Supporting Diagnosis of Requirements Violations in Systems of Systems

Michael Vierhauser, Jane Cleland-Huang
Department of Computer Science and Engineering
University of Notre Dame
South Bend, IN, USA
mvierhau@nd.edu, JaneClelandHuang@nd.edu

Rick Rabiser, Thomas Krismayer, Paul Grünbacher
Christian Doppler Lab MEVSS, ISSE
Johannes Kepler University
Linz, Austria
rick.rabiser@jku.at

Abstract—Industrial software systems are often systems of systems (SoS) whose full behavior only emerges during operation. They therefore require monitoring techniques to observe systems and detect deviations from their requirements. The focus of existing monitoring approaches, however, is mainly on detecting violations of expected behavior, while support for diagnosing violations is typically limited or even neglected. Diagnosis is particularly challenging in SoS due to their technological heterogeneity and the diversity of development tools in use. Uncovering the root cause of a violation typically requires developers to trace violations to artifacts such as source code or requirements documents, which is difficult without detailed domain knowledge. In this paper we describe our experiences of developing a tool-supported approach facilitating the diagnosis of requirements violations in SoS. We describe how we complemented a requirements monitoring model with a system artifact model relating SoS artifacts needed for diagnosis and required monitoring events. We customized our approach to an industrial SoS and conducted a scenario-based walkthrough with engineers developing the SoS and engineers unfamiliar with it. The results of our evaluation have shown that our approach can significantly ease diagnosing violations in a real-world SoS.

Index Terms—Requirements Monitoring, Systems of Systems, Requirements Traceability

I. INTRODUCTION

Many industrial software systems today are systems of systems with decentralized control, support for multiple platforms, inherently volatile and conflicting requirements, and independent and continuous evolution of its heterogeneous parts [30]. Interactions between an SoS and its environment can only be fully checked during operation when all of its software systems – including legacy and third-party software – interoperate with each other and the monitored events. Furthermore, unforeseen changes, implementation errors, or the deterioration of mechanical parts may change system behavior and thus render requirements invalid even after successful deployment. Runtime monitoring is thus required to observe and check the behavior of SoS during operation.

Various research communities have developed runtime monitoring approaches for diverse domains and different types of checks [33]. Examples include requirements monitoring [43], [23], [35], monitoring of architectural properties [27], complex event processing [46], and runtime verification [12]. The expected runtime behavior is often expressed formally using temporal logic [15]. Also, domain-specific languages have been proposed to define and check runtime constraints [35], [2], [45] through analyzing the stream of events and data collected by probes instrumenting the systems [24].

Unfortunately, few monitoring approaches support diagnosis activities once requirements violations have been detected [43], thus reducing their usefulness in industrial SoS. Due to the complexity and heterogeneity of SoS, engineers diagnosing a requirements violation cannot be familiar with all systems. For diagnosis, engineers thus need easy access to system artifacts such as requirements, design models, or code [26], [34], and the tools needed to manage these artifacts. For instance, trace links to the source code are important for detecting the origin of a requirements violation and debugging it in an IDE, while links to requirements are needed to fully understand the expected behavior [3].

In this paper, we describe our experiences in developing a novel solution for diagnosing requirements violations in SoS and applying it to an industrial automation SoS of our industry partner Primetals Technologies. Our work has been guided by an established model for technology transfer [13] and relies on close cooperation with stakeholders from industry. More specifically, we describe how we extended our existing requirements monitoring framework for SoS (ReMINDS) [45] with a system artifact model (SAM) that establishes traceability between monitored events, requirements violations, and the diverse artifacts needed to support diagnosis. We also discuss how we managed the heterogeneity of existing tool landscapes – an important characteristic of SoS – when developing our diagnosis tool support.

Our paper provides the following contributions: it demonstrates how we developed support for the (i) fine-grained analysis of requirements violations detected by a monitoring infrastructure, by facilitating (ii) traceability to the diverse artifacts related to a violation, and providing (iii) generic interfaces to arbitrary domain-specific tools needed to diagnose and fix violations. The paper also summarizes (iv) what we learned from customizing our solution to an industrial SoS and performing a scenario-based walkthrough involving both engineers developing the SoS and engineers unfamiliar with it to assess the usefulness of our solution.
II. RESEARCH PROCESS

Our research has been conducted in close collaboration with industrial stakeholders. We followed a context-driven research approach [6] to address concrete needs in a specific domain and development project. Specifically, we loosely followed Gorschek’s technology transfer model [13] to investigate the industrial needs and to develop and evaluate a solution for a specific industrial context of a concrete SoS, existing tools, workflows, and processes. Fig. 1 provides an overview of these steps and the degree of involvement from industry and academia. In this paper we focus on the steps we took, starting with discussing a concrete problem with industrial stakeholders and ultimately providing a ready-to-use prototype that was evaluated and refined with industrial experts. We also report plans for staging the release of our approach.

