Requirements Engineering for Feedback Loops in Software-Intensive Systems

Eunsuk Kang Rômulo Meira-Góes
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA USA
{eunsukk,rmeirago}@andrew.cmu.edu

***Abstract*—A *feedback loop* occurs when the output of a system induces certain changes in the environment, which, in turn, influences the system through its input. Certain *self-reinforcing* feedback loops can inflict significant harm on the environment, by amplifying the existing bias or other undesirable effects over time. In this paper, we argue that such feedback loops are becoming more prevalent in software-intensive systems, and propose a set of requirements engineering activities and research problems for understanding, modeling, and dealing with feedback loops.**

I. Introduction

A *feedback loop* occurs when a system makes decisions that induces certain changes on its environment, which, in turn, influences the system through its input. A structure of a typical feedback loop is shown in Figure 1. Feedback loops have been studied extensively in systems engineering [7] and control theory [6], [4], [1]. Although not a commonly discussed concept in software engineering, feedback loops can be found in a wide variety of software-intensive systems that closely interact with the world, ranging from distributed applications and social networks to financial and credit scoring systems. Their prevalence is becoming more apparent through the rise of systems that rely on machine learning (ML) for decision making: These systems constantly evolve by retraining a model using data that is collected from the world, which, in turn, is influenced by the decisions made using the model.

Feedback loops are classified into two types: *negative* and *positive* feedback loops [1]. A feedback loop is said to be negative if the feedback induced by the output and the input are of the opposite polarity; i.e., it negates or reduces the magnitude of the input. On the other hand, a feedback loop is positive (or *self-reinforcing*) if the feedback is *in phase* with the input, in that it adds to the input. When left uninterrupted over a long period of time, a positive feedback loop can result in an amplification of an effect on the environment that is difficult to contain or reverse (this phenomenon is also called *system instability* [1]).

There are numerous well-documented examples of harmful effects caused by positive feedback loops in software systems [8]. For example, consider a system that automates decisions about allocation of police patrol to different neighborhoods in a city. These decisions (corresponding to the machine output in Figure 1) are made by an ML model that is trained on the historical data of prior arrests; the neighborhoods that have a historically high arrest rate are thus allocated more patrol. An increase in police presence leads to more arrests; this, in turn, skews new data being collected by the system for retraining (i.e., input from the environment in Figure 1), further biasing the model towards allocating police in those neighborhoods. Over a long period of time, this feedback loop could exacerbate the perceived level of crime activity in the neighborhoods, to the detriment of the residents.

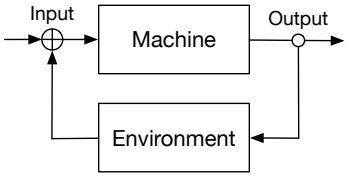

Fig. 1. Structure of a feedback loop. Incoming and outgoing arrows from the input and output, respectively, point to parts of the world that are not explicitly captured by the environment under consideration.

Although this example may be brushed aside as a problem in policy making or social science, software engineers have a responsibility to be aware of this kind of feedback loops and design the software to minimize their harmful impact. This can be, however, an extremely challenging task. The impact of a positive feedback loop often does not become apparent until after the system has been deployed for a long period of time. Most of the existing testing and validation tools available to developers assume a snapshot of the environment at *a particular point in time*, and is inadequate for reasoning about how the system interacts with the environment as the latter evolves over time.

In this paper, we argue that software-intensive systems should be designed explicitly with mechanisms for detecting and intervening with harmful effects of feedback loops. We also propose a set of requirements engineering activities that are crucial for supporting the design of a such system, including (1) understanding potential harms that can be incurred on various stakeholders, (2) modeling and articulating the structure of a feedback loop, and (3) runtime monitoring and intervention to reduce the impact of an undesirable feedback loop. We first begin by giving examples of positive feedback loops, and then further elaborate on the above activities, including research challenges and opportunities.

II. Feedback Loops: Examples

Feedback loops are frequently observed in numerous different domains, such as electronics, ecology, biology, economics, sociology, and climatology. One type of positive feedback loop commonly seen around us is the *bandwagon effect*. For example, under a certain economic or political trigger, people may begin to withdraw their money from a bank, spreading panic and encouraging others to also rush to their local bank; this, if sustained over time, may cause the bank to run out of money and face bankruptcy.

Not all positive feedback loops are necessarily undesirable, and some are deliberately built into a system. An example is audio feedback at a live concert, where a signal picked up by a microphone is passed through a loudspeaker, which produces audio that is again picked up by the microphone, enabling amplification of the sound. However, in this paper, we will focus mostly on *unintended* positive feedback loops that have undesirable or harmful effect on the environment.

