Is Bidirectionality Important?

Perdita Stevens(✉)

School of Informatics, University of Edinburgh, Edinburgh, UK
perdita.stevens@ed.ac.uk

Abstract. Bidirectional transformations maintain consistency between information sources, such as different models of the same software system. In certain settings this is undeniably convenient – but is it important? I will argue that developing our ability to engineer dependable bidirectional transformations is likely to be crucial to our ability to meet the demand for software in coming decades. I will discuss some of the work that has been done so far, including some I've had a hand in, and what challenges remain.

1 Introduction

It is usually held [15] that whenever the title of an article is given the form of a boolean question, the answer to the question should be **No**. For, if I believe the answer to be **Yes**, why have I not titled this paper, and the talk it accompanies, "Bidirectionality is Important"?

Any reader who started at the abstract, or indeed, who knows me, will guess that – pace Betteridge [4] and Hinchcliffe [12] – this maxim does not apply here. In this case, the question is a marker for some complexity that I wish to discuss. What do we, or should we, mean by "important"? What is "bidirectionality"? Where are we going? What is it all for?

In order to tackle these ridiculously large questions, let me start with some philosophical background.

2 What Does "Important" Mean to Humans?

To say, to an audience of researchers in software engineering, that a topic is "important", is to say that it deserves research attention, because paying it this attention may eventually have a positive impact on the practice of software engineering.

This involves not only a prediction ("may eventually") but also a value judgement ("positive"). Indeed, whenever we decide to classify an issue as "important", we are making a value judgement. Such a judgement may involve an appeal to an individual's philosophy or religion; but some things are certain.

1. We make the judgement using our brain, which has evolved, over many millions of years of natural selection, to enhance our individual survival by prioritising what to pay attention to. What an individual, of any species, needs to pay attention to depends on the nature of the species. As humans:

A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 1–11, 2018.
https://doi.org/10.1007/978-3-319-92997-2_1

2. We are members of a social species. That is – like a variety of animals from lions to mole-rats to bees – we tend to form cooperative groups which we will call societies. Sometimes it is important to be aware that this fact underlies the things we think of as worst, as well as those we think of as best, about humanity. *A* society has a variety of relations between its individuals, and there are most likely also individuals who are considered to be entirely outside it. We cooperate, but not always, and not perfectly.
3. We are members of a species that uses tools. Indeed, like primates that strip leaves off twigs to fish for insects, we *make* tools, and like beavers, we engineer our own ecosystem. We take deliberate action to change our environment from how it is, to how (we think) we would prefer it to be.

Were any of these things not the case, this conference could not exist. What, though, are the implications for our subject matter?

Software systems, just like the insect-fishing twigs, are tools. The purpose of developing a software system is to modify something about the environment, broadly conceived (e.g. in that people are part of one another's environments). Because we are a social species, this may not always be immediately apparent to someone who is working on the software. Somebody, somewhere, considers the modification to the environment, that developing the software system effects, to be beneficial; but that someone may be socially a long way removed from those who develop it.

For example, I am writing this paper using a text editor named Emacs, and I consider the environment in which I can do this much better than one in which I would have to write it using a typewriter. The earliest editors were developed by people who wanted to use them themselves, as well as to make them available to others. When I was developing software to help my employer chase up non-paying customers, though, the benefit to me was only indirect (I got paid), and the people most affected by the existence of that software certainly did not consider it beneficial.

Thus, it may not be evident to one person that another person considers a piece of software to be having a beneficial effect. It is worse than that: neither within, nor between, societies are our interests perfectly aligned. Therefore, wherever more than one person is affected by some software, there is the potential for conflict, which must somehow be resolved. Indeed, all human conflict is about reconciling competing interests. On a small scale – between people who are socially close – we do this informally, using our faculties of empathy and perspective taking. On a larger scale, where these faculties prove insufficient, we resort to making agreements, often explicit treaties, or *contracts*, expressing desired relationships between things different people care about.

In the technical context, we shall return to this in Sect. 6. But first, let us look more closely at computer systems.

3 What Do Computer Systems Do?

Evolutionarily speaking, the important thing that computer systems do is to affect the environment of one or more human beings. They may do this rather directly, like computer games, providing stimuli that affect the brain's sensations of pleasure. They may give someone more, or less, money. They may enable communication between several humans. They may be instrumental in the growing of food, or in the transportation of a human from Edinburgh to Toulouse.

It is not important that computer systems implement computable functions from some inputs to some output. That is merely part of how they do the important things. As soon as we move from mathematics to software engineering, we have to move up and out.