Sections III–VIII of this paper describe each step in detail. We started by analyzing the industrial scenario through conducting workshops and working with domain experts to analyze concrete scenarios (cf. Sec. III). We then worked with domain experts to define key capabilities of the diagnosis approach (cf. Sec. IV). This activity was informed by studying the state-of-the-art and features of existing approaches, which led to multiple refinements of the defined capabilities to address gaps identified in existing work. We then developed a candidate solution that provides these capabilities. It includes: (i) a system artifact model (SAM) (cf. Sec. V), which extends a requirements monitoring model (RMM) [44] to establish traceability between monitored events and different artifacts needed for diagnosis and (ii) a tool (cf. Sec. VI) we developed based on the model, together with APIs and interfaces to other tools. Our approach is customizable to fit the actual needs of industrial users and to seamlessly integrate with the processes and tool landscape in the company (cf. Sec. VII).

Finally, we performed both a ‘static’ and a ‘dynamic’ evaluation [13], resulting in further refinements and improvements of the solution based on feedback from users and domain experts: first, we discussed the solution in a guided walkthrough with both researchers and experts from our industry partner, collected their feedback and adapted the solution accordingly, and second, we conducted a series of scenario-based user studies with domain experts highly familiar with the industrial domain as well as engineers and researchers unfamiliar with the domain to assess whether both groups could benefit from our approach and the provided tools (cf. Sec. VIII).

III. ANALYZING INDUSTRIAL SCENARIO

As a first step towards developing a concrete solution, we investigated the motivation and challenges of monitoring and diagnosing an industrial plant automation SoS of Primetals Technologies. This SoS automates, optimizes, and tracks different stages of the metallurgical production process. It comprises both software and hardware systems for process automation of melting iron ore and raw materials to produce iron, refining liquid iron to produce steel, as well as casting liquid steel into solid slabs. The iron, steel, and continuous casting automation systems include several million lines of code and are developed independently by different teams at different locations.

Although each software system is engineered independently, manifold dependencies need to be considered when planning their joint operation. For example, the YieldMaximizer system, optimizing the arrangement of steel slabs on a strand in the caster, relies on information provided by the material tracking system and can be triggered by the user as well as by other systems. The plant automation SoS is further connected to legacy or third-party systems leading to additional complexity. All systems in the SoS are first tested separately using simulation environments. During commissioning, the software and hardware systems are integrated with third-party and legacy systems for the first time. Monitoring behavior across systems in the SoS is crucial for detecting inaccurate and erroneous runtime behavior during both simulations and operation.

We use the scenario shown in Fig. 2 as a motivating example throughout this paper. Our starting point is the violation of a requirement of the YieldMaximizer system, monitored at runtime. The requirement specifies that data needs to be available from the Tracking system before the YieldMaximizer results can be calculated and that the results of the computation need to be sent to the Analyzer component within 10 seconds. In the monitoring infrastructure this is described as the constraint YieldMaximizer Execution Cycle defining the required occurrence and order of a sequence of events monitored from the (instrumented) running systems, as well as the required time for completion of the event sequence (10 seconds).

If the constraint is violated at runtime, a service engineer (1) is notified about the violation and starts investigating the issue by first trying to understand the immediate reasons that led to the violation (e.g., whether an event was missing or the sequence was not completed in time).

The engineer (2) then continues further reviewing the details and origin of the violation. Especially in an SoS with tight interactions between hardware and software, or between several
Most requirements monitoring approaches do not support engineers and maintenance personnel in performing diagnosis [43] activities (cf. steps 2–4) after a requirements violation has been detected, thus requiring these tasks to be performed manually. As a precursor to automating the process, we met with engineers to identify key capabilities for analyzing requirements violations. We also analyzed support for these capabilities in existing work. We provided an initial description of the capabilities in a short research preview paper [42] and discuss them in more detail below.

**Providing details about violations.** Engineers analyzing a requirements violation require detailed information to reveal its root cause. This includes basic information such as the violation type, the time it occurred, and potentially involved components and systems in the SoS. Additionally, the history of recorded events, event data, and violations are needed to diagnose certain types of problems such as the gradual degradation of performance. Some monitoring approaches provide visualization capabilities or basic explanations for violations. Baresi and Guinea present the ECoWare framework [2], which provides a dashboard to visualize runtime data in live charts and an event history. The Kieker monitoring framework [40] provides visualization capabilities for analyzing performance violations. The approach by Müller et al. [28] supports explanations of runtime violations to service-level agreements in service-based systems. However, as reported by these authors, existing monitoring approaches (for service-based systems) do not support a fine-grained and detailed analysis of violations.

**Providing easy access to related system artifacts.** Engineers analyzing the origin of a violation in an SoS typically need to inspect diverse artifacts such as source code, documents, models, or log files, often at multiple locations and machines. Their task could be simplified by providing trace links to these artifacts. Existing approaches use code-level assertions [20] to establish traceability in case of violations. For instance, the approach by Cito et al. [7] automatically generates trace links between runtime traces and source code to support diagnosis in large-scale cloud environments. Some existing monitoring approaches also link requirements to higher-level models, e.g., goal models [35] or UML diagrams [40], but do not link violations or other events to system artifacts. Traceability tools [17], [10] have partially addressed these problems but do not incorporate runtime events and data.

