ON USING A MODEL FOR DOWNSTREAM RESPONSIBILITY

Marty J. Wolf, Frances S. Grodzinsky, Keith W. Miller

Bemidji State University (USA), Sacred Heart University (USA),
University of Missouri – St. Louis (USA)

mjwolf@bemidjistate.edu; grodzinskyf@yahoo.com; millerkei@umsl.edu

ABSTRACT

In the software development process, it is common for developers to be unaware of all the potential uses of the software they create. Developers typically do not control who buys their software, and importantly, they do not control the sorts of systems that their software is used in. This work guides responsibility analysis of complex systems that involve upstream software (developed earlier in the process) and downstream software (developed later). To that end, we analyze a responsibility attribution model and recent papers that address responsibility attribution to identify and categorize software features that scholars use in their analysis. This work gives key features of software that developers can use to address questions of responsibility surrounding their work. Furthermore, this work supports a move toward adopting sound "software provenance" practices as a way to address the many hands problem.

KEYWORDS: responsibility, software developer responsibility, models of responsibility, ethical analysis.

1. INTRODUCTION

In recent years, harm caused through the use of software surfaces at all too regular intervals. Authors often try to identify who is responsible for the harm caused by failures in software systems. Such analyses typically involve some consideration of social, economic, and technical factors. There are times when the analysis is done in such a way that it places responsibility on "software developers." These sorts of analyses, while appropriate, often do not account for the complexity of software and the fact that many software developers were likely involved in creating the disparate pieces of software that make up the software system under consideration. In "On the Responsibility for Uses of Downstream Software," (Wolf, Miller, and Grodzinsky 2019), we took a slightly different approach by considering the technical features of the software and the software development process and argued that these aspects play a role in the attribution of responsibility to the software developers whose software is part of the complex computing system that caused harm.

In the software development process, it is common for developers to be unaware of all the potential uses of the software they create. Developers typically do not control who buys their software. We set aside those cases where the software is designed for a specific narrow purpose and somehow fails at that purpose. Rather, we are interested in guiding analysis of complex

systems, especially when pieces of the software are put together by different groups of developers.

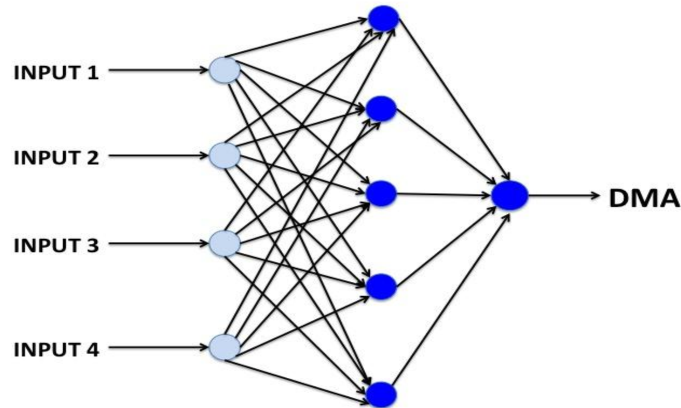
Consider an example: Assume that a software system is designed to ensure that a bomb hits its intended target; assume further that the software fails, and the bomb hits someplace else. We typically assume that the developers of the targeting software bear responsibility for that software failure. However, the complexity of any targeting software raises other concerns about whether that responsibility should also be shared by others; for example, should we attribute some responsibility for the failure to the developers who produced the driver for the GPS unit that the target software developers purchased "off the shelf?" In our analytical system, we call analogues of the GPS driver "upstream software," and we call analogues of the targeting software a "downstream software." Note that mistakes that cause the errant bomb could have been exclusively in the upstream software, the downstream software, or from an interaction between the two; localizing the fault would be part of the analysis.

The bomb targeting example illustrates another dimension to this issue: responsibility for intended effects, and unintended effects. The developers of the GPS driver may not have foreseen specific military uses for their product; the developers of the bomb guidance software clearly would. (The use of open source software in military applications was at the center of a discussion with some relevance to this article; please see Miller (2007) and Wolf et al. (2009).)