The attraction of abstracting what our tools do as mathematical functions is that these are easy to think about. However, *effects* such as those discussed are, compared to pure functions, difficult to reason about, and because we do not (so far) have a relationship of empathic trust with our computer systems, we need to be able to reason about what they do. The paradigm of software structured as objects, which have (encapsulated) state, behaviour and identity [5], and communicate by a predefined collection of messages that have limited capacity to be parameterised, fits easily into a human mind. It is an admittedly limited approach to managing just the effect of statefulness, but it is hard to argue with the overwhelming success of object orientation. (Managing state and other effects using monads is more powerful, but jokes about the plethora of object tutorials would fall flat [11].[1])

If reasoning about effects is hard, but software must have effects, what gives? To date, it has been relatively straightforward to step over the gap between inside and outside a computer system without noticing it. We abstract the gap in terms of sensors and actuators, or in terms of a user interface in which all the important interfacing is done by the user. The interaction between a computer system and a human is typically of the same order of complexity as that between a human and a company they do business with: it can be governed by a relatively impoverished contract.

The time in which that gap has been easy to ignore, though, may now be coming to an end. Artificial intelligence is talked of by the general public again, and concern is rising about the way in which computer agents can be disguised as human ones – even though, so far, the disguises (of nefarious Twitter bots for example) are rather crude. The common feature is that the effects these computer systems may have on our environment – social, political or physical – are not predictable; given that we also do not have reason to trust them, the result is fear. As the interface between computer systems and human individuals and societies becomes more fractal, the interaction becomes more like that between humans, in how we must reason about it, than we are used to; this happens

[1] See also https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/,https://wiki.haskell.org/Monad_tutorials_timeline.

even if the computer side does not have properties we are ready to label as intelligence.

In order to get to the implications of this for software engineering, let us think about how we handle predicting or trusting one another. When we talk about our own behaviour, we talk about having good *habits*; interpersonally, the specification each person thinks they should obey is called their *principles*; our attempts to ensure that other people behave in a principled way towards is us called having healthy *boundaries*.

What does it mean for a software system to have good habits, principles, boundaries? How can we possibly manage such concepts? Before we return to these matters let us consider, more concretely, the problems of today's software engineering and how models, and bidirectionality, fit in.

4 What Are the Problems of Software Engineering?

Does software engineering have problems? The term "software crisis" seems to have been coined at the NATO Conference on Software Engineering in 1968. From then on, peaking about 1990, it was a commonplace that we had a software crisis: that is, an inability to develop enough software, with high enough dependability, to meet demand.

Then, use of this phrase began to fall away, and we came to take for granted that software systems, even large ones, even safety critical ones, can indeed be developed predictably enough – at least without hugely more difficulty than, say, tram systems [6]. Concern turned to the supply of talented developers needed to meet the ever-increasing demand for software; to attempt to tackle this, we see educational initiatives aimed at persuading young children to learn to code and consider software engineering as a career.

Throughout, the "pain" experienced in software engineering has been in two main areas: requirements, and maintenance. Curiously, I suggest, the innovations that have led to the demise of the "software crisis" have not focused on either of these areas. Books could be written on what those innovations are: my point here is that what we have *not* seen is a revolution in how requirements are gathered and managed, nor radically new ways to handle maintenance of software systems. Rather, in adopting *object orientation* we have learned to structure our system in terms of relatively stable units, viz. classes, rather than functions, setting boundaries around state. We have streamlined the process of developing dependable software. For example, we have developed ways to catch errors early, in the form of *automated testing*. We have derived some benefit from verification, especially lightweight fully automated verification such as powerful *type systems*, which render certain classes of error impossible even in principle. In parallel, we have increased the *agility* of the software development process, in which developers with highly disciplined habits are able to "embrace change" [3]. Some of these advances work well together; others, as yet, do not.

It is, I think, no accident that these key elements connect so clearly with the habits, principles and boundaries we identified in the previous section. In the end,

all of the problems of software engineering come down to one thing: a human being can only hold so much in their head. We all hate the feeling of being overwhelmed, of knowing that there are vital facts that we have temporarily forgotten. In order to make progress, we invent means of managing complexity, and putting in front of ourselves all, and only, the information that we need at a given time for a given purpose. (Indeed, arguably this is also our motivation for developing habits, principles, and boundaries: all of these limit the range of possible behaviours that we need to consider.) The key challenge is typically to identify what to leave out, whether that is possible future requirements (YAGNI, "you ain't gonna need it") or detail about what an existing piece of software does (a more detailed specification, even if it is correct, is not better, if it includes information that is of no benefit to its user). That brings us, naturally, to models.

