, it will make finding bugs easier by being able to eliminate from consideration parts of the codebase that you can quickly show are correct.

If you ever get an opportunity to learn the techniques of formal methods from a practical perspective, take it. The techniques will change how you write code, making you a more efficient and accurate developer.

What is remarkable about this (unsolicited) article is not so much its content, but that it was written at all. One is surprised that the author was surprised...

Of course formal methods can be practical. In fact they are basic, and essential to any kind of good programming style. As the writer points out, it's a question of degree: it does not have to be "to the n^{th}".

We knew that already... *didn't we?*

1.2 Who is This "We"? And Where Did Formal Methods Go?

Who We Are... and Who "They" Are. Imagine this situation, black-and-white simple in its significance: the "we" above is having a casual conversation with a fourth-year computer-science student who – it is already established – is a very good programmer, respected for that by both peers and teachers. We're discussing a small loop.

"Ah, I see. So what's the loop invariant?" I ask.

"What *is* a loop invariant?" is the reply.

"We" are talking to a "they"; and that we-they conversation, and others like it, is where the motivation for an "Informal Methods" course came from. It was

not that these students couldn't understand invariants etc. It was that they had never been given the chance.

We have to turn "them", those students, into "us".

Where Formal Methods Went. Formal Methods did not start in one place, with one person or one project. But in the early 1970's there were a few significant publications, from a small number of (already, or soon to be) prominent authors, that set the stage for a decade of progress in reasoning carefully about how programs should be constructed and how their correctness could be assured. Amongst Computer Science academics generally the names, at least, of those authors are well known: not a few of them are Turing Award winners. But outside of Formal Methods itself, which is a relatively small part of Computer Science as a whole, few academics can say what those famous authors are actually famous for. And even fewer can say what some of them are still doing, right now, as this article is being written.

Why aren't those authors' works now the backbone of elementary programming, not a part of every student's intellectual toolkit? Part of the reason is that although the ideas are simple, learning them is hard. Students' brains have to be ready, or at least "readied" by careful conditioning: the ideas cannot be pushed in by teachers. They have to be *pulled* in by the students themselves. Teachers operate best when answering questions that students have been tricked into asking. Even so, many excellent teachers simply are not interested in those ideas: they have their own goals to pursue, and not enough time even for that.

Another problem with Formal Methods is evangelism: there was, and continues to be, an urge to say to others "You have to do this; you must follow these rules... otherwise you are not really programming. You are just hacking, playing around." Formal Methods, like so many other movements, generated a spirit of epiphany, of "having seen the light" that encouraged its followers too much to try to spread the message to others, and too eagerly. And often that brought with it proposals for radical curriculum re-design that would "fix everything" by finally doing things right. Another "finally", again. And again.

In general those efforts, so enthusiastically undertaken, never made it out of the departmental-subcommittee meeting room — except to be gently mocked in the corridors, by our colleagues' shaking their heads with shrugged shoulders, wondering how we could be so naïve. And now, years later, those efforts are mostly gone altogether, forgotten.

So that is where Formal Methods went, and why it has not become a standard part of every first-year Computer-Science course. And "we" are the small group of Computer Scientists who are old enough to have had no way of avoiding Formal-Methods courses during those "decades of enthusiasm". Or we might be younger people who are intellectually (genetically) pre-disposed to seek, and achieve that kind of rigour: after all, in every generation there are always some.

But now we leave the matter of "us", and turn to the question of "they".

1.3 Operating by Stealth: Catching "Them" Early

In my own view, the ideal place for an informal-methods course is the second half of first year. A typical first-half of first year, i.e. the first computing course, is summarised

> Here's a programming language: learn its syntax. And here are some neat problems you never thought you could do; and here are some ideas about how you can program them on your very own.
> *Experience* for yourself the epic late-night struggle with the subtle bugs you have introduced into your own code; *savour* the feeling of triumph when (you believe that) your program finally works;*gloat*over the incredible intricacy of what you have created. (1)

This is true exhilaration, especially for 18-year-olds, who are finally free to do whatever they want (in so many ways), in this case with "grown-up" tools to which earlier they had no access: most first-years have been exposed to computers' effects for all of their conscious lives, and there is nothing like the thrill of discovering how all these mysterious and magical devices actually work and that, actually, you can do it yourself.

Become a Tenth-Level Cybermancer! Your younger siblings, non-CS friends and even parents will now have to cope with cryptic error messages and bizarre functionality that you *have created.*

Once we accept (or remember ☺) the above emotions, we realise that the key thing about formal-methods teaching is that for most students there is *no point* in our trying to introduce it until after the phase (1) above has happened and has been fully digested: if we try, we will just be swept out of the way.

A typical second-half first-year course is elementary data-structures; and indeed second-half first-year is a good place for it. But informal methods is more deserving of that place, and it is far more urgent. Survivors of the first half-year will now understand something they didn't have any inkling of before:

> Programming is easy, but programming *correctly* is very hard.

This moment is crucial, and must be seized.

They have not yet *mistakenly* learned that making a mess of your initial design, and then gradually cleaning it up, is normal, just what "real programmers" do — and that lost nights and weekends, sprinting with pizza and Coke, are simply how you play the game. And it is not yet too late to stop them from *ever* learning that mistaken view. This is the moment we must teach them that programming should not be heroic; rather it should be *smart*.

We want programmers like Odysseus, not Achilles.[2]

[2] See this equivalently as whether you'd like to have Sean Bean or Brad Pitt on your programming team.

So the "stealth" aspect of informal methods, in second-half first year, is that it should be called something like

- Programming 102, or
- Taking control of the programming process, or
- Practical approaches to writing programs that Really Work, or
- Getting your weekends back.[3]

Having just experienced the pain (and pleasure) of over-exuberant coding, and its consequences, they are now – just for this small moment of their lives – in a small mental window where their teen-aged energy can be channelled into forming the good habits that will help them for the rest of their professional careers. In short, they are vulnerable, at our mercy — and we should shamelessly take advantage of it.

By second year, it could be too late.

1.4 Making Contact

After six months' exposure to university-level programming, many first-year students are on a trajectory to a place that you don't want them to go: they think that their journey in Computer Science is going to be a series of more elaborate languages with steadily increasing functionality, a growing collection of "tricks of the trade" and more skill in dealing with the unavoidable quirks of the software tools we have. And it will be all that. But it should be more: how do we catch that energy, and deflect these would-be tradesmen onto a path towards proper engineering instead?

In my view, our first move has to be making contact: you must first "match their orbit", and then gradually push them in the right direction. It's like saving the Earth from an asteroid — first you land on it, and then a slow and steady rocket burn at right-angles does the job. The alternative, a once-off impulse from a huge space-trampoline, simply won't work: the asteroid will just punch through and continue in its original direction.

In Sect. 2, below, some suggestions are made for doing this, the gradual push. For second-years,[4] it's to develop in the first lectures a reasonably intricate (for them) program, on the blackboard in real-time, in the way they are by then used to: guesswork, hand-waving and even flowcharts. You indulge in all the bad habits they are by now developing for themselves, and in so doing establish your programming "street cred" — for now, you are one of them.[5]

[3] Only this last suggestion is meant as a joke: for undergraduates, that phrase is more likely to mean "breaking up with your boy/girlfriend".

[4] I say "second years" here because that is what I have actually been able to try. As should be clear from above, in my opinion this is better done in first year.

[5] Your aim in the end is, of course, that they should become one of you.

But this is where, secretly, you are matching their orbit. (For fourth years, paradoxically, you should use an even simpler program; but the principle is the same.[6])

In both cases, once you have caught their attention, have "landed on the asteroid", by going through with them, sharing together, all the stages mentioned in Sect. 1.3 above – the epic struggle, the subtle bug, the savour of triumph, the gloat over intricacy – you "start the burn" at right angles, and gradually push them in the direction of seeing how they could have done it better.

Your aim is to train them to gloat over simplicity.

2 The Very First Lecture: *Touchdown*

2.1 Begin by Establishing a Baseline

Much of this section applies to any course, on any subject, and experienced lecturers will have their own strategies already. Still, as suggested just above, it's especially important in this course to make a strong connection with the students and to maintain it: you are going to try to change the way they think; and there will be other courses and academics who, with no ill will, nevertheless will be working in the opposite direction, suggesting that this material is unimportant and is consuming resources that could be better used elsewhere. With that said...

First lectures usually begin with administrative details, and that is unavoidable. But it's not wasted, even though most people will forget all that you said. It's useful because:

- Although no-one will remember what you said, to protect yourself later you will have to have said it: "Assignments are compulsory", "Checking the website is your responsibility", "Copying is not allowed".
- It will give the students a chance to get used to your accent and mannerisms. To reach them, you first have to let them see who you are.

[6] A simpler program is better for more advanced students because they have developed, by then, an impatience with complexity introduced by anyone other than themselves. (First-years are still indiscriminatingly curious.) Furthermore, older students have begun to realise that their lecturers actually might have something to teach them. Remember Mark Twain:

> When I was a boy of fourteen, my father was so ignorant I could hardly stand to have the old man around. But when I got to be twenty-one, I was astonished at how much the old man had learned in seven years.

And finally, older students have learned to suspect that if something looks really obvious than there's probably a catch: so warned, they'll stay awake. The "find the maximum" program used for fourth years was the topic of the quote in Sect. 1.1. For the second-years, I used part of an assignment they had been given in the first-year introduction-to-programming course.

– You can say really important things like "No laptops, phones, newspapers, distracting conversations, reading of Game of Thrones are allowed in the lectures." *Really, not at all.* (If you don't say this at the beginning, you can't say it later because, then, it will look like you are picking on the person who is doing that thing. Only at the very beginning is it impersonal.) This one is really important, because part of getting students to want to understand what you are presenting is showing them that you are interested in them personally. If you don't care about students who are not paying attention, you will be perceived as not caring either about those who are.
– You can warn them not to be late for the lectures. (Do it now, not later, for the same reasons as just above.) Being on-time is important, because if they are not interested enough to be punctual they won't be interested enough to be responsive in the lecture itself, and so will form a kind of "dark matter" that will weigh-down your attempts to build a collaborative atmosphere.

But do not spend too much time on this initial stage: remember that most of it won't sink in and that, actually, that doesn't matter. Your aim with all the above is simply to get their attention.

Once all that is done, continue *immediately* with a programming exercise — because by now they are beginning to get bored. Wake them up!

The details of the exercise I use are the subject of Sect. 4.1. The exercise is not used for making people feel stupid; rather it's used instead for showing people, later, how much smarter they have become.

The exercise is handed out, very informally, on one single sheet of paper, handwritten (if you like); and then it's collected 10 min later. It's not marked at that point, and no comment is made about when it will be returned: just collect them up; store them in your bag; say nothing (except perhaps "thank you").

Like the introductory remarks, this exercise will probably be forgotten; and that's exactly what you want. When it returns, many weeks later, you want them to be (pleasantly) surprised.

2.2 Follow-Up by Cultivating a Dialogue with the Class: And Carry Out an Exercise in Triage

Delivering a course like this one can be seen as an exercise in triage, and the point of the initial dialogue is to carry this out. A (notional) third of the students, the very smart or very well-informed ones, will get value from the course no matter how badly you teach it, and you should be grateful that they are there. Amongst other things, they provide a useful validation function — for if they don't complain about what you say, you can be reasonably sure that any problems you might have are to do with presentation and not with correctness. And they can be used as "dialogue guinea-pigs" (see below). In fact these students *will* listen to and attempt to understand your introduction. But it wouldn't matter if they didn't, since they have already decided to do the course, and for the right reasons.

Another third of the students might be there because they think the course is easy, and that they can get credit while doing a minimal amount of work. They are not a problem in themselves; but neither should they consume too much of your effort and resources. They will not listen to your introduction, and it will make no difference to them, or to you, that they do not. If you're lucky, they will get bored and leave the course fairly early (so that they don't have to pay fees for it).

It's the last third, the "middle layer", who are the students you are aiming for. It's their behaviour you want to change, and it's them you want to excite. They will listen to your introduction, but not so much because they will use it to help them in planning or preparation; instead they will be trying to figure out what kind of person you are, and whether the course is likely to be fun. You must convince them that it will be.

So your constraints are mainly focussed on the top- and the middle groups: for the first, tell the truth (because they will know if you do not); for the second, simply be enthusiastic about what you say and let them see your genuine pleasure that they have come to take part in your course. The introduction could therefore go something like the following:

> *(In-)Formal Methods* are practical structuring and design techniques that encourage programming techniques easy to understand and to maintain. They are a particular kind of good programming practice, "particular" because they are not just rules of thumb. We actually know the theory behind them, and why they work. In spite of that, few people use them. But – by the end of this course – *you* will use them, and you will see that they work.
>
> Unusually, we do not take the traditional route of teaching the theory first, and only then trying to turn it into the practice of everyday programming methods. Instead, we teach the methods first, try them on examples; finally, once their effectiveness is demonstrated, we look behind the scenes to see where they come from.
>
> Thus the aim of this course is to expose its students – you – to the large conceptual resource of essentially *logical and mathematical* material underlying the construction of correct, usable and reliable software. Much of this has been "lost" in the sense that it is taught either as hard-core theory (quite unpopular) or – worse – is not taught at all. So there will be these main threads:
>
> 1. How to think about (correctness of) programs.
> 2. Case studies of how others have done this.
> 3. How to write your programs in a correctness-oriented way from the very start.
> 4. Case studies of how we can do that.
> 5. Why do these techniques work, and where would further study of them lead.

For (1) the main theme will be the use of so-called *static* rather than operational reasoning, that is thinking about what is true at various points in a program rather than on what the program "does" as its control moves from one point to another. This is harder than it sounds, and it takes lots of practice: explaining "static reasoning" will be a major component of the course.

The principal static-reasoning tools, for conventional programs, are *assertions*, *invariants* and *variants*. If you have never heard of those, then you are in the right place. Usually they are presented with an emphasis on formal logic, and precise step-by-step calculational reasoning. Here however we will be using them informally, writing our invariants etc. in English, or even in pictures, and seeing how that affects the way we program and the confidence we have in the result.

For (2), the programs we study will be chosen to help us put the ideas of (1) into practice, as they do need *lots* of practice. Usually the general idea of what the program needs to do will be obvious, but making sure that it works in all cases will seem (at first) to be an almost impossible goal. One's initial approach is, all too often, simply to try harder to examine every single case; and "success" might then be equated with exhaustion of the programmer rather than exhaustion of the cases.

Our alternative approach (3) to "impossible" programs will be to try harder to find the right way to think about the problem they are solving — often the obvious way is not the best way. But getting past the obvious can be painful, and tiring, though it is rewarding in the end: a crucial advantage of succeeding in *this* way is that the outcome – the correctness argument – is concrete, durable and can be communicated to others (e.g. to yourself in six months' time). Further, the program is much more likely to be correct.

Doing things this way is fun, and extremely satisfying: with (4) we will experience that for ourselves. It's much more satisfying that simply throwing programs together, hoping you have thought of every situation in which they might be deployed, and then dealing with the fallout when it turns out you didn't think of them all.

Finally, in (5), we recognise that the above (intellectual) tools all have mathematical theories that underlie them. In a full course (rather than just this article), we would study those theories — but not for their own sake. Rather we would look into the theories "with a light touch" to see the way in which they influence the practical methods they support. Those theories include *program semantics*, *structured domains*, *testing* and *compositionality*, and finally *refinement*.

2.3 Making Contact with the Top Layer: The "Dialogue Guinea Pigs"

In this first lecture, you should begin figuring out who the smart, confident people are. You will use them as a resource to help the other students learn

to take the risk of answering the questions you will ask throughout the course. Although the goal is to make the students feel that they can dare to answer a question even if they are not sure, in the beginning you have to show that this is not punished by embarrassment or ridicule.

So: find the smart people; but still, when you ask your first question, ask the whole room and not only them. Pause (since probably no-one will answer), and then pick a smart person, as if at random, who will probably give an answer that will be enough to work with even if it's not exactly right. Make having given an answer a positive experience, and others will be keen to join in.

As the weeks pass, you will increase your pool of smart targets; and you will establish that answering the questions is not threatening. Furthermore, having established a question-and-answer style allows you to fine-tune the pace of a lecture as you go: a conversation is always easier to manage than a prepared speech.

2.4 Making Contact with the Middle Layer: The Importance of "street Cred"

By now, the students have sat through your introduction, and they have probably completely forgotten that they did a small test at the beginning of the lecture (Sect. 2.1).[7] Do not remind them.

Instead, grab the middle layer of your audience by actually writing a program collaboratively, i.e. with their help on the board, right before their eyes — that is, after all, what the course is supposed to be about. Writing programs. But choose the program from material they know, ideally a program they have encountered before, and do it in a way they expect. Your aim here is not to dazzle them with new things which, you assure them, they will eventually master. Rather you are simply trying to *make contact*, and to reassure them that you really understand programming in exactly the way they do, and that you "can hack it" just as they can. Roll up your sleeves as you approach the board; literally, roll them up. This is the Street Cred.

A second aim of this first exercise is to establish a pattern of joint work, between you and them and among them. You aim to form a group spirit, where they will help each other; this is especially important later, when the students who understand the new ideas will enjoy helping the ones who have not yet "got it". Again there is something peculiar about this course: the message sounds so simple that people, initially, will think they understand when in fact they do not understand it at all. A camaraderie operating *outside* the lecture room is the best approach to this. It is terribly important.

In the second-year version of this course, I chose a programming assignment from those very students' first-year course and abstracted a small portion, a slightly intricate loop, and obfuscated it a bit so that they wouldn't recognise it except perhaps subliminally. (It's described in Sect. 5 below.) I then did the

[7] Have you forgotten too, as you read this? That's precisely the idea.

program with them, on the board, in exactly the same style I thought a second-year student might do it.[8] It was careful, operational and hand-waving... but the program worked,[9] and the class experienced a sense of satisfaction collectively when, after a hard but enjoyable struggle, we had "got it out".

I then had an insight (simulated), suggested a simplification, and did the same program again, the whole thing all over, but this time using a flowchart. Really, an actual flowchart in a course on "formal methods". The resulting program, the second version, was simpler in its data structures (the first version used an auxiliary array) but more complicated in its control structure: exactly the kind of gratuitous complexity a second-year student enjoys. When we finished that version, as well, we had begun to bond.

This program, and its two versions, are described in Sect. 5 below. What the students did not know at that stage was that the very same program would be the subject of their first assignment, where they would develop a program that was smaller, faster and easier to maintain than either of the two versions we had just done in class (and were, temporarily, so proud of). Having developed the earlier versions with them, all together, was an important emotional piece of this: the assignment shows them how to improve *our* earlier work, not theirs alone. It's described in Appendix B below.

3 Follow-Up Lectures, "Mentoring", and the Goal

A difference between the fourth-year version of the course and the second-year version was that "mentoring" was arranged for the latter. Neither course was formally examined: assessment was on the basis of assignments, and a subjective "participation" component of 10 %.

Mentoring (explained below) was used for second-years because of the unusual nature of this material and, in particular, its informal presentation. As mentioned just above (the "second aim") the risk is that "young students" can convince themselves that they understand ideas and methods when, really, they do not. The point of the mentoring is to make sure every single student is encouraged to attempt to solve problems while an expert is there who can give immediate feedback and guidance. In that way, a student who does not yet understand the material will, first, find that out early and, second, will be able to take immediate action.

The 40+ students in the second-year course in 2014 were divided into groups of 8 students, and met once weekly for 30 min with one of three mentors who (already) understood the material thoroughly. (Two of the mentors were Ph.D. students; and all three were veterans of the fourth-year version of the course.) The student-to-mentor allocations were fixed, and attendance at the mentoring sessions was compulsory (and enforceable by adjusting the participation mark).

[8] In fact I tested this beforehand on a small sample of such students, to find out whether they agreed that my hand-waving and picture-drawing could be regarded as a typical approach.

[9] Almost: see the "small problem" identified in Fig. 3.

The mentors were instructed (and did) ask questions of every student in every session, making sure no-one could avoid speaking-up and "having a go". In addition, the three mentors and I met for about 30 min each week so that I could get a bottom-up view of whether the message was getting through. The mentors kept a week-by-week log with, mostly, just a single A, B or C for each student: excellent, satisfactory or "needs watching". (If you ask the mentors to write too much, instead they won't write anything: this way, the more conscientious mentors will often put a comment next to some of the C results without your having to ask them to.)

With all that said, it is of course not established that the mentoring was effective or necessary: the course has not been run for second-years without it, and so there is no "control" group. And the course has no exam. So how do we know whether it was successful?

Success is measured of course against a goal; and the goal of this course is not the same as the goal for a more theoretical course on rigorous program-derivation. For the latter, it would be appropriate to an examination-style assessment testing the ability (for example) to calculate weakest preconditions, to carry out propositional- and even predicate-logic proofs, to find/calculate programs that are correct by construction wrt. a given specification etc.

In fact the two goals for this course are as follows: the first targets the "upper layer" identified in Sect. 2.2 above. Those students will do very well in this course, and it will probably be extremely easy for them. Indeed, it will probably not teach them anything they could not have taught themselves. Are they therefore getting a free ride?

No, they are not: the point is that although they *could* have taught themselves this material, they probably *would not have done so* — simply because without this course they'd have been unaware that this material existed. As educators we strive to maintain standards, because the society that pays for us relies on that. In this case, the good students, the longer view is that it's precisely these students who will take a real "formal" Formal Methods course later in their curriculum, and then will become the true experts that our increasing dependence on computers mandates we produce. Without this course, they might not have done so.