In an earlier article, we argued that there are at least five features of software and the software development process that should be considered during deliberations about responsibility attribution. The five are: closeness to the hardware, risk, sensitivity of data, degree of control over or knowledge of the future population of users, and the nature of the software (general vs. special purpose). Software that deals directly with capabilities of the hardware (such as the GPS driver mentioned above) are close to the hardware. Applications, such as a web browser that can run on a wide variety of devices, are farther from the hardware. Risk in software is tied to the predictability of its behavior and outputs. Sensitivity of the data refers to how private the data are that the software is processing. The degree of control over the downstream users of the software refers to the software developer's ability to limit who uses the software or their ability to ensure that the downstream user of the software meets certain professional standards. Finally, the nature of the software matters in responsibility attribution for downstream uses. How these features interact, and which feature take priority over others is a matter of judgment, and is case-specific.

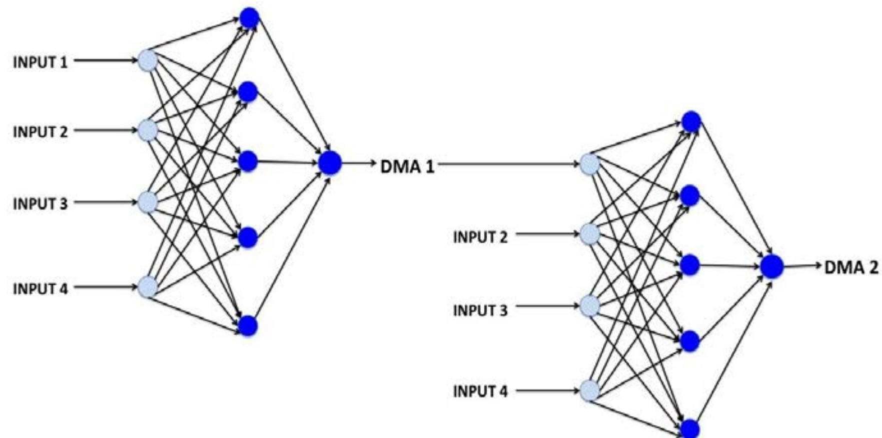
Close analysis of these features in the context of responsibility in computing ethics led us to develop an analytical system with two different models that might be used to assign responsibility in the use of downstream software: the Fixed History Model and the Chained History Model. In the Fixed History Model (see Figure 1), we assume that within the system of events (such as choosing to use an existing piece of software as part of a system) that led to an ethical breach (or an ethically positive outcome), there are certain inputs that are immune to the assignment of any moral responsibility for the Distributed Moral Action (DMA). This model does not consider assigning any portion of Distributed Moral Responsibility (DMR) to those who produced the inputs—INPUT 1, INPUT 2, and so on in Figure 1. The Fixed History Model is appropriate for certain types of software. For example, the developers of database software are rarely considered for the assignment of moral responsibility in the event of a breach. Typically, in such a case, responsibility stops at the database implementers who would have been responsible for initiating and maintaining security.

Figure 1. Fixed History Model.



The Chained History Model (see Figure 2), however, applies in cases where one of the inputs to the system is a piece of software and the attribution of moral responsibility propagates back to the developers of that software. In Figure 2 DMA 1 is a piece of software that becomes part of a larger software system that results in DMA 2. For complex computing systems, having more than two levels may be appropriate. Additionally, it is the case that multiple inputs may be upstream software and each of them may play different roles in contributing to the DMA under analysis.

Figure 2. Chained History Model.



In Wolf, Miller, and Grodzinsky (2019) we argue that, taken independently, each of the five features suggests a particular model to use. The Fixed Model fits well when the upstream software is close to the hardware, its developers have control over the downstream use, or the software is more general purpose. On the other hand, the Chained Model is more appropriate when the upstream software is riskier, computes over more sensitive data, or is more special purpose.

In this paper, we review a selection of recent papers (2017-2019) related to the attribution of responsibility in emerging technologies. Our analysis will determine situations when responsibility attribution for a moral action could have been clarified by using either the Fixed History Model or the Chained History Model. We will demonstrate how applying these models might help clarify ethical issues associated with distributed responsibility for software developers in a few complex and interesting cases.

The next section reviews six papers, identifies the strategy used in the analysis, and compares that strategy to the two models we proposed. The section that follows analyses the applicability of the analytical system and adapts them to more closely parallel how scholars think through issues surrounding responsibility in software. We offer some final thoughts in the conclusion.

