Web-Based Tracing for Model-Driven Applications

Jörg Chrisitian Kirchhof, Lukas Malcher,
Judith Michael, Bernhard Rumpe
Software Engineering
RWTH Aachen University
Aachen, Germany
https://se-rwth.de

Andreas Wortmann
Institute for Control Engineering of
Machine Tools and Manufacturing Units (ISW)
University of Stuttgart
Stuttgart, Germany
https://www.isw.uni-stuttgart.de

***Abstract*—Logging still is a core functionality used to understand the behavior of programs and executable models. Yet, modeling languages rarely consider logging as a first-level activity that is manifested in the language through modeling elements or their behavior. When logging is part of the code generated for the respective models or the corresponding runtime environment only, it must be generic, as the modeler cannot influence, through the models, what and when logging takes place. To enable modelers to log model behavior, we devised a method based on language extension and smart code generation that can integrate logging into arbitrary textual modeling languages. Based on this method, log entries can be produced, traced, and presented through a web application. This method and its infrastructure can facilitate lifting logging to the model level and, hence, improve the understanding of executable models.**

***Index Terms*—Software Engineering, Model-Driven Development, Internet of Things**

I. Introduction

Logging is a core functionality used to understand the behavior of programs [1] and executable models. Logs are used by software developers "to locate the sources of errors and problems in [running] software systems" [2], and enables system administrators and operators to understand the actual use of software systems in the normal operation of the same system. In this paper, we use the term logging to refer to the production of log messages and the term tracing to refer to the process of locating the root cause of a particular behavior or condition. Logging in combination with model-driven software engineering (MDSE) is challenging, as modeling languages rarely consider logging as a first-level activity. MDSE treats models equal to code and, thus, leads to increased productivity, quality and maintainability [3].

Existing approaches in research and practice lack comprehensive user support for the combined tracing and filtering of logs (*cf.* Sec. V). The tracing of execution flows is common in microservices infrastructures, where requests are tagged with meta-data and handled in a central instance to visualize the traces. Major cloud providers provide systems for logging and filtering but leave the implementation of the logs themselves up to the developers. Other approaches, *e.g.,* [4] consider offline analysis: They analyze log files and provide visualizations together with additional functionality such as filtering or querying. Languages such as ThingML [5] or MoniLog [6] include concepts for logging, whereas the first does not put logs in relation to each other which makes tracing a time-consuming task and the second one has no focus on traceability and filterability.

This paper discusses what is needed in modeling languages to support web-based debugging. The main contribution of this paper is a method based on language extension and smart code generation that can integrate logging into arbitrary textual behavior modeling languages. Based on this method, log entries can be produced, traced, and presented in a web application. Our approach eliminates the need for developers to search for semantically related log messages in very long log files.

The paper is structured as follows: Sec. II presents background technologies and concepts. Sec. III introduces our approach to integrate logging capabilities into base modeling languages, which is evaluated in Sec. IV using the MontiThings language. In Sec. V, we show related work and discuss our approach in Sec. VI. The last section concludes.

II. Background

A. MontiCore

MontiCore [7] is a language workbench [8] for the engineering of compositional modeling languages. Its languages are based on a context-free grammars (CFGs) that define both concrete and abstract syntax of the language under development. Based on this CFG, it generates abstract syntax classes as well as infrastructure for parsing, model checking, code generation, and more to efficiently engineer and compose modeling languages. After parsing, the models are translated into instances of abstract syntax trees (ASTs), processed by MontiCore's extensional function library, checked for well-formedness and other properties, transformed, and ultimately translated into other models, reports, source code, or other target representations. MontiCore also supports the definition of symbols–meaningfully abstracted model parts–based on grammar rules. Symbols are stored in symbol tables and can be resolved within a language as well as by other languages, thus enabling efficient forms of language composition [9]. MontiCore languages, documentation, and tutorials are available[1] online.

[1]MontiCore website: http://www.monticore.de

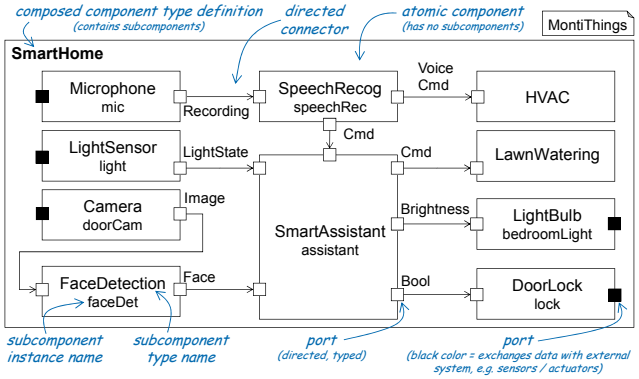


Fig. 1. Exemplary smart home application modeled using MontiThings (taken from [10]).

B. MontiThings

MontiThings is an architecture description language (ADL) for the model-driven development of Internet of Things (IoT) applications [11]. Fig. 1 shows an example of a MontiThings application. Being a component and connector (C&C) ADL, applications consist of components that exchange data by connecting their (typed and directed) ports. Ports are also used to communicate with the outside world, *e.g.,* sensors and actuators. This is denoted by ports having a black filling. The data types used by ports can be defined using class diagrams. Components can be either composed, *i.e.,* define their behavior by instantiating and connecting subcomponents, or atomic, *i.e.,* define their behavior directly using statecharts, a behavior language, or handwritten C++ code.