The second goal concerns the "middle layer". These students would never have taken a real Formal Methods course and, even after having taken this course, they probably still won't. But, as the comments in Appendix F show, their style and outlook for programming have been significantly changed for the better — at least temporarily. They will respect and encourage precision and care in program construction, for themselves and – just as importantly – for others; and if they ever become software-project managers or similar, they will be likely to appreciate and encourage the selective use of "real" formal methods in the projects they control.

These people will understand this material's worth, and its benefits; and they will therefore be able to decide how much their company or client can afford to pay for it — they will be deciding how many top-layer experts to hire and,

because they actually know what they are buying, they will be able to make an informed hard-cash case for doing so. Given the current penetration of rigour in Software Engineering generally, that is the only kind of case that will work.
References to Section 4.1 and beyond can be found in Part II; appendices are at the end of Part II.

4 Part II – Specifics

4.1 Binary Search: The Baseline Exercise in Lecture 1

We now look at some of the actual material used for supporting this informal approach to Formal Methods. We begin with the "baseline" exercise mentioned in Sect. 2.1.

Binary Search is a famous small program for showing people that they are not as good at coding as they think. But that is not what we use it for here. Instead, it's used to establish a starting point against which the students later will be able to measure their progress objectively.

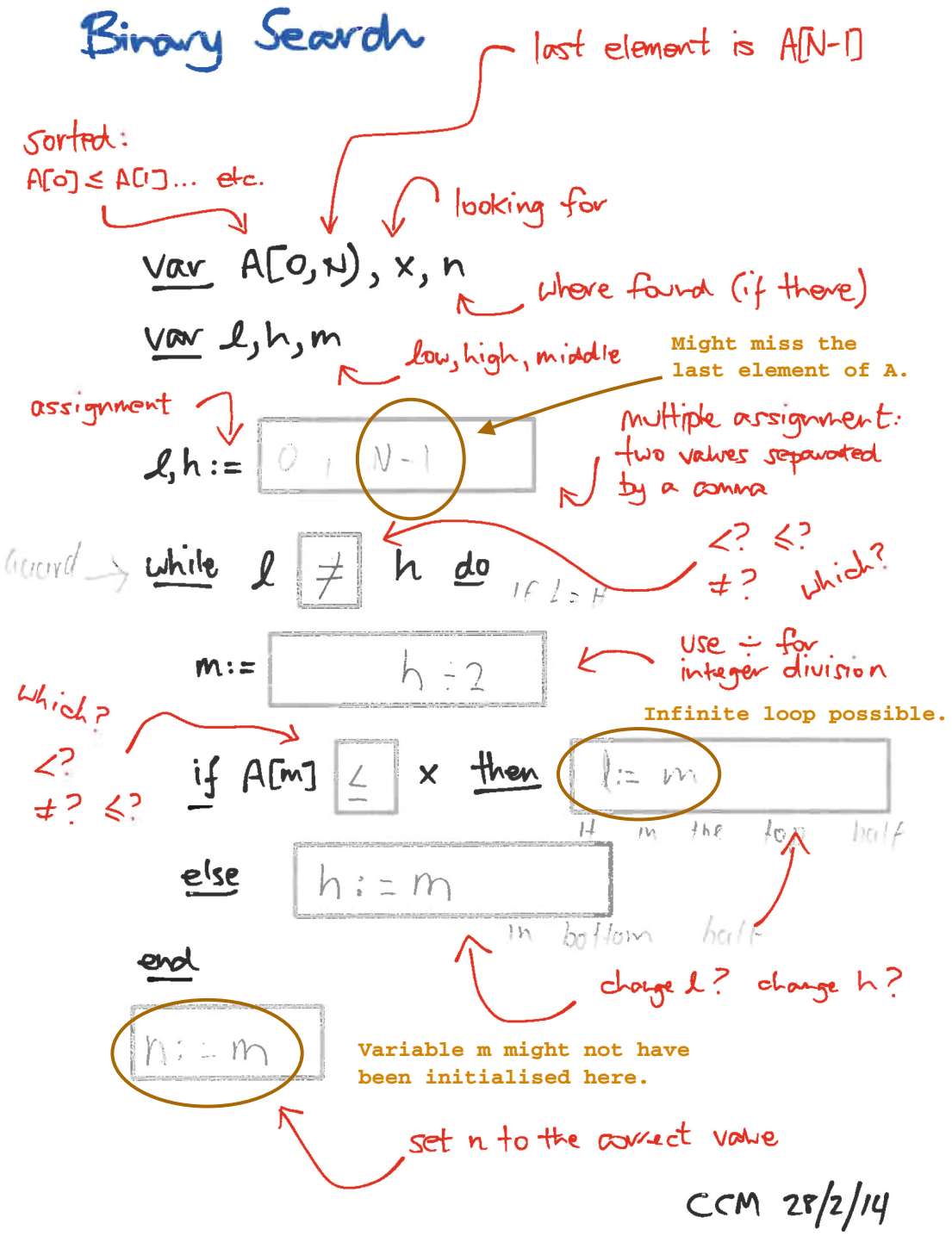
The exercise is deliberately made to look informal, "unofficial", so that very little importance will be attached to it, and it's done (almost) as the very first thing, to give people the greatest chance of forgetting that it happened at all. Thus it's a single sheet, with instructions on the front and boxes to fill-in on the back, almost like a questionnaire or survey. The actual version used was handwritten,[10] further increasing its casual appearance. Furthermore, doing it that way makes the point that the emphasis is on *concepts* and precision in thought, not on LaTeX- or other word-processed "neatness" of presentation. (For this article, however, I have typeset it to avoid problems with my handwriting: in class, they can just ask what any particular scrawl is supposed to be.)

A typical answer is in Fig. 1, on the reverse side of the handwritten version. (The front side, including the instructions, appears in Appendix A.) The typewriter-font text and ovals were added in marking. The hand-written comments and arrows were there in the handed-out version, in order to give very explicit help about what was expected and what the possible answers were. The light-grey text in the boxes of Fig. 1 are the student's answers.

The (typewriter-font) comments in the marking are important, even if it's only a few: if the program is wrong, for your credibility as a teacher you must identify at least one place it clearly *is* wrong. That way the student won't blame you; and it also makes the point that you actually bothered to read the work.[11]

[10] The handwritten version was done with a tablet app, and then converted to PDF. It's important that it be done that way so that – even handwritten – it can easily be corrected and improved as it's used and re-used in subsequent years.

[11] Annotating a PDF on a tablet is a remarkably efficient way of doing all this. Not only can you come back and alter your remarks later, but you can do the marking on the bus or train, a few each day, in time you wouldn't otherwise be using. And it's important to spread-out the marking as much as possible, so that your comments are fresh each time: as much as possible your comments should seem personal.



The exercise itself is at App. A: this figure is the answer. The (orange) notes in `courier` are the marker's comments; all other material, including (red) handwriting with arrows, was present in the handout beforehand.
The student's answers are within the grey boxes.

Fig. 1. Typical marked answer to the Binary-Search exercise of Sect. 4.1(Color figure online).

You don't have to find *all* errors, however, since this test is not marked numerically for credit. The aim is merely to find *enough* errors so that the students will be pleased, later, when they realise that the techniques they have learned would have allowed them to write this program, first time, with no errors at all.

5 The Longest Good Subsegment

This problem forms the basis for the first assignment, whose role in the overall course was described in Sect. 2.4 above: in summary, the problem is solved using "traditional" methods, in class – in fact, it is done twice. Once that has been endured, this assignment does it a third time, but using instead the techniques that the course advocates. The hoped-for outcome is that they will achieve a better result, by themselves but using these methods, than they did with the lecturer's help without these methods.

5.1 Problem Statement

Assume we are given an array `A[0,N)` of `N` integers `A[0]`, `A[1]`,..., `A[N-1]` and a Boolean function `Bad(n)` that examines the three consecutive elements `A[n]`, `A[n+1]`, `A[n+2]` for some (unspecified) property of "badness". We write `A[n,n+3)` for such a subsegment, using inclusive-exclusive style.

Just what badness actually is depends of course on the definition of `Bad`; but for concreteness we give here a few examples of what `Bad` might define. If subsegment `A[n,n+3)` turned out to be $[a, b, c]$ then that subsegment might defined to be "bad" if a, b, c were

- All equal: $a = b = c$.
- In a run, up or down: $a{+}1 = b \wedge b{+}1 = c$ or $a{-}1 = b \wedge b{-}1 = c$.
- Able to make a triangle: $a{+}b{+}c \geq 2(a \max b \max c)$.[12]
- The negation of any of the above.

Note that however `Bad` is defined, using `Bad(n)` is a programming error, subscript out of range, if `n` < 0 or `n`$+3 >$ `N`.

Now a subsegment of `A` is any consecutive run `A[i,j)` of elements, that is `A[i],A[i+1],...,A[j-1]` with $0 \leq$ `i` $\leq$ `j` $<$ `N`. We say that such a subsegment of `A` is *Good* just when it has no bad subsegments inside it. A Good subsegment of `A` can potentially have any length up to the length `N` of `A` itself, depending on `A`; and any subsegment of length two or less is Good, since no Bad subsegment can fit inside it. (Bad subsegments of `A` have length exactly 3; and since they can overlap, it's clear that `A` can contain anywhere from 0 up to `N`-2 of them.)

The specification is of our program is given in Fig. 2.

[12] The operator (max) is "maximum".

We are to write a program that, given `A[0,N)`, sets variable `l` to the length of the longest Good subsegment of `A[0,N)`. More precisely, the program is to

Set `l` so that for some n[a]
- `A[`n`,`n`+l)` fits into `A`, that is $0 \leq n \wedge n+$`l` $\leq$ `N`; and
- There is no bad subsegment `A[`b`,`$b+3$`)` such that both

$n \leq b \wedge b+3 \leq n+$`l` — `A[`$b$`,`$b$`+3)` fits into `A[`$n$`,`$n$`+l)`
and `Bad`(b) . — `A[`b`,`b`+3)` is bad

[a] The "`l`" at left is the letter ℓ, not the digit one. The latter is `1`.
We write actual program variables in `Courier` (i.e. typewriter) font. Mathematical variables like n are in *mathsfont*, to avoid giving the impression that there has to be some variable `n` in the program. (There *might* be an `n` that stores the value n; but there does not have to be.)

Fig. 2. Specification of the first programming assignment

5.2 First Idea

An Operational Approach. A typical second-year student's first thoughts about this problem might be along these lines. First go through all of `A` and store its "bad positions" b in an auxiliary array `B`. Then find the largest difference *maxDiff* between any two adjacent elements b of `B`. Then – after some careful thought – set `l` to be *maxDiff*+2... or something close to it.[13]

Notice the operational phrasing "go through...", and the slightly fuzzy "something close to it" (one more than that? one less? exactly that?) A typical student would code up the program at this point, using the guess above, and see whether the answer looked right. It's only an assignment statement, after all, and if it contains a one-off error well, it can be "tweaked", based on trial runs, without affecting the structure of the rest of the program. This is how second-years think. (And they are not alone.)

We developed such a program (given below), interactively at the board, without hinting even for a moment that there might be a better way: indeed the whole

[13] The "careful thought" here, which most students will enjoy, is to figure out what the longest Good subsegment can be that includes neither of the bad subsegments `A`$[b_0, b_0+3)$ and `A`$[b_1, b_1+3)$, where $b_0 < b_1$ are the two adjacent bad positions with $maxDiff = b_1 - b_0$.

It must start at-or-after b_0+1, in order to leave out `A`$[b_0, b_0+3)$, and it must end at-or-before b_1+2-1 to leave out `A`$[b_1, b_1+3)$. So its greatest possible length is $(b_1+2-1) - (b_0+1) + 1$, that is $b_1 - b_0 + 1$. Since the largest such $b_1 - b_0$ is $maxDiff$ itself, the correct value for length `l` is $maxDiff+1$ for the largest $maxDiff$ — and not $maxDiff+2$ as suggested above. This "guessing wrong" and then "calculating right" can be simulated in your presentation, and increases the students' sense of participating in the process.

point of sneaky exercise is to get away with having the students agree that this is a reasonable solution, one indeed that they might have come up with themselves.

"Interactive", by the way, is important. The ideas of "static reasoning", invariants, assertions and so on are so simple when finally you understand, but so hard to grasp for the first time. (That's why the mentoring described in Sect. 3 is important: the students who are not yet assimilating the approach must be helped to *realise for themselves* that they are not.) Building a "let's all work together" atmosphere encourages the students to help each other; and so another contribution of this program-construction exercise carried out collectively is to build that collegiality in the context of something they understand: operational reasoning about a program. They *understand* operational reasoning.

Our aim is to change the reasoning style while maintaining the collegiality.

The Program Developed at the Board. The program we came up with is given in Fig. 3.[14] It uses an auxiliary array but, in doing so, does achieve a conceptual clarity: first find all the bad segments; then use that to find the (length of) the longest Good segment.

5.3 Second Idea

Use a "Conceptual" Extra Array. The more experienced programmers in the group will realise that, although having the auxiliary array `B` is useful for separating the problem into sub-problems, logically speaking it is also "wasteful" of space: a more efficient program can be developed by using a *conceptual* `B`'s whose "currently interesting" element is maintained as you go along. After all, that's the only part of `B` you need.

This idea of a "conceptual" `B` is very much a step in the right direction. (It's related to auxiliary variables, and to data-refinement.) This is an insight that can be "simulated" during the presentation of this problem.

A second more advanced technique that the better students might consider is to use "sentinels", in order not to have to consider special cases.

But the resulting control structure turns out to be a bit complicated; and so, in order to see what's going on, we use a flowchart. The flowchart in Fig. 4 is the one we developed at the board, all together interactively in class. Again, this was very familiar territory (at least for the better students).

The code shown in Fig. 5 is the result.

5.4 Static Reasoning and Flowcharts

Ironically, the flowchart Fig. 4 gives us the opportunity to introduce static reasoning just as Floyd did in 1967:[15] one annotates the arcs of a flowchart. Experience seems to show that, initially, students grasp this more easily than the idea of something's being true "at this point in the program text".

[14] The syntax is based on `Dafny`, for which see Sect. 6. `Dafny` does not however have `loop`, `for`, `repeat` or `exit` constructions.

[15] R.W. Floyd. Assigning Meanings to Programs. *Proc Symp Appl Math.*, pp. 19–32. 1967.

```
// Find length of longest subsegment of A[0,N)
// that itself contains no "bad" subsegments.
// (Array A is referred to from within Bad.)

m:= 0
for b:= 0 to N-3 { // Don't go too far!
    if (Bad(b)) { B[m],m:= b,m+1 }
}
// Now B[0,m) contains the starting indices of all bad subsegments.

l= 0
for i:= 0 to m-1 { // For each bad-subsequence index...
    if (i==0) { l= l max (b[i]+2) } // compare first with start...
    else { l:= l max (B[i+1]-B[i]+1) } // else compare with previous.
}
l:= l max (N-B[m-1]-1) // Compare end with the last one.
```

This program is not supposed to be especially well written (it isn't), but neither is it supposed to be really badly done (it isn't *that* bad, for a second-year). It's supposed to be *credible*, something the students might have done themselves.

The special end-cases (the `i==0` case) and the `+1`'s and `-1`'s lying around: these should be familiar instances of the sort of irritating details that the students, by now, are beginning to accept as "just a part of programming".
We want them to learn *not* to accept them.

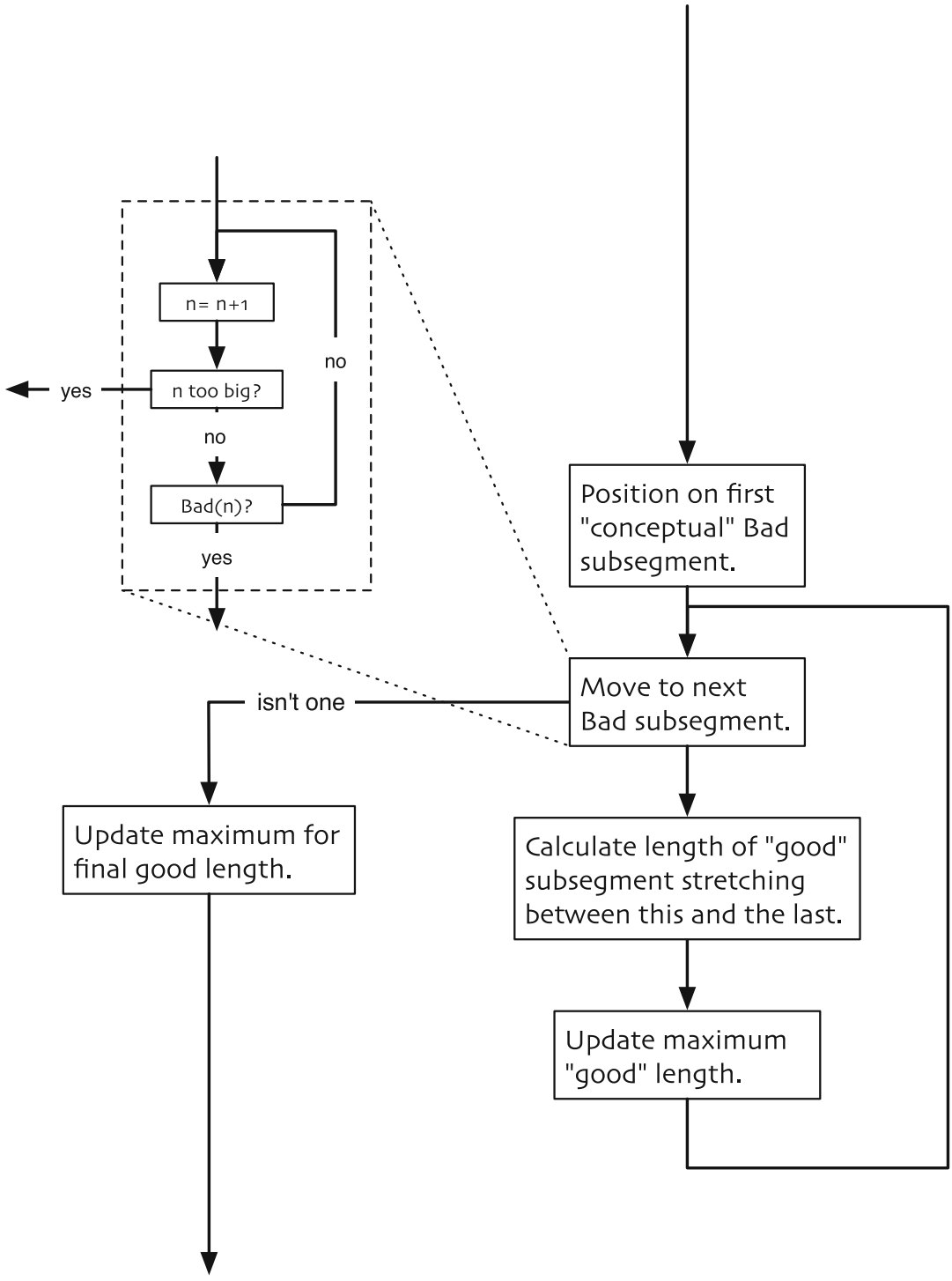
A "small problem" though is that this program will fail (indexing error in its last statement) if there are no "bad" subsegments at all. How many students would miss this? And why is the first case `b[i]`+2 rather than `b[i]`+1?

Of those that understand those points, how many would revel in discussing these "deep subtleties" with their friends, over a beer? Is that part of the fun of programming? Should it be?

Fig. 3. First approach to the "bad segment" problem

An example of that applied to this problem is given in Figs. 6 and 7: no matter that the assertions there are complicated, even ugly. The point is that the assertions are *possible at all*, a new idea for the students, and crucially that their validity is entirely local: each "action box" in the flowchart can be checked relative to its adjacent assertions alone. There is no need to consider the whole program at once; no need to guess what it might do "later"; no need to remember what it did "earlier".

That's a very hard lesson to learn; experience with these courses suggests that it is not appreciated by course's end, not even by the very good students. It is only "planted", and grows later.



Like the first approach, this one also has a problem if `A` contains no `Bad` subsegments.

Fig. 4. Flowchart for the second approach to the "bad segment" problem

```
// Find length of longest subsegment of A[0,N)
// that itself contains no "bad" subsegments.
// (Array A is referred to from within Bad.)

b,n,l:= -1,-1,0
outer: loop {
    repeat {
        n= n+1
        if (n+3>N) { exit outer }
    } until (Bad(n))
    l,b:= l max n-b+1,n
}
l:= l max N-b-1
```

No auxiliary array `B` is needed in this program. Instead, variable `b` is the value that `B[m]` *would have* — if there were still a `B`. (As a data-refinement, this would have introduced `b` with coupling invariant $\mathtt{b} = \mathtt{B[m]}$.)

The sentinels are "virtual" bad segments at `A[-1,2)` and `A[N-2,N+1)`: they automatically prevent `A[-1)` and `A[N]` from being considered as candidates for our longest Good subsegment.

Fig. 5. Code for the second approach to the "bad segment" problem

5.5 The First Assignment, Based on This Example

The assignment based on this example is handed out about a week (i.e three hours of lectures) *after* the operational developments described above. Other material is presented in between. (The material of Sect. 6 works well for that.)