**Dealing with heterogeneous tools and development environments.** Due to the diversity of SoS, different teams use diverse methods and tools when working with system artifacts. Diagnosis support thus relies on “plugging” domain-specific tools into a monitoring infrastructure, e.g., various IDEs are needed to inspect and edit source code linked to a violation, while other violations may require access to modeling tools or document editors. Even though existing monitoring approaches [35], [40] interface with other tools to visualize monitoring data, they do not support the integration of third-party tools to aid in the inspection of violations.

---

**IV. Defining Diagnosis Capabilities**

software systems, the actual root cause of a violation is often not immediately obvious. For example, the violation could be the result of a database performance problem preventing completion of the sequence in time, or it could have been caused by a broken communication link between the YieldMaximizer and the Analyzer component (event result_sent did not occur). It could also result from a defective hardware sensor not reporting the correct parameters needed for the calculations (event data_received did not occur). A first step is thus to find out if the events have been executed in the expected order and if data has been provided.

This analysis typically also involves checking the related artifacts of the involved systems. For instance, the engineer would explore the source code related with this scenario (of the Tracking, YieldMaximizer, and Analyzer systems) as well as requirements specification documents providing additional information on the systems, and their potential interaction and expected behavior. For that purpose the engineer uses external tools such as repositories or IDEs to (3) further investigate the causes of the violation by exploiting traceability to documents and source code.

Depending on the severity and criticality of the violation the engineer will (4) perform different actions on the artifacts related to the violation. For instance, in a simple case the engineer may just review the code artifacts linked to the violation, i.e., the monitored methods in the YieldMaximizer code. A more experienced engineer may further decide to debug the potentially faulty locations in the simulation environment by automatically creating breakpoints and remotely connecting a debugger. Finally, if the violation cannot be fixed without deeper analysis, the engineer might create an issue in an issue-tracking system to notify the responsible developers.
**Supporting diagnosis activities with tool actions.** Depending on the violation to be diagnosed, different tool actions and services will be required. Researchers in the area of Activity-Based Computing [31], [38] have stressed the importance of an abstraction mechanism for describing and collecting different user actions and resources required for these actions. For instance, to determine why a certain event did not occur, the developer may review and edit source code in an editor of an IDE, or to investigate a failed data check she may inspect configuration files or use a debugger.

V. DEVELOPING A MONITORING AND DIAGNOSIS MODEL FOR SYSTEMS OF SYSTEMS

Models are increasingly used to monitor and verify different aspects of runtime behavior and support runtime decision-making [4]. We developed a software model for monitoring and diagnosis that relates development-time and runtime elements to provide the previously described diagnosis capabilities. Specifically, it uses an existing requirements monitoring model (RMM) [44], extended with model elements for diagnosis and artifact traceability (Fig. 3).

A. Requirements Monitoring Model (RMM)

The RMM [44] comprises the following elements: Monitoring Scopes define the areas of interest to be monitored. They hierarchically represent the architecture and, to a certain extent, the organizational structure of the SoS. For instance, a scope may represent a particular system, one or more components, or connectors (such as interfaces or APIs) between different parts of an SoS. SoS Requirements describe functionalities, properties, or behavior of the SoS that has to be fulfilled at runtime. Modelers define Constraints that formalize SoS requirements to be checked at runtime based on events and event data. The RMM is not limited to specific types of constraints as long as they are related to events and event data provided by probes. Typical constraints define expected event sequences, event timing [11], or allowed data ranges:

- **Data-related constraints** check data attached to a monitored event, for example, in the plant automation SoS the “Run Id” of a YieldMaximizer run has to be a number greater than zero.

- **Event-related constraints** are used to check the occurrence, timing and expected correct order of events. For example, one future occurrence constraint for the plant automation SoS checks that after receiving a request from another system such as material tracking or the human-machine interface, the YieldMaximizer starts calculations within 10 seconds.

Another example is an event sequence constraint checking that the YieldMaximizer operates in the expected order: reading data from the underlying machinery or production planning system, performing optimization calculations, validating the calculations, and distributing the calculations to other systems. The constraint is violated if any event in the defined sequence is missing or if events occur in a different order.

SoS Events and data are collected by Probes instrumenting different systems in the SoS. They conform to event models providing a uniform representation and common scheme of events and event data. Each constraint in the RMM is related with one or more events needed to check for requirements violations (and each requirement may be checked using one or more constraints). The elements defined in the RMM provide the foundation for monitoring the SoS at runtime and are created, e.g., by domain experts based on their knowledge and experience of the system. The links between requirements and scopes as well as between probes and scopes allows tracing violations to their origin, such as the affected elements of the SoS architecture.
B. SoS Artifact and Diagnosis Model (SAM)

We extend the RMM with the SAM comprising additional elements to manage violation details and support diagnosis tasks via tool actions for different system artifacts.

**Constraint Check Result.** Storing the results of constraint checks is a prerequisite for supporting extended diagnosis beyond simple violation reporting. As we assume an event-based monitoring approach, constraint checks are triggered by events as soon as they occur. The SAM thus stores – for each constraint check – information on the event which triggered the check. In case of a constraint violation this allows retrieval of the trigger event for further analysis. The attribute type allows distinguishing different types of constraint check results, e.g., event timing violation, event order violation, failed data check, and consistent state (if no violation was detected). Revisiting the history of recorded constraint check results may be of interest during diagnosis – even for consistent results – to analyze points of failure or to uncover trends of gradual degradation. For example, if a violation of the YieldMaximizer constraint described above is detected, the history might reveal potentially suspicious behavior, such as degradation of the completion time. Furthermore, different severity levels (e.g., error, warning) can be defined to classify violations.