MontiThings generates distributed applications in the form of C++ code from the models. The generated partial applications communicate using standard message broking protocols, *i.e.,* message queue telemetry transport (MQTT) or data distribution service (DDS). Using a configuration language, MontiThings models can be adapted to different platforms and connected to their respective hardware components, *e.g.,* sensors and actuators. Regarding this, it is also possible to connect to hardware whose drivers are written in other languages, *e.g.,* Python, as long as the language supports MQTT. The generated code can be automatically packaged into container images and (cross-)compiled for different platforms.

III. Concept

We use logging as the basis for tracing. This section presents the concepts for the syntactic integration of logging through language extension, for the integration of logging behavior, and for the tracing of logs.

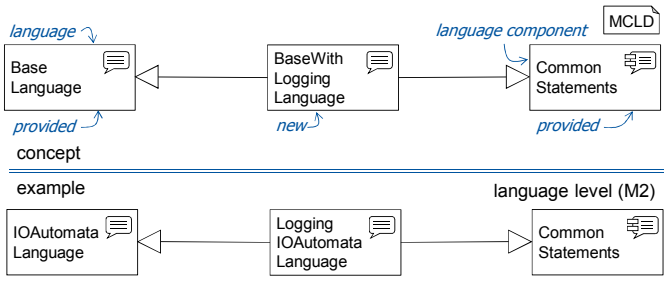
A. Logging Syntax

Integrating logging capabilities into arbitrary base modeling languages, leverages language extension [12] and reuses a library of language components [13], including components for common expressions and statements, to express the concrete syntax and abstract syntax of the logging language elements to be integrated. The language extension produces a new, looging-aware, language inheriting from the base modeling language and extends it with language elements intended for logging. Our concept to extend an arbitrary textual modeling language with logging capabilities, thus, demands to create a new language that inherits from both the base language and a language (component) providing the language elements related to logging. In this new inheriting language, the language designer makes the design choices of adding logging capabilities to specific parts of the language to be extended, e.g., for transitions in statecharts or ports of a C&C ADL. The integration of logging behavior then follows the newly added syntax (*cf.* Sec. III-B).

Leveraging the wealth of language components available in the technological space of MontiCore, we integrate logging into a base language by extending from the `CommonStatements` [13] language component. The latter is a language component in the sense that it is not meant to be used a stand-alone language but comprises concepts related to statements (such as method calls) to be reused by other languages [12]. The logging-capable extension of the base language then inherits from `CommonStatements`. However, if the base language already features language elements that can be use for logging, this additional extension is not necessary. And in the new language, the language designer integrates concepts from `CommonStatements` to enable logging by extending production rules of the language elements that should support logging. Through our notion of language extension [12] this amounts to introducing new alternatives for production of the base language, such that logging of extended elements is optional. Moreover, as a language can extend multiple other languages, the base modeling language can extend from other languages to integrate modeling constructs of arbitrary complexity, *e.g.,* to introduce new data structures, conditional logging, of even complete logging programs.

Following our concept, the extension made to a base modeling language to support logging is *access-conservative* [7] on the abstract syntax, *i.e.,* the AST of models of the inheriting language are still valid ASTs with respect to the base language. Hence, tooling such as model analyses operating on the AST can understand models of the new language but will not be able to process its new elements. The extension further may be *concrete syntax conservative* [7], *i.e.,* tools operating on the concrete syntax, such as parsers, may be able to process models of the inheriting language unless new, mandatory, keywords are enforced. This can be achieved if logging model elements reuse syntax available in the base language.

For the example of extending the `IOAutomata` language of Fig. 2 (1st listing) with logging capabilities, the language designer creates the new language `LoggingIOAutomata`, which inherits from `IOAutomata` and from `CommonStatements` (2nd listing, l. 1). In this new language, the designer choses which elements of the base language `IOAutomata` should support logging. In this case, she decides that transitions and final states should support logging. To this end, she introduces new productions `LogTransition` and `LogFinalState` (2nd list-




```
01 grammar IOAutomata extends CommonExpressions {
02   IOAutomaton = "automaton" Name Body;
03   Body        = "{" States Variable* Transition* "}";
04   Variable    = "var" Type Name ("=" Value)?;
05   States      = "states" (State || ",")* ";";
06   State       = ("initial" | "final") Name;
07   Transition  = src:Name@State
08                 Guard? ("/" Action)?
09                 "->" tgt:Name@State ";";
10   Guard       = "[" Expression "]";
11   Action      = "/" Name "=" Value;
```

```
01 grammar LoggingIOAutomata extends IOAutomata, CommonStatements {
02   LogTransition extends Transition = src:Name@State
03                                      Guard? ("/" Action)?
04                                      "->" tgt:Name@State
05                                      ">>" MethodCall ";";
06   LogFinalState extends State      = ("initial" | "final") Name
07                                      ">>" MethodCall;
08 }
```

example — model level (M1)

```
01 automaton RoboVac {
02   states initial idle, cleaning, returning, final finished,
03          final error >> debug("terminated with error");
04
05   var Room r = 0;
06
07   idle [Monitor.receivedTask()] / r = Monitor.first() -> cleaning;
08   cleaning [Navi.completed()]   / r = Monitor.HOME    -> returning
09                                 >> log("returning to station");
10
11   returning [Navigation.completed() && Navigation.atHome()]
12                                 / r = Montior.BASE -> finished
13                                 >> log("finished");
14 }
```