5 What Are Models, and What Are They for?

My favourite one-sentence definition of a model is this:

> A model is an abstract, usually graphical, representation of some aspect of a system.

As is often observed, this, like other definitions of models, does not technically exclude much – "everything's a model". What it does do is to emphasise what is important about models: they represent some things, *but not other things*, that are true about systems. A model has, conceptually, a boundary: a piece of information may be inside the model, or not. That is, they allow us to separate concerns [8]. A model may be designed for a particular purpose – which may be prescriptive or descriptive – to include all and only that information that is necessary for the purpose.

Often the purpose is supporting the work of a particular group of stakeholders in the system: that is, different people may use different models. Models still provide benefit, though, and people still spontaneously develop them, any time there are discernibly different concerns that it is helpful to separate. The purpose of a model is to focus attention on what is important to some person at some time.

Different concerns require access to different information. Life is easiest when they require access to *completely* different information: the models are orthogonal, in that a change in one can be made without any implications for another. Recalling that everything is a model, we may for example think about two classes in different subsystems, that do not interact, as possible models. The developer of each one may be able to work quite happily on their own class, unaware of what the developer of the other is doing. However, this happy state of affairs is rare.

6 What Is Bidirectionality?

Finally, it is time to mention bidirectionality, and bidirectional transformations. There is a wide field of research on this topic which we will not survey: one place

to turn for further reading is a recent set of tutorial lectures [9], especially its introductory chapter [2]. The Bx Wiki[2] gives further pointers.

The essence of bidirectionality in a situation is:

- There is separation of concerns into explicit parts such that
- more than one part is "live", that is, liable to have decisions deliberately encoded in it in the future; and
- the parts are not orthogonal. That is, a change in either part may necessitate a change in the other.

Bidirectionality may be present, and it may be helpful to think in these terms, even without there being any relevant automation. The management of a bidirectional situation may be automated to a greater or lesser degree, and this is the job of a bidirectional transformation.

A bidirectional transformation is a means of maintaining consistency between some data sources, such as models of a software system. It is often convenient to separate this job conceptually into two tasks:

1. check whether the sources are consistent;
2. if not, change at least one of them (we may or may not wish to specify which), such that they become consistent.

Each of these tasks could, of course, be carried out by a conventional (unidirectional) program. The key observation justifying the study of bidirectional transformations as a distinct idea is that the tasks are so tightly coupled that, to ensure that they have behaviour that is jointly sensible, they should be engineered together. For example, we normally want a guarantee that the process of consistency restoration should, indeed, result in consistent sources (it should be "correct"); and this should remain true, even if the notion of consistency changes during the course of development, so that the bidirectional transformation must itself be updated. Writing separate programs to carry out the tasks involves duplication of effort and requires a separate check of whatever coherence properties between the tasks are required. Therefore it is extremely helpful for the bidirectional transformation to be written as a single program in a bidirectional language, one artefact incorporating both the definition of consistency and the instructions about how to restore it properly.

Thus, a bidirectional transformation must include a definition of what it is to be "consistent". The term sometimes gives difficulty to people who are used to using it in a logical sense. For our purposes here, consistency is nothing but a mathematical relation on the sets of models we consider. Given a tuple of models, it is possible (in principle) to say whether they are, or are not, consistent. If they are, and if their own groups of stakeholders are each happy with their own model, we consider that they are in a (relatively) good state from which to continue development; if not, something needs to be fixed – perhaps not immediately, but eventually. This is a very flexible notion. Given two sets of models, there

[2] http://bx-community.wikidot.com/.

is a wide choice of possible consistency relations, depending on what kind of consistency we are interested in for the particular development scenario, and how much automation we choose. Crucially, consistency need not be a bijection, but this does *not* imply consistency restoration will be non-deterministic (the process may look at both models, or even more information than that).

As a concrete example, suppose that one model is the Java source file for a class, and the other is that of a JUnit test class. (Recall that while models are usually graphical, everything's a model – this certainly includes everything we call code, which sometimes yields the most familiar examples.) Our bidirectional transformation might incorporate any of the following notions of consistency (in which of course we elide some details), or many others:

1. The files compile together without error.
2. 1. holds, and the JUnit file includes a test for every public method.
3. 2. holds, and all the tests pass.
4. 3. holds, and a certain coverage criterion is met.

