Self-Portrayals of GI Junior Fellows

Andreas Vogelsang*

Explainable software systems

https://doi.org/10.1515/itit-2019-0015
Received May 14, 2019; accepted May 14, 2019

Abstract: Software and software-controlled technical systems play an increasing role in our daily lives. In cyber-physical systems, which connect the physical and the digital world, software does not only influence how we perceive and interact with our environment but software also makes decisions that influence our behavior. Therefore, the ability of software systems to explain their behavior and decisions will become an important property that will be crucial for their acceptance in our society. We call software systems with this ability *explainable software systems*. In the past, we have worked on methods and tools to design explainable software systems. In this article, we highlight some of our work on how to design explainable software systems. More specifically, we describe an architectural framework for designing self-explainable software systems, which is based on the MAPE-loop for self-adaptive systems. Afterward, we show that explainability is also important for tools that are used by engineers during the development of software systems. We show examples from the area of requirements engineering where we use techniques from natural language processing and neural networks to help engineers comprehend the complex information structures embedded in system requirements.

Keywords: Explainability, quality attributes, cyber-physical systems

ACM CCS: Software and its engineering → Software creation and management → Designing software, Software and its engineering → Software organization and properties → Extra-functional properties

1 Introduction

Explainability has recently gained attention due to research efforts on *Explainable AI*. Complex algorithms from the field of deep learning reveal the demand for additional information on the output of the algorithm to be able to comprehend, assess, and finally accept and trust the algorithms. Whereas projects on Explainable AI focus on explaining machine learning results, many cyber-physical systems (CPS) make context-dependent decisions that are not based on ML. Nevertheless, in many cases, it is not obvious for the user of a system why the system behaves in a certain way.

This shows that explainability is a challenge not solely for deep learning but for any software system that reaches a certain level of complexity. In a time, where systems become more and more connected, autonomous, and interactive, users and society notice the need to understand what systems are doing and why they are doing that. Just as we have the same need when interacting with other humans, we would like to ask systems questions like: *"why did you do that?"*, *"why not something else?"*, *"can I trust you?"* or even better, we would like the systems to provide us with explanatory information while they are running.

We call systems that have the ability to answer such questions *explainable software systems*. We use the following informal definition of this term: An explainable software system provides hints or indication on the rationale why the system made a decision. An extended property that builds upon explainability is *actionability*. An *actionable* software system provides hints or indication for how the user can influence system decisions by changing the system's environment (e. g., its inputs).

Making software systems explainable needs to be considered from the beginning of the development—similar to other quality attributes. In this article, we highlight explainability from two directions. First, we introduce an architectural framework that can be used to build self-explainable software systems by adding components for monitoring and analyzing the demand for explanations and then creating user-specific explanations. Secondly, we show the importance of explanations also for tools that are used by engineers during the development of software systems.

We are convinced that explainability is a key characteristic that needs to be addressed for the engineering of future software systems.

***Corresponding author: Andreas Vogelsang,** Technische Universität Berlin, Berlin, Germany, e-mail: andreas.vogelsang@tu-berlin.de

2 Designing self-explainable software systems

The complexity of software systems is constantly increasing because they control more and more complex processes in the physical world, possibly with multiple users, changing contexts, and changing environmental conditions. Hence, their software is distributed, concurrent, and combines discrete and continuous aspects. Due to this complexity, it becomes increasingly difficult for engineers, but also users, auditors, and other stakeholders, to comprehend the behavior of a system. Thus, it will be increasingly relevant for future software systems to explain their behavior to their stakeholders. This is essential to improve the trust and understanding between the user and the system [6], to enhance collaboration, and to increase confidence [5].

Our vision is to enable the development of *self-explainable systems* that can – at run-time – answer questions about their past, current, and future behavior (e. g., why a certain action was taken, what goals the system tries to achieve and how).

An example of an ambiguous action that might need explanation could be that a user in an autonomous car wishes to know an answer to the following question: *"Why are we leaving the highway?"* Here, the observed behavior is "leaving the highway". However, there could be several explanations for the behavior, e. g., *"We are leaving the highway..."*

- *"...because there is a traffic jam ahead"*; or
- *"...because we reached our travel destination"*; or
- *"...because we need to drive to a gas station"*.

Adding such self-explainability capabilities, however, is difficult. Self-explainability requires that the system has some understanding (i. e., a model) of itself, its environment, the requirements that it shall satisfy, and more: an understanding of the stakeholder that requires an explanation, and mechanisms that can reflect on the current behavior and provide hindsight and foresight. To date, there is no requirements engineering or design methodology for building such systems, and there is no reference framework for building self-explainable systems. We propose such a reference framework for building self-explainable systems, which is based on the MAPE-loop for self-adaptive systems [4].

