Towards Self-Explainable Cyber-Physical Systems

Mathias Blumreiter[*], Joel Greenyer[†], Francisco Javier Chiyah Garcia[‡], Verena Klös[¶],
Maike Schwammberger[||], Christoph Sommer[**], Andreas Vogelsang[††] and Andreas Wortmann[‡‡]
[*]Institute for Software Systems, Hamburg University of Technology, Germany
[†]Software Engineering Group, Leibniz Universität Hannover, Germany
[‡]Heriot-Watt University, United Kingdom
[¶]Software and Embedded Systems Engineering, Technische Universität Berlin, Germany
[||]Department of Computing Science, University of Oldenburg, Germany
[**]Heinz Nixdorf Institute and Dept. of Computer Science, Paderborn University, Germany
[††]Automated Systems Engineering Technologies, Technische Universität Berlin, Germany
[‡‡]Software Engineering, RWTH Aachen University, Germany
mathias.blumreiter@tuhh.de, greenyer@inf.uni-hannover.de, fjc3@hw.ac.uk,
verena.kloes@tu-berlin.de, schwammberger@informatik.uni-oldenburg.de,
sommer@ccs-labs.org, andreas.vogelsang@tu-berlin.de, wortmann@se-rwth.de

***Abstract*—With the increasing complexity of Cyber-Physical Systems, their behavior and decisions become increasingly difficult to understand and comprehend for users and other stakeholders. Our vision is to build self-explainable systems that can, at run-time, answer questions about the system's past, current, and future behavior. As hitherto no design methodology or reference framework exists for building such systems, we propose the *Monitor, Analyze, Build, Explain* (MAB-EX) framework for building self-explainable systems that leverage requirements- and explainability models at run-time. The basic idea of MAB-EX is to first *Monitor* and *Analyze* a certain behavior of a system, then Build an explanation from explanation models and convey this EXplanation in a suitable way to a stakeholder. We also take into account that new explanations can be learned, by updating the explanation models, should new and yet unexplainable behavior be detected by the system.**

***Index Terms*—Explainability, self-adaptive systems, cyber-physical systems**

I. Motivation

The complexity of Cyber-Physical System (CPS) 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 system- and software engineers, but also users, auditors, and other stakeholders, to comprehend the behavior of a system. Thus, it will be increasingly relevant for future CPS to explain their behavior to their stakeholders. This is essential to improve the trust and understanding between the user and the system [1], to enhance collaboration, and to increase confidence [2]. 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, etc.

An example for 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.

In this paper, we propose such a reference framework for building self-explainable systems which bases on the *Monitor, Analyze, Plan, Execute* (MAPE) loop for self-adaptive systems from IBM [3]. The MAPE loop proposes to continuously *monitor* relevant system and environment data, and, based on this, *analyze* whether an adaptation is necessary to satisfy system goals/ improve the performance. According to the analysis results, the system *plans* and *executes* a suitable adaptation. As we need similar self-reflection capabilities for a self-explaining system, we adapt this feedback loop to our needs. We demonstrate the applicability of our approach by sketching realizations in an example use case of a *Vehicle-to-X* (V2X) driver assistance system, which is prototypical for cooperative mobile systems in smart cities [4].

We introduce details on our *Monitor, Analyze, Build, Explain* (MAB-EX) framework and how it adapts the MAPE loop in Section III. Afterwards, we illustrate its application to our use case in Section IV. We discuss challenges yet to face and potential extensions of our framework in Section V. For related work on explainability of CPS see the following Section II.

II. Explainability in Software-Intensive CPS – An Overview

Explainability has gained attention due to research projects on *Explainable AI*. Whereas these projects focus on explaining machine learning results, many CPS make context-dependent decisions that are not based on ML. To explain these decisions, some approaches focus on *explainable planning*: In [5], Assumption-based Argumentation is used to model planning problems and to generate explanations for planning solutions as well as for invalid plans. [6] explicitly focus on CPS. This work-in-progress aims at providing interactive explanations based on Why and Why-Not questions from end-users about specific behaviors of the system. Answers are provided in form of contrastive explanations. Explanations contain the consequences or properties of choices, and how the choices affect goals and objectives of the system. In [7], verbal explanations of multi-objective probabilistic planning are automatically generated. They also use contrastive justification as explanation for why a generated behavior is preferred to other alternatives. In contrast to our work, these approaches focus on how to generate explanations and do neither provide a framework for identifying situations that need to be explained nor provide automatic customization to users and operation contexts.