2. REVIEW OF SELECTED PAPERS

Summary A

In the article "Digital health fiduciaries: protecting user privacy when sharing health data" Chirag Arora (2019) argues that when it comes to privacy concerns surrounding health data, it is the responsibility of the digital health data controllers to take steps to protect the privacy of those whose data is being collected and stored. Arora argues for a fiduciary relationship between data subjects and the data controller. Arora's argument uses "security, anonymization, and data minimization as examples of contextualization and flexibility required to deal with privacy issues." Even though Arora brings up the WannaCry ransomware attack on the UK's National Health Service, the responsibility for the system failure is not mapped back to flaws in the software that was infected by WannaCry, but rather to the failure to upgrade. Arora places the ethical breach at the feet of the data controller and makes no attempt to push any responsibility back to those who created the software with the flaw in it. We take this as evidence that Arora assumed a Fixed History Model.

Analysis A

Arora uses the term "controller" as defined by the EU's General Data Protection Regulation and refers to the entire collection of legal entities that might bear some responsibility. Thus, this analysis does not directly address the various responsibilities of the many individual contributors to complex digital health systems. The question we are concerned with is whether our analysis technique can advance this analysis to include statements about the responsibility born by the developers of upstream software. On one hand, since a digital health system presumably includes hardware, the closeness to the hardware features leans toward the Fixed History Model. Also, given that Windows 7 was an integral part of NHS, it is easily argued that some of the upstream software is general purpose, also suggesting a Fixed History Model.

On the other hand, it may be the case that at least some of the upstream software in the system is deemed risky. For example, software that controls various medical devices and is deployed in situations where there may be life and death decisions being made surely involves risk. Further, it is clear that such a system is handling medical data that are highly sensitive. Both of these observations suggest that the Chained History Model ought to be in play. It is clear in the context of this complex system that its developers could have had some control and fore knowledge of the sorts of downstream developers who would be using it. It is also likely that at least some of

this software would be special purpose. Therefore, these two observations are in tension. The upshot of this analysis is that for complex software systems there could be a tension among what features to prioritize in the analysis. Identifying those upstream parts of the system is important. In this case, trying to resolve and prioritize the features might be a step in moving the analysis towards the Chained History Model that more accurately reflects the complexity of the digital health system. It is also important to conceptualize that software development is not just design, code, and deploy; but also includes maintenance as part of the life cycle, and maintenance is a crucial part of the story of this failure. The developers who create the patches can only go so far. It is the responsibility of system administrators to actually apply them in mission critical systems.

Summary B

In the article "First steps towards an ethics of robots and artificial intelligence (RAI)," John Tasioulas (2019) investigates the problem of trying to build moral norms into RAIs. He distinguishes between RAIs that follow top-down algorithms that are prescriptive and closed-rule and bottom up or stochastic algorithms that use machine learning. In the first case, the RAI is largely functional and failure to accomplish its task can be attributed back to the developer. "In ... RAIs operating on the basis of top-down algorithms that render their behavior highly predictable, the argument for attributing legal responsibility to manufacturers, owner, or users seems compelling (Tasioulas, 2019:70). In our language, responsibility analysis can be served by the Fixed Model. Those developing the upstream software are likely not to share in the responsibility for harm caused by this sort of RAI.

In RAIs that use machine learning, the cases are more varied and complex. The author asks "...whether a good case exists for attributing legal personality to RAIs with corresponding legal rights and responsibilities …" (Tasioulas, 2019:70). The European Union and UNESCO have been grappling with this issue and it is beyond the scope of our paper to delineate the various arguments. However, Tasioulas does raise the question of traceability as particularly difficult with RAIs using bottom-up algorithms. He asks "... how do we ensure the 'traceability' of RAIs in order to be able to assign moral or legal responsibility in relation to them? Traceability involves being able to determine the causes that led an RAI to behave in the way that it did…" (Tasioulas, 2019:71). Whenever we grapple with tracing responsibility, we are closer to the Chained History Model.

Analysis B

In Tasioulas' first case, our analysis is similar to Tasioulas'. If in the development of the RAI's software, the developers used "off the shelf" software, the upstream developers cannot reasonably be expected to predict that their software might be used in such a system. In the second case, we see Tasioulas considering one of the components of the larger system and recognizing the risk associated with it. The Chained History Model seems to be called for here since machine learning software is inherently risky software; it requires great caution and complex software to prohibit dangerous, unanticipated actions. Tasioulas' argument supports a claim that lack of traceability increases the risk of such software. Thus, applying the Chained Model to this sort of system seems warranted. Developers of machine learning software ought to bear some responsibility for its downstream use.

