Model Assisted Distributed Root Cause Analysis

1st Michael Mayrhofer
Cognitive Robotics and Shopfloors
Pro2Future GmbH
Linz, Austria
michael.mayrhofer@pro2future.at

2nd Christoph Mayr-Dorn
Institute of Software Systems Engineering
Johannes Kepler University
Linz, Austria
christoph.mayr-dorn@jku.at

3rd Ouijdane Guiza
Cognitive Robotics and Shopfloors
Pro2Future GmbH
Linz, Austria
ouijdane.guiza@pro2future.at

4th Alexander Egyed
Institute of Software Systems Engineering
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

2021 IEEE 26th International Conference on Emerging Technologies and Factory Automation (ETFA) | 978-1-7281-2989-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ETFA45728.2021.9613684

***Abstract*—Cyber-physical production systems are composed of a multitude of subsystems from diverse vendors and integrators, connected in a distributed fashion. An undesirable phenomenon in one system might cause a misbehavior in another connected system. Searching for the root cause of this misbehavior quickly becomes very tedious as many possible search directions exist. This paper proposes an approach and algorithm to tie together information available in design-time and runtime models. This then allows, in conjunction with observed and desired status of a system, to recommend search options and concrete solution steps to guide workers along the fixing process without being overwhelmed by the complexity of the overall system of systems. We demonstrate the feasibility of our approach using a lab-scal production cell model.**

***Index Terms*—Automation systems; information models; worker assistance; debugging; root cause analysis; cyber physical systems**

I. Introduction

A production plant often requires the integration of cyber physical production systems implemented by different manufacturers. Manufacturers use different platforms to develop their platforms, which interact and form a system of systems (SoS). In case of bugs occurring in this SoS, a debugging tool tailored to a certain platform will only be able to access systems built on this platform.

Due to the interconnected nature of shopfloors, an undesirable phenomenon in one machine might cause a misbehavior in another connected machine, and hence searching for the root cause quickly becomes very tedious as there are many possible connections. Connections exist at the same hierarchical level, between systems (horizontally) as well as between layers like hardware, software, communication (vertically). Guidance based on system models helps to better understand how these machines depend on each other.

In this paper, we identify available information at the level of system and SoS and an approach to consolidate this information to enable tracking of bugs for systems of cyber physical production systems.

Section II will describe a small scenario to further illustrate the challenges addressed by this paper. Section III discusses existing works and their limitations. Section IV shows how to generate debugging assistance by reasoning on a holistic model created from model fragments. Section V discusses in detail which information was applied in our use case. We demonstrate the feasibility of our approach in a use case presented in Section VII. We conclude our paper in Section VIII and provide an outlook.

II. Motivating Scenario

A simplified production cell in our motivating scenario, inspired by one of our industry partner's setups, is composed of the following systems: an injection molding machine (IMM), a handling robot, and a safety fencing. The safety fencing activates a light barrier when closed. IMM and handling robot operate only with a closed safety fencing.

We assume that a worker closes the safety fencing and intends to start production, being stopped by an error message on the IMM's display: "Safety fencing is open". Many error sources are likely to lead to this unexpected behaviour:

- The safety fencing might not have been closed properly.
- The light barrier might be broken.
- The signal interface of the safety fencing's controller might be broken.
- The safety fencing might have lost network connection.
- In case robot and IMM are daisy chained, the robot might not have re-transmitted the information to the IMM.

Most items in this list could be investigated by the worker directly, without remote assistance from a support technician. Yet, it cannot be expected that every worker conceives every possible error source. Moreover, with sufficient information on the systems' states and their interconnection, some error

This work has partially been supported by the FFG, Contract No. 854184: "Pro²Future is funded within the Austrian COMET Program Competence Centers for Excellent Technologies under the auspices of the Austrian Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology, the Austrian Federal Ministry for Digital and Economic Affairs and of the Provinces of Upper Austria and Styria. COMET is managed by the Austrian Research Promotion Agency FFG."
Part of this work has also received support by LIT-2019-8-SEE-118, LIT-ARTI-2019-019, and the LIT Secure and Correct Systems Lab funded by the state of Upper Austria. Additional partial support by Engel Austria GmbH.

sources can be excluded beforehand, while others become more probable to cause the observed malfunction.
This is where our proposed approach comes in, to help with the reasoning about which elements to examine, and which examination order might lead to a bug fix the quickest.