Additional **Diagnosis Data** can be provided via the SAM for each constraint check result if required. This data varies depending on the type of constraint and the type of constraint check result: while a data constraint requires details on the violated data condition (e.g., expected vs. actual values), a constraint evaluating the execution order of an event sequence requires information on the actual sequence, the time between events, as well as missing or unsuitable events. This diversity of diagnosis data also affects tool support for diagnosis, requiring custom features for handling certain kinds of violations.

**Artifacts.** In an SoS many different stakeholders are involved in requirements engineering, implementation, and maintenance. Thus, a wide variety of artifacts exist that need to be retrieved to provide detailed information on a violation. In the plant automation SoS this includes specification documents, system administrator documentation stored in document management systems, source code maintained in repositories, or problem reports managed in issue-tracking systems. To support the heterogeneity of needed artifacts we employ a dedicated meta-model for defining domain-specific **Artifact Types** such as Requirements or Source Code. Each type may provide additional attributes, e.g., specifying the file type or the storage type, which can then be used by external tools to retrieve, display, or modify the related artifact.

**Trace Links.** Diverse system artifacts are related with the elements of the requirements monitoring model [34], [14], [8]. For instance, requirements or goal models are commonly related with a constraint [9], source code and configuration files are related to events and probes, and architectural models and diagrams are often related with monitoring scopes [39]. Artifacts related to a violation can be presented together with the intended actions for the type of violation. Depending on the type of violation, certain types of artifacts will have higher importance: for a data violation a trace link into source code or configuration files may quickly reveal the root cause of the problem, while diagnosing a violation of a sequence constraint may require code reviews across different repositories and documents describing the desired behavior. The trace that supports our model can be constructed manually [34], [32], collected as a byproduct of the development process [1], or constructed with the aid of machine-learning and information retrieval techniques [22].

**Diagnosis Tasks & Actions.** Different diagnosis tasks can be performed depending on the type of violation, e.g., displaying requirements documents related to the identified problem, or highlighting a method in a source code file. Following Activity-Based Computing [31] concepts, the SAM defines the actions and resources required for diagnosing violations. **Internal Actions** can be performed directly within a monitoring and diagnosis tool (e.g., viewing related events) while **External Actions** exploit trace links to external artifacts and tools.

C. Customizing the Model

Before the SAM can be used together with the RMM at runtime to detect and analyze violations it needs to be tailored and set-up for a specific system by: (1) assessing available artifacts and selecting artifacts relevant for diagnosis support, (2) establishing trace links between SAM artifacts and RMM model elements, and (3) defining actions and implementing handlers for the respective artifacts.

**Assessing & Selecting Available Artifacts.** As SoS typically consist of hundreds of components and potentially thousands of artifacts, a brute force approach using all available artifacts will likely fail. We thus propose to carefully assess and select artifacts that provide potential benefits to the diagnosis process. A valuable starting point is the source code instrumented with probes. Together with domain experts, further related and relevant artifacts can be identified for the various components of the monitored system. These artifacts can be added to the **SoS artifact model** by first defining their type in the SAM meta-model and then creating instances for the concrete artifacts.

**Establishing Trace Links.** The artifacts are then linked to one or more elements of the RMM. This can be done manually, e.g., by assigning a linked specification document to a particular scope, or automatically, by exploiting information already present in the RMM. For instance, trace links between source code artifacts and constraints can be created automatically based on the information provided by the probes instrumenting the system. Further, when using AspectJ to instrument Java systems the respective pointcuts establish trace links between probes and source code fragments.

**Defining & Implementing Actions.** The monitoring tools and their extensions enable the use of the SAM artifacts at runtime. Thus, **application-specific action handlers must be implemented** as extensions to the monitoring tool. This step is highly dependent upon the previously selected artifacts and the set of existing tools that are in use. For example,
a simple action handler might only be capable of opening a PDF document, while a more advanced one might handle Java source code files by directly opening an engineer’s IDE and highlighting a specific code fragment.

To support evolution of the SAM, our tools provide means for adding, updating, and modifying linked artifacts and associated actions at any time. This means, for example, that new documents can be added at runtime, storage locations can be modified and links to source code can be updated when new probes register at the monitoring infrastructure.

VI. DEVELOPING DIAGNOSIS TOOLS

We implemented the previously described diagnosis capabilities within ReMINDS. In this section we focus on the extensions made for providing diagnosis support that go beyond what the monitoring tool [21] already provides. Specifically, we extended our tool with capabilities to define and use the SAM for diagnosis (cf. Fig. 4). This includes additional interfaces and editors for managing artifacts, tool support to perform diagnosis tasks, and extensions to interface with external artifacts and tools.