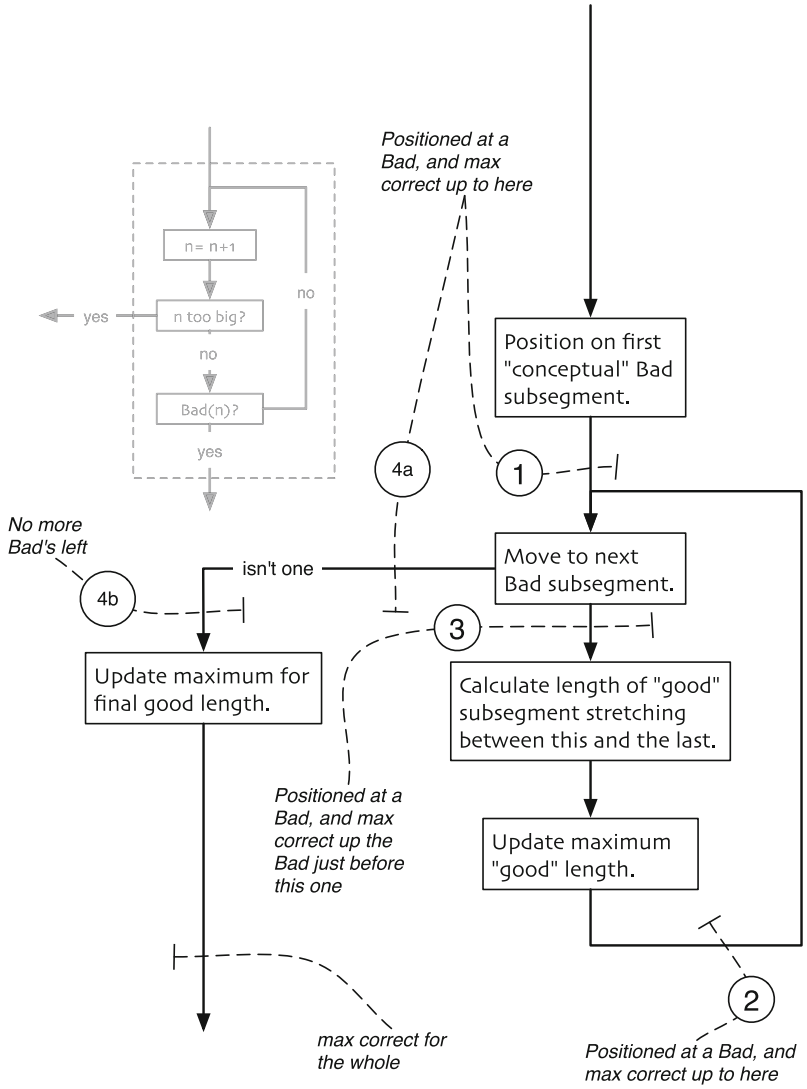
Handed out too soon, the whole exercise looks like trickery or even mockery: why should the lecturer go through all that stuff only to say, just afterwards, that it's all wrong? Left to with the students to "marinate", it will come to be seen as a solution that they thought of themselves. And then they are ready to appreciate a better one.

The assignment is reproduced in Appendix B.

6 Tools for Development and Proof: Software Engineering

6.1 Proofs by Hand; Proofs by Machine

It's tempting to base a computer-science course on a particular programming language, a particular *IDE*, a particular program-verifier. If the lecturer is already familiar with the tools, then the lectures and exercises can very easily be generated from the specifics of those tools; and indeed "not much energy required" lectures can easily be given by passing-on information that you already know to people who just happen not to know it yet (but don't realise they could teach themselves). The lecturer can simply sit in a chair in front of the class, legs crossed, and read aloud from the textbook.

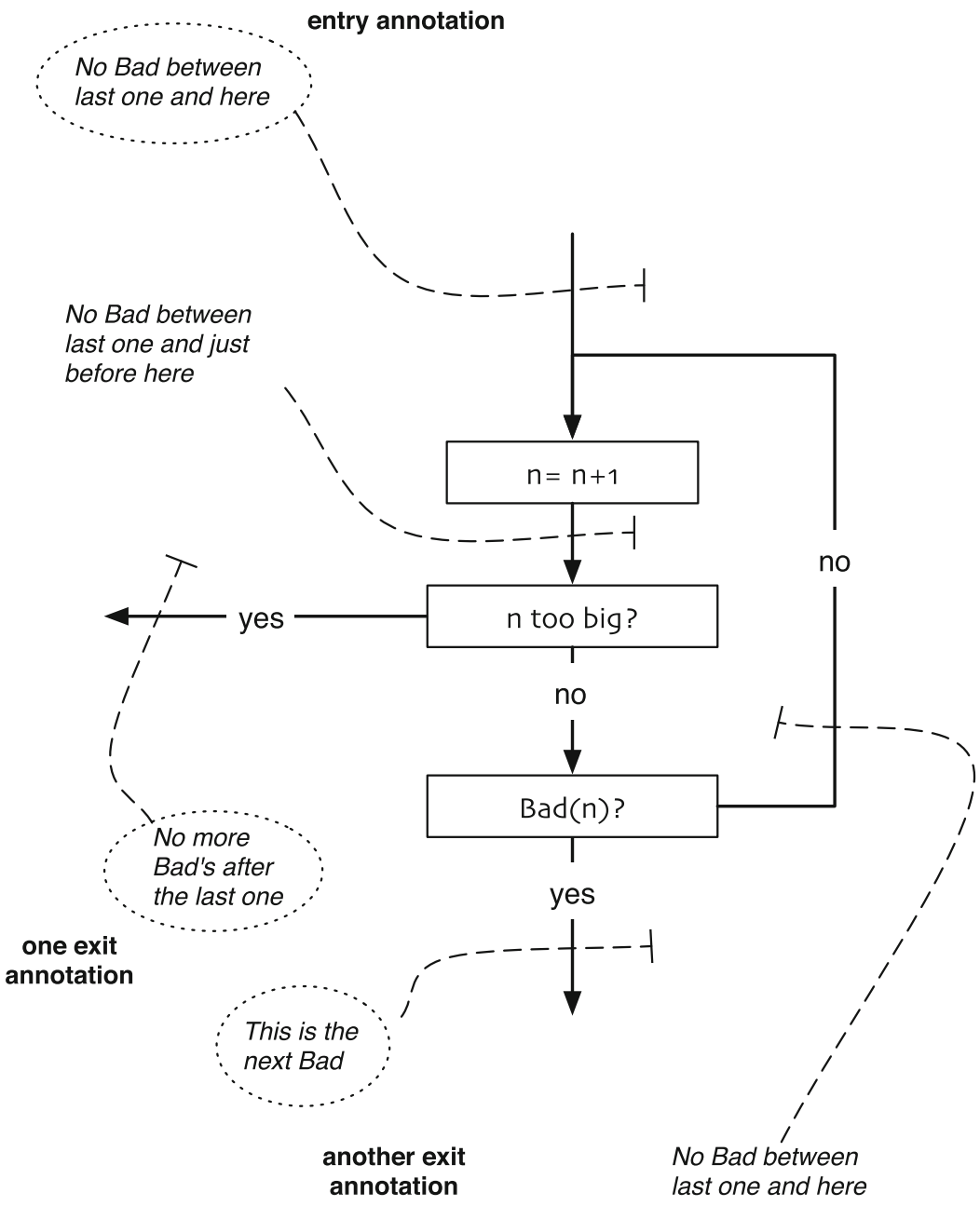


To understand a flowchart, we can annotate the arcs that connect one action to the next. The annotation states a fact that must be true whenever program execution reaches that arc, *no matter how execution got there*. This is called "static" reasoning. To understand static reasoning like this, you can check the action boxes of the flowchart one-by-one: there is no need to understand them all at once. Just make sure for each box that if the fact annotating an entry arc holds, and the action takes place, then any fact annotating an exit arc holds too.

For example, the box "Move to next Bad segment" has two entry arcs (1 and 2) and two exit arcs (3 and 4). The facts on the input arcs (1,2) are the same, "Positioned at a Bad, and max correct up to here". We therefore can *assume* it is true whenever this box is reached.

The box has two exits (3,4); one of them is labelled by two facts (4a,4b). All those facts hold *given* that the entry facts held, the box's action was taken *and* that control left the box along the path labelled by the fact.

Fig. 6. Annotated flowchart for "bad segment" problem: outer code



These are the annotations for the "inner" portion of the "bad segment" program. If we understand this part, we can the understand its role in a larger program (e.g. Fig. 6) simply by concentrating on the annotations of its entry and exits.

Fig. 7. Annotated flowchart for "bad segment" problem: inner code

There are, however, some disadvantages too. The first two of them apply to any tool presentation in class; and the third is more specific to formal methods. They are:

– Even if it's easy to describe programming-language syntax to the class, there is no guarantee that they will be listening or, if they are, that they will remember what you have said. In the end, they'll have to look it up anyway. And then what were the lectures for?
– From the fact that the lecturer seems to think the particular sequence of buttons to push in an *IDE* is important, the students will conclude that the sequence indeed is important. But is it really?
– Since an automated prover can check your reasoning and find any mistakes, one could conclude that it's not necessary to try to avoid making mistakes in the first place. In that case, why don't we get machines to write the programs, instead of merely check them?

In spite of all those caveats, there is an overwhelming advantage to using tools: without them, no serious (software) project of any size can be rigorously completed. It is simply not possible to master that complexity, either alone or in teams. How do we navigate between these two opposites?

What's important is how a tool is *presented*: it's just a tool. If you're lecturing about building circuit boards, you don't spend hours describing which end of a soldering iron to hold: students will figure that out pretty quickly for themselves. And in the end, a robot will do the soldering anyway. What's crucial is that they know what solder is for, and what therefore the robot is doing. The details of a *particular* robot are uninteresting and unimportant.

In this course, the (experimental) principle was adopted that things should not be done using *with* computer-based tools until those same things, perhaps in smaller instances, have been done by hand *without them.*[16]

In the case of static reasoning over programs, then, do we explain what an invariant is informally, conceptually, and how you intuit your way into finding

[16] The *reductio ad absurdum* arguments, e.g. that students must translate their programs manually into assembler before being allowed to use a compiler, are escaped by recognising the role of abstraction.

A compiler provides a layer of abstraction that its user can pretend is reality. The programmer can believe, and reason as if, assignment statements really do assign (instead of loading into a register and then storing somewhere else), as if while loops really do "loop while" instead of comparing, setting condition bits and then (conditionally) branching back, whatever that latter might mean to a second-year. And such abstractions are usually good enough for a first course: more hostile, demanding applications can break them; but by that stage, students are ready to go to lower levels.

On the other hand, a typical *IDE* doesn't abstract from anything: it's a cookbook, not a chemical analysis of edible compounds and their reactions with each other. To make and run a program, you type its text into the left-hand window and keep pressing a button on the right, and fiddling with your text, until all the red highlights go away. Press another button at that point, and some outputs might appear somewhere else. It's hard to explain what is going on without knowing the primitives over which these actions operate: source files, compilers, libraries, linkers, archives, object-files, debuggers. And indeed some teachers think that they are doing their students a favour by hiding those details from them. If, on the other hand, the primitives are explained and used first, on small examples, the *IDE* can be explained as a convenience rather than as an incantation.

one? Or do we explain an invariant as something which when typed into a particular tool will allow it to say "loop verified". (Or do we pretend that the invariant follows inescapably, inevitably by careful manipulation of the symbols in the predicate-calculus formulation of your loop's postcondition?)

In this course the motto for finding invariants is "beg, borrow or steal". It makes absolutely no difference where an invariant comes from; but it makes all the difference that you insist on having it, and then what you do with it from that point on.

6.2 Proofs by Hand

The very first example in the list below is used early in the course, informally:

> "If you know that x equals 1 now, and the statement $x := x+1$ is executed, what can you say about the value of x afterwards?"[17]

It's easy, and indeed it looks too easy. With the further exercises in that list, we build our way from easy cases to hard ones, looking for the painful point where the problem is still small enough that you think you should be able to work it out in your head but, whenever you try you keep getting the wrong answer. By doing that, we are creating an irritation, a curiosity and, actually, a need.

That need should be satisfied by the "Hoare assignment rule", but instead of presenting it as an axiom of a beautiful programming logic (while remembering that the very idea of have a *programming* logic is beautiful in itself), it is simply presented as an algorism[18] for assertions and programs. Presenting it as a definition is premature: remember that the students have not yet realised that there even is a thing that needs defining.[19]

Thus what these exercises are designed to do is to take the students from simple assertional reasoning, where operational thinking works, to more complicated examples where they still understand what they are trying to do but find the details have become difficult to control. At that point, they are ready to be given a method — but still, crucially, a method to be used by hand.

---

[17] Just after introducing flowcharts is a good moment. (Recall Sect. 5.3.).

[18] *Algorisms* are the techniques for calculating with notations denoting numbers, viz. they are algorithms for arithmetic.

[19] Although the compelling rigour of logic, meta-language and object language, is what guided the creators of formal methods and is what makes sure that, in the end, it all comes together into a coherent whole, for most programmers it's best not to present it that way initially.

Try explaining to a second year that "actually" there are at least four kinds of implication in Formal Logic: the ordinary "if/then" of natural language, the horizontal line in a sequent, the single turnstile, the implication arrow... and then, underneath it all, the double turnstile (which makes five).

Those things have to be *asked for* when a person reaches the point of being too confused to proceed without them. Only then will you be thanked for giving the answer.

precondition	program	postcondition
$\{x{=}1\}$	$x{:=}\ x+1;$	$\{???\}$
$\{x{=}2\}$	$x{:=}\ x/2;$	$\{???\}$
$\{x{=}3\}$	$x{:=}\ x/2;$	$\{???\}$
$\{???\}$	$x{:=}\ x/2;$	$\{x{=}1\}$
$\{x{=}A \wedge y{=}B\}$	$x{:=}\ y;$	$\{???\}$
$\{x{=}A \wedge y{=}B\}$	$x{:=}\ y; y{:=}\ x;$	$\{???\}$
$\{x{=}A \wedge y{=}B\}$	$x{:=}\ x+y; y{:=}\ x-y;$	$\{???\}$
$\{x{=}A \wedge y{=}B\}$	$x{:=}\ x+y;$ $y{:=}\ x-y;$ $x{:=}\ x-y;$	$\{???\}$
$\{x{=}A \wedge y{=}B\}$	$t{:=}\ x;$ $x{:=}\ y;$ $y{:=}\ t;$	$\{???\}$
$\{\ x=A \wedge y=B \wedge z=C\ \}$	$???;\ ???;\ ???;\ ???;$	$\{\ x=B \wedge y=C \wedge z=A\ \}$
$\{???\}$	$y{:=}\ x{*}x-2{*}x+1;$	$\{y=0\}$
$\{???\}$	$y{:=}\ x{*}x-3{*}x+2;$	$\{y=0\}$
$\{x{=}A\}$	`if` $(x{<}0)$ `{`$x{:=}\ -x;$`}`	$\{???\}$
$\{x{=}A \wedge y=B\}$	`if` $(x{<}y)$ `{`$x,y{:=}\ y,x;$`}`	$\{???\}$
$\{x{=}A \wedge y=B\}$	`if` $(x{\le}y)$ `{`$x,y{:=}\ y,x;$`}`	$\{???\}$
	$s{:=}\ x;$ $s{:=}\ s{+}y;$ $s{:=}\ s{+}z;$ $\{???\}$	$\{???\}$ $\{???\}$ $\{???\}$
	$p{:=}\ x;$ $p{:=}\ p{*}y;$ $p{:=}\ p{*}z;$ $\{???\}$	$\{???\}$ $\{???\}$ $\{???\}$
	$s{:=}\ 0;$ $s{:=}\ s{+}x;$ $s{:=}\ s{+}y;$ $s{:=}\ s{+}z;$ $\{???\}$	$\{???\}$ $\{???\}$ $\{???\}$ $\{???\}$

precondition	program	postcondition
	p:= 1;	$\{???\}$
	p:= p∗x;	$\{???\}$
	p:= p∗y;	$\{???\}$
	p:= p∗z;	$\{???\}$
	$\{???\}$	
	m:= x;	$\{m = x\}$
	`if` (m<y) `{`m:= y;`}`	$\{m = x \max y\}$
	`if` (m<z) `{`m:= z;`}`	$\{???\}$
	$\{???\}$	
	$\{$`minInt` $\le x\}$	
	m:= `minInt`;	$\{m =$ `minInt` $\wedge$ `minInt` $\le x\}$
	`if` (m<x) `{`m:= x;`}`	$\{???\}$
	`if` (m<y) `{`m:= y;`}`	$\{???\}$
	`if` (m<z) `{`m:= z;`}`	$\{???\}$
	$\{???\}$	
	m:= $-\infty$;	$\{???\}$
	`if` (m<x) `{`m:= x;`}`	$\{???\}$
	`if` (m<y) `{`m:= y;`}`	$\{???\}$
	`if` (m<z) `{`m:= z;`}`	$\{???\}$
	$\{???\}$	
	m:= $+\infty$;	$\{???\}$
	`if` (m>x) `{`m:= x;`}`	$\{???\}$
	`if` (m>y) `{`m:= y;`}`	$\{???\}$
	`if` (m>z) `{`m:= z;`}`	$\{???\}$
	$\{???\}$	
	m, n:= $-\infty, 0$;	$\{m = \max A[0, 0)\}$
	`while` (n≠$\lvert A\rvert$)	invariant $\{m = \max A[0, n)\}$
	`{` m:= $m \max A[n]$;	$\{m = \max A[0, n{+}1)\}$
	n:= n+1;	$\{???\}$
	`}`	
	$\{??? \wedge n{=}\lvert A\rvert\}$	$\{m = \max A\}$
	s, n:= $0, 0$;	$\{???\}$
	`while` (n≠$\lvert A\rvert$)	invariant $\{s = \sum A[0, n)\}$
	`{` s, n:= s+$A[n], n$+1; `}`	
	$\{???\}$	
	p, n:= 1, 0;	
	`while` (n≠$\lvert A\rvert$)	invariant $???$
	`{` p, n:= p∗$A[n], n$+1; `}`	
	$\{???\}$	
$\{\lvert A\rvert \ge 1\}$	m, n:= $A[0], 1$;	$\{???\}$
	`while` (n≠$\lvert A\rvert$)	invariant $\{m = \max A[0, n)\}$
	`{` m, n:= $m \max A[n], n$+1; `}`	
	$\{m = \max A\}$	

6.3 Proofs by Machine

By the time the examples above have been worked through, using by-hand calculation based on the substitution definition of an assignment's action on a post-condition, a second boundary has been reached. Even with that assignment rule, it's become too easy to make simple mistakes in the calculations — not because they are difficult to understand, but because they have become larger and are more encumbered with the trivial complexities common in programs generally. And, indeed, the amazing effectiveness and the novelty of the rule has, by then, worn off as well. Having grown used to calculation rather than guesswork, the students are ready for the transition to a tool that does those calculations for them. Crucially, however, they know what those calculations are and thus what the tool is doing (although not, perhaps, how it does it).

It's the same two transitions that move, first, from counting on your fingers to using positional notation on paper and then, second, from paper to a pocket calculator. For us, the pocket calculator is a program verifier. Having done this in stages, though, we understand that the calculator/verifier is just doing what we did on paper, but faster and more reliably. And what we were doing on paper was just, in turn, what we were doing with our fingers/operationally, but with a method to control detail.

In Appendix C we give the same exercises as above, Figs. 12, 13, 14, 15, 16, and 17, written in the language of the program-verification tool `Dafny`: it will be our pocket-calculator for program correctness. Naturally the exercises should however be done by hand, in class, before the "pocket calculator" is used. Or even before it's revealed that there is one.

6.4 `Dafny`

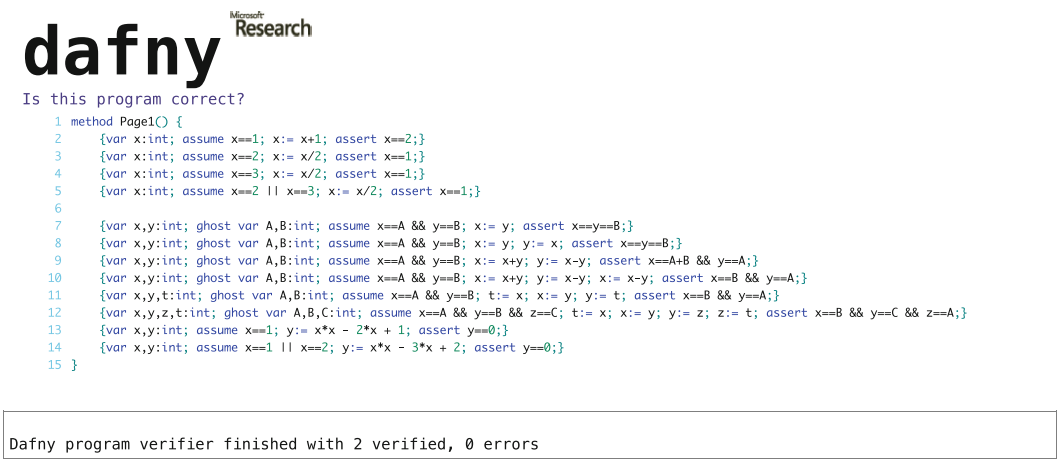
From the `Dafny` website:[20]

> ***`Dafny`*** *is a programming language with a program verifier. As you type in your program, the verifier constantly looks over your shoulders and flags any errors.*

One presentation of `Dafny` is embedded in a web-page, where you can simply type-in a program and press a button to check its correctness with respect to assertions you have included about what you want it to do.[21] In Fig. 8, the first part of the assertion exercises from p. 25 above has been (correctly) completed using the template of Fig. 12 (in Appendix C), and that has been pasted into the `Dafny` online-verification web-page.