III. Related Work

Assistance in fixing bugs in pure software environments has been investigated for a long time already [1][2][3]. Developments towards higher-level programming languages have led to more complex software requiring a multitude of debugging strategies [4] for correct execution.
Bugs in distributed and event-based systems are an active field of research. Some are well applicable to our scenario, whilst others are restricted by certain properties.
Causal Consistent Replay Debugging [5] logs every interaction in functional and concurrent programming languages that are based on message passing. The paper presents rules that allow to replay the program backward and forward in time, allowing to follow clearly how a certain bug arose in a run of a program. So far, only an Erlang implementation exists. Also, to produce the logs, the program has to be compiled with additional instrumentation code.
Causality-Guided Adaptive Interventional Debugging [6] predicts the location of faults in source by running a high number of program variants, with different inputs and versions with injected faults. Such techniques are only applicable to digital twins of production equipment.
Consistent Retrospective Snapshots [7] are used to create a totally ordered event-log for a system of event-sourced systems. This log allows to replay the system by feeding it with the logged events. An implementation exists only in Java so far.
In [8] a formalization of a debugging framework for communicating event-loops is given. The paper leaves the implementation of such a debugger as future work. Communicating event-loops are a generalisation of actor-based systems in the sense of Hewitt [9], which are suited well to model systems of cyber physical production systems.
In [10], Marra et. al. connect a monitoring application developed in the Pharo programming language to access sensor values on a GrovePi board. Although the case study is limited to a very tiny example, it allows to showcase different debugging techniques.
Mixed Dimensional Displays as described in [11] are a great tool to convey debugging information to workers, but still need a concept to create recommendations for next steps with a high probability to solve the issue.
Reflective Epidemic Debugging as described in [12] allows to debug an actor-based system while actors are in an idle state. The framework is implemented in the Ambient Talk language. One concern is, that bugs will very likely arise in non-idle-states.
Geels, Altekar, Shenker, and Stoica describe replay debugging for distributed systems in [13]. Their approach targets processes and applications hosted on one linux machine and requires a modified GNU debugger.
Several approaches study debugging at a level of behavior models, like UML [14] [15], COLA [16] and UML-RT [17] [18]. These approaches assume inputs before execution and hence cannot be applied in production, where system states and inputs arise by execution in an incompletely controllable context and environment.
In case of production plants, systems of many different vendors are working together. Many of them will be closed-source and expose information only via a standardized and restricted set of interfaces. This limits the use of above approaches.

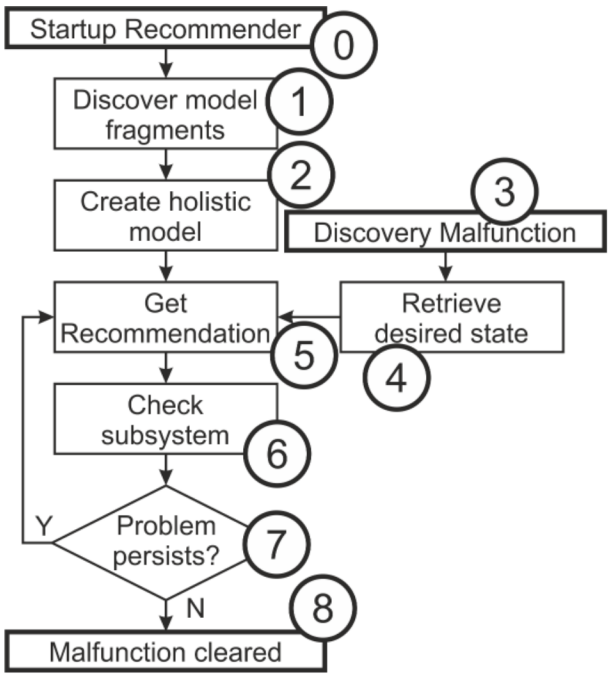

Fig. 1. Overall structure of model based debugging assistance

IV. Model Based Assistance for Debugging

Our approach makes use of partial information available in different models, each a standard interface for production machinery. A rough outline of the overall approach is given in Figure 1.
For a given production plant, the recommender has to run through a starting procedure once (0). First, the production machinery is discovered and browsed for usable information (1). Information is processed and stored as model fragments. Wherever possible, links are created between the model fragments to form a holistic model of the production plant (2). The recommender is now ready to assist in debugging.
Latest at discovery of a malfunction (3) in the production plant, the recommender ought to be started. The worker fixing the malfunction expresses the expected and observed behavior of the malfunctioning subsystem to an oracle, which retrieves the desired state (4) of the subsystem. The current state of the subsystem can be polled from the subsystem. Current state, desired state, and the holistic plant model allow reasoning to

create recommendations (5) for the worker. Recommendations take the form of checking a certain component of a subsystem, or checking a link between subsystems (6). If the problem was not solved by this check (7), the next recommendation is computed, until the malfunction is cleared (8).