Fig. 2. Concept and example of integrating logging capabilities into arbitrary languages.

ing, ll. 2-6) that extend `Transition` and `State` of the `IOAutomata` base language (1st listing, ll. 5-9) respectively. Each new production reuses the syntax of its base production but adds a `MethodCall`, which is a production provided by the `CommonStatements` language component. The extensions `LogTransition` and `LogFinalState` of `Transition` and `State` ultimately introduce new alternatives for their base productions but do not enforce their use. This means that models of the `IOAutomata` base language can be processed with the tooling for the new, logging-aware, language, *i.e.,* changes are concrete syntax conservative and that tooling for the base language (*e.g.,* model checkers) can be used with the abstract syntax of the new language, *i.e.,* the changes are access conservative. Modelers, hence, can use logging in final states and transitions as required but are not forced to use logging on each transition or final state.

The `RoboVac` model (3rd listing of Fig. 2) illustrates the resulting language and its logging capabilities. To this end, it introduces five states (ll. 2-3) out of which the state `error` (l. 3) uses the new, optional, production `LogFinalState` and its language elements to log a message about erroneous termination of the cleaning process. Moreover, the 2nd and 3rd transition each use the optional production to log when the respective transition is fired. Hence, *e.g.,* a deadlock checking algorithm for the `IOAutomata` base language, that does not care about logging, still can perform its computations against the concrete and abstract syntax of `LoggingIOAutomata`, by ignoring the alternatives `LogTransition` and `LogFinalState`. Also, a model of `LoggingIOAutomata` that does not use the new production still is a valid model of the `IOAutomata` base language. Both conservations enable reusing tooling and models related to these languages.

While these language components of this example are specific to the technological space of MontiCore [7], the concepts presented in the following can be applied to any technological spaces of language engineering that support reusing multiple languages by another language and extending its abstract syntax rules or classes, such as GEMOC Studio [14], MPS [15] or Neverlang [16].

B. Logging Behavior

Models of the extended modeling language then are translated to logging-aware code. To this end, the generator can translate the method calls, and especially log statements, into code by simply printing a corresponding method call in the target language. The generator is unaware of the fact that it generates code for a log statement. It only knows how to generate code for method calls. The runtime environment of the generated code is then responsible for providing a suitable implementation. Thereby, the generated code can be flexibly, *i.e.,* without modifying the generator, linked against different log implementations. It is also possible to add further functionality in the course of the implementation of the log function, which goes beyond the actual writing of the log. For tracing the logs using a web application, additional information about the log entry can be tracked to be able to link different logs together later. Also, additional data about the state of the system can be logged that was not specified by the developer in the log message. For example, the generator can store all variable assignments so that they become available in the log as if the developer had also set a breakpoint at the time of the log call and examined the state of the system with a debugger.

C. Log Tracing

If the logs shall be traceable, they must be relatable to each other. For this purpose, the implementation against which the generated log method call is linked provides each log entry with additional information. Each log entry is assigned a unique identifier and a timestamp. Furthermore, the developers of the generator and the logging web app decide the granularity with which model elements belong together in the logging web app. The log entries are then grouped according to the model elements in whose context they were created. For example, if a log is created as part of an entry action of a state in a UML

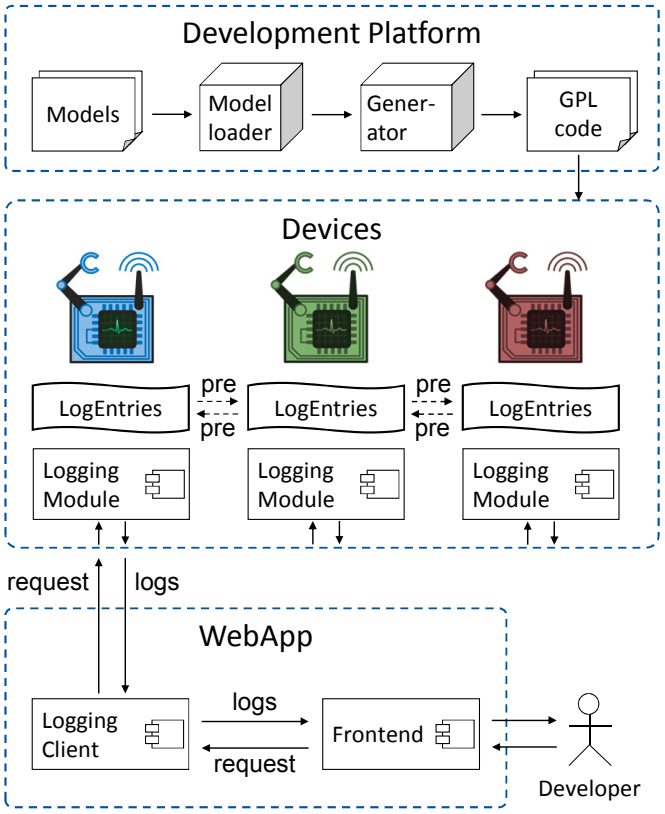


Fig. 3. Models are used to generate code that is deployed to one or more devices. Each device creates and stores groups of log entries, that may reference each other as predecessor. The logs are provided on demand via a logging module deployed to each device. The developer can request these logs via a web application.