To achieve this, we proposed the MAB-EX framework as depicted in Figure 1. Similar to the MAPE-loop, we first ***M**onitor* the control system, its environment, and possibly also the recipient. To this end, we capture and sample relevant sensor data, (a history of) commands from controller components, and possibly also a history of user-system interactions and former explanations.

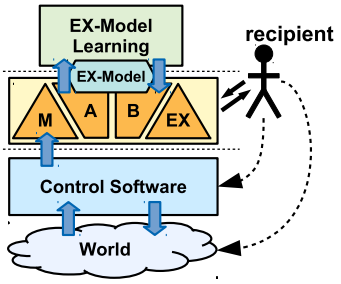

Figure 1: The MAB-EX framework for (self-)explainable software systems.

Then, we ***A**nalyze* the monitored data to detect an explanation need. This need can either be triggered because a recipient requires it (e. g., "Why are we leaving the highway?") or because the system shows behavior that requires an explanation (e. g., "We are slowing down soon because the road ahead is in poor condition."). The latter can be detected by identifying deviations from formerly observed behavior that might indicate an explanation need. Examples are irregularities in the monitored sensor data or sudden changes in the way the user interacts with the system. In the former case, we additionally need to analyze whether the change can be expected, e. g., due to user interaction. Furthermore, the history of controller commands or user commands can be analyzed to identify aimless sequences of interactions (e. g., contradicting commands over time that lead to nowhere). In case of explanation queries from the recipient, the query can be processed in this phase.

Instead of planning new behavior like in the MAPE-loop, our third phase is to ***B**uild* an explanation by evaluating an internal model of the system, which we call *explanation model*, based on the currently monitored system behavior, in order to extract relevant information. An explanation model is a behavioral model of the system that captures causal relationships between events and system reactions. It allows for identifying possible causes for the behavior that needs to be explained, e. g., traces of events that may lead to the behavior. It may also allow for look-ahead simulation to enable answering questions like "What happens if...?" or "When will ...be possible again?" Possible implementations for an explanation model could be fault/decision trees that connect observations to possible reasons [1], or executable behavior models (e. g., state machines). Such models may be constructed from requirements or from a behavior model, constructed manually, or

learned from observations. Synthesized explanations from these models are not yet in a recipient-understandable format. With *recipient*, we refer to the addressee of an explanation, which can be a user or an engineer.

Thus, the fourth and last phase is to ***EX****plain* the behavior in question to the recipient, meaning to transfer the result of the building phase to an understandable explanation for the target group. The explanation should be target-specific, as, e. g., an engineer might need more detailed information than a user, and a user might not understand technical terms that are useful for the engineer. To this end, we use a *recipient model*, e. g., mental model of a human recipient or an explanation interface between control software of different systems (e. g., to allow for collaborative learning and operation). It describes the preferences of the recipient w. r. t. explanation format (e. g., textual, image, voice, or machine-processable) and kind of information that should be included in an explanation (e. g., level of abstraction, points of interest). These recipient models can range from general mental models for target groups (e. g., engineers vs. users) to models for individual users.

As both, the system that needs to be explained and the recipient of the explanation may evolve over time or are subject to uncertainties at design time (about the system behavior, its operational context, and the recipient and its preferences), we include a *Model Learning* component into our framework that is responsible for updating both our explanation model and our recipient model. To update the recipient model, preferences of the recipient can be inferred from the interaction with the recipient (e. g., based on follow-up questions that indicate the wish for further information).

3 Explainable development tools

Many engineering tasks are nowadays supported by tools that either check the quality of manual work or perform the tasks completely automatic. Examples from the area of Requirements Engineering (RE) are requirements categorization [8], prioritization [7], trace link recovery [3], or detection of language weaknesses [2]. The increasing abilities of these tools are driven by the availability and accessibility of complex technologies. RE tools make use of advanced natural language processing techniques [2], information retrieval mechanisms [3], and machine learning (e. g., by artificial neural nets [8]).

Despite the complex technologies used, such development tools are very appealing to practitioners because most of the technology is hidden from the user. However, when tools produce results that a user finds strange or that a user cannot explain, tools often fail to give evidence or hints *why* it made this decision and what the consequences are [10]. Moreover, for some of the complex technologies used it may even be impossible to provide reasons for some decisions. For example, it is very hard to explain why a neural net makes a specific decision.