The holistic model is represented as an undirected graph of three different node types: components, ports, and model references. The edges between nodes are annotated to provide additional information when navigating the graph. For example, the edge from a port into a model reference node identifies which element in the model itself represents the port. The model reference node describes where to find the actual model. An edge between two ports identifies how two components interact with each other. A component has edges to its subcomponents. One can think of the holistic model as a jigsaw puzzle where individual puzzle pieces represent different model fragments that describe different concerns (see Figure 2). Without a recommender, the procedure of identifying the root cause has the user start at a node where a behavior deviation is observed (circled start node Figure 3 upper left). Navigating one edge down, the user encounters a well behaving node (indicated by "OK") and then proceeds to investigate other nodes (marked grey) until identifying the root cause node (indicated by "error"). In a realistic scenario the possible paths are many and long, hence our approach aims to reduce the set of navigation paths that are most likely to lead to the root cause. Section VII provides a visualization excerpt of an example holistic model.

The following Section describes a representative selection of model fragments and how the holistic model may be put together from various sources.

V. Model Fragments

During its life cycle, a production plant is described by many partial models. To support the search for the root cause horizontally and vertically in a distributed SoS, we need at least three types of models: device information (describing how components are deconstructed into subcomponents down to hardware), behavior models (how software components react), and wiring models (the topology of distributed component interactions across process boundaries). We introduce one candidate model for each of these categories.

A. Device Topology

Each system may consist of subsystems. Subsystems may be connected with each other, forming a topology. Connections are realised via ports, that are connected to other ports. Connections are either uni- or bidirectional. Usually in systems, a coordinator subsystem aggregates information of subsystems and orchestrates the components' behaviors. A reduced example for the subsystems of a manufacturing cell is given in Figure 4 where the cell consists of a Moulding Machine, a Tempering Device, and a Safety Fence, which in turn consists of a Light Barrier, an Emergency Button, and a Fence Gate.

Information of systems and subsystems is available at connection points, and is exchanged via connections between connection points. Some connections within the Safety Fence subsystem are given in Figure 5. Here the Safety Fence exposes three ports, one connected to each subcomponent. The OPC UA Device Integration specification [19] details, how such information is structured in case the system uses OPC UA as communication protocol.

B. Wiring Information

The devices providing their topological self-description interact at the production plant level. In order to standardise such information, skills [20][21] have been proposed and modeled as OPC UA programs. Alternative model approaches describe the provisioning and requiring of capabilities [22]. In [23] we have shown, how a dynamic reconfiguration of the plant level topology can be realised with little overhead. The topology configuration ("wiring") is split and the necessary parts are distributed to the devices. From the devices these parts can be gathered later to reconstruct the overall topology.

C. System Behavior

Software and control engineers use state machine diagrams to capture behavior of a system, device, or component. To ensure interoperability, machine vendors agree on standard behaviors. For example, a widely adopted standard is the PackML state machine [24].

To model and store state machines in a machine-readable way, we relied on UML stored in standard XML format. Our recommender utilizes the following UML elements: UML events are described in UML interfaces. UML components then exhibit in ports and out ports that reference these interfaces. The ports of UML components are linked to describe which components interact with each other within one system context (i.e., within on system process). Recall that interaction across system and network processes is described via wiring information (see previous subsection). Each component contains a UML state machine that describes which UML events cause which state transitions. For each UML state, there exists a minimal UML activity diagram that describes whether that state leads to the sending of an event via one of the UML component's ports.

Note that UML is just one possible candidate to describe such behavior. Our approach merely requires the description of behavior in the form of a state machine, what messages/requests are dispatched in which states via which ports to what other components, and which events arrive at which port to trigger transitions in the state machine. The State machine in Figure 6 describes that the safety fence transitions from the "Active" state to the "Inactive" state when the light barrier is free, and the emergency button is released. With the connected device models one can obtain insights that the events for these two conditions arrive via port A and B of the Safety Fence, hence navigate further down into the light barrier and emergency button to inspect their status.

Component Hierarchy, Wiring Information, and System Behavior are, of course, not a complete set of models to handle