The more stringent the notion of consistency we use, the more difficult may be the task of restoring consistency when one of the models is changed; on the other hand, the more work may be saved for the users of the models. There is a trade-off between work invested in automating the bidirectional transformation, and work invested in manually updating the models.

(The connection to the logical – strictly, model-theoretic, for a different sort of "model"! – sense is that *if* we identify a model with a set of statements about a system, then the relation we are interested in *may* be that the union of the statements given by all the models is logically consistent, so that there is a system about which all the statements are true. But for reasons we will not go into further here, this is not normally a helpful perspective e.g. because models and their relationships do more than make statements about a hypothetical system: they also facilitate development.)

To use bidirectional transformations in practice, we need both theoretical underpinnings and support from languages and tools. At present, the design space of ways to represent bidirectional transformations is wide open. Many languages have been proposed, a few of which have gone beyond being academic prototypes; despite early successes, none has yet achieved more than a tiny degree of real-world penetration. We should not be despondent about this: the problem is hard. While, collectively, we have decades of experience designing hundreds of unidirectional languages in several paradigms, for bidirectional languages that experience does not yet exist. Even basic questions remain unanswered. In some cases, we may eventually reach consensus; in others, it will likely turn out that the right answer depends on the circumstances. Here are a few examples, each of which has both theoretical and engineering aspects.

– What properties should a language enforce on every bidirectional transformation, and what will we need other mechanisms to check? For example, should the language enforce any formal least-change property [7], to capture the idea that consistency restoration should not change a model *gratuitously*?

- To what extent, if any, should a bidirectional transformation maintain and use information beyond the models themselves, e.g. a record of the relationship between parts of the models it relates (trace links)?
- Should we directly program transformations which are symmetric (that is, between models *each* of which contains information not present in the other? Or should we program asymmetric transformations (between a source and a view which is an abstraction of the source), relying on span or cospan constructions [10] to give the symmetric behaviour?
- Should our languages directly support maintaining consistency between more than two models? Or should we program binary bidirectional transformations, and rely on separate mechanisms to maintain consistency in networks of models related by these [13,14]?
- How should our language handle effects, such as state, non-determinism, exceptions, and user interaction, while still maintaining guarantees of good behaviour [1]? (Getting this right is the key to managing the other perennial trade-off of automation, between increased reliability of the automated process and decreased flexibility, compared to the manual process.)

If we can develop good-enough bidirectional transformation languages, and other supporting tools, the rewards should be great. To date, the adoption of model driven development has been limited by the difficulty of incorporating it into agile development. As long as models have to be maintained manually, separately from code and from one another, this difficulty remains. Bidirectional transformations have potential to allow the automation of this process. Imagine if developers could work with whatever model was most appropriate to the change they were making, with all others, including the final system, automatically updated to match. This would be an advance worth working for, even just considering the needs of today's software engineering. It could increase the speed of producing software, by enabling developers to work with the most cognitively efficient representations of what they need to have in mind. It could enable the development of software that is both agile and continuously verified and documented by means of tools working on appropriate, automatically updated, models.

7 What Is the Future of Software Engineering?

So far, we have been thinking inside the confines of software engineering as it is now. It is inconceivable, though, that the development of software fifty or a hundred years hence will be done in the same way as today. We have already alluded to one of the forces for change: the inexorably increasing demand for software which is updated ever more frequently. Pushing to solve this just by recruiting more software developers is absurd. We will have more important things for those five-year-olds to do when they grow up. Simultaneously, we are fractalising the boundary between software and the rest of our environment and ourselves, even as we demand greater dependability – trustworthiness – from

the software. The more our software becomes intelligent, the harder it will be to blame its developers for every wrong decision of the software – but we will demand someone be blamed.

Software will be so different, and will be developed so differently, that we may even have some difficulty in even recognising the software and its development process. I would not be surprised if today's programming languages look as puzzling to my grandchildren (if any) as the punched cards my father used to work with look to me. Doubtless, I am completely failing to foresee some important changes. Fundamentally, though, it is economically essential that more of the decisions about what our software should do will be moved away from software specialists, and towards people whose jobs only touch on software development. They may include people who use the software in different roles, but also, specialists in, say, safety, protection of personal data, or support for users with special needs. Rather than having to commission changes in the software from software specialists, or put up with a poor fit with their needs, they will be able to make changes directly. In order to make their decisions, these people will have to be provided with information, some of which they will change. The information given to a particular person is what we term a model. Such models blur the distinction between developer and user. (Lest this sound too fanciful, bear in mind that we already live in this world to some extent: what proportion of the software systems you work with on a daily basis have a settings screen, or similar? This is nothing other than a model, albeit one that may often seem too simple for your needs.) In any setting where more than one person has a model, there will have to be a task of reconciling decisions that the people make and record in their models; that is what we have termed bidirectionality.