A. Implementing & Populating the SAM

We implemented both parts of the SAM, the meta-model and the concrete instances created based on this meta-model, in Java. Our tool support manages the models as XML files and provides Eclipse RCP-based editors for engineers. One editor supports artifact meta-model definition for a specific domain including hierarchies of artifact types and supporting assigning actions to certain artifact types. Based on a particular meta-model, another editor in our tool allows engineers to populate a SAM with concrete artifacts and trace links to RMM elements. The types of valid trace links depend on the specific artifact type. For example, a source code artifact can be linked to a certain event type – which means that a trace link contains information on the deployment entity (e.g., a jar file), the class name, and the method. Our implementation also provides a Java-based API enabling queries against the SAM for trace links and artifact information.

B. Creating Actions/Handlers interfacing with External Tools

Multiple development environments and tools are used to develop and maintain the diverse artifacts of various systems in an SoS. We have thus implemented an Artifact Registry that allows different Handlers to register for different kinds of artifacts and declare actions that can be executed. The handlers decouple the monitoring tool from the actual mechanism responsible for retrieving the artifact, e.g., from a repository, document storage or shared network folder. Our tool subsequently can perform diverse Actions on the artifact as defined in the SAM. For example, for Java source code, we implemented a concrete handler that is registered with the Artifact Registry and manages this type of artifact. This handler is realized as an extension added to the Eclipse IDE used for managing Java source code. In case a corresponding action in ReMINDS is requested the handler retrieves information on the artifact and trace link as defined in the SAM using our API – e.g., the location of the Java class and the name of the method – and then opens the file in the Eclipse IDE.

C. Tool Support for Diagnosis

We implemented the diagnosis capabilities on top of ReMINDS. Our tool can access the interfaces of both the RMM and the SAM and accumulate information on constraint checking results and access information on artifacts linked to related model elements via the SAM API. We extended the Eclipse RCP-based user interface of ReMINDS (cf. Fig. 4) to present – for each violation – an overview of all related events, probes, constraints, and requirements from the RMM, as well as the related artifacts as defined in the SAM. This overview enables engineers to review detailed information on the actual purpose of the constraint, the immediate reason of the violation, the involved events and their status, and the artifacts related. If external tools have been linked via handlers, the user can directly open the selected artifact with the linked tool.

VII. CUSTOMIZING THE DIAGNOSIS PROCESS

A key factor that impedes the successful use of academic tools and prototypes in industry is the lack of compatibility and integration with existing tools and processes [6]. Thus, in addition to providing tool support for our implementation, tailoring the tools and customizing the approach to Primetals Technologies’ environment was crucial. This step comprises the selection of artifacts relevant for diagnosis, providing customized user interfaces in the tool and establishing interfaces to well-established tools used within the company. An overview of the actions and artifacts that were deemed relevant for diagnosis purposes for the plant automation SoS can be found in Table I.

Assessing & Selecting Available Artifacts. Through conducting a series of workshops we collected internal and external actions. Specifically, in a first step we selected 15 actions that were deemed important for the monitoring and diagnosis process and could be implemented with reasonable effort. The artifact types included documents, such as requirements specifications; source code (Java source files or jar files) and C# code for the casting system, C++ code for the ironmaking system; issues such as Mantis issues for documenting problem reports and requesting missing features; in addition to system files such as property, configuration, or log files.

Establishing Trace Links. We then established trace links between the artifacts and constraints for the different systems we collected in earlier work [44]. Depending on the type of artifact, we supported trace link creation via a UI of ReMINDS (e.g., linking requirements documents to certain constraints) or automated support for establishing trace links between source code and runtime events and their attached data (e.g., analyzing the Java classes instrumented via probes).

Defining & Implementing Actions. For each diagnosis task we then assigned internal actions supported by the diagnosis tool and external actions for the different artifact. The five
TABLE I: Internal and external diagnosis actions for the plant automation SoS with associated RMM elements & SAM artifact types.

<table>
<thead>
<tr>
<th>Internal Actions</th>
<th>Associated RMM/SAM Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Review Violation Reason</td>
<td>Constraint Check Result, Diagnosis Data</td>
</tr>
<tr>
<td>Review All Events Uncover Events</td>
<td>Diagnosis Data, Events (stored in DB)</td>
</tr>
<tr>
<td>Inspect Constraint / Requirement</td>
<td>Constraint, Requirement</td>
</tr>
<tr>
<td>Inspect Event History</td>
<td>Events (stored in DB)</td>
</tr>
<tr>
<td>Inspect Constraint Check History</td>
<td>Previous Constraint Check Results</td>
</tr>
<tr>
<td>Inspect Event Details</td>
<td>Event Data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>External Actions</th>
<th>Artifact Type</th>
<th>Artifact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Code Artifact</td>
<td>Code</td>
<td>File, Jar File, Groovy</td>
</tr>
<tr>
<td>Open Issue (Category)</td>
<td>Issue</td>
<td>Mantis Issue</td>
</tr>
<tr>
<td>Open Log File</td>
<td>Log</td>
<td>System Log or Comp.</td>
</tr>
<tr>
<td>Open System File</td>
<td>Sys. File</td>
<td>Text File, Java Property File</td>
</tr>
<tr>
<td>Highlight Code Artifact</td>
<td>Code</td>
<td>Java Source Code, Jar File</td>
</tr>
<tr>
<td>Create Breakpoint</td>
<td>Code</td>
<td>Java Source Code, Jar File</td>
</tr>
<tr>
<td>Create New Issue</td>
<td>Issue</td>
<td>Mantis Issue</td>
</tr>
</tbody>
</table>

internal actions comprise viewing a detected violation, inspecting events relevant for a violation, inspecting event details of a certain event (e.g., the attached data), inspecting the constraint check history of a specific constraint, and inspecting past events (of the same type). For the external actions we identified actions for opening different types of artifacts, e.g., a source file at a certain line or method, highlighting certain areas of a file (e.g., a method responsible for a constraint violation), creating breakpoints in the source code, or creating a problem report in the issue-tracking system.