In Fig. 9, the same has been done but with a deliberate error introduced; and it shows that `Dafny` found the error. When explaining this to the students, *it cannot be stressed too much* what a remarkable facility this is. They will be used to syntax errors (annoying, but trivial); and they will be used to run-time errors (fascinating, unfortunately, but also time-consuming). What they

[20] http://dafny.codeplex.com.
[21] http://rise4fun.com/dafny.



The first part of the exercises from 26 has been completed, and put into `Dafny`-format based on Fig. 12. `Dafny` reports "verified".

Fig. 8. The `Dafny` on-line verifier applied to Fig. 12 completed correctly.

will never have seen before is a computer-generated warning that means "Your program compiles correctly but, even if it does not actually crash at runtime, there are circumstances in which it will give the wrong answer." It is – or should be – one of those moments we all remember from our undergraduate days that we were truly amazed, that mentally we "went up a level".[22]

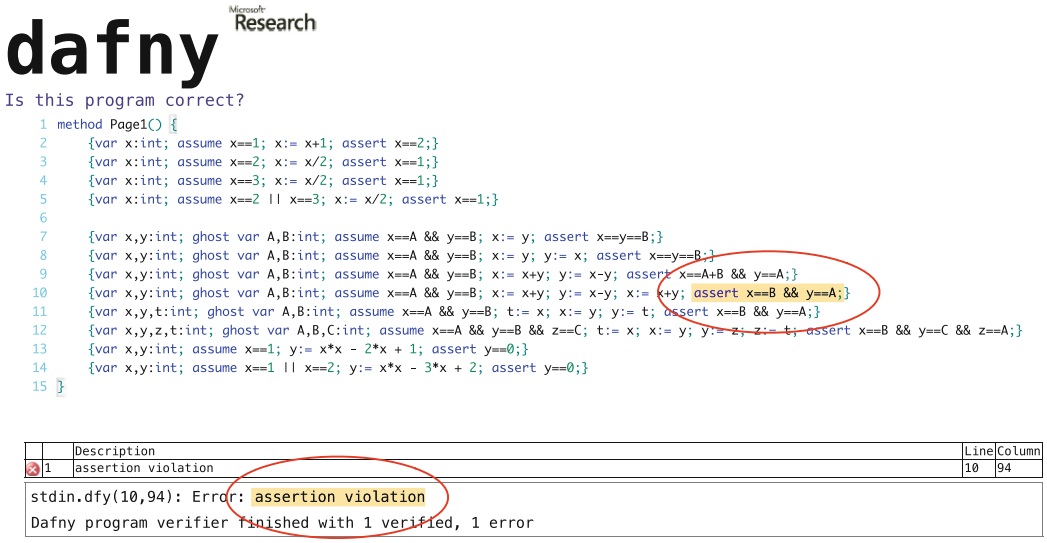
7 Return to Binary Search

In Sect. 2.1 the one-page handwritten *Can you write Binary Search?* questionnaire was described, given almost at the beginning of the very first lecture and then (we hope) forgotten. What happens when we return to it? What material should have been covered in the meantime?

7.1 Why Return to Binary Search at All?

Our aim is to convince students that the techniques we are presenting are worth their while. Most people, at least when they are young, are curious: curiosity can almost be defined as the urge to find out things irrespective of their expected utility. But university students have begun to lose that, at varying rates of course; but the effect that teachers must confront is that many students will learn things

[22] As a teacher, however, be prepared for the later moment when they realise that Dafny can sometimes fail to prove correctness even when the program is correct. Of course it shares that problem with all verifiers: but the pedagogical issue is "How far can you get before reality bites?".



The first part of the exercises from P. XX has been completed, and put into `Dafny`-format based on Fig. 12 but with a deliberate mistake introduced. `Dafny` reports an error: it's not a syntax error; and it's not a run-time error, since indeed the program hasn't been run yet. It's an error-report of the kind the students probably will *never have seen before*.

Fig. 9. The `Dafny` on-line verifier applied to Fig. 12 with a deliberate error.

(from the teachers) only if they believe it will somehow repay the effort. So we have to convince them of that.

Once they are reminded of the Binary-Search exercise in Lecture One, they will probably also recall that they felt uncomfortable, a bit at a loss. That is exactly what we want, because we want them to feel comfortable now, a few weeks later, to notice that difference, and then to attribute their happiness to what they have learned between then and now.

7.2 The Invariant: Begging, Borrowing, Stealing? or Maybe Donating?

In Sect. 6.1 we suggested that, for students at this level, it does not matter where an invariant comes from. In simplest terms, that is because techniques of invariant *synthesis* can't effectively be taught unless the students are already convinced that they want an invariant at all; and that will take more than a few weeks, or even a few terms. We begin here.

For the moment, we will simple give invariants to the students, without pretending that they should be able to figure them out for themselves. What is

the point of reproaching them for not being able to do a thing we know they cannot do (yet)? For the Binary-Search program, we donate the invariant[23]

$$\texttt{A[0..l)} \quad < \quad \texttt{a} \quad \leq \quad \texttt{A[h..N)}, \tag{2}$$

a piece of sorcery, a Philosophers' Stone and, without worrying (now) about where it came from, we simply show them what it can do. It converts their earlier unease into a feeling of confidence and even a little bit of power: it's a magical item, as if gained in an RPG, that gives them an ability that can set them apart.

The key issue is whether we present an invariant as a friend or an enemy. If we say "You can't develop loops properly without knowing first how to find the loop invariant." then the invariant is an enemy, trying to stop them from doing something they know perfectly well they can do without it (even if, actually, they can't).

The invariant is a friend if you say "Remember that program, Binary Search, that seemed so hard to get right? If only you'd had one of these, like (2) above, then it would have been easy." And then you go on to show how easy indeed it now is *for them*. (Showing that it is easy for you would be missing the point: more about that immediately below.) Once they understand that, then they will be motivated to learn how to find "one of these", by themselves, for other programs that they encounter later.

There will be an exhilarating moment, for you, when finally one of the students asks "How do you figure out invariants in the first place?" The spark of pleasure you will get is that they *want to know*. That's completely different from your having to convince them that they need to know.

[23] The comparison operators here operate over all elements of the structure. It's a neat bit of notation, but at some point it must be mentioned that it is not transitive (when the intermediate structure is empty). Given that most of the students in the class won't have heard of transitivity, now is probably not the time. Remember that the idea of operators' having (algebraic) "properties" itself is a higher level of awareness than most will have at this stage.

Save this, thus, until at some later stage in the class the topic of algebraic reasoning comes up naturally. It will: how do you initialise a loop whose invariant is that some variable holds the product "so far"? What's the product of an empty sequence? Why is there a "right" answer? (It's so that product is a homomorphism.) At that moment, you can suddenly remember this operator, and discuss its abstract properties too. Having a store of deferred "Did you notice?" items, like this one, is useful for time-management during your interactive-style lectures. If you look like you're going to run out of material, pull one of them out and connect it to earlier material. Spend a happy ten minutes discussing with them how to think about it properly.

Have also a few really intriguing "puzzles" that you can look at with the upper-layer students; from about one-third of the way through the course, the others will be happy to listen and they won't be bored. A good one for "What's the sum/product of...?" is "What's the determinant of an empty matrix?" Only the upper layer will know what matrices and determinants are: but they will be pleased that you recognise their extra knowledge and expertise.

7.3 Collaboration and Local Reasoning; Cardboard and Masking Tape

Now the aim is to use (2) to solve the problem posed in Sect. 2.1. Recall that the class has been divided into groups, for mentoring (Sect. 3).

Write the mentoring-group names on small pieces of paper, and bring them to class together with a small cup.[24] Pick one of the slips from the cup, thus picking a mentoring-group "randomly". (In fact, you should rig this to make sure you get a strong group. The theatre here is not in order to seem clever — rather it's to avoid any sense among the students that you might be picking on individuals. Standing in front of the class is a challenge, for them, and you defuse that by making them feel relaxed by the process and maybe even amused by way you carry it out.)

The First Group: Writing the Loop Skeleton. Call the selected group to the front, and explain to them that they will write the loop initialisation, guard and finalisation. Write on the board at the left and explain that "we" (you and they) are going to fill in the `???`'s.[25]

The `(something)`, you explain as you glance at the cup, will be another group's problem. (Nervous laughter from the audience.)

```
l,h:= ?L?,?H?
{ A[0..l) < a <= A[h..N) }
while ?G? do
{ A[0..l) < a <= A[h..N) }
   (something)
end
{ A[0..l) < a <= A[h..N) and not ?G?}
{ A[0..l) < a <= A[l..N)}
n:= ?E?
{ A[0..n) < a <= A[n..N) }
```

Now you have to lead them through the steps of filling-in the missing pieces. You will have to say most of it; and it will be hard because, at first, either they will not speak at all, or they will give incorrect answers. The point of having them at the front, even if they say little, is that afterwards what they will remember is that they contributed collectively to finding the solution (just by standing there) even if actually they contributed nothing concrete. The alternative, of pointing to the group still sitting in the audience, does not work nearly so well.

A good piece to start with is `?L?,?H?`, because it's the simplest but also a little bit unexpected. What, you ask them, will establish $A[0..l) < a \leq A[h..N)$ trivially? Drag out of them (you will probably have to) that $A[0..l) < a \leq A[h..N)$

[24] An Australian styrofoam "stubby holder" gets a good reaction, especially a brightly coloured one whose beer-brand logo will be recognised even from the back of the classroom.

[25] The use of `?L?` etc. just below is to be able to refer to them in this text. On the board, simply `???` is fine, since you can point to the one you mean.

is trivially true when `l` = 0 and `h` = `N`, because both array segments are empty. *Watch them struggle* to interpret what this might mean for the coming loop; *discourage them* from doing so; *assure them* that it is unnecessary.

Work similarly on `?G?` — what is the simplest test you could pick so that `A[0..l)` $<$ `a` $\leq$ `A[h..N)` $\wedge$ $\neg$`?G?` implies `A[0..l)` $<$ `a` $\leq$ `A[l..N)`. Again, it's "obviously" `l` $\neq$ `h`, since $\neg$(`l` $\neq$ `h`) is of course `l` = `h`, just what's needed for that implication. Again you will have to drag; again they won't really believe that you can program this way.

Finish off with `?E?`, and then ask them to sit down.

The Second Group: Sketching the Loop Body. Now take a piece of cardboard big enough to cover the work just done, and tape it over the board so that the work is no longer visible. Pick a second group's name from the cup, and invite them to come to the front. Write in the middle of the board

```
{ A[0..l) < a <= A[h..N) and l!=h}
m := ?M?
{ l <= m < h }
if ?C?
   then { l <= m < h and A[m]<a }  (something for then)
   else { l <= m < h and A[m]>=a } (something for else)
(something)
{ A[0..l) < a <= A[h..N) and h-l has strictly decreased }
```

and go through the same process with this group. Of course the `A[m]` $<$ `a` after the `then` gives the text `?C?` away; but they will hesitate because they think it can't be that simple. You are teaching them, by doing this, that it *is* that simple.

Cover-up the results when it's done.

The Third Group: Finishing Off. For the third group, write

```
{ A[0..l) < a <= A[h..N) and l <= m < h and A[m]<a }
?T?
{ A[0..l) < a <= A[h..N) and h-l has strictly decreased }
```

and

```
{ A[0..l) < a <= A[h..N) and l <= m < h and A[m]>a }
?E?
{ A[0..l) < a <= A[h..N) and h-l has strictly decreased }
```

and split the group in two. Get one half to do `?T?` and the other half to do `?E?`.

You might have to explain why it is allowed for the `A[0..l)` $<$ `a` $<=$ `A[h..N)` suddenly to reappear. Explain (remember this is *in*formal methods) that it carries through from the beginning of the loop body and you really should have written it for the previous group — but you left it out to reduce clutter: there was no assignment to any of its variables.

Once it's done, just ask them to sit. No need to cover this up.

The Fourth Group: Finishing Up. Now remind the class what has been done: three independent groups, working effectively in separate parts of the program, from the outside-in (not from first-line to last-line), and "no idea" whether the three bits will fit together, however correct they might be with respect to their individual assertions.

Then invite a fourth group to come up and remove the cardboard, and to write a single program combining the pieces already there. Tell them that they don't have to write the assertions, except for two: the invariant, just at the beginning of the loop body, and the overall postcondition, at the end of the whole program.[26]

Be sure they write it neatly, so that the whole class can read the program (they believe) they collectively have just written. Then hand back their earlier answers, from the first lecture long ago, without commenting on them:

> Really, no comment: say nothing. Nothing at all.[27]

Just wait until you see that people are looking up, starting to pack their things away, ready to go. The minute or two (in my experience) before that is usually so quiet you can almost hear the neurones firing. Make it last, let it run: you won't get many moments like it.

Then simply end the lecture at that point: no announcements or reminders. Just "See you next week."

8 Using `Dafny` to do Top-Down Development

Getting the most out of Binary Search, in Appendix D we hammer home the idea of program development "from the outside in" rather than from first statement to last. This approach, though known instinctively by many, was brought especially to prominence by Niklaus Wirth in the early 1970's.[28]

Surprising to me personally, though taught this principle as an undergraduate, was that use of `Dafny` on a program of any complexity actually forces you to do this "stepwise refinement": it is not an option. In effect, `Dafny` makes a necessity out of a virtue. And here is why.

Novices with `Dafny` (as I still am) at first expect to be able to write the complete code of a small program (i.e. Binary Search in this case) and then to "do the right thing" by including the assertions and invariants that they have

[26] Depending on the grip you feel you have on their attention at this point, you could ask them whether they found it odd to be developing a program without a specification of what it should do. But do not force this: if they look confused enough already, do not add to it.

[27] Saying nothing (aside from, obviously, "Here are your answers from Lecture 1") is important: it's right now, for a few minutes only, that they will be most receptive and will draw conclusions of their own. You cannot draw *their* conclusions for them.

Any distraction (e.g. your voice) will dilute the effect. Silence!.

[28] N. Wirth. Program development via stepwise refinement. *Communications ACM* 14,4, pp. 221–7. 1971.

worked out for themselves, perhaps on paper. Then – one push of the magical button – `Dafny` will tell you whether those assertions verify. That's the theory.

In practice, `Dafny` can sometimes give a third answer, and that in two forms. One is "Can't verify." (paraphrased) which means literally that: `Dafny` has not found your assertions to be wrong; but neither can it prove them to be right. The second form is the "whirring fan" where your laptop simply never returns from the proof effort; eventually you must interrupt the verification yourself.[29]

Once you have experienced this often enough, you realise that a step-by-step approach is much better, where you check at each stage that `Dafny` can verify what you have done so far. And – crucially – the "so far" has to be from the outside-in if you are not to run the risk ultimately of finding e.g. that your very last assertion won't verify, forcing you to start all over from the very beginning.

Appendix D shows this step-by-step approach with Binary Search. Although it looks like an awful lot of text for such a simple program, in fact each stage is made from the previous one by a cut-and-paste copy followed by alteration of a very small part. It's much more efficient than it looks.

And, most important of all: it works.

9 Fast Forward: Meeting the Real World

9.1 Motivation and Principles

Finally, we must leave (in this presentation) the Binary Search: we "fast forward" from there to the very end of the course.

The material above describes a prefix of the course, perhaps the first third. The middle of the course (not described here) charts a course through examples and techniques that aim to end with a real-world example, briefly treated in this section. The precise trajectory taken through that middle part depends on the characteristics – strengths and weaknesses – of the students in that particular year; and being able to adapt in that way requires a store of ready-made topics that can be selected or not, on the fly. That store will naturally build up as the course is repeated year by year.

But however we pass through the middle, we want to end with the feeling "This stuff really works. It's practical. And it makes a difference." The topic chosen to finish off the course, and its associated assignment, is deliberately designed to be one that, as a side effect, dispels the mystery around some part of the undergraduate computing experience.[30]

[29] The `Dafny` documentation explains why this is a risk with "SMT solvers", which is the kind of prover (Z3) that `Dafny` uses.

[30] For me, as an undergraduate, the computing courses that had the most impact, both at the time and lasting even until now, were the "de-mystifiers" — the course on compiler construction, that showed how that impossible task could be routinely done if only you looked at it the right way (and read the literature); the course on operating systems, that followed a single character from the moment you typed it in until its arrival in your program's char-buffer; and the course we would have had, had it been 10 years later, of how to program a full-screen editor.

In this case the mystery was "What *really* happens when you type in a command to a terminal window?" As we all know (but some students do not), a program is run. But how does that program access the resources it needs? And how do you use the methods here in order to make sure it does that correctly?

9.2 Programming the `cp` Command

We chose the copy command "`cp`", which must read and write files from... somewhere. To do that, it uses system calls, and it must use them correctly.

The assignment begins with an abstract model of the system calls `open`, `creat`, `read` and `write` that – crucially – are based on *real* operating system calls, complete with the nondeterminism of how much is read or written (not necessarily as much as you asked for), and the operating-style convention for end-of-file that is indicated by reading zero characters.[31]

Only this faithful modelling of what really is "out there" will give the students the conviction that they can and should apply these methods to what they will find when they really get "out there" themselves.

The assignment text is given in full in Appendix E, and is commented upon there. About one third of the students got high marks for this, more than expected. They used all these techniques that were new to them in February, but which after June they will never forget:

- Abstraction of interfaces.
- Refinement of code through increasing levels of detail.
- Invariants and static reasoning.
- Automated assistance with verifying program correctness.
- Programs that work first time, even under the most bizarre scenarios.

Imagine what Software Engineering would be like today if all students experienced those exciting accomplishments in their very first year.

Can we make it that way tomorrow?

10 Conclusions... and Prospects?

It would be tempting to conclude from the remarks in Appendix F that this course has been an outstanding success. But we can't. Whether the students enjoy a course, and whether they *think* they benefitted is quite different from whether a course actually achieved its objectives.[32] And, so far, we have no real,

[31] Calling it "creat", rather than sanitising it to something sensible, is an important part of this experience. The students should be able to find that *exact* command using "`man 2 creat`", and they should see that the behaviour described in the manual page matches their abstraction.

[32] Indeed it's often the unpopular lecturers and courses that turn out in the end to have been of the most value. Remember your time at school, what you thought then and what you realised twenty years later.

objective evidence that these are better programmers than they were before or, more significantly, than they would have been without (In-)Formal Methods.[33]

I think it's fair, though, to say this course has at least not been a failure. Why is that important? It's because so many Formal Methods courses *have* failed. That is a critical problem: the skills the students never get a chance to learn, because of those failures, are precisely the skills they need. And they need them at an early stage, to bring about a significant improvement in both their own accomplishments *and* the expectations that they have of others both now and later, of their future colleagues and employers. It's ridiculous, actually scandalous that they do not have these elementary techniques at their disposal.

As for prospects: it depends on whether this approach is *portable* and *durable*. It has already been taught by two other lecturers, who report positively. But true progress will come with an integration so inextricably into the matrix of conventional curricula that it cannot be undone when its patron moves on. Rather than being the icing on the cake, which can always be scraped off, Informal Methods must be the rising agent distributed throughout.

Invariants, assertions and static reasoning should be as self-evidently part of the introductory Computer Science curriculum as are types, variables, control structures and I/O in the students' very first programming language.

Can you help to bring that about?

Acknowledgements. The ideas in this course description distill what I have learned from many years of teaching students and of interaction with my fellow lecturers, both in Australia and, earlier, at Oxford in the UK. Some of those ideas I thought of myself; but most I have copied from colleagues whose style I admire. The key is, of course, in having consistent principles of what to copy and what to leave aside. In spite of the difficulty Formal (or Informal) Methods has had in gaining traction against more conventional courses, I have personally never felt that I lacked the support of my fellow academics in trying this material out. In earlier teaching of rigour in programming, I took a very strict approach; here (obviously) it is not strict at all. I have been encouraged by others in both cases, and I appreciate it. It is not clear yet how to combine the informal and the formal: there is still more experimenting to do. Thanks therefore to all my students, friends, colleagues and even skeptics who have allowed this exploration the space to breathe, and who have given me fair and constructive criticism that has helped to make it better. Finally, I would like to thank Zhiming Liu, Jonathan Bowen and Zili Zhang for organising the *Summer School on Engineering Trustworthy Software Systems* at which lectures based on this "users manual" were given, and for the opportunity to publish it here. I am also grateful also for the institutional support of the University of New South Wales and of NICTA, both during the running of these courses and during the preparation of this article.

[33] This is a problem with any form of teaching, of course. But it's especially an issue with Formal Methods because those who haven't "got it" don't want it and, furthermore, don't realise that actually they need it. On the other hand, those who have got it are so amazed at their new perspective that they tend to run ahead of the evidence and so discredit the whole enterprise. Formal-Methods proselytisers must play by the same rules as anyone else if they are not to be branded as zealots — which is the usual prelude to being ignored.