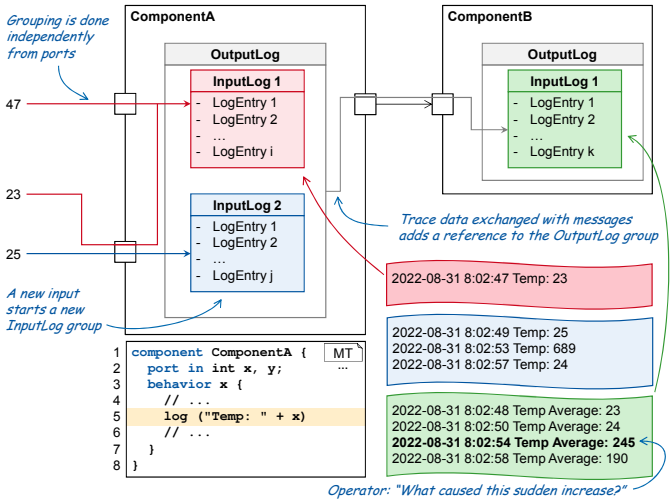


Fig. 4. Motivation and concept of logging in MontiThings.

statechart, this log can considered to be in the context of the state instead of considering the entry action an independent model element. A new log group is created with each use of a model element. For example, in the case of a statechart or automaton, logs can be grouped for the states, or in the case of an activity diagram, the activities. If a state is entered several times in a statechart, several log groups are created.

Each such group of log entries can then be assigned a predecessor group. This creates a concatenated list of log groups over time. If, in the case of a distributed system, the logs of different devices are to be related, the devices can communicate the unique ID of their respective log group to each other. In this way, devices can also tell other devices the ID of their log groups to use them as predecessors without having to exchange the concrete log messages.

At runtime, developers should now be able to track the logs of their models through a web application (*cf.* Fig. 3). In the runtime environment of the generated code, a logging module is built in, which is able to answer requests for logs or log groups. The web application in turn can use these logging modules to request logs. The web application exposes the models and allows developers to select model elements. When a model element is selected, the web application requests the necessary logs from the devices and presents them to the developer. If the developer now selects one of the log entries, the web application can request the predecessors of these logs. Thereby, a trace of the requested log entries is created, similar to a stacktrace. In contrast to the classic stack traces of a debugger, traces can thus also be shown across multiple devices of a distributed device. Through this tracing mechanism, only the logs relevant to a developer's request in each case are displayed, as only the specifically requested logs are shown. This eliminates the need for developers to search for semantically related log messages in very long log files.

IV. Evaluation

To validate our approach, we implemented web-based tracing functionality for the MontiThings language. In the following, we first present how we integrate the tracing into MontiThings and then evaluate the overhead caused by the tracing on the IoT devices.

A. Evaluation Setup

IoT applications are inherently hard to trace as they are not only distributed systems but also depend on a large number of external influences such as sensor inputs and might be deployed to unreliable hardware. As a result, tracing IoT applications often requires developers to manually analyze a large number of log messages. In addition, imperfect synchronization of the IoT devices' clocks, and the wrong timestamps caused by this in the log message, might lead developers to draw wrong conclusions and search the root cause of a particular behavior in the wrong places of the code base or logs. Overall, the understanding of IoT applications based on their logs is error-prone and time-consuming.

In MontiThings, components exchange data via ports. The connectors between the components' ports define which components interact with which other components. Implicitly, these connectors also tell us how the components could influence each other. If data flows from component A to