A. Software systems

Given that software is increasingly being embedded into our society, it is no surprise that it is becoming a primary driver behind many of the positive feedback loops around us. One well-known example is the flash crash that occurred on May 6, 2010, which temporarily wiped out over one-trillion dollars from the U.S. market. In essence, this crash was a failure caused by a software-driven feedback loop: A sell order initiated by a high-frequency trading company caused an automated algorithm to increase the volume of stocks being traded, which, in turn, triggered other market participants to sell the stocks at a faster rate. Although this type of feedback loop is possible without software, increased automation and the lack of human oversight can make it much more likely.

Another area where a feedback loop can lead to an unpredictable and often undesirable behavior is distributed systems. In this domain, this type of behavior is sometimes called a *cascading failure* [2]. It typically begins with the failure of a single node or component within a distributed system; this, in turn, spreads the workload across the remaining nodes, increasing the likelihood of additional nodes failing. Without intervention, this chain effect may quickly propagate across the entire network, damaging system availability. In 2015, DynamoDB (a distributed database from Amazon AWS) experienced this type of cascading failure, disrupting access to popular services such as Netflix and Airbnb [10].

B. ML-based systems

ML-based systems share a common feedback loop structure: They rely on one or more ML models to make decisions that influence the environment, from which further data is collected to retrain the models. The problem is that the initial model may reflect undesirable properties of the environment (e.g., existing bias in a human decision making process) that is captured in the initial training data; through this feedback loop, these properties may be further reinforced and amplified over time. Adding to this problem is that these models are typically designed to optimize a certain metric (e.g., accuracy) that is not necessarily aligned with the interests of the stakeholders.

There are numerous examples of harmful feedback loops in ML-based systems deployed today [8]. Beside the policing example mentioned earlier, another good example is algorithmic hiring systems, which use various information about a job candidate (e.g., prior experience, education, skills) to predict how likely the person is to succeed in a position. In an infamous public case, the automated hiring system by Amazon was found to consistently rate female candidates lower than their male counterparts with equal qualifications [3]. The source of this problem was that the model used for prediction was initially trained on resumes that had been rated manually by human interviewers; unfortunately, the interviewers had a bias against female candidates, which was reflected through this data and ultimately, the ML model.

The harmful effect of this bias extends beyond unfair hiring decisions and deep into parts of the environment that are not directly interfacing with software. When an unfair system like this one is deployed for a long period of time, it may lead to a decrease in the number of female employees; this, in turn, may give rise to a perception that a certain profession is not welcoming for women, discouraging them from applying. Over time, this self-reinforcing loop is likely to damage diversity and overall quality of the profession.

Recently, researchers in ML have recognized harmful effects of bias in ML and are actively working to address them. However, relatively few techniques have been developed in understanding the problem at the requirements level, before a model is developed and deployed into the world.

III. Research Challenges

Despite their increasing prevalence, feedback loops have been studied relatively little by the software engineering community. As with many other problems in software, we believe that the most effective ways of dealing with feedback loops will begin in the requirements stage. In this section, we propose research problems to support requirements engineering activities that are crucial for understanding how a feedback loop can arise in software-intensive domains, and for informing the design of a system that is capable of minimizing its harmful effect.

A. Identifying Harmful Feedback

One of the very first (and perhaps most important) steps is to identify relevant stakeholders and understand potential harm that can be inflicted on them by the system. It is easy to overlook this step, however, as organizations often have objectives that do not necessarily align with minimizing harm to the stakeholders. For instance, one of the major goals of a social media company is to maximize the amount of user engagement (and ultimately increase the revenues generated). The company may implement various features on a mobile app to achieve this goal, such as infinite scrolling, personal recommendations, and push notifications (e.g., to encourage disengaged users to return to the app). While these features

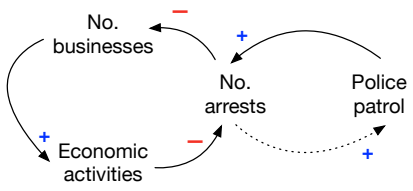

Fig. 2. A causal loop diagram for the policing system. A dotted edge represents a causal relationship that is realized by a machine (i.e., software).

may be effective in achieving the goal, they have also been found to cause a number of harmful side effects, such as addiction to smartphones, teen mental health problems, and the rise of echo chambers on the social media. For example, when a user reacts positively to a politically biased article, the app may suggest more of similar articles; over time, this can result in an echo chamber where people with similar beliefs congregate and insulate themselves from other views, contributing to increased political division within a nation.