We note here that developers should not be allowed to claim that they are absolved of any ethical responsibility for a machine learning algorithm, just because it can change after deployment. We have argued elsewhere (Wolf et al., 2017) that the developers should not deploy something unless than are willing to take responsibility for its behaviour after launch.

Summary C

In the article "The ethics of cloud computing" Boudewijn de Bruin and Luciano Floridi (2017) argue for certain regulatory restrictions that prevent various players in the cloud computing arena from engaging in certain practices. They argue for "rather intense pressure on business clouders" (de Bruin & Floridi, 2017:33). Their analysis rests on a clear delineation of the business interests in the cloud computing space. At the hardware level, there are companies that host cloud services. At the next level are the cloud services themselves—the applications and the software (e.g. recruiting software, document storage and sharing). The third level is made up of the business users of the software. It is this third level that they call the "business clouders". They argue that when cloud computing was still in its infancy (at least five to seven years ago), the providers of cloud computing should have been given a lot of latitude to experiment and thus, should have had few regulatory restrictions. On the other hand, they "suggest that proscriptive pressure must be exerted primarily on the business *users* of software as a service" (de Bruin & Floridi, 2017:23, emphasis in the original). In their analysis, they leave open the possibility of enacting more restrictive regulations on the hosting companies and software-as-a-service providers, should risks become evident.

Analysis C

While the focus of their analysis is establishing an ethical and well-justified regulatory framework for cloud computing, it is clear that their analysis is consistent with using the features of software and software development. Software developers closest to the hardware, the general-purpose nature of cloud services and software-as-a-service, and control over downstream use all point to using the Fixed History Model. De Bruin and Floridi make a clear argument that using cloud services mitigates risk. "… [T]he probability of this kind of crime" (the stealing of actual servers) "is likely to decrease when firms start opting for cloud services, because criminals will find it very hard to determine which servers in the datacentres contain the data they are interested in. Whereas a bank's server has only one purpose and is an easy target …" (2017:34). Thus, because of the less risky nature of cloud computing, the risk feature also points to using the Fixed History Model for responsibility attribution to the upstream software providers. Finally, cloud hosting services provide general purpose software. When this software is used by downstream developers, this feature also points to using the Fixed History Model. On the other hand, software-as-a-service is more special purpose: it is for storing and sharing files or recruiting employees. De Bruin and Floridi focus on the business clouders. In our analysis, the final feature, control over downstream use, comes into play. A software-as-a-service provider enters into a contract with the business clouder that integrates the upstream software (software-as-a-system) into the business clouder's computing infrastructure. This again points to the Fixed History Model.

Summary D

In the article "Artificial intelligence, responsibility attribution, and a relational justification of explainability," Mark Coeckelbergh examines the question: "What does the development of responsible AI mean?" (Coeckelbergh, 2019) In other words, who is responsible for the benefits and harms of developing and using AI technology? He initially explains the Aristotelian conditions of responsibility: control and knowledge of the responsible agent and starts with the premise that only humans can be responsible agents. He then adds the complication that AI systems are not always reducible to one agent. In these complex systems there is both the problem of many hands and of many things (systems and subsystems) interacting. Coeckelbergh also addresses the idea of relational responsibility: those who develop the software (agents) and those who are affected by it (patients) enter into a relationship mediated by the software. Moral agents should have a moral requirement to provide or explain reasons for a decision or action caused by their software to moral patients. He assumes that AI systems do not meet the criteria for full moral agency and therefore, the responsibility for their actions and decisions remains with the humans "who develop and use this technology." Coeckelbergh draws his examples from automation technologies, especially self-driving cars and airplanes.

Analysis D

The conceptual analysis in Coeckelbergh (2019) is illuminating. In deconstructing the problem of many hands and the concept of distributed responsibility, Coeckelbergh (2019) makes the following point that speaks to our attempt to develop a model that is helpful to developers:

> Acknowledging the distributed character of responsibility in the case of AI does not solve the practical problem of *how* to distribute the responsibility, given that one may not know (the extent of) all the contributions and interactions, and given a number of other challenges.