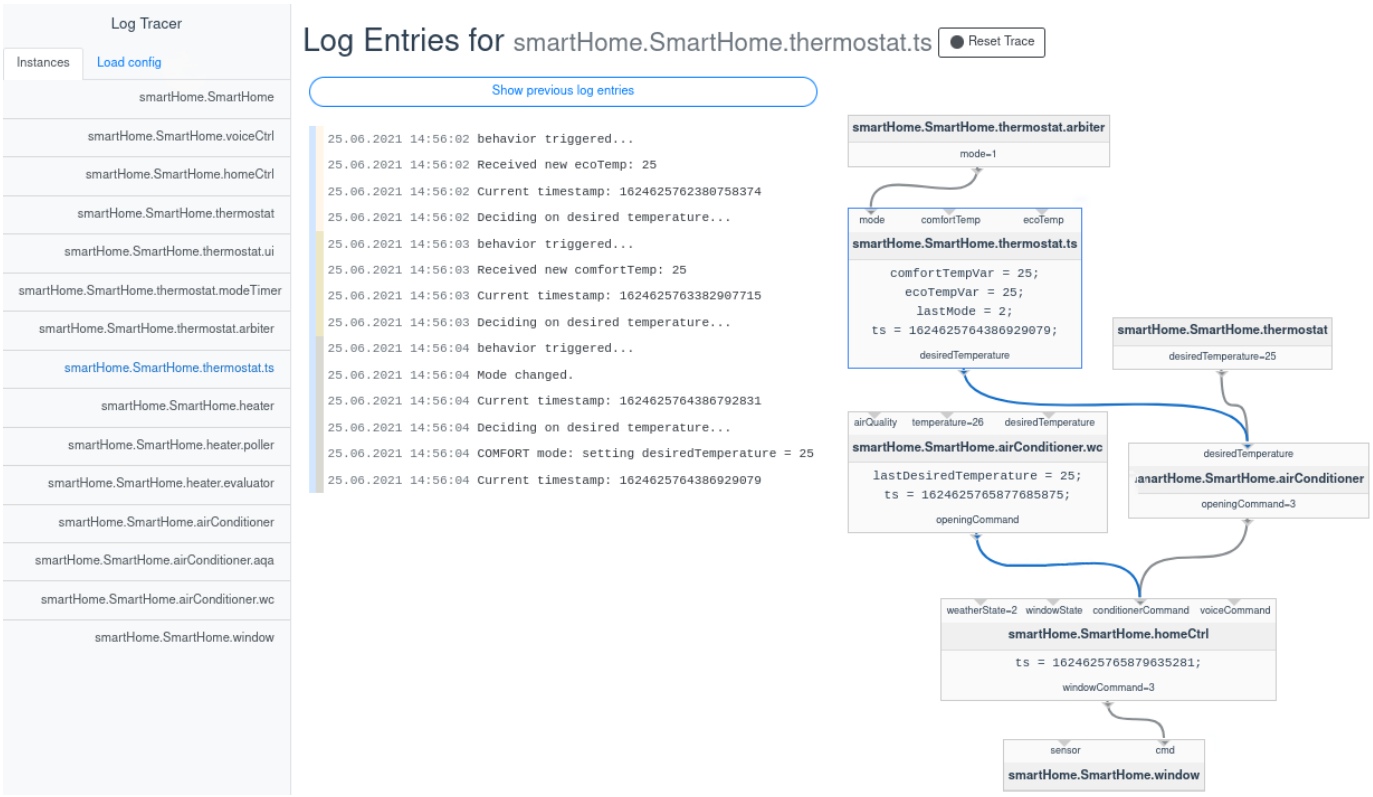


Fig. 5. Screenshot of the tracing web application user interface. Developers can select a component in the menu on the left. In the middle, the application shows the component's (filtered) log messages. On the right, the application shows a visual trace of the currently selected log message through the architecture.

component B, component B might be influenced by component A. In contrast, if the components do not even exchange data indirectly, *i.e.,* there is no path from component A to component B when viewing the components and their connectors as a directed graph, then they do not influence each other[2].

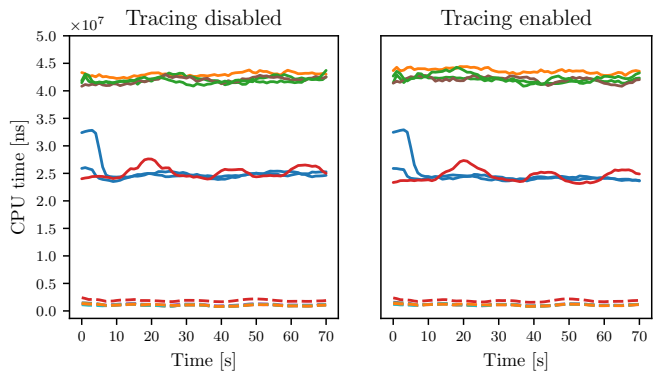
Fig. 4 gives an overview of how MontiThings provides tracing of log messages. As ports and the connections between them define a dependency relationship between the components, we group log messages based on the messages a component exchanges with other components. Whenever a component receives a message on one or more of its incoming ports, we create a new *input log group*. An input log group is an ordered list of log messages. Unfortunately, developers sometimes forget to log values, such as a variable, that they need to later understand the system. To mitigate this, input log groups also implicitly log a snapshot of the component upon creation. This snapshot contains the values of all variables and messages. To avoid creating full snapshots every time a log statement is called, snapshots are reconstructed based on historical change logs of individual variables. Each time a variable is changed, a new time slice is created for the variable and the previous value is considered invalid. In this way, previous variable states can be reconstructed based on the closest previous time slice in which a value was valid when the log statement was invoked.

If a component sends a message on one of its outgoing ports we further create an *output log group* consisting of all input log groups created since the last outgoing message. To relate log groups to each other, each log group also may reference another log group that serves as its predecessor. To convey this predecessor relationship across a distributed IoT application, each log group contains a unique identifier. When a component sends a message on one of its outgoing ports, this message includes the identifier of the log group associated with the message. Thereby, the receiving component can set the received identifier as the identifier of the predecessor of the input log group it creates upon receiving the message.