Instead of being treated as an afterthought, these harms must be identified early in the requirements stage and explicitly considered throughout the rest of the product life-cycle. Identifying these harms may require a change in a typical developer's mindset: In addition to thinking about how stakeholders could benefit from the software product being developed, one must also understand how some of them could be negatively affected by it. Requirements elicitation techniques such as surveys, interviews, and prototyping are helpful but may not be sufficient: Even the stakeholders themselves may not be able to fully comprehend or predict harms that they could face over time. In addition to these techniques, this process is likely to require close collaboration with domain experts. For example, understanding the mental health effect of a social media app would be best carried out in collaboration with psychologists; to understand the long-term effect of racial bias in recidivism, one would consult sociologists and experts in policy design.

B. Modeling Feedback Loops

Understanding the impact of a feedback loop involves predicting how the environment evolves and interacts with the machine over a long period of time; this will require a model of the environment that can be simulated or analyzed for this purpose. Developing notations and analysis techniques to support this activity is an important research direction.

a) Causal modeling: One type of modeling approach that may be useful here is the *causal loop diagram (CLD)* [13]. Originally developed by the system dynamics community, CLDs have been used to model, simulate, and reason about feedback loops in systems from various domains, including business, economics, sociology, and biology. A hypothetical CLD for the algorithmic policing system introduced earlier in the paper is shown in Figure 2. A CLD contains a set of *system variables* and *causal relationships* between them, where each relationship (depicted as an edge) is labeled as a positive (+) or negative (-) relationship. A variable x has a positive relationship with another variable y if an increase (decrease) in x leads to an increase (decrease) in y. For example, in Figure 2, an increase in the number of arrested made in a particular neighborhood results in an increase in the amount of police patrol that is assigned to that neighborhood. On the other hand, when the neighborhood experiences an increased level of perceived crime, it may discourage business owners from continuing their activities; thus, the two variables—the number of arrests and the number of businesses—have a negative causal relationship.

A sequence of edges that form a cycle represents a feedback loop. A CLD enables a quick test to determine whether a feedback loop is positive (i.e., self-reinforcing) or negative: If the number of negative causal links is even, then the loop is positive. The CLD in Figure 2 contains two feedback loops that are both positive. In particular, when the number of arrests in a neighborhood increases, this may lead to a decrease in the number of businesses and ultimately, the amount of economic activities; the economic hardship experienced by the population, in turn, may lead to an increased level of crime, forming a self-perpetuating cycle. This example also shows how multiple positive feedback loops may originate from the decisions that are made by the software.

Of course, the difficult part in devising a useful CLD is to identify the relevant set of system variables and causal relationships for the problem. As with understanding harms, this modeling task will require working with domain experts (e.g., in the case of the policing system, sociologists or economists) or building on existing causal models.

b) Integration into existing modeling methods: It may also be possible to augment existing requirements modeling approaches (such as i* [15], goal models [14], and problem frames [5]) with the kinds of causal relationships present in CLDs. For example, in the problem frames approach, environmental domains and the machine are modeled as interacting through a set of *phenomena*. If these phenomena are treated like system variables in a CLD, then one could introduce positive and negative causal relationships between phenomena, enabling reasoning about feedback loops in problem frames.

C. Monitoring and Intervention

Once the structure of a feedback loop and its potential harms have been identified, the next step is to design the system with mechanisms for recognizing when an undesirable feedback loop occurs, and performing appropriate intervention to limit its harmful effect.

a) Control theory: Control theory is a long-established field that is, at its core, about using feedback as a mechanism for controlling the behavior of a dynamical system [6], [4], [1]. Thus, we can look to this field for both examples of feedback loops as well as methods that can be potentially leveraged to reduce the impact of feedback loops in software systems. In control, a positive feedback loop is typically considered undesirable, as it can cause instability in the system behavior over time. Given a model of the system (called the *plant* in their terminology), typically specified as a set of

mathematical equations, the goal is to synthesize a *controller* that manipulates the input to the plant such that its output remains within some desired boundary (i.e., it becomes stable over time). Conceptually, in Figure 1, the controller would be placed before the machine and adjust the input that is received from the environment, to ensure that the resulting feedback loop is negative (i.e., stable).