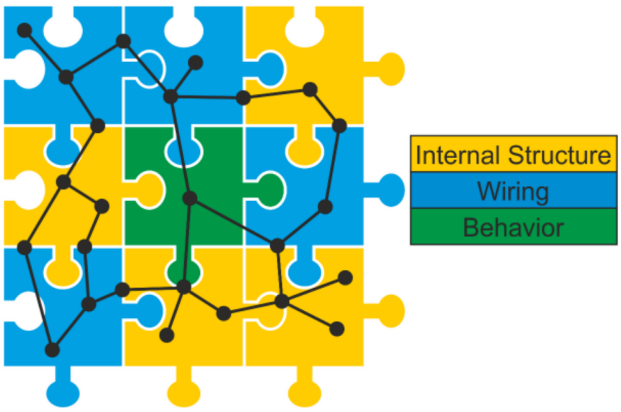

Fig. 2. Elements of the overall system model

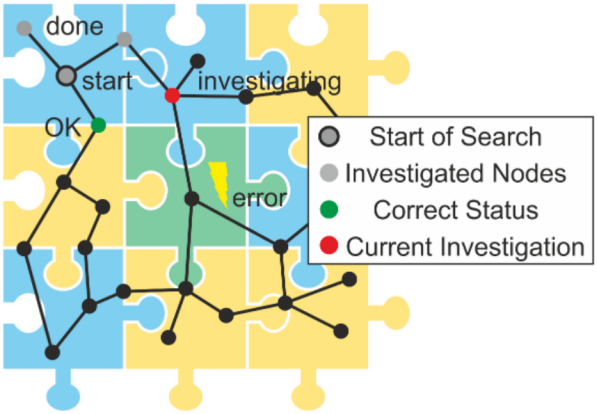

Fig. 3. Example Recommendation Path

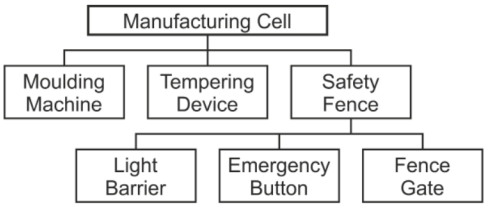

Fig. 4. Hierarchical System Structure of a Manufacturing Cell

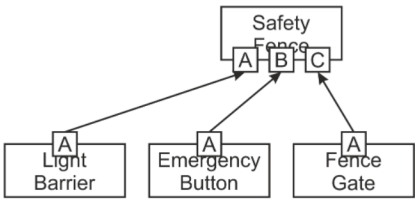

Fig. 5. ConnectionPoints and Connections within a System

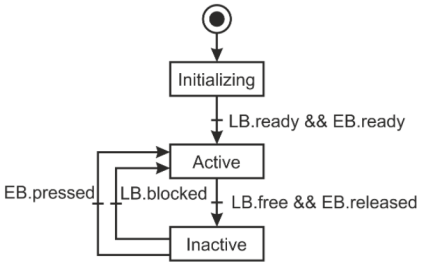

Fig. 6. Simplified behavior of a Safety Fence subsystem

all issues arising during production. Still, they are examples sophisticated enough to show the benefits of our approach when tracing the links between these models. Our approach allows to include additional model types through the use of an adapter pattern between models and the reasoning engine. One example would be the inclusion of the electrical layout (e.g., eplan models) to understand which combination of input signals will lead to which combination of output signals to identify deviations from the expected signal status.

D. Model Fragment Discovery and Merging

The model fragments are combined to form a graph, as illustrated in Figure 2. Nodes are enriched with information containing type of information, metadata, pointers to detailed models. For our recommender prototype implementation, we rely on our OPC UA wiring information model to describe the dependencies between capabilities. Each capability then refers to a UML model and provides the UML port identifier that establishes the mapping between the OPC UA level capability and the UML port defined in the UML model. An alternative mechanism to provide model information is via AutomationML (e.g., see [25]).

VI. Recommendation Generation

The recommender is given the node of the observed malfunction, together with the desired state of the node. Such information is typically available in the form of system documentation or defined in requirements. The recommender then retrieves the current status of the component associated with the node. Here, a standardized mapping of state machine status to OPC UA simplifies the lookup of such information, e.g., for PackML see [21]. It then starts to traverse this graph from this given node, as is illustrated in Figure 3. Depending on the types of the connected nodes, specific reasoning is carried out on each traversal step to compute the desired state for the connected node. The table in Figure 7 lists the possible scenarios for the model used in the current use case.

- *Check Connection* If an expected message did not arrive via the *IN port*, either the connection is faulty. Here faulty depends on the connection type. A physical connection might be subject to a broke wire or a loose plug. A network connection might be subject to an incorrect endpoint configuration, network partitioning, or authentication failure. The extent of recommender support here depends on the available modeling granularity. Or, if the connection is working, the message was not sent at the *OUT port*. Hence, the algorithm continues to search via edges from the *OUT port*.
- *Message Inferring* If a message is required to transition from the current state to the desired state, and the message is received via this port, then the algorithm proceeds to check this port for this message.
- *State Inferring* If a message was expected to be sent from the port, the algorithm looks up the behavior model for the state in which this message would have been sent

from. If the current state differs from this desired state, the search is continued from this node.

- *Message and State Inferring* If the *IN component* is in a different state than expected, the algorithm looks up the behavior model for the transitions from current to desired state. The next expected trigger message is then used in an immediate *State Inferring*. If the message is sent by another component matching the *OUT component*, the desired state is determined for that component and the search continues from there.

These steps continue, until the problem is solved, or the graph has been exhaustively traversed.