A Binary-Search Class Test

A.1 Teacher's Notes

1. Hand the exercise out as a *single double-sided A4 sheet* with the instructions on the front and the template on the back. The fact it's just a single sheet reinforces the feeling of informality that we want to achieve: this exercise should not be a big deal.
2. Use an audible, but gentle alarm to give them 10 min to complete it. (I used a countdown timer on a smart phone that played *2001: A Space Odyssey*'s theme *Thus Sprach Zarathustra*. It begins softly, and so doesn't startle anyone; and in the end it gets a laugh.) Use an alarm sitting on the table, rather than e.g. checking a clock or wrist-watch yourself, because that removes you from the "enforcement zone" — you become the good cop. It's the automated alarm that's the bad cop.
   (This is the same strategy used by some libraries that put their photocopiers on a timer that switches them off automatically at ten minutes before closing. Can't blame the nasty librarian, in that case.)
3. Just collect the answers, and then move on immediately. You want the students as quickly as possible to forget that they have done this test, because they'll regard Binary Search as trivial, as "old stuff" (in spite of the fact their answer is almost certainly wrong), and if you dwell on it they will start to wonder whether they have enrolled in a course that's beneath them.
4. Look at the answers only later, e.g. when you get home: you will probably be amazed. Out of a class of 42 beginning second-year Computer-Science students I found just one answer that was correct. A second one was nearly correct; and the remaining 40 (= 95 %) were quite wrong. Figure 1 in Sect. 4.1 above shows a typical example.
5. Scan them all to PDF's and mark them (at your leisure — you won't need them for a while) by annotating them. (See example marking also in Fig. 1.)
6. Remember that the point of this exercise isn't humiliation, of course. What you will do is choose your moment, somewhere further down the course, where their coding is clearly better than it was on the first day.
   At that stage you return the marked PDF's of Binary Search, and you let them see for themselves how much they have improved.

A.2 The Test Itself

The next two pages are the test itself. The second page is a typeset version of Fig. 1 before it was answered. Probably the handwritten version is more effective, since it reinforces the informality.

(In-)Formal Methods
In-class exercise

Binary Search

Student Name:

Your program is to read from three variables `A`,`N`, and `a`:

– A sorted array `A` of integers, of size `N`, indexed from 0 (inclusive) up to `N` (exclusive). Thus the array elements are

$$\texttt{A[0]} \quad \leq \quad \texttt{A[1]} \quad \leq \quad \cdots \quad \leq \quad \texttt{A[N-1]}.$$

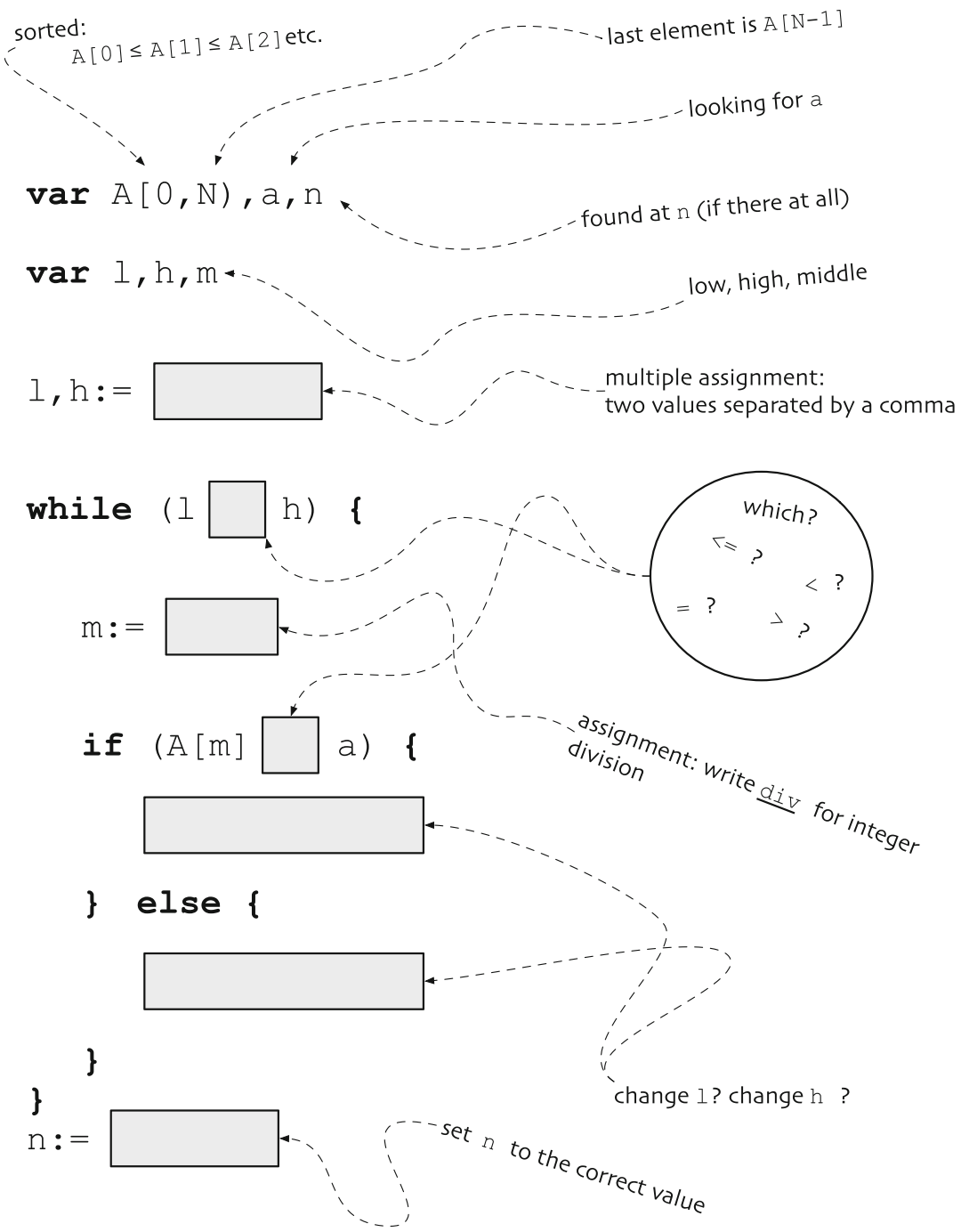
– An integer `a` to be sought in the array.

The program is to assign to a single variable `n` the least value such that `A[n]` $=$ `a`, if there an `a` in `A` at all. Otherwise `n` should be the least value such that `A[n]` $>$ `a`; and if no value in `A` is greater than `a`, then `n` should be set to `N`.

You can use other, temporary variables.

Write your binary-search program by filling in the seven boxes on the back of this sheet.

Do not change anything else.



The original version of this was handwritten. (See Fig. 1.)

B Assignment 1: Good Subsegments

The assignment follows, adapted for this article. Footnotes in italics have been added in this text; footnotes in normal font were in the original.

(In-)Formal Methods
First assignment

The longest good subsegment

B.1 Motivation, Presentation, Evaluation

In class we went in detail through the steps needed to program-up a solution to a problem inspired by one of the assignments you had last year:[34] The techniques we used for the two versions of that program were intended to be what is normal for a second-year student: indeed, they (or similar) are normal for more experienced programmers too.

Motivation. In this assignment we deal with the same programming problem, i.e. we do it for a third time. But now we will be using the techniques of this course rather than the introductory techniques of last year. The aim is that *these* techniques are what will become normal for you.

Presentation. Below (p. 43) there is a (hand-written) section "The Assignment — detail " that explains the assignment further, and contains the eight questions you should answer.[35] (The blurry green portions are model answers, to be used for marking. Note their approximate size!)

Your submission must be a single PDF file named `Ass1.pdf`, and it must have "Informal Methods Session 1 2014 Assignment 1" at the top, then your name (with your family name in capitals), and then your student number. That must be followed by your answers, clearly labelled Answer 1, Answer 2 etc. *Note that the number-of-sentence limits are mandatory.* If you write more sentences than allowed, the extra might be ignored. An example of that format is given in Fig. 10.

An easy, efficient approach is to use a text editor, i.e. with an ASCII file, and then print-to-pdf and submit that PDF. Using *Word*, *Pages* or LaTeX is *not recommended*. Although the example below fits on one page, you may take as many pages as you like — but you may not write more than the allowed number of sentences for each answer.

Remember: PDF file `Ass1.pdf` — not `.txt`, `.doc`, `.pages` or `.ps` etc. And "Informal Methods 2014 Assignment 1" at the top, then your name (with family name in capitals), and then your student number.

[34] *For these students, that was first-year introductory programming in C.*

[35] *The hand-writing of assignments is deliberate. (In the course as given, most of the material was hand-written; much of this article has been typeset from those hand-written notes just for this publication.)*

First, hand-writing is a much faster way of getting material ready when it mixed text, program code, marginal notes, arrows etc. . . .

Second, and more important, is that it sends the message that clever, glossy, beautiful typesetting is not the aim of the course: we are interested in clever, glossy, beautiful ways of thinking. Hand-written notes and on-the-board lectures reinforces that.

```
Informal Methods Session 1 2014
Assignment 1
Jack SPRAT
#1234567

Answer 1: All work and no play makes Jack a dull boy.

Answer 2: All work and no play makes Jack a dull boy. All work and
no play makes Jack a dull boy. All work and no play makes Jack a
dull boy.

Answer 3: All work and no play makes Jack a dull boy. All work and
no play makes Jack a dull boy.

Answer 4: All work and no play makes Jack a dull boy. All work and
no play makes Jack a dull boy.

Answer 5: All work and no play makes Jack a dull boy.

Answer 6: All work and no play makes Jack a dull boy. All work and
no play makes Jack a dull boy. All work and no play makes Jack a
dull boy.

Answer 7: All work and no play makes Jack a dull boy.

Answer 8: All work and no play makes Jack a dull boy. All work and
no play makes Jack a dull boy.

All work and no play makes Jack a dull boy. All work and no play
makes Jack a dull boy.

    All work and
    no play makes
        Jack a dull boy
        All work
        and no play makes Jack a
    dull boy
```

Fig. 10. Sample answer format

Evaluation. The markers will be trying to make sure that you understand the material that the questions are covering. That's a two-part process: first figuring out, from what you've written, what you are actually thinking; and second, figuring out whether you are thinking the right thing. *Make the first part easy for the markers by writing clear and concise answers.*

You will get marks only for the second part (evidence that you are thinking the right thing); but you cannot get marks there unless the markers successfully interpret the first part. So neatness is important.

Your marked answers will be emailed to you as annotated PDF's. The general marking conventions will be as follows:[36]

- Phrases the marker wants to emphasise as *good* will be highlighted in green. Green means "You got it."
- Phrases the marker wants to emphasise as not understood, or dubious, will be highlighted in yellow. Yellow means "Are you sure?" There might be a short query nearby.
- Phrases the marker wants to emphasise as *bad* (e.g. completely wrong) will be highlighted in red. Usually those will have a short explanation nearby.[37]
- Next to each question in blue will be a fraction: that's the number of marks gained (numerator) over the number of marks available (denominator) for that question.
- At the top of the assignment will be an overall mark as a blue fraction in a box, obtained by summing the individual numerators and denominators for each question. The numerator is your mark, and the denominator is the total mark for the whole assignment. (That total mark will be scaled to a percentage, later, depending on the proportion of marks this assignment represents in the whole course.)
  The "numerator/denominator" scheme makes it easy to check for marking errors: check the numerator-sum to make sure the marks given were added correctly; check the denominator-sum to make sure every question was marked and its mark included in the total.
- Any annotation numbers that are *not* fractions are merely marker's notes, and should be ignored.

[36] An example is given in Fig. 11.

[37] Often the red highlight and the short explanation will be enough for you to see what is wrong. But not always. This scheme is chosen to make things efficient *for the marker*, to reduce fatigue and so allow more real thought while marking.

Thus the marker's principles in choosing the explanation will be to keep it short, and to act as a reminder *to the marker* what the problem really was. That way the marker can check your work more thoroughly (less fatigue), but will also be able to remember what the problem was and explain it face-to-face if you ask, afterwards, for more help.

Informal Methods

Assignment 1 detail

In Week 1's lecture we did together, in class, a programming exercise based on an assignment that you had last year. The two programs we developed on the board looked like this:

```
m := 0
for b := 0 to N-3 do
    if Bad(b) then B[m], m := b, m+1
l := 0
for i := 0 to m-1 do
    if i=0 then l := l ⊔ B[0]+2
    else if i=m-1 then l := l ⊔ N-B[i]-1
    else l := l ⊔ B[i+1]-B[i]+1
```

12 lines and an auxiliary array

```
b,n,l := -1,-1,0
loop
    repeat
        n := n+1
        if n+3 > N then exit
    until Bad(n)
    l,b := l ⊔ n-b+1, n
end
l := l ⊔ N-b-1
```

12 lines no auxiliary array

They were each about 12 lines long. Our first attempt used an auxiliary array, but was very straightforward — though a bit tricky in the small details. In our second attempt we got rid of the auxiliary array by using a flowchart to help us introduce a more sophisticated control structure.

We think both programs are correct: but we aren't sure.

In this assignment you will find an answer to this programming problem that is shorter and simpler than either of the above and you will be able to explain why it is correct.

But how?

Fig. 11. Example of marked assignment with colour conventions.

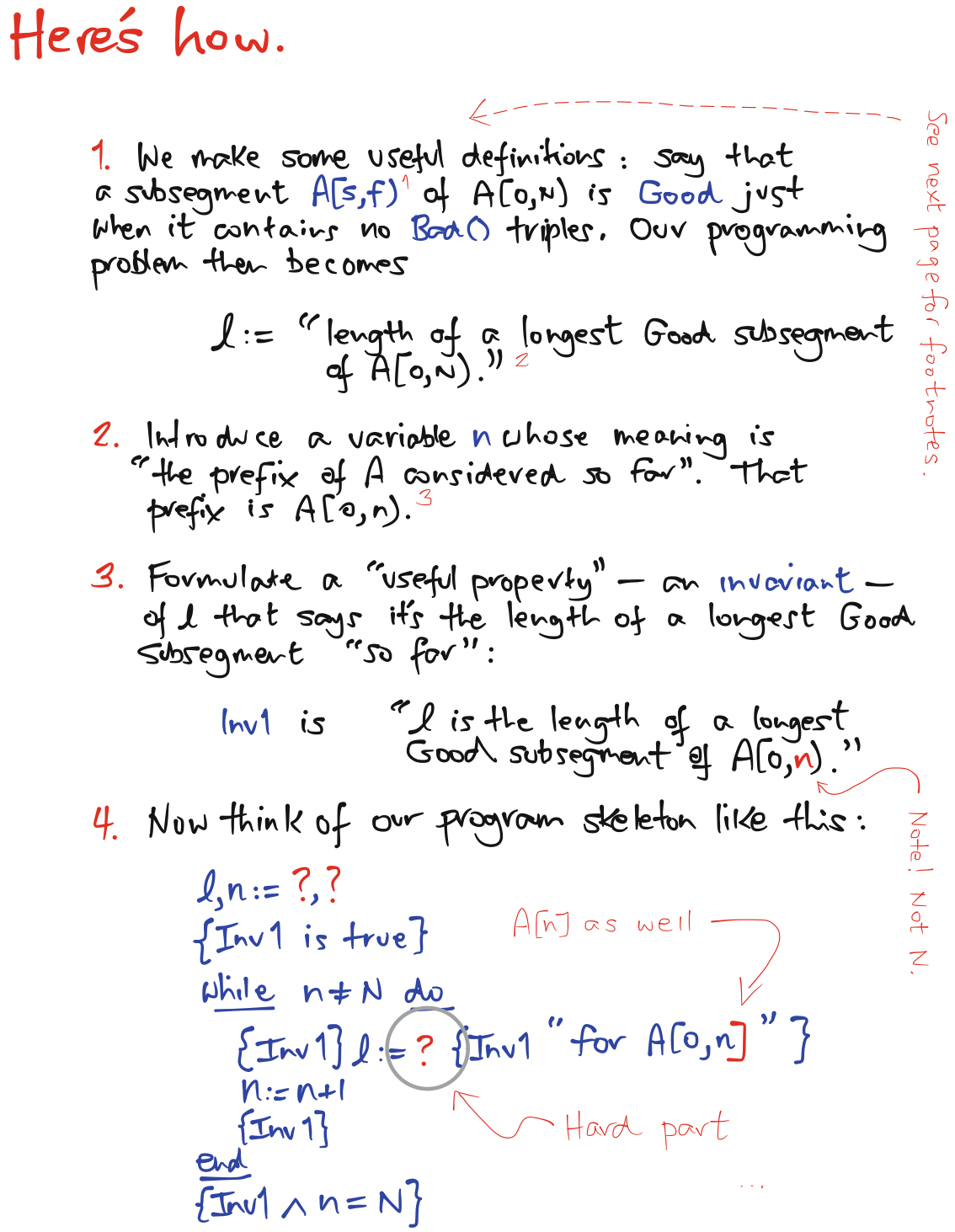


Fig. 11. (*continued*)

5. Introduce a new variable e, and a new invariant for it:

   Inv2 is "A[e,n) is a longest Good suffix of A[0,n)." [3]

6. Refine our program to this:

```
l,n,e := ?,?,?
{Inv1 ∧ Inv2}
while n ≠ N do
    {Inv2}
    if n-2 ≥ 0 ∧ ? then e := ?
    {Inv2 "for A[0,n]"}
    l := ?  // Use e to help with this.
    {(Inv1 ∧ Inv2) "for A[0,n]"}
    n := n+1
    {Inv1 ∧ Inv2}
end
{Inv1 ∧ n = N}
```

---

[1] Start is s, (one more than) finish is f.
[2] There might be several "longest" – but they will all have the same length.
[3] A[s,f) is a prefix of A[0,n) when s=0; it's a suffix when f=n.

Fig. 11. (*continued*)

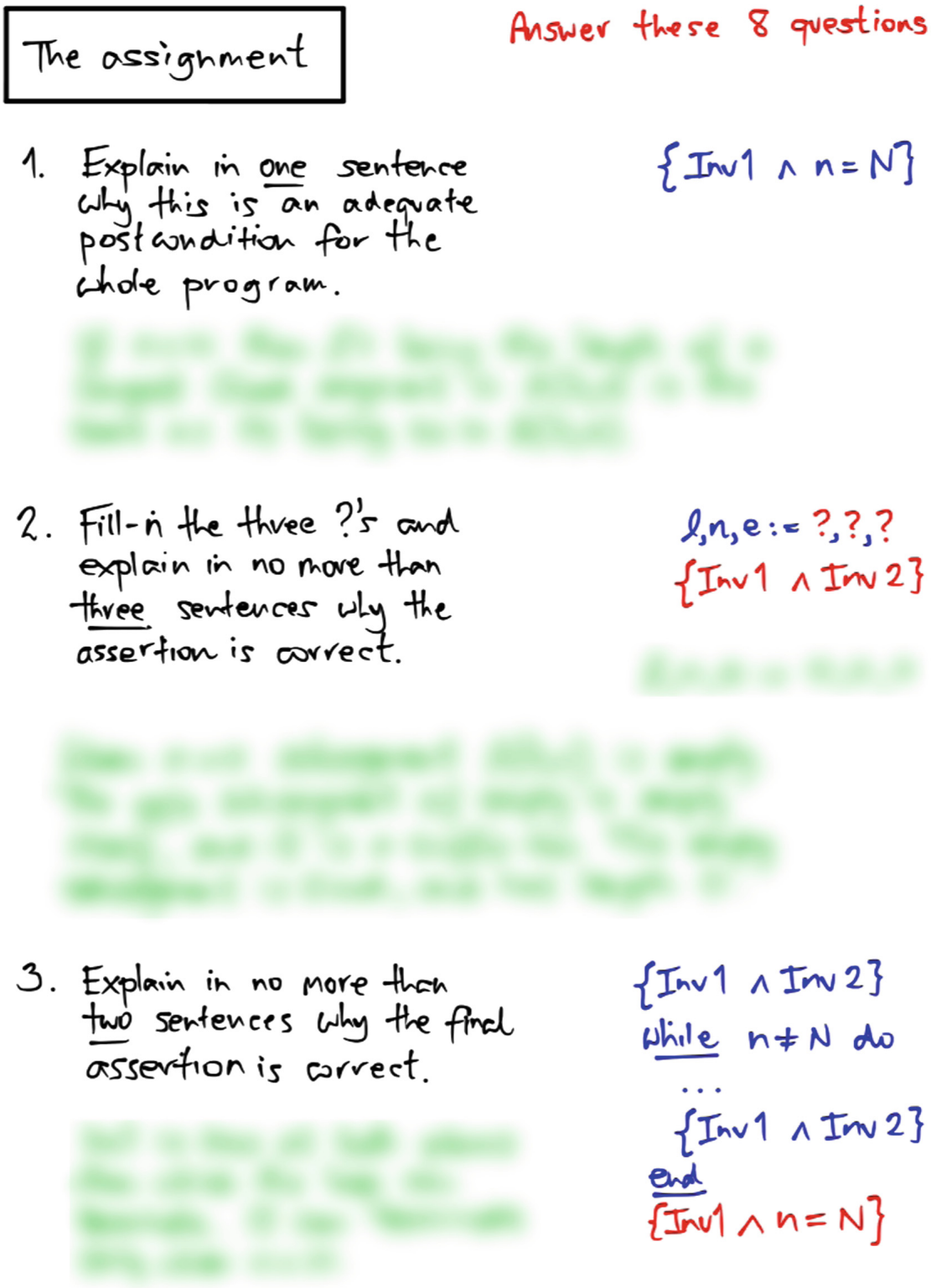


Fig. 11. (*continued*)