In [8], the authors sketch first steps towards a conceptual framework for self-explaining CPS. Similar to our approach, they propose to add a layer for self-explanation that includes an abstract model of the system, and they propose to adjust the granularity of explanations for different user groups. In contrast to our work, they propose to construct cause-effect chains for observable actions using the abstract model. Users can access these chains to understand the cause of actions.

In [9], a feedback loop approach is used to identify situations where it is valuable to ask a user for feedback about system behavior. There, the authors compare the user behavior with a goal model and ask for feedback when users achieve sub-goals or when they deviate from an expected sub-goal. This is similar to our detection of situations that might need an explanation.

Other work has focused on rationalizing and verbalizing the behavior of autonomous agents. Rationalizations do not need to accurately reflect the true decision-making process, but give some explanations like humans would give in similar situations. In [10] an agent's actions are rationalized by using an encoder-decoder neural network to translate between state-action information and natural language. In [11] the agent's experiences on a route are verbalized by converting sensor data into natural language as answer to user queries with varying levels of abstraction, specificity and locality. Another approach to generate explanations at run-time is to use a multi-modal agent that can be queried 'on-demand' [12], [13]. There, the system behaviors are mapped into a modified version of fault trees, which the authors call *model of autonomy*, that capture the possible states of the system [14]. The authors found that the explanations given by the agent helped improving the fidelity of the operators' mental model, increasing the operator's understanding of what the autonomous vehicles were doing and why, as well as how they work [13].

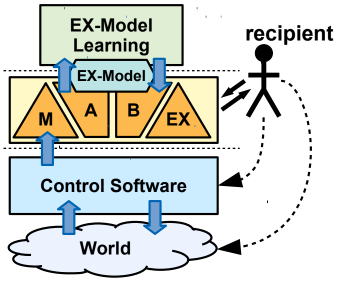


Fig. 1. The *Monitor, Analyze, Build, Explain* (MAB-EX) framework.

III. The MAB-EX Loop for Explainability

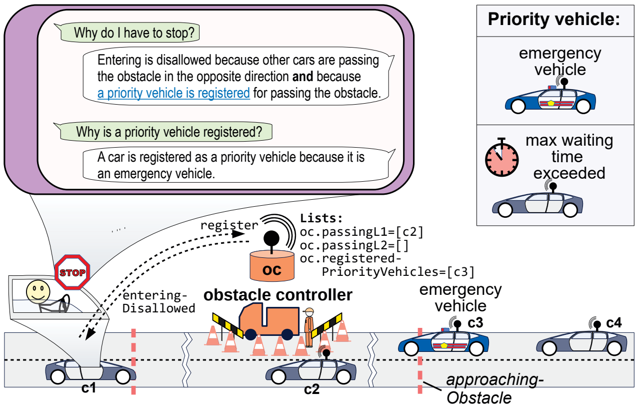
Our framework for self-explaining systems is inspired by the MAPE loop for self-adaptive systems, as we need similar self-reflection capabilities to detect the possible need for an explanation and to provide context-specific explanations. To achieve this, we propose the *Monitor, Analyze, Build, Explain* (MAB-EX) framework as depicted in Figure 1. Note that the underlying system does not need to be self-adaptive. Our MAB-EX loop can be added to any kind of computing system. However, if the underlying system is self-adaptive, our approach can also be integrated into the existing (MAPE) feedback loop. Similar to the MAPE loop, we first ***Monitor*** the control system, its environment and possibly also the recipient of explanations. To this end, we capture and sample relevant sensor data, (a history of) commands from controller components, and possibly also a history of user and/or system interactions and former explanations. To identify whether the user is satisfied with an explanation, we could also monitor the users face expressions (cf. Cowie et al. [15]).