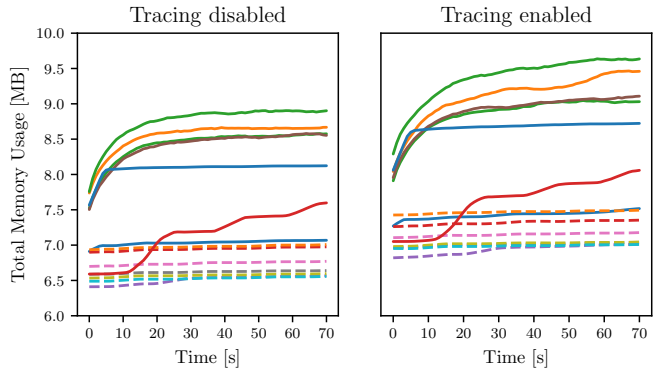
For each component, MontiThings generates a small application binary. When enabling tracing, the code generator injects a communication module into this binary that collects logs. Furthermore, it answers the requests for these logs. The logs are requested by a web application developers use to trace their generated software. Its interface is shown in Fig. 5. When using the web application, the developer can request the logs of any component using the menu on the left. After first requesting the logs from a component, the developer will receive a list of all log messages. The developer can now click on one of the log messages to investigate in more detail how and why it arose.

For this purpose, the web application requests all pre-

[2]From a data flow perspective. From a technical point of view, components deployed on the same device can influence each other, *e.g.,* by each accessing the network and thus lowering the data rate available per component. Our approach to tracing such technical issues is described in [10].



(a) Processing performance overhead



(b) Memory performance overhead

Fig. 6. Performance overhead of the tracing. Each color / stroke type represents a different component instance. A moving average is calculated over the last 9 values. This smoothes the trends and makes them more visible.

decessor log groups of the selected log message from the influencing components. Predecessor log groups occurred on the timeline immediately before the log statement invocation and likely influenced the computation containing the original log message. These log groups may be distributed across several other components and are visualized in a so-called trace tree. Developers can further expand the trace by selecting a relevant log group in the tree. This also gives developers the option, if a component consists of multiple subcomponents, to decide whether to continue the trace at a hierarchically lower level or remain at the current level of abstraction. This can be useful since components at lower levels are probably not as relevant as components at higher levels.

Thereby, we reduce the number of logs that the developers need to inspect to find the log message a log message originated from. Of course, we are aware that our filtering may also hide messages that might have been filtered even though they were relevant to the log message at hand. To mitigate not being able to further inspect the components' behavior in these cases, developers can choose to view all previous logs regardless of whether our tool considers them relevant and then continue with the tracing.

B. Performance Evaluation

To evaluate the overhead of our log tracing, we constructed a heating, ventilation, and air conditioning (HVAC) case study. This case study has first been used in [10] and provides a scenario that deterministically stimulates the cyber-physical system (CPS) with input values for at least one minute. It was run 100 times and the average value was taken, whereas log tracing was disabled from the half[3]. We assessed the overhead by inspecting the applications' containers' performance using Google's cAdvisor[4]. Fig. 6 shows the results of our performance evaluation. Each line color and style represents the CPU time and memory consumed by one of the components.

Processing-wise (Fig. 6(a)), our approach causes almost no overhead. The difference in total CPU time is less than 1 %. This expected as the only additional processing caused by our approach is storing log messages in memory. The difference between the three groups of lines visible in Fig. 6(a) comes from the purpose and execution frequency of the components. Some components needed to be executed every 500 ms while other components were processing messages every second. Composed components generally have the lowest CPU requirements as they only forward data to their subcomponents without doing their own processing.

Memory-wise (Fig. 6(b)) our approach causes a constant overhead for the log collection module. Additionally, the memory consumption of the components also gradually increases over time. This is caused by storing log messages. This overhead could be mitigated by either storing the logs to a file (optionally in a remote storage) or by discarding old log messages. The latter, however, could make behavior impossible to trace if the discarded log messages contain the predecessor of a log message the developer is interested in.

V. Related Work

The idea of tracing execution flows in distributed systems has been implemented in microservice infrastructures in particular. The more services collaborate to process a request, the more difficult it becomes to understand what exactly is causing undesirably slow response times or anomalous behavior. Widely used tracing tools include instrumentation and trace collector tools like OpenTelemetry [17], together analysis tools such as Jaeger [18], Splunk [19], Dynatrace [20], and Lightstep [21]. Each request is automatically or manually tagged with a correlation id, timing information, and custom tags. It is ensured that the correlation id is maintained during the propagation of the request through the network. Trace data is sent to a central instance, which in turn, provides a web interface where the traces can be visualized, *e.g.,* in the form of waterfall diagrams or latency plots. In this way, it is possible to observe the entire flow of individual queries, including time measurements ranging from microservice processing to time spent in specific methods to the duration of database queries.

[3]The benchmark host was a virtual machine running Ubuntu 20.04 on 8 cores of AMD Ryzen 2600X equipped with 10GB physical memory.