For each external action and its associated artifact type we implemented a handler to execute the desired action, e.g., to open Java files in the Eclipse IDE and highlight certain areas of a file (e.g., a method responsible for a constraint violation), creating breakpoints in the source code, or creating a problem report in the issue-tracking system.

For the purpose of issue tracking, we used the SOAP API provided by the Mantis Issue Tracker [25] used by Primetals Technologies, which allows us to create new issues and automatically add relevant information (e.g., contained in the diagnosis data) to an issue. Regarding the population of the SAM and implementing the handler this took about 4 to 5 person days and was performed by one of the authors of this paper, supported by an engineer of Primetals Technologies.

VIII. EVALUATING THE DIAGNOSIS APPROACH

Following Wieringa’s classification of design science research [47] our work can be considered an improvement problem: to develop and evaluate the SAM and its associated tools with the aim of improving practice and satisfying the needs of real-world stakeholders [6]. This requires us to consult relevant stakeholders in order to understand their goals and improvement criteria for a tool-supported method facilitating the diagnosis of violations in SoS. Specifically, we are interested if our approach eases diagnosis in SoS monitoring. We thus investigate the perceived benefit of using our approach for diagnosing violations in an industrial SoS. We were also
interested in the extent to which engineers with and without domain knowledge of the SoS benefit from the approach. As an initial step, we performed a guided walkthrough [37] and then continued with an in-depth user study involving different types of user, who explored the effects and perceived value of our diagnosis tool in a real-world scenario.

A. Guided Walkthrough

We conducted the guided walkthrough with two experienced users: one engineer familiar with the domain from our industry partner and a PhD student with three years of industry experience as a software engineer but without familiarity with the domain. In two different sessions, lasting approximately two hours each, we used mock-ups of different YieldMaximizer constraint violations (cf. Sec. V) and asked the two participants to work with our tool and diagnose the problems while thinking-aloud [18].

Both participants mentioned, for example, that in some cases the meaning of the elements displayed was hard to comprehend or ambiguous, and that using the tools and the different views was not always straightforward due to the lack of guidance. Based on their feedback we updated and improved REMINDS regarding its usability, e.g., by adding additional descriptions and improving the arrangement of events and data in the view. We also provided additional capabilities to accommodate both domain experts and engineers less familiar with the given violations and artifacts.

B. Scenario-based User Study

Using this improved version of REMINDS, we conducted a study with six participants: two experts from our industry partner familiar with the domain and SoS, two experienced software engineers unfamiliar with the domain and SoS, and two researchers (not involved in the development of the tools or the approach and also unfamiliar with the domain and SoS). Similar to the evaluation performed by Cito et al. [7] for their runtime diagnosis approach, we compared our tool against a baseline established by one of the two experts who had been provided with all necessary information about the constraint violations but not the diagnosis tool. We discussed with the expert what tools he would use, how he would find the reason for each violation, and what concrete activities he would perform (which we documented and counted as steps), thus forming our baseline of action steps. For example, one activity we counted as a step was searching for a particular method or class name in the source code.

We defined three different violation scenarios of increasing complexity which the study participants had to diagnose. These scenarios addressed: (A) a data violation, (B) a future event occurrence violation, and (C) a sequence violation (these were violations of the YieldMaximizer constraints we described in Sec. V). REMINDS provides different information for each scenario such as the data attached to an event or the order of occurring events and links to different artifacts such as source code or requirements documents. Here, an example that we counted as a step is opening and reviewing source code in the IDE at a specific location after clicking on a linked artifact. All internal and external actions we collected and created as part of customizing our approach (cf. Sec. VII) were used in the study.

The steps could be performed in any order and as many times as necessary for both the baseline and the tool-supported diagnosis, e.g., when switching back and forth between different artifacts and the tool. Following the diagnosis activities, we asked the participants to describe positive and negative aspects of using our tool to diagnose violations and to explore linked artifacts. One researcher served as moderator introducing REMINDS and its capabilities. In total, three researchers (the moderator and two additional people) documented the steps using observer and think-aloud protocols. Each participant took about 30 minutes to complete all three tasks (diagnosing the three violations associated with the three constraints) followed by a brief discussion. We deliberately only counted steps and did not measure the time for completing a task as this would have conflicted with our qualitative goals of eliciting discussions and encouraging think-aloud statements useful for further improving and evolving the approach and the provided tool-support.

Table II provides an overview of the participants of our study and shows the number of steps they performed for the three different violation scenarios of our study.