Then, we ***Analyze*** 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 a user interaction. Furthermore, the history of controller commands or user commands can be analyzed to identify aimless sequences of commands/ 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 ***Build*** 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, e.g., be (fault/decision) trees that connect observations to possible reasons, or executable behavior models (e.g., state machines), as illustrated in our case study in Section IV. Such models may be constructed from requirements or from a behavior model, constructed manually, or learned from observations. Possible implementations also could be goal models that capture goals, objectives and motivations for the systems' behavior. Note that this synthesized explanation is not yet in a recipient-understandable format, but in an intermediate format. With *recipient* we refer to the addressee of an explanation, which can be a user (e.g., engineer or end-user) or a (sub-)system.

Thus, the fourth and last phase is to actually ***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 an end-user, and an end-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 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. end-users) to models for individual users.
The final explanation is provided to the recipient and thus, we do not have a loop anymore. However, we might have an indirect loop by monitoring the recipient's reaction to the given explanation or because the recipient itself asks for more details on the explanation.

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*** into our framework that is responsible for updating both our explanation model and our recipient model. Consider, e.g., the case that an emergency maneuver is executed due to a spontaneously changing extreme weather condition for autonomous driving. If the monitored and analyzed behavior is not contained in the explanation model, an explanation cannot be built immediately in the building phase. However, after having integrated this new behavior into the explanation model, an explanation can be provided later or if the behavior should occur again. Model Learning can be realized using machine learning algorithms, or as an expert system, where domain experts (and probably also other cooperating systems) are asked to provide an explanation for the observed situation, or as combination of both. This cooperative updating process could, e.g., be realized by connecting *Model Learning* components of different systems and experts via a cloud service. 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).


Why do I have to stop?

Entering is disallowed because other cars are passing the obstacle in the opposite direction **and** because a priority vehicle is registered for passing the obstacle.

Why is a priority vehicle registered?

A car is registered as a priority vehicle because it is an emergency vehicle.

Fig. 2. *Vehicle-to-X* (V2X) narrow passage coordination assistance system

IV. Example Realizations of MAB-EX

We illustrate the MAB-EX framework by instantiating it for an example of an advanced V2X driver assistance system such as is typically envisioned for future cooperative mobile systems [4]. This system helps drivers safely pass obstacles on the road (see Figure 2). In this example, cars that approach the obstacle register at an obstacle controller and await permission to enter the narrow street section. The system's response (pass or stop) is displayed to the driver. We focus on a car (c1 in Figure 2) that must stop and where the driver wonders why passing the obstacle is not possible—even though the roadworks is on the opposite lane and the road ahead seems free. We envision that an interface (top left of Figure 2) provides an explanation to this question. The explanation in this case is twofold: There is a car in the narrow street section approaching from the other side (which the driver may not see yet), and, moreover, there is an emergency vehicle approaching from the other side, which is not yet in the narrow section, but has registered at the obstacle controller as a priority vehicle. Other reasons to stop could be fairness to cars that already waited for a long time.

We illustrate the four building blocks of the framework for this example.

A. Monitor

As stated in Section III, we monitor the controlled system to identify situations that demand an explanation. In the example, we need information about the position of a car in the lane (L1 or L2) and the controller's response towards the event of approaching the obstacle (enteringDisallowed or enteringAllowed). Since we are only interested in one specific