[4]cAdvisor project website. [Online]. Available: https://github.com/google/cadvisor. Last accessed: 17.01.2022

Instead of focusing on request-response modeled applications, *ShiViz* [4] aims to be a more general solution for all types of applications. It is an offline tool that processes log files and outputs a web interface that shows a space-time graph visualizing the relative order of log entries and communication within the network. It supports filtering, querying communication patterns, and identifying repetitive patterns. To mitigate distributed clock shifts, ShiViz provides a library that intercepts each network call in order to maintain a vector clock. The clock can then be used to annotate events such as the creation of log entries or communication with other nodes.

Unfortunately, applications in the CPS domain cannot rely on these approaches because their communication patterns are more complex and inconsistent. Events may no longer refer to a single request, but may even be relevant for much later inputs to the application. In such a case, the trace grows continuously. Thus, the main focus should be on the filtering aspect.

ThingML adds logging to its language by extending it with a tagging mechanism [5]. The tagging mechanism enables developers to mark model elements that they want to log information on (using the `@monitor` annotation). As noted by the authors of [5], this approach is similar to aspect-oriented programming, which can likewise be used to annotate (code) elements. The code generator uses these tags to add log statements to the generated code whenever the model elements are modified. For example, if a variable is marked to be logged, changing its value will cause a new log to be created. The logs are then sent to a central web application where they can be inspected. Unlike our approach, however, the individual log messages are not set in relation to each other. Thus, tracing errors in the collected logs is still a time-consuming task.

The major cloud providers, *i.e.,* Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform (GCP), all provide systems for logging. Logs are either sent to the log service using a software development kit (SDK) provided by the cloud provider or automatically sent from services offered directly by the cloud provider. This, however, leaves the burden of implementing such logs completely to the developers. Unlike our approach, cloud logging services cannot implicitly log data about the observed system as the semantics of the system is unknown to the cloud provider (unless the service is offered by the cloud provider itself). As cloud providers offer comprehensive suites of interacting products, inspecting the logs collected by their log services can be a time-consuming task for developers. Thus, they offer filtering mechanisms. For example, GCP's log explorer offers a logging query language [22] for filtering logs. Azure[5] and AWS[6] offer similar languages. The downside of this approach is that it requires developers to deeply understand all parts of the system for which they are inspecting the logs. As our approach automatically relates log messages to each other, we can automatically remove irrelevant logs without needing such manual filtering rules. However this requires our approach to understand the semantics of the language to some degree.

MoniLog [6] provides a language for defining combinations of monitoring and logging, called moniloggers. Language engineers define interfaces that enable the interpreter of the moniloggers to observe certain properties of the system. While both MoniLog and this paper present techniques for extending modeling languages with logging capabilities, MoniLog is more focused on the efficient specification of which logs to create whereas our method focuses on relating logs to each other to make them traceable, filterable, and, thus, more understandable. As both MoniLog and this paper automatically monitor certain events of the observed system, both methods are only applicable if the modeling language contains clearly defined events, *e.g.,* a state change in a statechart.

VI. Discussion

Our concept assumes that language extension considers the abstract syntaxes and related concrete syntaxes. In grammar-based technological space of language engineering, considering both is often part of the extension mechanisms [7], [16], [23], whereas in metamodel-based approaches, this often demands additional infrastructure to inherit the editing capabilities accordingly. While this limits the immediate application of our method to language engineering technological spaces with less powerful extension mechanisms, this is not a limitation of the presented method itself.

To relate logs to each other, our system assumes that there are clearly defined events, which in turn are also related. This relationship is especially of temporal nature, *i.e.,* events depend only on temporally earlier events and never on future events. Further, this relationship is also of a contextual nature, *i.e.,* there can be events that are influenced by other events as well as events that are completely independent of each other. Whether and how events depend on each other in terms of their context is determined by the semantics of the modeling language. As both MoniLog and this paper automatically monitor certain events of the observed system, both methods are only applicable if the modeling language contains clearly defined events, *e.g.,* a state change in a statechart.

As an application developer, it can sometimes be difficult to predict at development time what information needs to be logged to understand an error. By using a model-driven approach, this problem can be mitigated by having the generator automatically log the context of each log message, *e.g.,* variable assignments, CPU load, or (if applicable) sensor data. The automatically logged information can in some cases enable the application developer to trace even those errors that cannot be traced by the content of the log messages alone.

In our prototype implementation, network overhead was avoided by receiving log messages from IoT devices only on request. In production use, a tradeoff must be made here, since the storage of the devices executing the generated code is not unlimited. One way to handle the limited storage is

[5]Azure uses the SQL-like *Kusto query language*. Log queries in Azure Monitor [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-monitor/logs/log-query-overview Last accessed: 26.01.2022

[6]AWS uses a Bash-like language to pipe the results of query commands into each other. CLoudWatch Logs Insights query syntax. [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html Last accessed: 26.01.2022

to offload log data to cloud storage. However, this creates network overhead that can be unacceptable, especially in cellular networks. It is also possible to delete old log data to make room for new ones. However, this can make the causes of even current errors untraceable if their causes lie too far in the past. In practice, therefore, a tradeoff must be made between hardware and storage costs (storage on devices and in the cloud), network costs (transfer of log data to the cloud) and the probability of being able to trace an error. Depending on the specific IoT system, this can be a serious problem if there is data to be logged every few milliseconds, but the devices have very limited memory. During normal execution, a lot of data may be irrelevant for later troubleshooting. However, if errors build up gradually, data that was generated during a normal execution and was considered inconspicuous may later become more relevant for tracing an error.