Coeckelbergh sees the way upstream software, AI in this case, is used downstream as a problem of distributed responsibility for AI applications. Citing cases where AI applications have been developed for one use but then applied in totally different contexts and cases where maintenance by systems administrators is not done responsibly, he argues that pro-active development is necessary for AI systems not to fail. Trying to attribute causal responsibility after the fact is very challenging. He identifies problems that surround "a long causal chain of human agency" such as the developers, the choices of data sets, potentially using software in one context when it was developed for a different one. He also identifies the role that software maintenance plays in establishing responsibility (2019). AI is generally viewed as risky, not close to the hardware, and in Coeckelbergh's examples there is little control over the downstream use. Each of these points to using the Chained Model. This is consistent with what Coeckelbergh is trying to do. One of Coeckelbergh's arguments, however, seems to be that some AI might be considered general purpose. "[I]n principle 'medical' face recognition software can also be used for surveillance purposes and become 'police' AI" (2019). This pushes back against our claim in our earlier paper that general purpose software points toward using the Fixed History model in responsibility attribution. That is, this observation says that the downstream use of AI in this way ought to remove the face-recognition AI developers from consideration for sharing in the responsibility of the moral impact when it is used. At first blush, this seems to point to a

shortcoming in our analytical system in that it gives conflicting indications as to how to ascribe responsibility. On the other hand, it is better to view this as an issue to be resolved by additional means.

Summary E

Vakkuri, Kemell, and Abrahamsson (2019) compare the literature in AI ethics with interviews they did with AI practitioners working in the healthcare sector in their paper "Implementing ethics in AI: Initial results of an industrial multiple case study." They list four values that they consider central to AI ethics: transparency, accountability, responsibility, and fairness. Much of their paper focuses on the nature and openness of algorithms that are "hidden" inside AI software. This focus includes descriptions of software projects that rely upon components that are studiously treated as "black boxes," whose inner workings are not studied nor even observed. Even more troubling they note that "developers do not see this as a problem" (2019).

Analysis E

The values of transparency, accountability, and responsibility discussed in Vakkuri et al. are all are highly relevant to our discussion about our two models. Their findings suggest that the subjects of their study only use the Fixed History Model when considering systems as it does not require much transparency, and the resulting accountability and responsibility relationships are clear but "shallow." These practiioners do not consider ascribing any moral resopnsiblity to those whose software they use and further expect the same consideration by those who use their software. It is clear that for the subjects of this study that the primary ethical go was competent creation of the software artifact. There was little consideration for other ethical values.

An insight from Vakkuri et al. that is directly relevant to our analytical system is that when we decide that the Chained History Model is appropriate to a particular system, then we must insist on transparency about that system in order to facilitate accountability and responsibility. As transparency increases about who is responsible for what in AI (or other) software, then accountability and responsibility could be traced more deeply, and the Chained History Model better reflects the complexity of the shared responsibility. Only when we can trace responsibility back further than a single step, and downstream into the next step in development can we meaningfully engage about earlier and later developers and their accountability.

Summary F

In the article "Algorithms, governance, and governmentality: On governing academic writing," Lucas Introna makes an argument that the algorithmic action of *Turnitin*, the text similarity analysis service, is increasingly a governing force in academic writing in that it "has produced a very particular regime of practice when it comes to academic writing" (Introna, 2016:37) While this article does not focus on responsibility, per se, it does focus on the software development process. Introna goes to unusual lengths to demonstrate the difference between code (Figure 1 is a C++ implementation of bubble sort) and an algorithm. Additionally, Introna's analysis includes the notion of the "relational temporal flow" of a piece of software. "This temporally unfolding process (the doing of the algorithm) itself inherits from prior actions and imparts to

subsequent actions, in the temporal flow of the doing of everyday sociomaterial practices" (Introna, 2016:22). While Introna's focus is on governance and ours is on responsibility, the analysis clearly acknowledges that some software (an implementation of a sorting algorithm) is temporally distant from its use in a commercial product, e.g. *Turnitin*.