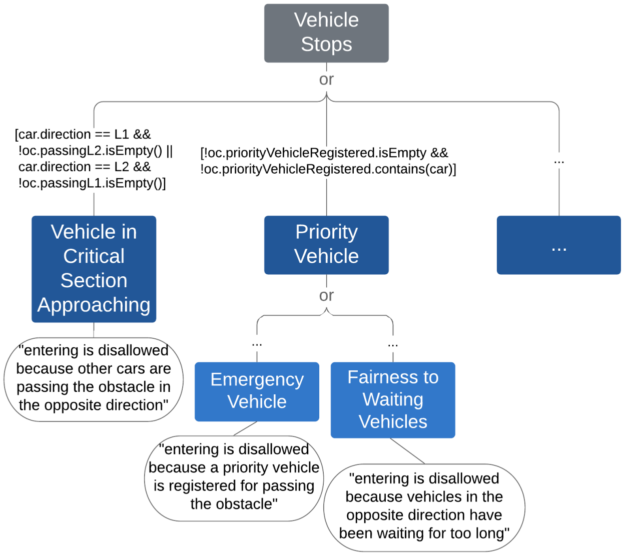

Vehicle Stops
or
[car.direction == L1 &&
!oc.passingL2.isEmpty() ||
car.direction == L2 &&
!oc.passingL1.isEmpty()]
[!oc.priorityVehicleRegistered.isEmpty &&
!oc.priorityVehicleRegistered.contains(car)]
...
Vehicle in Critical Section Approaching
Priority Vehicle
...
"entering is disallowed because other cars are passing the obstacle in the opposite direction"
or
...
...
Emergency Vehicle
Fairness to Waiting Vehicles
"entering is disallowed because a priority vehicle is registered for passing the obstacle"
"entering is disallowed because vehicles in the opposite direction have been waiting for too long"

Fig. 3. Model of causality for the car example. Each node has a condition of system variables and a natural language explanation. A model of causality may contain as many nodes and depth levels as necessary.

situation in this example, we do not need more information. In extended scenarios, it may be interesting to monitor, for example, the vehicle's speed to identify critical situations that may demand an explanation. If the system has a query feature for explanations in its HMI, we need to monitor user queries as well.

B. Analyze

We analyze the monitored data and identify situations that need to be explained. In our example, the only situation that needs an explanation is when a car is approaching the obstacle on lane L1 and the controller responds with enteringDisallowed.

C. Build

Building the actual explanation is usually the greatest challenge in the MAB-EX framework. We present two solutions that can be used to identify and compose the ingredients of an explanation (i.e., the causes of the event that needs to be explained). The first solution to provide such explanations is based on *models of causality*, which connect actions of the system to their (possibly internal) causes, including natural language descriptions. The second solution shows how behavioral models of the system that are created at design-time can be leveraged to assemble explanations at run-time.

1) Models of Causality Approach: This approach was explored before for providing explanations to operators of autonomous underwater vehicles [13], [14]. We can also apply this approach to the example above.

Figure 3 shows a tree for the situation of when a car stops in our case example. The root node is the observable event ("Vehicle Stops") and the branches give the possible reasons for the event. Traversing down the tree gives the explanations, which are attached to the nodes as natural language sentences. The explanations are increasingly detailed further down the tree, allowing to easily adapt to the user's needs. Together with the explanations, the nodes have a condition in terms of system variables that can be checked to determine if the node could be a plausible reason for the observable event.

A solution based on models of causality gives a high-level representation of the events without looking at the system details. Thus, the trees are independent from the system implementation, which allows to build these trees at any point of the system's life. They can be directly derived at the requirement specification phase or built after the system has been released. This approach requires minimal or no changes to the system that it explains, as they only monitor system variables to evaluate the node conditions. The level of abstraction over the system's internal processes makes controlling the amount of information disclosed easily adjustable.

However, the models of causality rely on manual modeling, which involves system knowledge that only those building or maintaining the system itself can provide. They also require knowing ahead of time which events can happen and the different explanations for the phenomena. Thus, they are limited (or focused) to explaining certain predefined state conditions.

2) Creating Explanations Dynamically from Run-Time Models: The above approach has the advantage that designers can easily model the system's explanation capabilities, specifically control the level of abstraction of the explanations, and that it can be easily integrated into a system: However, the explanations are limited to the phenomena anticipated at design time, and they are limited to explanations concerning specific (current or past) states—properties of sequences of states or predictions about the future are not possible. To achieve this, we require more elaborate models at run-time, which connect the behavior of the system and its environment with requirements specification, assumption specifications, and which can be queried and, especially, executed for look-ahead predictions.

Such an approach could be based on executable scenario-based behavior models, e.g., *Live Sequence Charts* (LSC) [16], that can be annotated with per-requirement/scenario rationales, or could contain trace links to natural language requirements, from which explanations can then be derived dynamically, at run-time. The executable scenarios could live solely within the system's explanation layer (as EX models), but they can even be used as the final implementation code for the distributed reactive behavior of CPS such as our example V2X system [17].