Applied to our motivating scenario, lets assume, the recommender has determined that the moulding machine state machine difference is a missing "Safety Fence Inactive" event via analysing the moulding machines UML model. It then identifies the port in the UML model through which this signal is received, maps this port to the holistic model port, traverses the holistic model to the port of the safety fence that provides this event and obtains the safety fence UML model. If this state machine is in the inactive state, the algorithm can identify the edge between the two ports as the potential root cause. If not, it analyses the safety fence state machine in which events are received to arrive in the Inactive state and the ports through which these events arrive and continues there. Note that this traversal requires no domain knowledge about any of the involved systems.

From \ To	Port	Model Reference	Component
Port	Check Connection	State Inferring	
Model Reference	Message Inferring		State Inferring
Component		Message and State Inferring	

Fig. 7. Reasoning Options

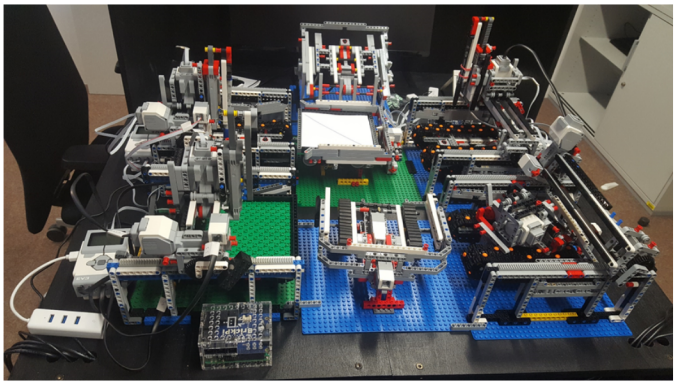

Fig. 8. Lab-scale Production Cell

VII. Evaluation based on Use Case

To demonstrate the feasibility of our approach we applied it on our existing lab-scale production cell, the "Factory in a Box" (FIAB) as depicted in Figure 8.

A. Use Case

The purpose of the FIAB is to serve as a whitebox production plant with full control over configuration and software. This allows to tailor software to the needs of flexible production and employ new control and monitoring concepts.
The FIAB is capable of creating pictures at a level of mass customization. Each station provides the capability of drawing in one color. The stations are able to draw arbitrary graphics. The production plant is completed by an input station, an output station, and turntables able to receive products from and deliver them to each of their sides.
Not shown in the picture is the manufacturing execution system (MES). The MES consists of an order agent managing the production orders and issuing drawing commands to plotting stations. The order agent also triggers a transportation agent that controls the routing of products between the stations.
Any synchronisation between stations happens without intervention from the MES. E.g. a plotting station is only informed to plot the product, its neighbouring turntable is ordered to load a product from its western side. Plotting station and turntable then establish a direct connection to negotiate readiness for handover and acknowledge the finished loading operation. Only the transportation agent is required to be aware of the absolute layout of the production plant, individual systems only know about the location of required skills provided by neighbours.
We used the Lego Mindstorms EV3 platform as it is a complete ecosystem of thoroughly tested actuators, sensors and controllers. This allowed seamless integration and cheap, fast prototyping of the stations. Communication between stations is based on OPC UA only. To create a more diverse environment, we used different programming languages for the stations. Plotters, as well as the input and output stations are programmed according to the IEC 61499 industry standard. We used the implementation provided by the Eclipse 4diac together with the FORTE runtime environment hosting an open62541 OPC UA server. The turntable software is implemented in Java, using the Eclipse Milo OPC UA server implementation. Thus, control software and communication infrastructure can be considered industry grade.

B. Models created for evaluation

The behaviour of plotting stations and turntable was modeled following the PackML [24] standard. To make the behaviour accessible to our debugging assistance, the state machines were modelled in Eclipse Papyrus standard UML, as illustrated in Figure 10. Transitions are guarded by events triggered by signals. A signal is sent on entering a state.
When developing a commercial system, such behavior models very likely were already created during early phases of the