A special property of development tools is that they are almost never used in a fully automated context. Most of the times, development tools are part of processes, where they support a human analyst in performing tasks or reviewing work products. Therefore, we argue that more research is needed towards explainable development tools.

In the past, we made some efforts to make our development tools explainable and actionable. Here, we provide an example. We have developed an automated approach to differentiate requirements from non-requirements (information) in requirements documents [9]. At one of our industry partners, it is the document author's task to label all elements of a requirements document manually as either *requirement* or *information*. Our approach uses an artificial neural net that is trained on a large set of well-labeled requirements documents. After the training, the neural net is able to classify text fragments as one of the two classes. We use this approach to check the quality of this classification in existing documents. To make the decisions of the tool explainable, we have developed a mechanism that traces back the decision through the neural net and highlights fragments in the initial text that influenced the tool to make its decision [9]. As shown in Figure 2, it appears that the word "must" is a strong indicator for a requirement, whereas the word "required" is a strong indicator for an information. While the first is not very surprising, the latter could indicate that information elements often carry rationales (why something is *required*).

Class	requirement information
requirement	the duration until the switch is recognized as hanging must be a configurable parameter .
information	the component conditionally drives an external fan . this fan is required for active ventilation of the headlight .

Figure 2: Automatic classification of textual specification objects into classes *requirement* and *information*.

4 Discussion and outlook

In this article, we have highlighted the importance of explainability for future software systems. An explainable software system provides hints or indication on the rationale why the system made a decision. Making software systems explainable needs to be considered from the beginning of the development similar to other quality attributes. Future challenges include the definition of flexible explanation models that allow controlling the level of explanations (e. g., coarse-grained explanations for users vs. fine-grained explanations for developers) and approaches to efficiently derive explanation models from other engineering artifacts such as requirements, user documentation, or code comments.

References

1. F. Chiyah Garcia, D. Robb, X. Liu, A. Laskov, P. Patron, and H. Hastie. *Explain Yourself: A natural language interface for scrutable autonomous robots*. Proceedings of the Explainable Robotic Systems Workshop (HRI), 2018.
2. H. Femmer, D. Méndez Fernández, S. Wagner, and S. Eder. *Rapid quality assurance with requirements smells*. Journal of Software and Systems (JSS), 123:190–213, 2016.
3. J. Hayes, A. Dekhtyar, and J. Osborne. *Improving requirements tracing via information retrieval*. Proceedings of the 11th IEEE International Requirements Engineering Conference (RE), pp. 138–147, 2003.
4. IBM. *An Architectural Blueprint for Autonomic Computing*. White Paper, 2005.
5. P. Le Bras, D. Robb, T. Methven, S. Padilla, and M. Chantler. *Improving user confidence in concept maps: Exploring data driven explanations*. Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp. 1–13, 2018.
6. B. Lim, A. Dey, and D. Avrahami. *Why and why not explanations improve the intelligibility of context-aware intelligent systems*. Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp. 2119–2129, 2009.
7. A. Perini, A. Susi, and P. Avesani. *A machine learning approach to software requirements prioritization*. IEEE Transactions on Software Engineering (TSE), 39(4):445–461, 2013.
8. J. Winkler and A. Vogelsang. *Automatic classification of requirements based on convolutional neural networks*. Proceedings of the IEEE 24th International Requirements Engineering Conference Workshops (REW), pp. 39–45, 2016.
9. J. Winkler and A. Vogelsang. *"What does my classifier learn?" A visual approach to understanding natural language text classifiers*. Proceedings of the 22nd International Conference on Natural Language & Information Systems (NLDB), pp. 468–479, 2017.
10. J. Winkler and A. Vogelsang. *Using Tools to Assist Identification of Non-requirements in Requirements Specifications–A Controlled Experiment*. Proceedings of the 24th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), pp. 57–71, 2018.

Bionotes

Prof. Dr. Andreas Vogelsang
Technische Universität Berlin, Berlin, Germany
andreas.vogelsang@tu-berlin.de

Prof. Dr. Andreas Vogelsang is an assistant professor (junior professor) for software engineering at the Berlin Institute of Technology (TU Berlin). He is leading the software engineering group at the Daimler Center for Automotive IT Innovations (DCAITI). He received a Ph. D. from the Technical University of Munich in 2015. His research interests comprise requirements engineering, model-based systems engineering, and software architectures for embedded systems. He has published his research in international journals and conferences such as IEEE Software, SoSyM, ICSE, and RE. In 2018, he was appointed as Junior-Fellow of the German Society for Informatics (GI).