We sketch a scenario-based explanation approach in the following. Listing 1 shows scenarios from the example V2X system in *Scenario Modeling Language* (SML) [18], a textual language for modeling LSC-style scenarios. An SML specification models, via *assumption-* and *guarantee scenarios*, how objects of a system and its environment interact by sending messages. Guarantee scenarios describe how the developed (software) system may, must, or must not react to environment events; assumption scenarios (not shown here) describe what may, will, or will not happen in the environment. Each scenario models valid sequences of events, using different *modalities*.

```
...
guarantee scenario CarRegistersAtObstacle
 bindings [oc = cp.obstacleCtrl] {
 sensor -> car.approachingObstacle()
 //@EX: when approaching an obstacle, the car must register at
       the obstacle control
 strict requested car -> oc.register()
}

guarantee scenario CarEnteringAllowedDefault {
 car -> oc.register()
 // @EX: entering is allowed because there is no indication to
       disallow it.
  requested oc -> car.enteringAllowed()
} constraints [
 interrupt oc -> car.enteringDisallowed()
]

guarantee scenario CarEnteringDisallowedWhenCarPassing {
 car -> oc.register()
 alternative [car.direction == L1 && !oc.passingL2.isEmpty() ||
       car.direction == L2 && !oc.passingL1.isEmpty()] {
  // @EX: entering is disallowed because other cars are passing
        the obstacle in the opposite direction.
  strict requested oc -> car.enteringDisallowed()
 } constraints [
  forbidden oc -> car.enteringAllowed()
 ]
}

guarantee scenario EnteringDisallowedForOtherPriorityVehicle {
 car -> oc.register()
 alternative [!oc.registeredPriorityVehicles.isEmpty()
   && !oc.registeredPriorityVehicles.contains(car)]{
  // @EX: entering is disallowed because a priority vehicle is
        registered for passing the obstacle.
  strict requested oc -> car.enteringDisallowed()
 } constraints [
  forbidden oc -> car.enteringAllowed()
 ]

guarantee scenario SetPriorityForEmergencyVehicle {
 car -> oc.register()
 alternative [car instanceOf EmergencyVehicle] {
  // @EX: car registered is a priority vehicle because it is an
        emergency vehicle.
  strict committed oc -> oc.registeredPriorityVehicles.add(car)
 }
}
}
...
```

Listing 1. SML scenarios for the V2X driver assistance system

For example, events can be *requested*, which means that the event must eventually occur; non-requested messages need never occur. Events can also be *strict*, saying that when the scenario is waiting for the event to occur, no event must occur that is expected within the same scenario at an earlier or later point. The *forbidden* modality models events that are forbidden while (a certain part of) a scenario is active; *interrupt* models events that are allowed, but will interrupt the scenario. The scenarios are *executable*; at execution-time, multiple scenarios can be active at the same time, each requesting or forbidding certain events, and events are chosen to satisfy all the constraints imposed by the scenarios.

The scenario CarRegistersAtObstacle specifies that when a car sensor detects that the car approaches an obstacle, the car must register at the obstacle controller oc. The scenario CarEnteringAllowedDefault specifies that the obstacle control shall allow the car to enter, unless the scenario is interrupted by the enteringDisallowed event that can be requested, for example, by the scenario CarEnteringDisallowedWhenCarPassing, which models the case of a car that is passing the obstacle in the other direction. Scenario EnteringDisallowedForOtherPriorityVehicle models the case where a priority vehicle is registered for passing the obstacle while the car that is subject to that scenario is itself not a priority vehicle. Last, SetPriorityForEmergencyVehicle specifies that when an emergency vehicle registers at the obstacle control, it will be added to the list of registeredPriorityVehicles (cf. Figure 2).

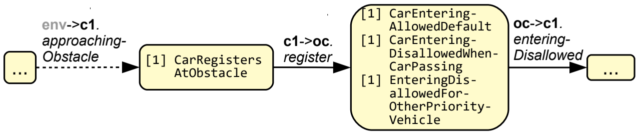


Fig. 4. Scenario run-time states for the V2X example

Figure 4 shows a sequence of states in the execution of these scenarios. For brevity we omit the states of the underlying objects. Starting with the approachingObstacle event the scenario CarRegistersAtObstacle is activated. Then, register terminates CarRegistersAtObstacle, but activates CarEnteringAllowedDefault, CarEnteringDisallowedWhenCarPassing, and EnteringDisallowedForOtherPriorityVehicle. In this case, enteringDisallowed is executed due to the conditions in the latter two scenarios that are satisfied in the state as in Figure 2.