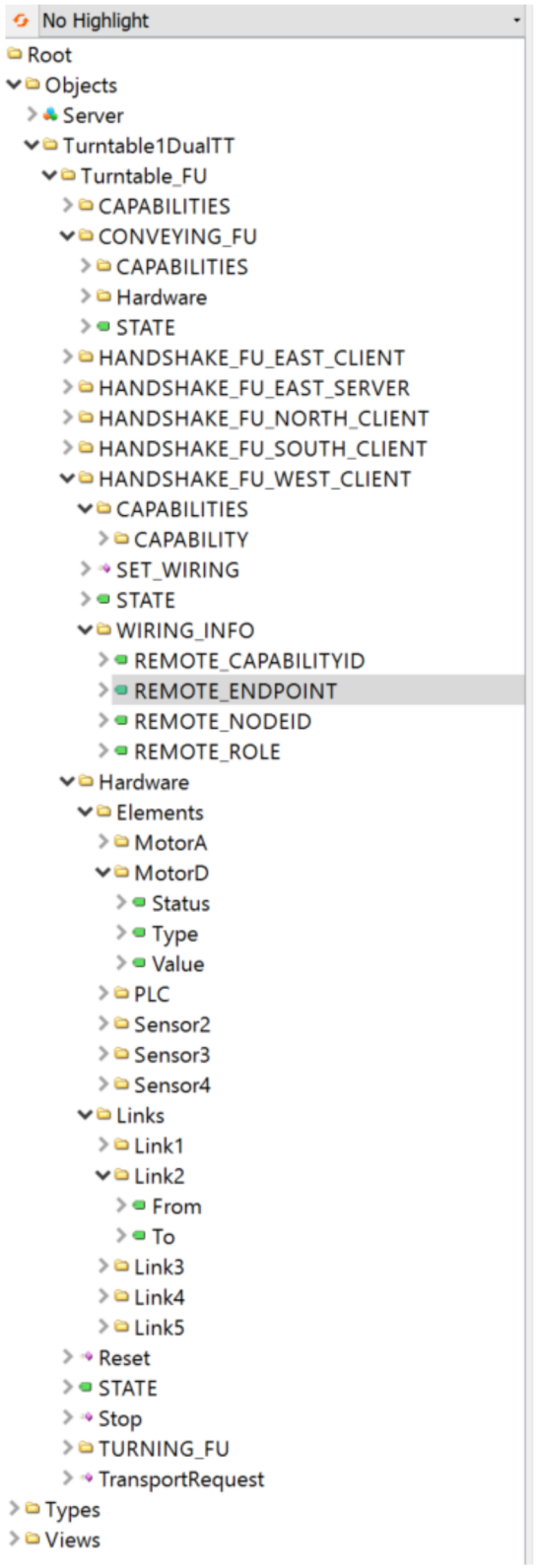

Fig. 9. OPC UA based provisioning of wiring and device information.

development process and are hence available during commissioning and runtime.

The device information and wiring information [23] is directly exposed via OPC UA. Figure 9 displays a partially unfolded OPC UA node set screenshot as browsed with the Unified Automation UAExpert client. Note the exemplary WIRING_INFO folder that provides information to which endpoint the handshake west component is connected to for fulfilling its "handshake" capability and the hardware folder describing the set of motors, PLC and sensors, as well as their links. This information is automatically generated from within the deployed software at runtime.

C. The resulting assisting graph

Our implemented prototype parsed structural hierarchy, wiring information and the UML diagrams to create graph fragments and scanned for matching system URIs, which serve as connection points. We are aware, that graphs are difficult to display. Yet, with the dimensions of our use case, a representation of the resulting information model is given in Figure 11.

D. Manual evaluation

By feeding the graph with status information from the discovered systems, seeding faulty states and triggering the debugging assistance with deviating desired states, the algorithm was inferring the missing messages and recommending correct actions along the search process.

VIII. Conclusion

Debugging the distributed systems of systems will become a common scenario during commissioning and operation of shopfloors in industry 4.0. In this papers we presented a novel approach to connect models present at design-time and runtime, together with states of individual systems, to recommend actions to resolve existing bugs in production machinery. We evaluated the feasibility of our approach in a lab-scale production cell.

Future work would include the assessment of additional overhead in creating models for real use cases, compared to the effort saved when tracking down errors. Further research could also include a history of bugfixes from other machines to give higher weight to likelier solutions.

References

[1] R. M. Balzer, "Exdams: Extendable debugging and monitoring system," in *Proceedings of the May 14-16, 1969, spring joint computer conference*, 1969, pp. 567–580 (cit. on p. 2).

[2] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transactions on Computers*, vol. 36, no. 4, pp. 471–482, 1987 (cit. on p. 2).

[3] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Computing Surveys (CSUR)*, vol. 21, no. 4, pp. 593–622, 1989 (cit. on p. 2).