Analysis F

Even though Introna gives a thorough analysis of the nature of software and software development, there is little to suggest that the downstream use of software has a bearing on issues surrounding governance. The analysis clearly parallels the Fixed History Model. Our analytical system also tilts in that direction. The software Introna considers is far from the hardware, and at least in the case of Turnitin, the software doesn't seem to bear much risk (but see below) and the data are not sensitive. On the other considerations, control over the downstream use and whether the software is special purpose or not, there is no clarity about what piece of upstream software might be under consideration.

Introna's work does bring to the fore another potential measure for consideration in the context of responsibility. Introna identifies numerous authors who argue for transparent algorithms. These arguments demonstrate an understanding that without some knowledge of how a system is designed, only the Fixed History Model can be applied. Introna then argues against transparency as an essential feature of software by suggesting a potentially undesirable outcome of transparency.

The argument considers someone who has the linguistic ability to understand the algorithm a hypothetical *Turnitin* might use for detecting how similar a piece of text in an essay is to an extant piece of text. Someone with that knowledge could then take steps to make changes to a copied passage: "those with linguistic skills could 'write over' copied fragments to make them undetectable" (Introna, 2016:38). Introna goes on to point out how this would disadvantage a non-native English speaker (assuming the essays were required to be written in English) in any sort of competitive environment.

This observation has a bearing on our analytical system, although in a less nefarious situation. In our case, we consider an upstream developer whose understanding of a downstream piece of software is incomplete. In this situation, should the Chained History Model come under consideration, part of the analysis should include the appropriateness of using the downstream software. In some cases, it may be that risk was introduced by the very act of choosing to use that software in the upstream system. In those cases, our conclusion is to tend toward the Fixed History Model.

3. ANALYSIS OF THE MODELS AND FEATURES

While a limited set of articles have been considered here, there is evidence that the analytical system we developed is supported by scholarship in computing ethics. Additionally, this work has identified two additional features of the software development process to be considered for inclusion in the analytical system: software maintenance and the use of existing software.

3.1. Software Maintenance

Generally, there is widespread agreement that software users should keep their software patched. That is, when an upstream software developer creates a patch, the downstream user of that software ought to install it—especially when that user is an end user (a business or a consumer). Upstream software developers might have reason to be more hesitant to apply such a patch. While the patch can be tested on non-production systems, the patch cannot be reasonably tested on all instances of the upstream software currently in production.

There are complications in trying to understand the overlapping responsibilities here. When software is released prematurely, patches may be frequent and inconvenient for users. Premature release is the ethical responsibility of developers, not users. Also, if patches routinely disrupt users' systems when installed, there is a disincentive for users to install them. Thus, it may not always be merely sloth or sloppiness that is the reason for users not promptly installing patches.

Another complication arises in environments where downstream developers (for example the developers in the NHS case considered by Aurora) have responsibilities to maintain stable mission critical systems. Such an environment calls for extensive testing to ensure that applying the update will not cause a failure. In these cases, it is not that the updated upstream software has flaws or that the downstream software has flaws, it is that the act of integrating those two systems introduces a system failure—the undesirable DMA.

We think some cases are clear: if a developer has acted responsibly with updates and patches, and if an ethically significant breakdown results in harm, and if an update or patch could have avoided that harm, then we think the Fixed History Model is appropriate, and users bear responsibility for the harm. Alternatively, if updates and patches are late or habitually disruptive, the Chained History Model seems more appropriate, as the developers share in the problem that resulted in the harm.

Software maintenance is an essential aspect of the software development process. Unlike other aspects of the software development process, its impacts often are in existing systems and dynamic environments that cannot be fully tested. The complexities of this socio-technical feature of the software development process do not point to using either the Fixed History Model or the Chained History Model to determine responsibility for DMAs. In real cases, careful analysis is required to ascertain an appropriate distribution of ethical accountability.

3.2. The Use of Existing Software

In practice this feature of the software development process is only narrowly applicable in the consideration of attribution of responsibility in the case of DMAs. In cases where a downstream developer has knowledge of and experience with the downstream software and the upstream software performs perfectly in testing, the responsibility attribution is consistent with the Fixed History Model. If there was some flaw with the upstream software and it still made it through the tests of the downstream software, then the developer of the upstream software is responsible for DMAs the upstream software contributed to (the Chained History Model). For this feature to come into consideration, the upstream software needs to behave in a way that is consistent with its specification, pass all of the tests of the upstream software, and still contribute to a DMA. There are perhaps very few of these sorts of situations. The downstream developer may have not completely understood the specifications of the upstream software.