The scenarios contain explanations annotated to all events that they request. This way, an explanation as show in Figure 2 can be produced by combining these explanations. Figure 2 also shows that the explanation component is able to answer the follow-up question *Why is a priority vehicle registered?*. This question can be answered by traversing over the past states in search of the events that contributed to rendering the condition true. In this example, a past activation of SetPriorityForEmergencyVehicle triggered by the register message from the emergency vehicle c3 caused the event of adding c3 to the list of registeredPriorityVehicles, which was the point from when the evaluation of the condition in EnteringDisallowedForOtherPriorityVehicle turned from *false* to *true*.

The scenarios could also be used for a forward-exploration of possible future behaviors that could be used to answer questions about the future, such as *When will I be allowed to pass the obstacle?* Moreover, instead of annotating the scenarios with explanations, these could also be extracted from textual requirements that could be referenced via trace links. It will be interesting to elaborate how also explanations for not executing certain events can be provided.

D. Explain

Finally, we need to generate the actual explanation from the information gathered in the *Build* component. In our example, the explanation is given as a text that is generated from the text fragments associated with the nodes in the model of causality or with the annotations in the scenario specifications. This way, the provided explanation for the detected situation is rendered as *"Entering is disallowed because other cars are passing the obstacle in the opposite direction and a priority vehicle is registered for passing the obstacle"*.

V. Conclusion and Research Roadmap

The MAB-EX approach towards explainability of system run-time behavior represents a first approach towards a generalizable architecture for self-explainable systems. We have shown how requirements- and models-at-runtime can be exploited as a basis for realizing self-explanation capabilities.

The road towards truly comprehensible, flexibly tailored explanations yields many challenges:

Comprehensible explanations: Useful explanations demand for a representation of decisions that supports tailoring the abstraction of explanation parts to the recipient, e.g., in contrast to the infamous Windows operating system blue screens 'explaining' its failure in terms of memory locations. Similarly, in engineering CPS with domain experts, a networking engineer might be very interested in communication decisions, but less in HCI decisions the system has made.

Explanation presentation: Depending on the facts to be explained or the receivers' background, different presentations of explanations will be of different usefulness. While engineers might prefer textual explanations (e.g., log files), users might prefer graphical explanations or conversational interfaces.

Focused explanations: To prevent systems from overwhelming receivers with potentially relevant information we need to conceive means for filtering and truncating explanation information based on, e.g., user studies or learned patterns.

Consultable explainers: When systems are capable of producing a wealth of explanations of different extent, abstraction, and personalization, being able to consult systems for specific explanations becomes necessary to support producing the best-possible explanations for different circumstances.

Interactive explanations: Similar to human discourse, self-explaining systems may produce explanations that entail subsequent queries about the reasons for a given explanation. Consequently, truly useful self-explaining systems should support interactive exploration of explanations, explanation sequences, and metadata (e.g., relations between explanations).

Explanation prediction: Systems equipped with means for self-explanation should be able to explain the future potential behavior as well as why expected events did not happen. This could have the form of explicit what-if queries or online explanation about expected behavior.

Cooperative explanations: To understand the behavior of systems cooperating in the Internet of Things, the smart factory of the future, or in V2X, systems must be able cooperatively explain their behavior. This demands for means to align their explanation terminologies (e.g., through explanation ontologies for specific domains) and might require reason about their own behavior based on implicit explanations (i.e., observations) of the cooperating systems' behaviors.

A posteriori explaining: The long-lived systems in industrial domains will need to cooperate with systems incapable of explaining themselves. Therefore, means to explain system behavior based on observations made by a posteriori deployed, dedicated explainers is necessary.

References

[1] B. Y. Lim, A. K. Dey, and D. Avrahami, "Why and why not explanations improve the intelligibility of context-aware intelligent systems," in *SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2009, pp. 2119–2129.