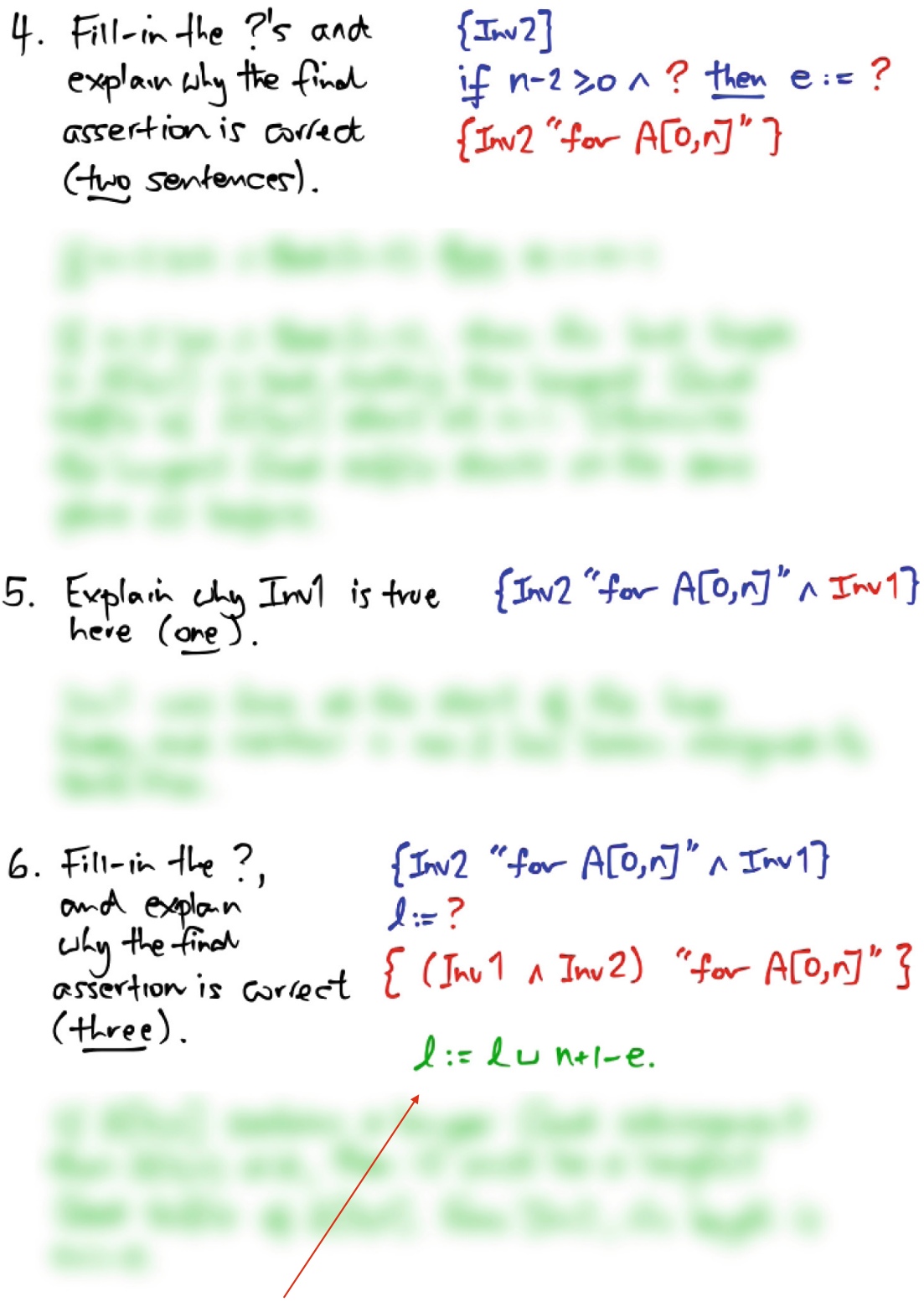


Fig. 11. (*continued*)

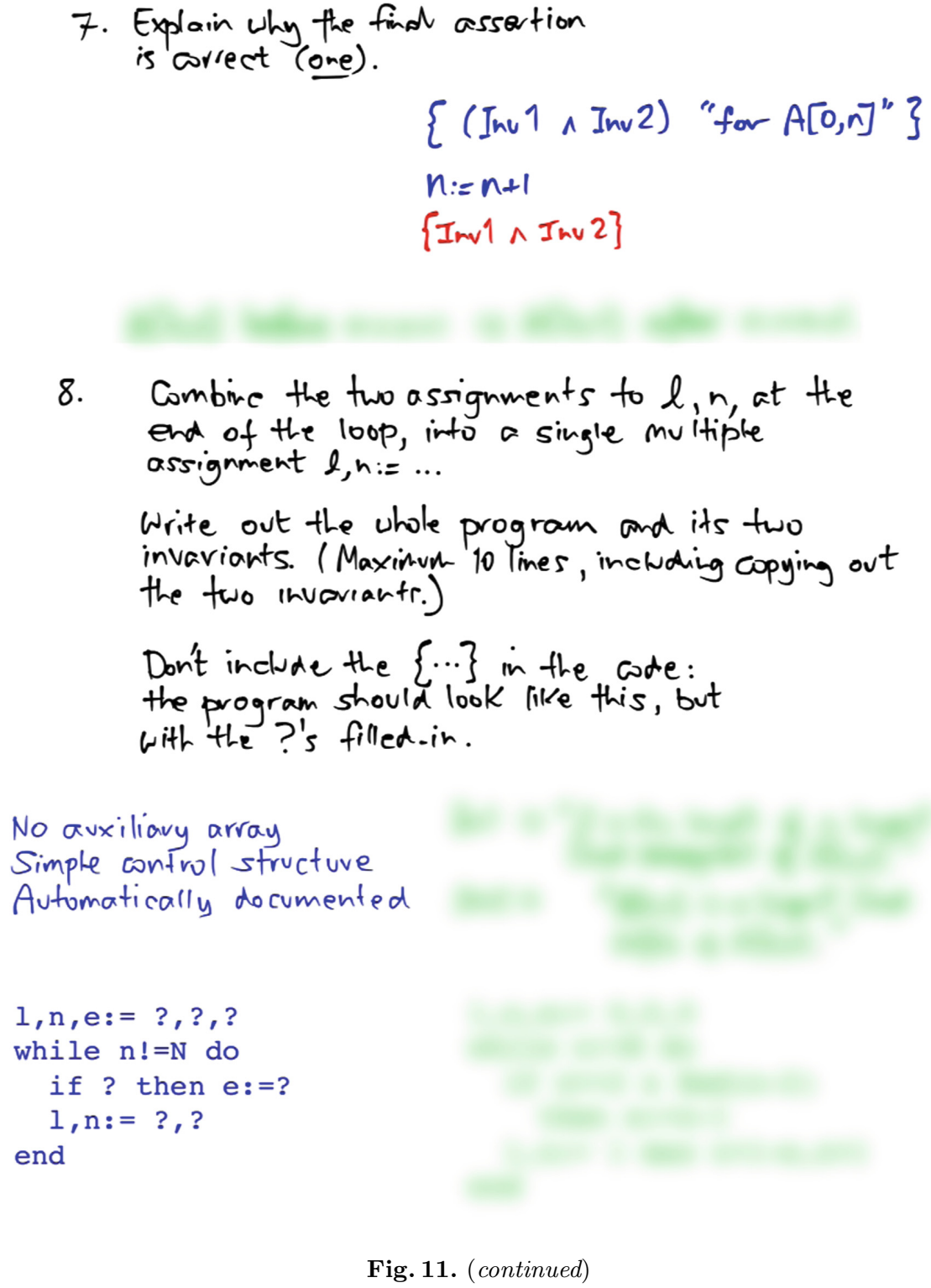


Fig. 11. (*continued*)

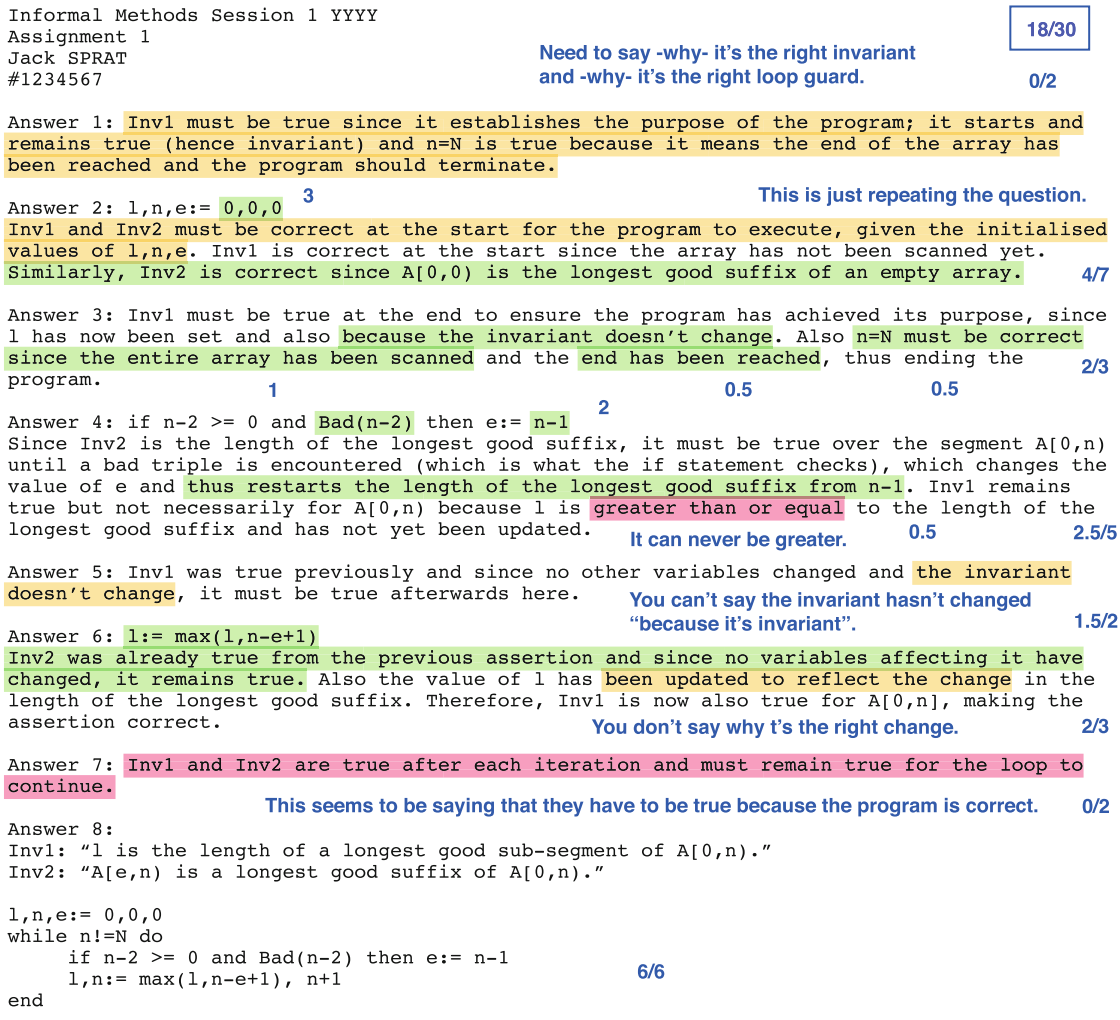


Fig. 11. (*continued*)

C Dafny Versions of Introductory Assertion-Exercises

```
method Page1() {
    {var x:int; assume x==1;  x:= x+1; assert true;}
    {var x:int; assume x==2;  x:= x/2; assert true;}
    {var x:int; assume x==3;  x:= x/2; assert true;}
    {var x:int; assume false; x:= x/2; assert x==1;}

    {var x,y:int; ghost var A,B:int; assume x==A && y==B; x:= y; assert true;}
    {var x,y:int; ghost var A,B:int; assume x==A && y==B; x:= y; y:= x; assert true; }

    {var x,y:int; ghost var A,B:int;
        assume x==A && y==B;
        x:= x+y; y:= x-y;
        assert true;
    }
    {var x,y:int; ghost var A,B:int;
        assume x==A && y==B;
        x:= x+y; y:= x-y; x:= x-y;
        assert true;
    }
    {var x,y,t:int; ghost var A,B:int;
        assume x==A && y==B;
        t:= x; x:= y; y:= t;
        assert true;
    }
    {var x,y,z,t:int; var A,B,C:int;
        assume x==A && y==B && z==C;
        t:= x; x:| x==A; y:| y==C; z:| z==A;
        assert x==B && y==C && z==A;
    }
    {var x,y:int; assume false; y:= x*x - 2*x + 1; assert y==0;}
    {var x,y:int; assume false; y:= x*x - 3*x + 2; assert y==0;}
}
```

Fig. 12. First part of assertion exercises

```
function abs(x:int):int {if (x>=0) then x else -x}
function method max(x:int,y:int):int {if (x>=y) then x else y}
function method min(x:int,y:int):int {if (x<=y) then x else y}
var minInf:int; var maxInf:int; // Infinities.

method Page2() {
    {var x:int; ghost var A:int;
        assume x==A;
        if (x<0) {x:= -x;}
        assert true;
    }
    {var x,y:int; ghost var A,B:int;
        assume x==A && y==B;
        if (x<y) {x,y:= y,x;}
        assert true;
    }
    {var x,y:int; ghost var A,B:int;
        assume x==A && y==B;
        if (x<=y) {x,y:= y,x;}
        assert true;
    }
```

Fig. 13. Second part of assertion exercises

```
    {var s,x,y,z:int;
        s:= x;   assert true;
        s:= s+y; assert true;
        s:= s+z; assert true;
        assert true;
    }
    {var p,x,y,z:int;
        p:= x;   assert true;
        p:= p*y; assert true;
        p:= p*z; assert true;
        assert true;
    }
    {var s,x,y,z:int;
        s:= 0;   assert true;
        s:= s+x; assert true;
        s:= s+y; assert true;
        s:= s+z; assert true;
        assert true;
    }
    {var p,x,y,z:int;
        p:= 1;   assert true;
        p:= p*x; assert true;
        p:= p*y; assert true;
        p:= p*z; assert true;
        assert true;
    }
    {var m,x,y,z:int; var minInt:int;
        m:= x;            assert m==x;
        if (m<y) {m:= y;} assert m==max(x,y);
        if (m<z) {m:= z;} assert true;
        assert true;
    }
    {var m,x,y,z:int; var minInt:int;
        assume minInt<=x;
        m:= minInt;       assert m==minInt && minInt<=x;
        if (m<x) {m:= x;} assert true;
        if (m<y) {m:= y;} assert true;
        if (m<z) {m:= z;} assert true;
        assert true;
    }
}
```

Fig. 14. Third part of assertion exercises

```
function maxSeq(A:seq<int>,n:int): int
    reads this; // To access minInf.
    requires 0<=n<=|A|;
{ if n==0 then minInf else max(A[0],maxSeq(A[1..],n-1)) }

ghost method maxLast(A:seq<int>,n:int)
    requires 0<=n<|A|;
    ensures max(maxSeq(A,n),A[n])==maxSeq(A,n+1);
{ if (n!=0) { maxLast(A[1..],n-1); } }

function sumSeq(A:seq<int>,n:int): int
    requires 0<=n<=|A|;
{ if n==0 then 0 else A[0]+sumSeq(A[1..],n-1) }

ghost method sumLast(A:seq<int>,n:int)
    requires 0<=n<|A|;
    ensures sumSeq(A,n+1) == sumSeq(A,n)+A[n];
{ if (n!=0) { sumLast(A[1..],n-1);} }

function prodSeq(A:seq<int>,n:int): int
    requires 0<=n<=|A|;
{ if n==0 then 1 else A[0]*prodSeq(A[1..],n-1) }

ghost method prodLast(A:seq<int>,n:int)
    requires 0<=n<|A|;
    ensures prodSeq(A,n)*A[n]==A[0]*prodSeq(A[1..],n);
{ if (n!=0) { prodLast(A[1..],n-1);} }
```

Fig. 15. Fourth part of assertion exercises

```
method Page3() {

    // Maximum of x,y,z.
    {var m,x,y,z:int;
        m:= minInf;
        m:= x; assume minInf<=x; // Property of minInf assumed.
        assert m==x;
        m:= max(m,y); assert m==max(x,y);
        m:= max(m,z); assert m==max(x,max(y,z));
        assert m==max(x,max(y,z));
    }

    // Minimum of x,y,z.
    {var m,x,y,z:int; var maxInt:int; assume maxInt>=x;
        m:= maxInt;
        m:= x; assume maxInf>=x; assert m==x; // Property of maxInf assumed.
        m:= min(m,y); assert m==min(x,y);
        m:= min(m,z); assert m==min(x,min(y,z));
        assert m==min(x,min(y,z));
    }
```

Fig. 16. Fifth part of assertion exercises

```
    // Maximum of A[0,N).
    {var A:seq<int>; var m,n:int;
        m,n:= minInf,0;
        while (n!=|A|)
            invariant 0<=n<=|A|;
            invariant m==maxSeq(A,n);
        {   maxLast(A,n); // Lemma needed: definition is foldr but program is foldl.
            m,n:= max(m,A[n]),n+1;
        }
        assert m==maxSeq(A,|A|);
    }

    // Sum of A[0,N).
    {var A:seq<int>; var s,n:int;
        s,n:= 0,0;
        while (n!=|A|)
            invariant 0<=n<=|A|;
            invariant s==sumSeq(A,n);
        {   sumLast(A,n);
            s,n:= s+A[n],n+1;
        }
        assert s==sumSeq(A,|A|);
    }

    // Product of A[0,N).
    {var A:seq<int>; var p,n:int;
        p,n:= 1,0;
        while (n!=|A|)
            invariant 0<=n<=|A|;
            invariant p==prodSeq(A,n);
        {   prodLast(A,n);
            p,n:= p*A[n],n+1;
        }
        assert p==prodSeq(A,|A|);
    }

    // Maximum of A[0,N) when N>=1.
    {var A:seq<int>; var m,n:int; assume |A|>=1;
        assume minInf<=A[0]; // Only place where minInf's being smallest is necessary.
        m,n:= A[0],1; // Start with the first element instead of minInf.
        while (n!=|A|)
            invariant 0<=n<=|A|;
            invariant m==maxSeq(A,n);
        {   maxLast(A,n);
            m,n:= max(m,A[n]),n+1;
        }
        assert m==maxSeq(A,|A|);
    }
}
```

Fig. 17. Sixth part of assertion exercises

D Stepwise Development in Dafny

In this appendix we give explicitly the stages through which one develops "from the outside in" a verified implementation of Binary Search. The virtues of this were explained in Sect. 8.

In Fig. 18 we have the *specification* of Binary Search given in the `requires`/`ensures` clause(s) just after the method header.

Then the method body sets `n` to an arbitrary value nondeterministically, via `n:= *` and, immediately afterwards, with `assume` statements forces that arbitrary value to be one that satisfies the very same `ensures` clauses as are above. Thus this "implementation" simply achieves the postcondition by setting `n` to a value that... satisfies the postcondition.

This extreme caution is brought about by experience: sometimes `Dafny` cannot prove that a universal quantification implies itself: in broad terms, that is because its general strategies for proving universal quantifications are sometimes confounded by simple instances. Here we are making sure at the very beginning that this won't happen to us here.

And what do we do if `Dafny` fails even this simple first step? In that case, we look for another way to specify what we want the program to do.

```
// Step 0: Write the body as a single step that
//         satisfies the requires/ensures trivially.

method BinarySearch0(A:seq<int>,a:int) returns (n:int)
    requires forall i,j:: 0<=i<j<|A| ==> A[i]<=A[j];
    ensures 0<=n<=|A|;
    ensures forall i:: 0<=i<n   ==> A[i]<a;
    ensures forall i:: n<=i<|A| ==> a<=A[i];
{   n:= *;
    assume 0<=n<=|A|;
    assume forall i:: 0<=i<n   ==> A[i]<a;
    assume forall i:: n<=i<|A| ==> a<=A[i];
}
```

Fig. 18. BinarySearch0.dfy

In Fig. 19 we make our first refinement step, preparing to replace the simple assignment by a loop that keeps most of the postcondition as an invariant, but splits one conjunct off to be established by the negation of the loop guard. We introduce the variables `low` and `high`, and anticipate a loop whose effect is to make them equal.