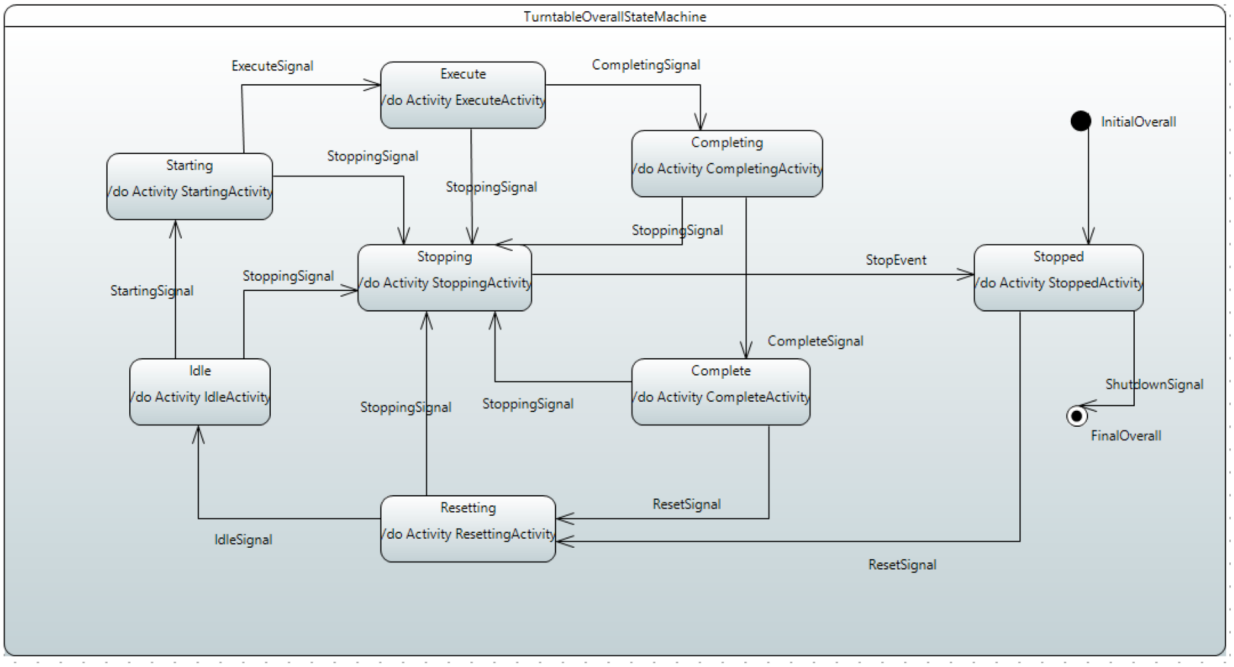

Fig. 10. UML State Machine for FIAB Turntable

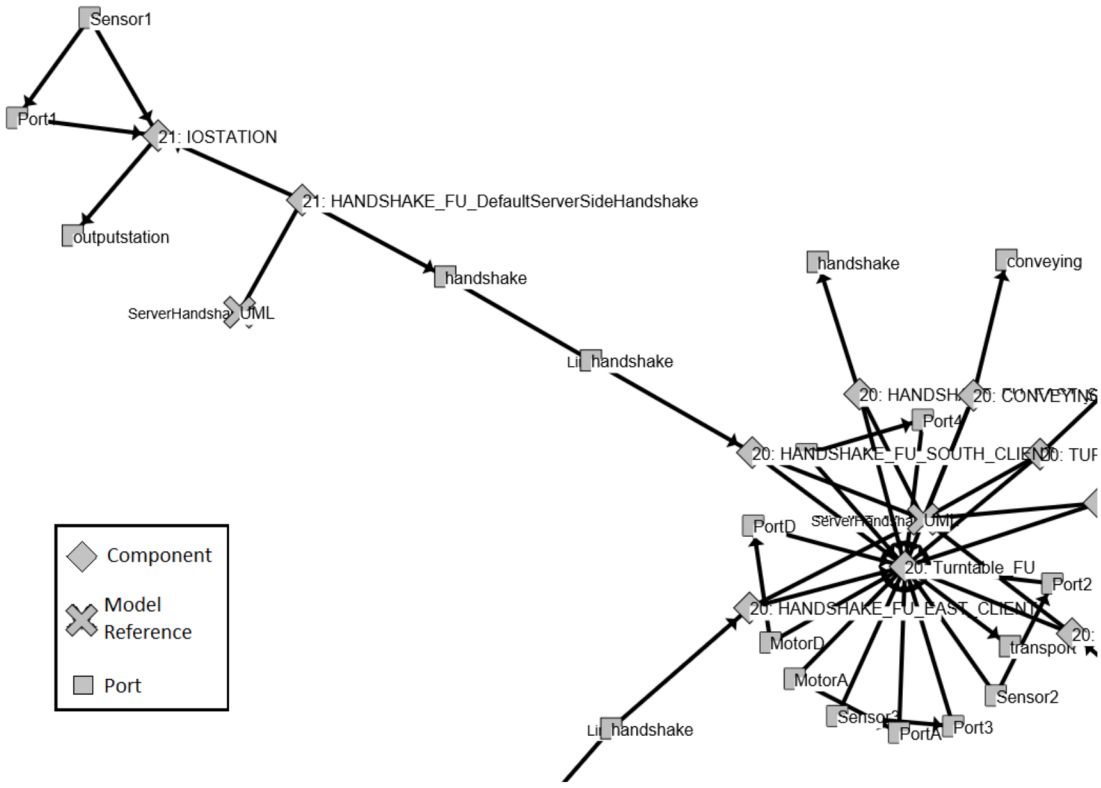

Fig. 11. Internal Visualisation of the Generated Production Cell Graph

[4] D. Spinellis, "Modern debugging: The art of finding a needle in a haystack," *Communications of the ACM*, vol. 61, no. 11, pp. 124–134, 2018 (cit. on p. 2).