We cannot possibly expect to specify in detail the behaviour of this exploding mass of ever-changing software. Rather, we will have to develop analogues of the habits, principles and boundaries that we expect human participants in our world to have. These analogues will abstractly represent certain aspects of the system's behaviour: that is, they are models. Just as humans adjust their behaviour to meet one another's (most vital) expectations, even when they encounter unexpected situations, so will our software have to do in future. This will require the models to adjust, automatically, to meet non-orthogonal expectations – bidirectionality again.

We already touched, in the more mundane setting of relating a class with its tests, on the fact that different degrees of automation, offering different guarantees, are possible. The same will apply in the more challenging future we envisaged. Indeed we do not expect to be able to rely on our fellow humans to behave perfectly first time in every interaction; it would be unreasonable to expect that of computers, once the interface becomes comparable in complexity.

Therefore, to my mind, the most fundamental change we must, as researchers and engineers, facilitate, is that software must in future be able to *explain* its past behaviour and *negotiate* its future behaviour. Our non-software-specialists manipulating their models will sometimes see behaviour they do not expect. They need to be able to ask "why did that happen?" and get an answer they

can understand, in order to know whether to continue to trust the software or not. "Computer says No" will not do in future: it must say why not. It may also take on board a human's explanation of why it should have said Yes, adjusting a model accordingly. (We have already explored simple cases where a bidirectional transformation may be refined by interaction with a human user [1]. Human-directed explanation will be much harder.) Explanations will have to be made in terms that both computers and humans can understand; non-software-specialists are not going to learn to do debugging the way software engineers do. Explanations must be trustworthy; they must stand up in a court of law; they must be framed like other such explanations, in terms of how the software interpreted its knowledge and goals in the context of its principles.

8 Conclusions

In this essay, I have given some background to the current interest in bidirectional transformations, and have tried to sketch how I think software engineering will change in future and the role of bidirectionality in that future.

A situation involves bidirectionality in an essential way once: there are separated concerns; more than one concern is live; and the concerns are not orthogonal. This already applies to most software development, so that better understanding and automation of bidirectionality would benefit us now. In the future, I think that the increasing complexity of interactions between humans and computers will render advances in the management of bidirectionality even more beneficial.

In conclusion: yes, bidirectionality is important.

References

1. Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Notions of bidirectional computation and entangled state monads. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 187–214. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19797-5_9
2. Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Introduction to bidirectional transformations. In: Gibbons and Stevens [9], pp. 1–28 (2018)
3. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
4. Betteridge, I.: Techcrunch: irresponsible journalism. Technovia.co.uk, February 2009. Accessed via [15] 18 Apr 2018
5. Booch, G.: Object Oriented Analysis and Design with Applications. Benjamin/Cummings, San Francisco (1991)
6. Brocklehurst, S.: Going off the rails: the Edinburgh trams saga. http://www.bbc.com/news/uk-scotland-edinburgh-east-fife-27159614
7. Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: On principles of least change and least surprise for bidirectional transformations. J. Object Technol. **16**(1), Article no. 3, 1–31 (2017)

8. Dijkstra, E.W.: Selected writings on Computing: A Personal Perspective. Chapter On the Role of Scientific Thought, pp. 60–66. Springer (1982). https://doi.org/10.1007/978-1-4612-5695-3
9. Gibbons, J., Stevens, P. (eds.): Bidirectional Transformations. LNCS, vol. 9715. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-79108-1
10. Johnson, M., Rosebrugh, R.: Cospans and symmetric lenses. In: Proceedings of the 7th International Workshop on Bidirectional Transformations. ACM (2018)
11. Petricek, T.: What we talk about when we talk about monads. Art Sci. Eng. Program. **2**(3), Article no. 12 (2018)
12. Shieber, S.M.: Is this article consistent with Hinchliffe's rule? Ann. Improbable Res. **21**(3), 18–19 (2015)
13. Stevens, P.: Towards sound, optimal, and flexible building from megamodels. Talk at Bx 2018 (paper in preparation)
14. Stevens, P.: Bidirectional transformations in the large. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 1–11. IEEE (2017)
15. Wikipedia contributors. Betteridge's law of headlines—Wikipedia, the free encyclopedia (2018). Accessed 18 April 2018