Note this does not mean that, when you do this yourself, you have to type in the whole program again. Copy the method `BinarySearch0`; paste the copy in and rename it to `BinarySearch1`; then alter its body. Then verify them both together. (For a larger program, you might use separate files to avoid constant verifying of the earlier steps; but for a small program like this one, it's so fast it makes no difference.)

```
// Step 1: Re-write the body as the specification of a loop
//         that maintains the first three asumptions as a invariants
//         and establishes the fourth assumption on loop-exit.

method BinarySearch1(A:seq<int>,a:int) returns (n:int)
    requires forall i,j:: 0<=i<j<|A| ==> A[i]<=A[j];
    ensures 0<=n<=|A|;
    ensures forall i:: 0<=i<n   ==> A[i]<a;
    ensures forall i:: n<=i<|A| ==> a<=A[i];

{   var low,high:= *,*;
    assume 0<=low<=high<=|A|;                    // Inv1
    assume forall i:: 0<=i<low    ==> A[i]<a;    // Inv2
    assume forall i:: high<=i<|A| ==> a<=A[i];   // Inv3
    assume low==high; // Negation of loop-guard
    n:= low;
}
```

Fig. 19. BinarySearch1.dfy

In Fig. 20 we insert a loop skeleton: its invariant and guard are as advertised in the previous step (Fig. 19). But at this stage, with `decreases *`, we indicate that we are not yet interested in proving that the loop terminates. (Experiment by commenting out the `decreases` clause.)

```
// Step 2: Add a loop that satisfies the post-condition given in the previous step,
//         with the loop body to be filled-in.

method BinarySearch2(A:seq<int>, a:int) returns (n:int)
    requires forall i,j:: 0<=i<j<|A| ==> A[i]<=A[j];
    ensures 0<=n<=|A|;
    ensures forall i:: 0<=i<n   ==> A[i]<a;
    ensures forall i:: n<=i<|A| ==> a<=A[i];
    decreases *; // Declare that the method (for now) is allowed not to terminate.

{   var low,high:= 0,|A|;
    while (low!=high)
        invariant 0<=low<=high<=|A|;
        invariant forall i:: 0<=i<low    ==> A[i]<a;
        invariant forall i:: high<=i<|A| ==> a<=A[i];
        decreases *;
    {   low,high:= *,*; // arbitrary values that re-establish the invariant
        assume 0<=low<=high<=|A|;
        assume forall i:: 0<=i<low    ==> A[i]<a;
        assume forall i:: high<=i<|A| ==> a<=A[i];
    }
    n:= low;
}
```

Fig. 20. BinarySearch2.dfy

In Fig. 21 we add the variant function that will guarantee loop termination. In this case it is that the variables `low` and `high` must move strictly closer together. First their current values are captured, and then the "set such that" statement requires that the difference has decreased.

With this done, a `decreases high-low` will be accepted by `Dafny`. But in many cases (including this one), `Dafny` can guess the loop variant itself: provided you code actually decreases some variant, `Dafny` will often figure out what variant that is.

```
// Step 3: Make progress towards termination by reducing the search,
//         in the loop body, to a strictly smaller portion of the sequence.
//         Note the "decreases *" is no longer needed.
//         Dafny figures out "decreases high-low" for itself.

method BinarySearch3(A:seq<int>,a:int) returns (n:int)
    requires forall i,j:: 0<=i<j<|A| ==> A[i]<=A[j];
    ensures 0<=n<=|A|;
    ensures forall i:: 0<=i<n   ==> A[i]<a;
    ensures forall i:: n<=i<|A| ==> a<=A[i];

{   var low,high:= 0,|A|;
    while (low!=high)
        invariant 0<=low<=high<=|A|;
        invariant forall i:: 0<=i<low    ==> A[i]<a;
        invariant forall i:: high<=i<|A| ==> a<=A[i];
        // decreases high-low; // Dafny figures this out for itself.
    {   var oldLow,oldHigh:= low,high;
        low,high:| high-low < oldHigh-oldLow;
        assume 0<=low<=high<=|A|;
        assume forall i:: 0<=i<low    ==> A[i]<a;
        assume forall i:: high<=i<|A| ==> a<=A[i];
    }
    n:= low;
}
```

Fig. 21. BinarySearch3.dfy

In Fig. 22 we implement the strategy "Choose some new variable `mid` to lie between `low` and `high`, and use it to change one or the other of those two variables." We don't know which, yet; but the decrease of the variant forces us even so to choose assignment right-hand sides that will have that strict-decrease effect. (Experiment by replacing the `mid+1` with just `mid`.)

Note the nondeterministic `if` statement whose both-`true` guards allow either of its two branches to be executed. At the moment, the `assume` statements further below live up to their name: they "assume" that the nondeterminism in the `if` statement has been resolved correctly, i.e. in a was that preserves the invariant. What will force us to code that up into "real" tests is that the `assume`'s are not allowed to be in our final program: in the *Refinement Calculus* it would be said that they are "not code".[38]

```
// Step 4: Introduce binary chop, so that termination is guaranteed.
//         But how do we chop?

method BinarySearch4(A:seq<int>, a:int) returns (n:int)
    requires forall i,j:: 0<=i<j<|A| ==> A[i]<=A[j];
    ensures 0<=n<=|A|;
    ensures forall i:: 0<=i<n   ==> A[i]<a;
    ensures forall i:: n<=i<|A| ==> a<=A[i];

{   var low,high:= 0,|A|;
    while (low!=high)
        invariant 0<=low<=high<=|A|;
        invariant forall i:: 0<=i<low    ==> A[i]<a;
        invariant forall i:: high<=i<|A| ==> a<=A[i];
    {   var mid:| low<=mid<high; // 1824: Set mid to anything that satisfies.
        if { // 1826: Which do we choose? Figure that out in the next step.
            case true => low:=  mid+1; // 1826.
            case true => high:= mid;   // 1826.
        }
        // 1824: The earlier assumption here is no longer necessary.
        assume forall i:: 0<=i<low    ==> A[i]<a;
        assume forall i:: high<=i<|A| ==> a<=A[i];
    }
    n:= low;
}
```

Fig. 22. BinarySearch4.dfy

[38] Carroll Morgan. *Programming from Specifications. Prentice Hall* 1994. Ralph-Johan Back, Joachim von Wright. *Refinement Calculus: A Systematic Introduction.* Springer 1998.

In Fig. 23 we have replaced the `true` guards with actual tests; and, having done that, we can remove the `assume`'s. Rather than `Dafny`'s *assuming* that they hold, it can now prove that they do.

Note however that we still have a nondeterministic statement choosing the value of `mid`. And yet the program is correct. What that means is that this program works *however* we choose `mid` strictly between `low` and `high`. That is, the "binary chop" step, which we are about to implement, is a matter of efficiency, not of correctness.

```
// Step 5: Make careful choices so that the loop invariant is maintained,
//         and the assumptions can be removed.

method BinarySearch4(A:seq<int>, a:int) returns (n:int)
    requires forall i,j:: 0<=i<j<|A| ==> A[i]<=A[j];
    ensures 0<=n<=|A|;
    ensures forall i:: 0<=i<n   ==> A[i]<a;
    ensures forall i:: n<=i<|A| ==> a<=A[i];

{   var low,high:= 0,|A|;
    while (low!=high)
        invariant 0<=low<=high<=|A|;
        invariant forall i:: 0<=i<low    ==> A[i]<a;
        invariant forall i:: high<=i<|A| ==> a<=A[i];
    {   var mid:| low<=mid<high;
        if {
            case A[mid]<a  => low:=  mid+1;
            case a<=A[mid] => high:= mid;
        }
        // All assumptions gone.
    }
    n:= low;
}
```

Fig. 23. BinarySearch5.dfy

Finally, in Fig. 24 we choose `mid` to lie somewhere approximately in between `low` and `high` and, finally, we have the traditional Binary Search.
Note though that by choosing `mid` differently (yet still in between), we end up with a linear search.

```
// Step 6: Choose binary chop specifically, by picking mid somewhere in the middle.

method BinarySearch6(A:seq<int>, a:int) returns (n:int)
    requires forall i,j:: 0<=i<j<|A| ==> A[i]<=A[j];
    ensures 0<=n<=|A|;
    ensures forall i:: 0<=i<n   ==> A[i]<a;
    ensures forall i:: n<=i<|A| ==> a<=A[i];

{   var low,high:= 0,|A|;
    while (low!=high)
        invariant 0<=low<=high<=|A|;
        invariant forall i:: 0<=i<low    ==> A[i]<a;
        invariant forall i:: high<=i<|A| ==> a<=A[i];
    {   var mid:= (low+high)/2;
        if { // We strengthen the guards.
            case A[mid]<a  => low:=  mid+1;
            case a<=A[mid] => high:= mid;
        }
    }
    n:= low;
}

// But we could also do a linear up-search by choosing mid:= low .
method Search6Up(A:seq<int>, a:int) returns (n:int)
    requires forall i,j:: 0<=i<j<|A| ==> A[i]<=A[j];
    ensures 0<=n<=|A|;
    ensures forall i:: 0<=i<n   ==> A[i]<a;
    ensures forall i:: n<=i<|A| ==> a<=A[i];

{   var low,high:= 0,|A|;
    while (low!=high)
        invariant 0<=low<=high<=|A|;
        invariant forall i:: 0<=i<low    ==> A[i]<a;
        invariant forall i:: high<=i<|A| ==> a<=A[i];
    {   var mid:= low;
        if {
            case A[mid]<a  => low:=  mid+1;
            case a<=A[mid] => high:= mid;
        }
    }
    n:= low;
}

// Or a linear down-search by choosing mid:= high-1. Try it!
```

Fig. 24. BinarySearch6.dfy

E Assignment 4: Real-World Programming

The assignment follows, adapted for this article. Footnotes in italics have been added in this text; footnotes in normal font were in the original.[39]

(In-)Formal Methods
Fourth assignment

Real-World Programming: A circular I/O buffer

E.1 Why "Real World" Programming?

With the programming techniques taught in this course, you will be able to develop code more quickly than before, and it will have far fewer errors than is normal in the *IT* industry.[40] And your code will be more easily maintained as well.

For that to happen, you must learn to apply our "perfectionist" techniques in an imperfect world, where systems have imprecise or incomplete specifications, and where most programs are too large and detailed to allow assertion-based reasoning by hand alone. *We need tools to help.*

The UNIX-style copy command (`cp` simplified) is our example, using (almost) the real UNIX system calls. We abstract the system calls' behaviour as `Dafny requires/ensures` specifications; and we will transliterate our `Dafny` programs into actual *C* code, and run it.

The remaining vulnerabilities are mainly that we have no real assurance that we have specified the UNIX calls correctly; and we have no assurance either that our transliteration into *C* of our own code did not itself introduce errors. The more-than-compensating strengths are that the algorithm is verified, that it

[39] In fact it was not possible to prepare for the assignment a fully "circular" buffer in the 2014 version of this course: getting the `Dafny` proof to go through proved too difficult to have prepared beforehand. But it was completed after the course, and the buffer will be fully circular next time.

[40] Is "Information Technology" a euphemism to disguise the fact that writing programs actually requires disciplined thinking and rigorous practices, more than just running spreadsheets, databases and word-processors? If so, it's good news for some: the people who *can* apply discipline and rigour, when it's required, will stand out from the pack. They'll be more valued, will have important projects and earn higher salaries. The rest of us will depending on them.

can easily be changed without introducing errors, and that its documentation is enforced (and, if necessary, updated) automatically — and all of these because *`Dafny` won't verify it otherwise.*

E.2 UNIX-Style Copying with a Single Buffer

In this section we take our first steps towards developing real code that copies standard input to standard output: it will be a scaled-down version of the UNIX command `cp`. For the moment, however, we abstract from UNIX by modelling the standard input, standard output and in-memory buffer all three as (`Dafny`) *sequences* rather than as actual files (input and output), or as a buffer-array with a pointer into it. That simplifies our initial sketch of the copying algorithm, so that we can see its overall structure.

The input-file sequence is fixed in value, modelling that in the real-life situation it is not being changed (by something else) as we read it; what does change as it is read is an *offset pointer* into the file that indicates the position from which the next read will occur. (UNIX stores that pointer as part of a "file descriptor" structure.) That pointer is initialised to 0 because the file is to be read from its beginning. The output-file sequence begins empty, modelling that we create a new file (rather than appending to an existing one); it is gradually extended by the buffer-loads of data that are successively written to it.

The effect of all this abstraction can be seen in the different answers required for the two questions marked by stars ★ below.[41] They refer to our `Dafny` code (Fig. 26) and its corresponding code in *C* (Fig. 25).

1. A simple *C* program for copying standard input to standard output is given in Fig. 25. It uses a single buffer of size `BUF_SIZE` which size, in your case, you will set to digits 1–3 of your student number. (In the example, the student number is `z7654321`.) It reads at most `IN_MAX` bytes of data at at time; you will set `IN_MAX` to digits 2–3 of your student number.
   Take the code of Fig. 25 and edit `BUF_SIZE` and `IN_MAX` to reflect your own student number as above: make it into a file `cpA.c`. Compile it using the command `cc cpA.c -o cpA`. Run it by typing `./cpA < cpA.c`, and check that it correctly copies its own text to the standard output.
   Now change `BUF_SIZE` to 0, then re-compile and re-run your program.
   ★ *What error message do you get, and when?*
2. In Fig. 26 appears the `Dafny` program from which the *C* program of Fig. 25 was transcribed.[42] Make it into a file `cpA.dfy` and edit its constants as above; verify it using the command `dafny cpA.dfy`. (It should get no errors.) Now change the `bufSize` parameter to 0, and re-verify it.
   ★ *What error message do you get, and when?*

[41] Since the questions are based on an actual assignment, references such as "you" etc. are to the students.

[42] That is, the code in Fig. 26 was written *before* the code of Fig. 25. Unfortunately, Fig. 26 is normally not made at all.

```
// For read() and write().
#include <unistd.h>

#define STD_INPUT 0     // File descriptor for standard input.
#define STD_OUTPUT 1    // File descriptor for standard output.

#define BUF_SIZE 765
#define IN_MAX 65      // Maximum read length: requires 0<IN_MAX<=BUF_SIZE.

int main() {
    // Initialisation of STD_INPUT and STD_OUTPUT is done for us.

    char buf[BUF_SIZE]; // Note: Characters, not integers.
    int eof= 0;

    while (!eof) {
        int count= read(STD_INPUT,&buf,IN_MAX);
        if (count==0) eof= 1; else write(STD_OUTPUT,buf,count);
    }
}
```

Fig. 25. *C* code `cpA.c` transcribed from Fig. 26.

E.3 Unit Testing: Harnesses and Stubs

In the `Dafny` code of Fig. 26 there are "simulations" of the environment in which our copy method is intended to run. The `read(...)` system-call is simulated by

```
if (inputPos==|inputData|) { eof:= true; } else {
    var count:nat:| 0<count<=inMax && inputPos+count<=|inputData|;
    inputPos,buf:= inputPos+count,inputData[inputPos..inputPos+count];
}
```

where the declaration and initialisation of `count` is *nondeterministic* — the symbols `:|` mean "...is given a value such that." And so the read system-call guarantees to set `count` to a natural-number value satisfying

```
0<count<=inMax && inputPos+count<=|inputData|
```

but, beyond that, it makes no guarantee at all about which value that will be.

Similarly, the `write(...)` system-call is simulated by

```
outputData:= outputData+buf;
```

where + is sequence concatenation.

In both cases these simulations can be compared with the informal descriptions given in the actual UNIX man-pages. (You can enter the UNIX commands `man 2 read` and `man 2 write` if you want to see them.) The code-fragments above are called *stubs* because they are not the real system calls. Similarly, the

```
// Input file, output file are sequences of integers.
var inputData:seq<int>,inputPos:nat;
var outputData:seq<int>;

method cpA(inMax:nat,bufSize:nat) modifies this;
    requires 0<inMax<=bufSize;
    ensures outputData==inputData;
{   inputPos:= 0; // Open inFile for reading.
    outputData:= []; // Open outFile for writing.

    var buf:seq<int>:=[]; var eof:=false;
    while (!eof) // Can't test file for EOF directly.
        invariant inputPos<=|inputData|;
        invariant outputData==inputData[0..inputPos];
        invariant eof ==> inputPos==|inputData|;
        decreases |inputData|-inputPos + (if eof then 0 else 1);

    {   // UNIX-style read() returns EOF-indicator only -after- you fail to read.
        if (inputPos==|inputData|) { eof:= true; } else {
            // Read "some" data into buf: set count "such that"...
            var count:nat:| 0<count<=inMax && inputPos+count<=|inputData|;
            inputPos,buf:= inputPos+count,inputData[inputPos..inputPos+count];

            // Write all data out from buf.
            outputData:= outputData+buf; // Here "+" is sequence concatenation.
        }
    }
}

method main() modifies this; {
    cpA(65,765);
}
```

Fig. 26. `Dafny` code `cpA.dfy` for simple UNIX-style read/write loop.

method-call `cpA(65,765)` is a simulation of what is using (rather than used by) our copy method: it is called a *harness*.

In both cases – in conventional program development – the simulations, the stubs and harnesses, are supposed to provide a great variety of behaviours typical of what the unit under test will encounter in practice, focussing particularly on the so-called "edge cases" where coding errors are likely to have occurred: when index-variables are smallest, or largest; when structures are empty, or full etc. Making an effective test environment requires lots of work.

With a modern software development method (such as we are now using) this work is much reduced and yet is more effective, as we now show.[43]

[43] Compare for example *our* read-stub to a traditional one in which nondeterminism is not available: then the stub would probably return one of three values for the number of characters read: the least, the greatest and one somewhere in between.

Our approach here in effect tests *all* values, not just three of them.

```
method cpB(input:Input,output:Output,inMax:nat,bufSize:nat) modifies this;
    requires input!=null && output!=null; modifies input,output;
    requires 0<inMax<=bufSize; // This takes the place of the harness.
    ensures output.data==input.data;

{   input.open(); // Open inFile for reading.
    output.creat(); // Open outFile for writing.

    var buf:seq<int>; var eof:=false;
    while (!eof)
        invariant input.pos<=|input.data|;
        invariant output.data==input.data[0..input.pos];
        invariant input.eof ==> eof;
        invariant eof ==> input.pos==|input.data|;
        decreases   (|input.data|-input.pos)
                  + (|input.data|-|output.data|)
                  + (if eof then 0 else 1);
    {   if (input.pos==|input.data|) { eof:= true; } else {
            var data:=input.read(inMax); //1036: See Fig. 28.
            buf:= data;
            output.write(buf); //1036.
        }
    }
}
```

A note on documentation: the arbitrary number `1036` links several comment-points together; only one of them has text. The choice of number is supposed to be random, and is in fact just the time of day I typed it in: that reduces the risk of "randomly" choosing a number more than once.
Doing multiple-relevance comments this way means you have to write the comment itself only once, and it automatically applies consistently in all the other places even if you update the comment text in that one place.
If you find such a comment `NNNN: Something.` then a search with a text editor for `NNNN.` finds all the (other) places it applies. And if you find a comment `NNNN.` then a search for `NNNN:` will find the relevant comment text. All this keeps everything in step with a minimum of effort.

Fig. 27. "Unit test" `cpB.dfy` of `method cpB`, no stubs or harnesses: Part I.

First, we can remove the harness altogether: its function is taken over by the `requires` clause(s) of the copy method itself, which describes *all* of the things a harness for this program is allowed to do, including the edge-cases automatically. (A conventional harness can only implement *some* of those things, in general). In Fig. 27 the harness is no longer there; and your student number is no longer necessary for selecting "random" block-sizes.

Second, we can remove the stubs by replacing them by a *specification* of what they do; again, this describes *all* of their possible behaviours, not just some of them. In Fig. 28 there is no code for reading or writing.

```
class Input { // 0902.
    var data: seq<int>; // The data in the (input) file.
    var pos:nat; // The current reading position.
    var eof:bool; // Whether end-of-file has been indicated.

    // Open file for reading.
    method open() modifies this;
        ensures pos==0 && !eof && data==old(data);

    // Read up to len from current position.
    method read(len:nat) returns(justRead:seq<int>) modifies this; //1036.
        requires pos<=|data|; ensures pos<=|data|; // Datatype invariant.
        ensures data==old(data) && pos>=old(pos);

        requires len!=0; // Can't ask to read 0.
        requires !eof; // Can't ask to read if EOF is already signalled.

        ensures old(pos)!=|data| ==> justRead!=[];
        ensures justRead==data[old(pos)..pos];
        ensures |justRead|<=len;
        ensures eof <==> old(pos)==|data|;
}

class Output {
    var data: seq<int>; // The data in the (output) file.

    // Create a new, empty file.
    method creat() modifies this;
        ensures data==[];

    // Append to file.
    method write(toWrite: seq<int>) modifies this; //1036.
        requires |toWrite|!=0; // Can't ask to write nothing.
        ensures data==old(data)+toWrite;
}
```

These two classes take the place of the stubs; note they contain *no* executable code.

Fig. 28. "Unit test" `cpB.dfy` of `method cpB`: Part II.

★ *Our* `read(...)` *specification replaces the traditional "read-stub". Describe very briefly in words the intention of the following four postconditions of the specification of* `read` *in Fig. 28:*

(a) `ensures old(pos)!=|data| ==> justRead!=[];`
(b) `ensures justRead==data[old(pos)..pos];`
(c) `ensures |justRead|<=len;`
(d) `ensures eof <==> old(pos)==|data|;`

★ *Describe very briefly in words what (bad things) the* `read` *method could do to its calling copy method if each the following three postconditions of* `read` *in Fig. 28 had separately been left out, i.e. in a case, for each one, where the* `read` *method violates it:*

(a) `ensures old(pos)!=|data| ==> justRead!=[];`
(b) `ensures justRead==data[old(pos)..pos];`
(c) `ensures |justRead|<=len;`

★ *Explain the purpose of the term* `+ (if eof then 0 else 1)` *in the* `decreases` *clause of the copy method in Fig. 27.*

★ *Explain the different purposes of the Booleans* `input.eof` *within the class* `Input` *and* `eof` *within the main program* `cp`*.*

E.4 The Buffer as Array; Reading/Writing in Blocks

The sequence abstraction for `buf` is very convenient for specification, but sequences are expensive to implement in real applications — and that is why it is is not used in the actual UNIX `cp` program.[44] Instead, an array (in *C*) is allocated for the buffer; and so we will model that now with an array in `Dafny`.