The upstream developers may have not had a clear statement of those specifications. Due to the narrowness of when this feature might apply, it does not warrant inclusion as a feature in the analytical system.

4. CONCLUSION

A possible criticism of this paper is "so what?" Why does it matter which diagram you select to model responsibility for software use in a particular project? We offer two responses to the "so what?" challenge:

First, we maintain that the exercise of applying one or both models to a project will require developers (and their managers) to explicitly engage with questions of responsibility and accountability. These questions are important to developing software with integrity, and we would be pleased if the introduction of these models encouraged that kind of engagement since they demonstrate these interactions graphically. This analytical system will help developers be more transparent and make complex systems more explainable to their moral patients. The examples show that the analytical system for downstream responsibility attribution can clarify thinking about responsibility for different kinds of software that is consistent with at least a small sampling of the scholarship in computing ethics. Each of the models captures distinct attitudes about responsibility for software: The Chained History Model requires a long range, deep view; the Fixed History Model allows a shorter range, more shallow view. In some of the examples above, choosing one model or the other may be different for different upstream pieces of a system: in some projects, the Fixed History Model is suggested for one subsystem, and the Chained History Model is suggested for another subsystem. This variation adds to total system complexity. This analytical system aids developers who are trying to sort out this complexity. We expect that in thinking about the two models with respect to the system they are building, they will be more likely to clarify and communicate their judgments about what level of responsibility they claim for downstream uses of their software.

Second, we hope that an emphasis on transparency, responsibility, and accountability (as in Vakkuri et al., 2019) could, in time, make it mandatory for software developers to adopt a rigorous process of what we call "software provenance." This process would make it clear (perhaps in a separate document, but more easily as part of commenting in source code) what subsystems were developed by whom, and when the software started and was revised. With such information available, especially if it were encoded formally, automated tools for tracing responsibility and accountability could easily be established.

If software provenance become widespread (either by custom or regulation), the problem of many hands could be greatly simplified using automated tools. Software developers, vendors, and other stakeholders could not hide behind the obscurity of software's history. The enforced transparency of software provenance might significantly change the attitude of all stakeholders about the accountability of software professionals for the impact of their work.

REFERENCES

Arora, C. (2019). Digital health fiduciaries: protecting user privacy when sharing health data, Ethics Inf Technol 21: 181. https://doi.org/10.1007/s10676-019-09499-x.

Coeckelbergh, M. (2019) Artificial intelligence, responsibility attribution, and a relational justification of explainability" Science and Engineering Ethics 1-18, https://doi.org/10.1007/s11948-019-00146-8. Published online October 24, 2019.

De Bruin, B. & Floridi, L. (2017). The ethics of cloud computing, Sci Eng Ethics 23, 21–39. https://doi.org/10.1007/s11948-016-9759-0.

Introna, L.D. (2016) Algorithms, governance, and governmentality: On governing academic writing, Science, Technology, & Human Values. 40(1), 17-49.

Miller, K. (2007). Open source software and consequential responsibility: GPU, GPL, and the no military use clause. In P. Boltuc (Ed.), APA Newsletter of philosophy and computers, 6(2), 17–22.

Tasioulas, J. (2019). First steps towards an ethics of robots and artificial intelligence, Journal of Practical Ethics. 7(1), 49-83.

Vakkuri, V., Kemell, K. K., & Abrahamsson, P. (2019). Implementing Ethics in AI: Initial Results of an Industrial Multiple Case Study. In International Conference on Product-Focused Software Process Improvement (pp. 331-338). Springer, Cham. https://doi.org/10.1007/978-3-030-35333-9_24

Wolf, M. J., Miller, K. W., & Grodzinsky, F. S. (2009). On the meaning of free software. Ethics Inf Technol 11(4), 279.

Wolf, M. J., Miller, K., & Grodzinsky, F. S. (2017). Why we should have seen that coming: comments on Microsoft's Tay experiment, and wider implications. ACM SIGCAS Computers and Society, 47(3), 54-64.

Wolf, M. J., Miller, K. W., & Grodzinsky, F. S. (2019). On the responsibility for uses of downstream software," Computer Ethics - Philosophical Enquiry (CEPE) Proceedings: 2019, Article 3. https://doi.org/10.25884/7576-wd27