For establishing the baseline the steps the engineer said he would conduct included remotely connecting to the system (running on a separate machine) and inspecting log files and other documents for retrieving details regarding the constraint violation. This includes activities such as manually searching through this files for similar items/names as the ones used in the constraint (the event names and data items), and searching the source code for identifier names (e.g., class names) similar or equal to the events and data items used in the constraint. The steps for investigating each violation were rather similar and, according to the participant, follow the established practices for how bugs and errors are investigated when they are reported by the customer. In total, the baseline for the scenarios consisted of eight steps for scenario A and B and twelve steps for scenario C. We then conducted the study separately with the five other participants using REMINDS to find the origin of all three constraint violations in the respective source code.

For scenario A (data violation) four to eight steps were necessary to pin down the origin in the source code. The expert from the industry partner was able to find the violation by performing only four steps since he was familiar with the code and had – in addition to the trace links provided by REMINDS – additional knowledge of how and where each particular data element was used. When compared with the baseline, on average 1.8 fewer steps were necessary when using our approach.

For scenario B (future occurrence violation), five or six steps were necessary. All participants analyzed the two events involved in the constraint and were able to identify the origin of the violation, which was “hidden” in a method called by the class to which a trace link was provided.
TABLE II: Overview of the steps performed by the participants compared to the Baseline and the Experience (Exp.) of the participants.

<table>
<thead>
<tr>
<th>Exp. (years)</th>
<th>A-Data (no. of steps)</th>
<th>B-Future Occ. (no. of steps)</th>
<th>C-Event Seq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Expert</td>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Engineer I</td>
<td>6</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Engineer II</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Researcher I</td>
<td>4</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Researcher II</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Avg. Delta</td>
<td>△ − 1.8</td>
<td>△ − 2.6</td>
<td>△ − 4</td>
</tr>
</tbody>
</table>

In addition to the fewer number of steps required when using the tool, we also observed for scenarios A and B that participants took less time to navigate within the artifacts. Specifically, participants were able to browse the code and look for related methods or parameters in the source code based on the information provided in the diagnosis view.

Scenario C (violation of an event sequence check) was intended to be the most complex one, which was confirmed by all participants. Again, all participants were able to identify the location of the violation in the source code. An average of four fewer steps were needed when using ReMINDS. However, all participants unfamiliar with the domain and the source code expressed that they “think that they found the problem [...] but are not 100% sure” and would seek to consult an additional expert, e.g., a developer of that component, to confirm their assumption.

C. Summary and Insights

Here we provide and discuss some qualitative insights provided by our participants. Specifically, we asked them about the positive and negative aspects (i.e., perceived advantages and disadvantages) when using the tool and how they would have diagnosed the violations without ReMINDS.

All study participants not familiar with the domain acknowledged that it would be significantly more time-consuming and cumbersome, if not impossible, to find the root cause without the trace links to the source code. Participants stated that they could “manually search for the source code in the IDE, [...] for example, by looking for similar parameter names” (in case of the data violation) or “for similar class and method names as the events”. The steps we counted therefore are likely to be an underestimation of the actual effort that can be saved. Another threat stems from the fact that only a single individual was used for creating our baseline. However, the chosen person was an expert with many years of experience with the SoS which we think is well-qualified to estimate the steps, activities, and tools he will use for analyzing an emerging problem in the system. For the collection and analysis of the steps performed, three researchers independently observed the participants and took notes. We are aware of the fact that different steps are not directly comparable in terms of the time spent for completing a certain step. It might take considerably longer to search for a specific method or class name in the IDE than to click on a button and follow a trace link to the location of the source code.

The study showed that even experts not familiar with the domain were able to find the origin of a constraint violation faster and that they felt more confident compared to just searching for similarly named classes or methods. Two participants explicitly mentioned as one of the advantages that “information is accumulated and presented in a single view” and that they do not need to switch between different windows of the monitoring environment to view the constraint, related events, and the actual violation. Two participants also mentioned they liked “the color coding” (events highlighted in red indicate a problem, events highlighted in green indicate unaffected events) “as one immediately can see where to start looking for the problem”.

We conclude that both, the guided walkthrough and the user study with participants with different levels of experience and domain knowledge demonstrate that our approach and our tool can be successfully used to diagnose runtime violations. We further showed that less effort is necessary to uncover the origin of a violation compared to a baseline established by a domain expert without tool support.

D. Threats to Validity

A threat to construct validity is the potential bias caused by the system we selected for our research. We selected the parts of the SoS for which we provided monitoring and diagnosis support and implemented tools based on a thorough analysis and discussion with industry experts. For the sake of study feasibility, we constrained the origin of all investigated violations to be within the source code, and assumed that it was not caused, for example, by sensors or other hardware parts (as this would have required access to actual hardware and machinery).

Regarding internal validity, i.e., the results might have been influenced by our treatment, we used a predefined research protocol and followed guidelines and best practices for conducting user studies [36], [18]. We selected subjects with different levels of experience and also included non-domain experts in the study. According to feedback from our industry partner it is very likely that service engineers responsible for initially diagnosing a reported violation lack in-depth knowledge of the overall system. We chose the most experienced engineer (i.e., the one most familiar with the domain) as the source for the baseline. The resulting difference in steps thus indicates the minimum amount of effort that can be saved. Another threat stems from the fact that only a single individual was used for creating our baseline. However, the chosen person was an expert with many years of experience with the SoS which we think is well-qualified to estimate the steps, activities, and tools he will use for analyzing an emerging problem in the system. For the collection and analysis of the steps performed, three researchers independently observed the participants and took notes. We are aware of the fact that different steps are not directly comparable in terms of the time spent for completing a certain step. It might take considerably longer to search for a specific method or class name in the IDE than to click on a button and follow a trace link to the location in the source code.