Researchers have successfully applied control theory to certain types of software systems (e.g., to achieve a desired level of performance and reliability in network applications) [4], [12]. These approaches, although promising, rely on a quantitative model of the system, where input and outputs are numerical variables, and the system behavior is captured using linear or non-linear equations. Thus, it is not immediately clear how control theory could be adapted to other classes of software systems that we have discussed above, and further research is warranted.

b) Self-adaptive systems: Self-adaptive systems is an active area of research on systems that are capable of adjusting to changes in their operating environment while achieving a desired quality attribute [11]. Techniques and tools developed by this community could be leveraged here, by monitoring the system for an undesirable feedback loop and applying different *strategies* (such as architectural reconfiguration) to limit its effect. However, it should be noted that self-adaptive systems can be a source of both solutions *and* problems for positive feedback loops. A typical approach to developing a self-adaptive system involves introducing some type of feedback loop into an existing system (most common one being the Monitor-Analyze-Plan-Execute, or *MAPE-K*, loop). If this approach is applied in a myopic manner, focusing on achieving a short-term objective (e.g., network throughput) while ignoring the long-term impact of the adaptation decision (e.g., system reliability), this itself could potentially introduce an undesirable feedback loop. Methods that combines both long-term and short-term planning (e.g., [9]) may be one promising approach to dealing with a positive feedback loop in a self-adaptive system.

c) Dealing with unanticipated feedback loops: There is likely to be some harms that one cannot anticipate at the requirements and design stage: As the environment evolves, the relationship among the existing and new stakeholders may change, giving rise to new types of harms or new ways in which harm can be inflicted. For example, as much as we may be tempted to blame the creators of early social media platforms, it would be unreasonable to expect them to have predicted all of the side effects that have emerged over the years (although one may fault them for not responding to them quickly enough to prevent the spread of the harm).

A more effective approach may assume that *unanticipated* feedback loops are likely to emerge at some point during the system deployment, and devise a plan for dealing with them when they do so. Such a plan involves both monitoring and intervention. For example, in the context of an ML-based system, a runtime monitor may be used to compute the distributions of system inputs and outputs, and look for an unexplained deviation from the expected norm. This may, for example, point to a shift in the environmental behavior that is caused by a positive feedback loop (e.g., in a loan lending system, an increasing proportion of applicants from a certain ethnic background being denied a loan).

Once such a deviation is investigated and confirmed to be an outcome of a feedback loop, different intervention strategies should be applied to limit its effect. These may include, for example, temporarily replacing an automated algorithm with a semi-automated or manual alternative that involves human oversight, re-training an existing ML model with a new dataset where bias has been removed, or if necessary, shutting down a service until the problem has been addressed.

IV. Conclusion

As software is increasingly being used to automate decisions that influence our lives, the risk of harm caused by self-reinforcing feedback loops is also likely to increase. To our best knowledge, however, there seems to be little prior research within the software engineering community on techniques and tools for understanding and dealing with feedback loops. We believe that requirements engineering will play a key role in improving the status quo, by providing methods for identifying and modeling feedback loops, and for detecting and mitigating their harmful effects as the environment evolves over time.

References

[1] K. J. Astrom and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2008.

[2] B. Beyer, C. Jones, J. Petoff, and N. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

[3] J. Dastin. Amazon scraps secret ai recruiting tool that showed bias against women. In *Ethics of Data and Analytics*, pages 296–299. Auerbach Publications, 2018.

[4] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. IEEE Press. Wiley, 2004.

[5] M. Jackson. *Problem frames: analysing and structuring software development problems*. Addison-Wesley, 2001.

[6] D. Kirk. *Optimal Control Theory: An Introduction*. Dover Books on Electrical Engineering Series. Dover Publications, 2004.

[7] D. Meadows and D. Wright. *Thinking in Systems: A Primer*. Chelsea Green Pub., 2008.

[8] C. O'Neil. *Weapons of math destruction: How big data increases inequality and threatens democracy*. Crown Publishing Group, 2016.

[9] A. Pandey, G. A. Moreno, J. Cámara, and D. Garlan. Hybrid planning for decision making in self-adaptive systems. In *International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 130–139. IEEE Computer Society, 2016.

[10] C. Patra. The DynamoDB-Caused AWS Outage: What We Have Learned. https://cloudacademy.com/blog/aws-outage-dynamodb, 2019.

[11] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *TAAS*, 4(2):14:1–14:42, 2009.

[12] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio. Control-theoretical software adaptation: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(8):784–810, 2017.

[13] J. Sterman. *Business Dynamics: Systems Thinking and Modeling for a Complex World*. Irwin/McGraw-Hill, 2000.

[14] A. Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *International Symposium on Requirements Engineering*, pages 249–262. IEEE, 2001.

[15] E. S. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *International Symposium on Requirements Engineering*, pages 226–235. IEEE, 1997.