[2] P. Le Bras, D. A. Robb, T. S. Methven, S. Padilla, and M. J. Chantler, "Improving User Confidence in Concept Maps: Exploring Data Driven Explanations," in *CHI Conference on Human Factors in Computing Systems*, ACM, 2018, pp. 1–13.

[3] "An Architectural Blueprint for Autonomic Computing," IBM, White Paper, Jun. 2005.

[4] C. Sommer and F. Dressler, *Vehicular Networking*. Cambridge University Press, 2014.

[5] X. Fan, "On Generating Explainable Plans with Assumption-Based Argumentation," in *International Conference on Principles and Practice of Multi-Agent Systems*, Springer, 2018, pp. 344–361.

[6] E. Zhao and R. Sukkerd, "Interactive Explanation for Planning-Based Systems," in *10th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS 2019)*, 2019.

[7] R. Sukkerd, R. Simmons, and D. Garlan, "Towards explainable multi-objective probabilistic planning," in *4th International Workshop on Software Engineering for Smart Cyber-Physical Systems*, ACM, 2018, pp. 19–25.

[8] R. Drechsler, C. Lüth, G. Fey, and T. Güneysu, "Towards Self-Explaining Digital Systems: A Design Methodology for the Next Generation," in *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*, IEEE, 2018, pp. 1–6.

[9] D. Wüest, F. Fotrousi, and S. Fricker, "Combining Monitoring and Autonomous Feedback Requests to Elicit Actionable Knowledge of System Use," in *Requirements Engineering: Foundation for Software Quality*, E. Knauss and M. Goedicke, Eds., Cham: Springer International Publishing, 2019, pp. 209–225.

[10] B. Harrison, U. Ehsan, and M. O. Riedl, "Rationalization: A Neural Machine Translation Approach to Generating Natural Language Explanations," 2017. arXiv: 1702.07826.

[11] V. Perera, S. P. Selveraj, S. Rosenthal, and M. Veloso, "Dynamic generation and refinement of robot verbalization," in *25th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, New York, NY, USA: IEEE, Aug. 2016, pp. 212–218.

[12] D. A. Robb, F. J. Chiyah Garcia, A. Laskov, X. Liu, P. Patron, and H. Hastie, "Keep Me in the Loop: Increasing Operator Situation Awareness through a Conversational Multimodal Interface," in *20th ACM International Conference on Multimodal Interaction (ICMI)*, ACM, 2018, pp. 384–392.

[13] F. J. Chiyah Garcia, D. A. Robb, A. Laskov, X. Liu, P. Patron, and H. Hastie, "Explainable Autonomy: A Study of Explanation Styles for Building Clear Mental Models," in *11th International Natural Language Generation Conference (INLG)*, ACM, 2018, pp. 99–108.

[14] F. J. Chiyah Garcia, D. A. Robb, X. Liu, A. Laskov, P. Patron, and H. Hastie, "Explain Yourself: A Natural Language Interface for Scrutable Autonomous Robots," in *Explainable Robotic Systems Workshop (HRI)*, 2018.

[15] R. Cowie, E. Douglas-Cowie, N. Tsapatsoulis, G. Votsis, S. Kollias, W. Fellenz, and J. G. Taylor, "Emotion recognition in human-computer interaction," *IEEE Signal Processing Magazine*, vol. 18, no. 1, pp. 32–80, 2001.

[16] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," in *Springer Formal Methods in System Design*, vol. 19, 2001, pp. 45–80.

[17] J. Greenyer, L. Chazette, D. Gritzner, and E. Wete, "A Scenario-Based MDE Process for Dynamic Topology Collaborative Reactive Systems – Early Virtual Prototyping of Car-to-X System Specifications," in *Modellierung 2018, Workshops zur Modellierung in der Entwicklung von kollaborativen eingebetteten Systemen (MEKES)*, vol. 2060, Braunschweig, Germany: CEUR-WS.org, 2018, pp. 111–120.

[18] J. Greenyer, D. Gritzner, T. Gutjahr, F. König, N. Glade, A. Marron, and G. Katz, "ScenarioTools – A tool suite for the scenario-based modeling and analysis of reactive systems," *Elsevier Science of Computer Programming*, vol. 149, pp. 15–27, 2017, Special Issue on MODELS'16.