Because an array (unlike a sequence) does not move around in memory once allocated, for efficiency reasons, our use of it will have to become more sophisticated: we will have start- and end pointers `s,e` into `buf` that indicate the part of it `buf[s..e]` that contains actual data. When the pointers get to the end of the buffer, we will reset them to the beginning.[45]

Having such pointers allows furthermore that input and output might have different preferred block-sizes, and that might be important depending on what the actual input- and output devices are. For example, if the input device prefers to deliver data in blocks of 100 elements but the output device prefers to receive data in blocks of 150 elements, again for efficiency reasons, then we should read *twice* into the buffer (200 elements) before we write once (leaving $200-150=50$ elements behind, which we should try not to write until we have read more).

[44] Sequences are expensive because they support so many convenient operations: concatenation, subsequencing etc. Arrays are much faster, but have fewer native operations.

[45] The circular-buffer version of this is more sophisticated.

Our more sophisticated `Dafny` code is given in Fig. 29; notice that it is written in the multiple-guard while-loop style, which is much less error-prone than the usual form.[46] (The updated stub-specifications are given in Fig. 30.)

★ *For* `cpC.dfy` *in Fig. 29, supply code for the missing portions according to the following hints.*

(a) `can write` Put a Boolean test here that ensures there is some data to write.
(b) `set n to how much to write` Put a "such that" assignment here to `n` that is as liberal as possible consistent with correctness of the program, but is not more than `outBlock`.
(c) `update s,e` Set `s,e` to the correct (new) values.
(d) `can read` Put a Boolean test here that ensures that an *EOF* indication has not already been received, and that there is room in the buffer for more data.
(e) `set n to how much to read` Put a "such that" assignment here to `n` that is as liberal as possible consistent with correctness of the program, but is not more than `inBlock`.
(f) `update e` Set `e` to the correct (new) value.

★ *Based on* `cpC.dfy`*, make a file* `MYcpC.dfy` *according to your answers above. Then verify it with* `Dafny`*.*

The *C* code corresponding to Fig. 29 is given in Fig. 31, where the `Dafny`-style multiple-guard `while` has been transliterated into a *C*-style

```
while (1) {if··· else if ··· else break;}.
```

What's especially interesting is that in doing that transliteration we have had to decide which `if` comes first, so to speak the "read `if`" or the "write `if`". Our choice in Fig. 31 has taken the second option: it gives priority to writing in the sense that if both reading and writing are possible, then writing will be

[46] If a loop **do** $G_1 \to S_1 \,[\!]\, G_2 \to S_2$ **od** were recoded as as a conventional while-loop, it would become

$$\textbf{while } G_1 \vee G_2 \textbf{ do } \quad \textbf{if } G_1 \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \quad \textbf{od},$$

which has the disadvantages that (1) it must repeat G_1 and (2) it is not obvious from the text what assertion holds at the beginning of S_2. (It is of course $(G_1 \vee G_2) \wedge \neg G_1$, that is G_2; but that might not be obvious if $G_1 \vee G_2$ itself has been simplified into some other form.)

So what we have written in Fig. 31 corresponds instead to

$$\begin{array}{l}\textbf{while } \mathit{true} \textbf{ do}\\ \quad \textbf{if} \quad\;\; G_1 \textbf{ then } S_1\\ \quad \textbf{else if } \; G_2 \textbf{ then } S_2\\ \quad \textbf{else break}\\ \textbf{od},\end{array}$$

which avoids both of those disadvantages. It still encodes a priority, however, favouring G_1 over G_2. To do the opposite, we would swap first two interior `if`-branches.

```
method cpC(input:Input,output:Output,inBlock:nat,outBlock:nat,bufSize:nat)
    modifies this;
    requires input!=null && output!=null; modifies input,output;
    requires 0<bufSize && 0<inBlock && 0<outBlock;
    ensures output.data==input.data;
{   input.open();
    output.creat();

    var buf:= new int[bufSize];
    var s:nat,e:nat:= 0,0; // Only buf[s..e] contains valid data.

    var eof:= false;
    while
        invariant input.data==old(input.data);
        invariant input.pos<=|input.data|;
        invariant s<=e<=bufSize;
        invariant e==bufSize ==> s!=e;
        invariant input.data[..input.pos]==output.data+buf[s..e];
        invariant input.eof ==> eof;
        invariant eof ==> input.pos==|input.data|;
        decreases   (|input.data|-input.pos)
                  + (|input.data|-|output.data|)
                  + (if eof then 0 else 1);

    {   case  can write   =>
            var n:nat;   set n to how much to write
            ghost var data0:= output.data;
            var count:= output.write(buf,s,n); // Write from buf starting at s.
            assert output.data==data0+buf[s..s+count];
            s:= s+count;
            if (s==bufSize) {   update s,e    }

        case  can read    =>
            var n:nat;    set n to how much to read
            ghost var buf0,pos0:= buf[..],input.pos;
            var count:= input.read(buf,e,n); // Read into buf starting at e.
            assert input.data[..pos0+count]
                ==input.data[..pos0]+input.data[pos0..pos0+count];
            if (count==0) { eof:= true; } else {   update e    }
    }
}
```

Fig. 29. Code `cpC.dfy` with array-buffer: reading/writing in blocks, Part I.

chosen. That is, the code of Fig. 31 reads only when it can't write; even though the original `Dafny` code does not have that property. Technically that represents a "resolution of specification-time nondeterminism".

But we could have put the `if`'s the other way, as in Fig. 32, in which case instead the code would write only when it couldn't read. *Both* Figs. 31 and 32 are valid transliterations of Fig. 29, and we can choose whichever we want

```
class Input {
    var data: seq<int>,pos:nat,eof:bool;

    method open() modifies this;
        ensures pos==0 && !eof && data==old(data);

    method read(buf:array<int>,p:nat,len:nat) returns(count:nat) modifies this;
        requires pos<=|data|; ensures pos<=|data|;
        ensures data==old(data) && pos>=old(pos);
        requires len!=0 && !eof;
        modifies buf; requires buf!=null && p+len<=buf.Length;
        ensures old(pos)!=|data| ==> count!=0;
        ensures count<=len;
        ensures buf[p..p+count]==data[old(pos)..pos];
        ensures eof <==> old(pos)==|data|;
        // Change only the part of buf into which we have read.
        ensures buf[..p]==old(buf[..p]) && buf[p+count..]==old(buf[p+count..]);
}

class Output {
    var data: seq<int>;

    method creat() modifies this;
        ensures data==[];

        method write(buf:array<int>,p:nat,len:nat) returns(count: nat) modifies this;
                requires buf!=null && p+len<=buf.Length;
                requires len!=0;
                ensures 0<count<=len; // Different from UNIX.
                ensures data==old(data)+buf[p..p+count];
}
```

The UNIX manual page (`man 2 write`) does not state that write is guaranteed to write more than zero bytes; but in our specification above, we have added that feature. Otherwise we could not be able to prove that our copy code terminates.

Fig. 30. Code `cpC.dfy` with array-buffer: reading/writing in blocks: Part II.

depending on implementation issues (like which of reading or writing should be given priority in our particular application).

★ *Based on* `cpC.c`, *make* `MYcpcC.c` *by filling-in the missing portions of Fig. 31 found in your verified* `MYcpC.dfy`. *Convert the* `Dafny` *such-that assignments to* `n` *to deterministic assignments in C that make* `n` *as big as possible consistent with the such-that's. Fill in the constants according to your student number.*

★ *Compile* `MYcpC.c` *with* `cc MYcpC.c -o MYcpC.c` *and run it using the command* `./MYcpC <yourFile 2>/dev/null` *on a test file of your choice.*[47]

E.5 Refinement of Multiple-Choice Iterations

In Footnote 48 we saw the general form

[47] The `2>/dev/null` merely hides the output of the `fprintf`'s.

```
#include <unistd.h>
#include <stdio.h>
#define STD_INPUT 0     // File descriptor for standard input.
#define STD_OUTPUT 1    // File descriptor for standard output.

#define BUF_SIZE  76543 // The first five digits of your student number.
#define IN_BLOCK  7654  // The first four digits of your student number.
#define OUT_BLOCK 765   // The first three digits of your student number.

#define MIN(a,b) ((a)<(b)?(a):(b)) // C has no built-in MIN.

int main() {
    char buf[BUF_SIZE]; int eof= 0;
    int s= 0; int e= 0;
    fprintf(stderr,"BUF_SIZE=%d, IN_BLOCK=%d, OUT_BLOCK=%d.\n\n",
        BUF_SIZE,IN_BLOCK,OUT_BLOCK);

    while (1) {
        if  can write    {
            int n=  set n to how much to write (maximum allowed)

            int count= write(STD_OUTPUT,&buf[s],n);
                       s+= count;
            if (s==BUF_SIZE)  update s,e
            fprintf(stderr, " Write: asked for %d, wrote %d and now s,e,e-s=%d,%d,%d.\"
                          , n,count,s,e,e-s);

        } else if  can read    {
            int n=  set n to how much to read (maximum allowed)
            int count= read(STD_INPUT,&buf[e],n);
            if (count==0) eof= 1; else  update e   ;
            fprintf(stderr, "Read: asked for %d, read %d and now s,e,e-s=%d,%d,%d.\n"
                          , n,count,s,e,e-s);

        } else break;
    }
}
```

Fig. 31. *C* code `cpC.c` corresponding to Fig. 29, with some `fprintf`'s.

$$\textbf{do } G_1 \to S_1 \;[\!]\; G_2 \to S_2 \textbf{ od} \tag{3}$$

of a multiple-guard iteration. It executes by first evaluating the *guards* G_1, G_2; if both are false, the loop terminates. If exactly one of G_1, G_2 is true, then the corresponding statement S_1, S_2 is executed. But if *both* G_1, G_2 are true, then *either* of S_1, S_2 can be executed. This is known as *nondeterminism*.

Nondeterminism might at first seem to make reasoning about programs harder. But – in this form at least – it actually makes it easier. What an alternative $G_i \to S_i$ says is that "if S_i is executed in a state where G_i holds, then it is guaranteed to maintain the invariant and to decrease the variant." It's that simple.

```
...
int main() {
...
    while (1) {
        if  can read  {
            ...
        } else if  can write  {
            ...
        } else break;
    }
}
```

Fig. 32. *C* code corresponding to Fig. 29, but with priority for reading. (`cpC.c`)

When the guards do overlap in this way, then it's possible in a refinement to alter the guards slightly in order to take implementation concerns into account. For example if we wanted a refinement in which the same overall effect was reached but, during the execution, the first guard was executed in favour of the second whenever both were ready, then we could use the modified loop

$$\textbf{do}\ G_1 \to S_1 \,[\!]\, G_2 \wedge \neg G_1 \to S_2\ \textbf{od},$$

in which the second guard G_2 has been strengthened to include "unless G_1". It is a refinement of (3). And the complementary $\textbf{do}\ G_1 \wedge \neg G_2 \to S_1 \,[\!]\, G_2 \to S_2\ \textbf{od}$ is also a refinement of (3), but one where we have given the priority to S_2 instead of to S_1.

The general refinement rule is that

$$\textbf{do}\ G_1 \to S_1 \,[\!]\, G_2 \to S_2\ \textbf{od} \quad \sqsubseteq \quad \textbf{do}\ G_1' \to S_1 \,[\!]\, G_2' \to S_2\ \textbf{od} \tag{4}$$

when $G_1 \vee G_2 \equiv G_1' \vee G_2'$ and $G_1' \Rightarrow G_1$ and $G_2' \Rightarrow G_2$. In words, the conditions are that the two loops have the same overall guard, and that whenever S_i is executed in the more-refined loop, it must have been permissible to have executed it in the less-refined loop.

In our read/write loop of Fig. 29 in fact we have actual non-determininism whenever it is both possible to read (because there's some space left in the buffer) and to write (because there's some data in the buffer). In the *C* code of Fig. 31 we resolved that nondeterminism in favour of writing; and in Fig. 32 we resolved it in favour of reading.

★ *By examining the guards you added to Fig. 29, write down exactly, in terms of the program variables, the conditions in which nondeterminism is present, that is when both reading and writing are possible.*

★ *Alter your guards so that writing has priority over reading whenever a full* `outBlock` *elements can be written, but otherwise the priority is not determined.*

★ *Use the refinement rule in* (4) *to check that your new loop, with its limited*[48] *output priority, is a refinement of the original.*

Code up your altered read/write method in the style of Fig. 31 *(with the* `fprint`*'s included); call the file* `cpD.c`*. When you resolve any remaining nondeterminism (i.e. as you transliterate the multiple-guard loop into the form* `if`–`else if`–`else`*), give the priority to reading.*[49]

★ *Compile it, and run it on the input file* `Ass4In`,[50] *capturing its* `fprint` *output in a file* `Ass4Out` *using the commands*

```
cc cpD.c -o cpD
./cpD <~se2011/Ass4/Ass4In >/dev/null 2>Ass4Out
```

E.6 How This Assignment Will Be Marked

1. The written answers to "why this" and "why that" will be checked. They should be very short, and precise.
2. The `Dafny` codes will be checked, by running `Dafny` on them, to see whether they verify.
3. The *C* codes will be checked to see whether they appear correctly to transliterate the `Dafny` codes. They won't be marked for style (otherwise), since conceptually the `Dafny` is our source code, and the transliterations are our assembly code. We don't usually mark the assembly-code output of a compiler for style (unless we are evaluating the compiler itself).
4. The test-file `Ass4In` was specially constructed to allow errors easily to be seen, and it will be used to check for run-time errors. But what kind of errors will it find? If the `Dafny` verified, the program should be correct as far as functionality goes. Thus this check helps to uncover transliteration errors; but it also captures cases where the nondeterminism was not resolved in the way the question required.

Written answers will be marked in the usual way, with partial credit available for answers that are partly correct. However...

Full credit for `Dafny` code is given only if it is the same structure (essentially) as the (supplied) code from which it is supposed to be derived, and has no verification errors when checked with `Dafny`. If it *does* have verification errors, then only partial credit is given. However if the Dafny code does not verify

[48] By "limited" we mean as above that output has priority only when it can write a full `outBlock` elements; otherwise (above) the choice between reading/writing remains nondeterministic.

[49] Thus in this question you are resolving the *remaining* nondeterminism in favour of reading, and you are doing it by choosing the way you transliterate the `Dafny` while-loop into *C*.

[50] This was a huge file, so large that the students could not tell just by looking what correct program output should be. Thus their confidence had to be based on the verification. They had to submit the `fprintf` output only: just the blocksizes read and written were checked.

completely (that is, if it gets *even just a single error*), then no credit is available for the other two (remaining) files `MYcpC.dfy` and `Ass4Out`. That is, if your `Dafny` doesn't verify then your *C* code gets zero, even if it looks right. Even if it *is* right. Our *C* code cannot be guesswork.

If the `Dafny` code does verify completely, i.e. with zero errors, then the remaining answers are marked simply as either *correct* or *incorrect* (i.e. either full credit, or none). To get full credit, the *C* code should compile without error and must accurately copy the marker's (not merely the student's) test-data file. For the `printf` outputs, the output you get must be byte-for-byte what is expected (based on the student number and the test file `Ass4In`). If it differs in even a single byte, it will be marked zero.

F Student Feedback: At Least Not a Failure

The comments below are verbatim, collected anonymously via UNSW's teaching-evaluation web-interface just after the course has ended. Any material about the lecturer personally, rather than about the course, has been omitted however. Otherwise they are complete.[51]

As remarked in Sect. 10, there is no sense in which student feedback in the short term can establish that a course has been successful: it indicates only what they thought of its style, delivery and content. In spite of that, for formal-methods related material especially, it's encouraging that none of these students felt the course was pointless or irrelevant.

F.1 From Second-Years in 2014

Best Features:

- The interactive and hands on approach of the teaching in the course, as well as the content itself.
- The content is relevant, lectures are interesting, tutorial is interesting.
- The assignments were an amazing learning experience, the lectures were helpful and so were the tutorials. The assignments eased you in and allowed me to learn a lot of the content while doing it.
- [This] course [was] interesting, challenging and overall awesome. The amount of content I learnt this semester in this course was huge. The structure of the course allowed a smooth transition for all students and the mentor sessions along with extensive notes provided allowed students to practice many examples before tackling the assignments.
- This course let us know how to design and plan[ning] to build a software which is really cool and interesting.
- [The] class room style teaching.

[51] That is, *all* the comments are included, not just the favourable ones. That is to give a fair picture of good vs. bad: there's no "cherry picking".

- Very easy to understand and interesting [...] Assignments were incredibly fascinating and were well thought out. Notes also aided in reinforcing knowledge.
- Encouraged thinking outside the box.
- Teaching methodology [...] and course content. The choice of tutors were mostly good. Structure of the course (except Project Management).
- [The] content was extremely interesting and useful.
- Clear [...] The relevance of content was made clear from the start to beginning. Very interesting course.
- Examples. Organisation.
- Relevance to past real world examples. Assignments were not testing as much but rather assessing through learning using ideas taught throughout the course. They were thought out and well constructed to make you think.
- [...] Interesting content. Challenging.

Suggested Improvements:

- Splitting the 3 h lecture slot into 2 time slots.[52]
- Nothing, it's already the best.
- Removing Assignment 4 and giving project management component an additional 2 weeks. I really enjoyed Assignments 1–3 but Assignment 4 seems somewhat repetitive (a summary of the other assignments in some ways). Also the invited guest speaker towards end of semester was also very interesting, I would love to see more in the future.
- No improvements needed.
- Iterating why it is important. I was not aware of how important proving correctness was until the guest speaker from NICTA visited.[53]
- More defined learning areas.
- Revise what Project Management requires and the aim of it.
- Better mentor sessions.
- Conducting better mentor sessions.
- Cover more content. We went a little slow at times.

F.2 From Fourth-Years in 2012

Best Features:[54]

- Encouraged a different way of thinking than I was used to, that makes much more sense. These concepts should be taught in first year.
- Teaching thinking method that I never use before.
- Subtlety, concurrency ..... No EXAM ..... Assignments.... Requirement to think in a different way.
- Everything. The content is amazing, very well structured, has a lot of interesting material and is well explained.

[52] Three 1-hour slots per week is best; but time-tabling forced one 3-hour slot in 2014.
[53] The guest speaker was June Andronick from NICTA.
[54] The course was not given at all in 2013.

- Good approach, but perhaps more appropriate for introducing people to programming properly than as a course numbered 6xxx which implies it should be taken later in the degree.[55]
- This course radically changed the way I view programming, and has certainly improved my programming skills immensely. This course should be compulsory for all Computer Science students, or have its content integrated into first year.
- The interesting problems and concepts presented. No other course makes you think like this, or presents methods of solving problems as this course does. Well structured and interesting topics. Assignments really helped to solidify lecture material.
- The subject matter, the way it was structured, and the way it was taught. One of the best courses offered at CSE.
- Fascinating content. Assignments were pitched at a good progression of difficulties. In general, the course was run extremely well.
- Made the abstract, theory side of computing very accessible, with clear practical applications. Not sure what determines the lecture times, but the one-hour-per-day split was very good.[56]

Suggested Improvements:

- It honestly could not be.
- Nothing actually. Maybe more students?
- Some lecture notes could [have] been released a week before the lectures so that you could have a better understanding of the content beforehand.
- Nothing.
- Having it in a room where you can hear from further back than the first row!
- Not much, maybe some of the concepts were a little too challenging, and that coupled with the new techniques of problem solving we were learning really sent your head into a spin. Although you set it out extremely logically, it can still become rather overwhelming.
- Not changing it in the slightest.

F.3 From Fourth-Years in 2011

Best Features:

- The subject matter [...] the interaction between the students and the lecturer, in short all of it.
- It changed my approach to programming in a way that I was then able to do better in my other subjects as well as in teaching programming.
- Very useful technique, impressive lecture.
- Class participation in lectures, interesting material.

[55] This comment is of course the thesis of this whole article.

[56] The courses in 2010–2 were taught in three 1-hour slots per week. In 2014 timetabling for second-years forced that to change to one 3-hour slot.

Suggested Improvements:

- Maybe some preview notes?
- More consistent marking. Although I now recognise and appreciate the difficulty in marking the assignments, disparity between marks for making the same mistakes seems odd.

F.4 From Fourth-Years in 2010

Best Features:

- Giving a thorough grounding to good programming techniques in computing through static reasoning.
- It was awesome, limited size group. I've benefited more from that course than from any other at UNSW. The best feature definitely is the informal style of the course, the high interaction between the lecturer and the students, the timetable (three times one hour instead of three hours in a row in most courses), and the fact that the teached material is actually quite a rare stuff.
- Interesting.
- Everything!!!
- The course content was really well thought out and prepared.
- The course content was extremely interesting.

Suggested Improvements:

- Better course notes.
- Hard to say really. I can't think of a simple way to improve it. But that doesn't mean there is no room for improvement!
- More time to do more stuff.
- A more concrete assessment schedule.