[5] I. Lanese, A. Palacios, and G. Vidal, "Causal-consistent replay debugging for message passing programs," in *International Conference on Formal Techniques for Dis-*

tributed Objects, Components, and Systems, Springer, 2019, pp. 167–184 (cit. on p. 2).

[6] A. Fariha, S. Nath, and A. Meliou, "Causality-guided adaptive interventional debugging," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 431–446 (cit. on p. 2).

[7] B. Erb, D. Meißner, G. Habiger, J. Pietron, and F. Kargl, "Consistent retrospective snapshots in distributed event-sourced systems," in *2017 International Conference on Networked Systems (NetSys)*, IEEE, 2017, pp. 1–8 (cit. on p. 2).

[8] C. Torres Lopez, E. G. Boix, C. Scholliers, S. Marr, and H. Mössenböck, "A principled approach towards debugging communicating event-loops," in *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 2017, pp. 41–49 (cit. on p. 2).

[9] C. Hewitt, "Actor model of computation: Scalable robust information systems," *arXiv preprint arXiv:1008.1459*, 2010 (cit. on p. 2).

[10] M. Marra, E. G. Boix, S. Costiou, M. Kerboeuf, A. Plantec, G. Polito, and S. Ducasse, "Debugging cyber-physical systems with pharo: An experience report," in *Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies*, 2017, pp. 1–10 (cit. on p. 2).

[11] P. Reipschläger, B. K. Ozkan, A. S. Mathur, S. Gumhold, R. Majumdar, and R. Dachselt, "Debugar: Mixed dimensional displays for immersive debugging of distributed systems," in *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–6 (cit. on p. 2).

[12] E. G. Boix, C. Noguera, T. Van Cutsem, W. De Meuter, and T. D'Hondt, "Reme-d: A reflective epidemic message-oriented debugger for ambient-oriented applications," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011, pp. 1275–1281 (cit. on p. 2).

[13] D. M. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," Ph.D. dissertation, University of California, Berkeley, 2006 (cit. on p. 2).

[14] D. Dotan and A. Kirshin, "Debugging and testing behavioral uml models," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 838–839 (cit. on p. 2).

[15] L. Guo and A. Roychoudhury, "Debugging statecharts via model-code traceability," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Springer, 2008, pp. 292–306 (cit. on p. 2).

[16] W. Haberl, M. Herrmannsdoerfer, J. Birke, and U. Baumgarten, "Model-level debugging of embedded real-time systems," in *2010 10th IEEE International Conference on Computer and Information Technology*, IEEE, 2010, pp. 1887–1894 (cit. on p. 2).

[17] M. Bagherzadeh, N. Hili, and J. Dingel, "Model-level, platform-independent debugging in the context of the model-driven development of real-time systems," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 419–430 (cit. on p. 2).

[18] M. Bagherzadeh, N. Hili, D. Seekatz, and J. Dingel, "Mdebugger: A model-level debugger for uml-rt," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, IEEE, 2018, pp. 97–100 (cit. on p. 2).

[19] *Opc unified architecture, part 100: Devices*, OPC Foundation. [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-100-device-information-model/ (cit. on p. 3).

[20] P. Ferreira and N. Lohse, "Configuration model for evolvable assembly systems," in *4th CIRP Conference On Assembly Technologies And Systems*, 2012 (cit. on p. 3).

[21] K. Dorofeev and A. Zoitl, "Skill-based engineering approach using opc ua programs," in *2018 IEEE 16th international conference on industrial informatics (INDIN)*, IEEE, 2018, pp. 1098–1103 (cit. on pp. 3, 4).

[22] M. Mayrhofer, C. Mayr-Dorn, O. Guiza, G. Weichhart, and A. Egyed, "Capability-based process modeling and control," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, vol. 1, 2020, pp. 45–52 (cit. on p. 3).

[23] M. Mayrhofer, C. Mayr-Dorn, O. Guiza, and A. Egyed, "Dynamically wiring cpps software architectures," in *2020 22nd IEEE International Conference on Industrial Technology (ICIT)*, IEEE, 2021 (cit. on pp. 3, 6).

[24] "Isa-tr88.00.02 machine and unit states: An implementation example of isa-88," Tech. Rep., 2008 (cit. on pp. 3, 5).

[25] X. Ye and S. H. Hong, "An automationml/opc ua-based industry 4.0 solution for a manufacturing system," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, vol. 1, 2018, pp. 543–550 (cit. on p. 4).