With respect to external validity, and the generalizability of our findings, we selected a system representative in terms of size and complexity for the automation software domain. Although the results and drawn conclusions depend on a single SoS and three constraints/violations only, we are confident that the results are valuable for other domains and systems as well. The intention of our work is to provide a generic approach explaining constraint violations and providing links to arbitrary artifacts with a tool that is easily extensible. While we can
not argue about the specific ROI describing the efficiency of investing in the approach, we are confident that the approach and the accompanying tools are useful in several different scenarios.

IX. LESSONS LEARNED

We learned the following lessons when working with industrial engineers to develop and evaluate our diagnosis approach:

Keep the mapping between monitored events and the underlying artifacts close. One lesson we learned when observing the participants is related to the Cognitive Dimension “closeness of mapping” [5]: while the domain expert that used our tool had no problems finding the data element mentioned in scenario A, in both the linked specification document and the source code, the other participants were initially confused. Although we provided the trace link to the respective method, they were confused by naming differences (e.g., “optimizeFinished” in REMINDS vs. “optimizeCompleted” in the source code). The same holds true for the other scenarios in which event names were similar but not identical. This leads us to the conclusion that providing additional information about the linked artifacts is necessary, e.g., a stack trace as suggested by one of the participants, a documented rationale for the link [16], or requirements documents that can provide important information. Not only can this information be used to establish trace links between source code artifacts and runtime event and data but moreover, this can also facilitate traceability to the respective requirements.

Consider the diversity of diagnosis actions required to support different users. When assigning actions for specific diagnosis tasks one should keep in mind that there is no ultimate set of actions fitting a task. In our evaluation we learned that people with different experience and background analyze violations differently. Some might start by looking at the actual reason for the violation, and then continue by reviewing related elements, while others prefer to directly navigate to the source code to investigate the issue. Furthermore, different kinds of stakeholders are interested in different kinds of information and artifacts. While engineers may be primarily interested in the source code, a project manager might prefer to explore the origin of the problem and its potential impact at the level of requirements. We thus suggest providing a broad and extensible set of actions for each diagnosis task to support different user preferences.

Let users keep their preferred tools. When instantiating the model and implementing tool prototypes we received feedback that not all actions should be included in the monitoring tool. For instance, when we suggested directly showing source code and property files in the monitoring tool engineers stated that they would prefer to view and modify such files in their own IDE. Thus, external actions should be defined for artifacts and tools already in place, allowing users to use the tools they have grown accustomed to. Our architecture provides this flexibility by implementing handlers to define a good mix of external and internal actions which can also include dedicated requirements management tools, supporting the definition of diagnosis actions that directly link to the involved requirements.

Automate model population and maintenance. A concern raised by industrial users is the maintenance of the models and trace links. Since the underlying systems are frequently evolving, the trace links to the artifacts have to co-evolve. It is thus important to reduce the maintenance effort by automating the population of the SAM and the trace links to artifacts. In our case, trace links between probes or event types and the associated source files can be created automatically: we use AspectJ [19] for instrumentation, which allows us to automatically identify the locations in the source code to then generate (and update) trace links to probes and event types. Additionally, consistency checking strategies can be employed to ensure consistency between source code artifacts and the models, e.g., as proposed by Murta et al. [29], and shown in our own earlier work [41].

X. CONCLUSION

Requirements monitoring in practice needs to provide more than a simple assertion about whether a constraint has been violated or not. Engineers need detailed information on violations and assistance for diagnosing the root cause of a problem. This is an inherently complex problem to solve and requires research solutions to be embedded within the complex operating environment of the target system.

This paper thus reported the collaborative process we engaged in to solve the pressing problem of diagnosing violations by extending an existing monitoring approach for SoS with diagnosis capabilities. Our approach supports the definition of a model describing different artifact types relevant for a particular domain, their relation to events and constraints, alongside the actions that can be performed using these artifacts. Engineers and tool developers can contribute custom-implemented artifact handlers to integrate existing tools into the monitoring environment. We performed a guided walkthrough and a user study to assess the effects of our approach in a realistic environment. The results suggest that engineers can benefit from the presented capabilities, regardless of whether they are familiar or unfamiliar with the domain and SoS.

In future work we will refine our approach by considering the lessons we learned. This should provide the foundation for releasing the approach within Primetals Technologies for the next phase of testing against selected parts of the plant automation SoS. We are confident that other organizations could benefit from our reported experiences when applying runtime monitoring for their systems.

ACKNOWLEDGMENTS

This work has been supported by the Christian Doppler Forschungsgesellschaft Austria, Primetals Technologies, and the Austrian Science Fund (FWF) under Grant No. 13998-N31. We particularly want to thank all the participants of our walkthrough and our user study.
REFERENCES