VII. Conclusion and Future Work

Within this paper, we have presented a method based on language extension and smart code generation that can integrate logging into arbitrary textual modeling languages. The application of this method allows to produce log entries, trace them, and visualize them through a web application. We have shown the applicability of the method for the MontiThings language. This method eliminates the need for developers to search for semantically related log messages in very long log files and improves the understanding of executable models. As future work, our approach should be evaluated on larger IoT systems, e.g., to further investigate the impact of limited memory of IoT devices.

Source Code

MontiThings is available on GitHub: https://github.com/MontiCore/montithings

Acknowledgements

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy—EXC 2023 Internet of Production—390621612. Website: https://www.iop.rwth-aachen.de

References

[1] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A Survey on Automated Log Analysis for Reliability Engineering," *ACM Comput. Surv.*, vol. 54, no. 6, July 2021.

[2] T. Galli, F. Chiclana, and F. Siewe, "Quality properties of execution tracing, an empirical study," *Applied System Innovation*, vol. 4, no. 1, 2021. [Online]. Available: https://www.mdpi.com/2571-5577/4/1/20

[3] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-Driven Software Development: Technology, Engineering, Management*, 1st ed., ser. Wiley Software Patterns Series. Wiley, 2013.

[5] B. Morin and N. Ferry, "Model-Based, Platform-Independent Logging for Heterogeneous Targets," in *ACM/IEEE 22nd Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, 2019, pp. 172–182.

[4] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst, "Visualizing distributed system executions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 2, pp. 9:1–9:38, Mar. 2020.

[6] D. Leroy, B. Lelandais, M.-P. Oudot, and B. Combemale, "Monilogging for Executable Domain-Specific Languages," in *14th ACM SIGPLAN Int. Conf. on Software Language Engineering*. ACM, 2021, pp. 2–15.

[7] K. Hölldobler and B. Rumpe, *MontiCore 5 Language Workbench Edition 2017*, ser. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[8] S. Erdweg, T. Van Der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh *et al.*, "Evaluating and comparing language workbenches: Existing results and benchmarks for the future," *Computer Languages, Systems & Structures*, vol. 44, pp. 24–47, 2015.

[9] A. Butting, K. Hölldobler, B. Rumpe, and A. Wortmann, "Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques—The MontiCore Approach," in *Composing Model-Based Analysis Tools*. Springer, 2021, pp. 217–234.

[10] J. C. Kirchhof, L. Malcher, and B. Rumpe, "Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins," in *20th ACM SIGPLAN Int. Conf. on Generative Programming: Concepts and Experiences (GPCE '21)*. ACM SIGPLAN, 2021, pp. 197–209.

[11] J. C. Kirchhof, B. Rumpe, D. Schmalzing, and A. Wortmann, "MontiThings: Model-driven Development and Deployment of Reliable IoT Applications," *Journal of Systems and Software*, vol. 183, p. 111087, January 2022.

[12] K. Hölldobler, B. Rumpe, and A. Wortmann, "Software Language Engineering in the Large: Towards Composing and Deriving Languages," *Computer Languages, Systems & Structures*, vol. 54, pp. 386–405, 2018.

[13] A. Butting, R. Eikermann, K. Hölldobler, N. Jansen, B. Rumpe, and A. Wortmann, "A Library of Literals, Expressions, Types, and Statements for Compositional Language Design," *Special Issue dedicated to Martin Gogolla on his 65th Birthday, Journal of Object Technology*, vol. 19, no. 3, pp. 3:1–16, 2020.

[14] T. Degueule, T. Mayerhofer, and A. Wortmann, "Engineering a ROVER Language in GEMOC STUDIO & MONTICORE: A Comparison of Language Reuse Support," in *MODELS 2017. Workshop EXE*, ser. CEUR 2019, 2017.

[15] F. Campagne, *The MPS language workbench: volume I*. Fabien Campagne, 2014, vol. 1.

[16] E. Vacchi, D. M. Olivares, A. Shaqiri, and W. Cazzola, "Neverlang 2: a framework for modular language implementation," in *13th International Conference on Modularity*, 2014, pp. 29–32.

[17] "Opentelemetry," [Online]. Available: https://opentelemetry.io/. Last checked 31. January 2022.

[18] "Jaeger: Open source, end-to-end distributed tracing," [Online]. Available: https://www.jaegertracing.io/. Last checked 2. August 2021.

[19] "Cloud-based data platform for cybersecurity, it operations and devops — splunk," [Online]. Available: https://www.splunk.com/. Last checked 15. January 2022.

[20] "Dynatrace — the leader in automatic and intelligent observability," [Online]. Available: https://www.dynatrace.com/. Last checked 6. February 2022.

[21] "Lightstep—The cloud-native reliability platform," [Online]. Available: https://lightstep.com/. Last checked 5. April 2022.

[22] Logging query language documentation. [Online]. Available: https://cloud.google.com/logging/docs/view/logging-query-language. Last accessed: 26.01.2022.

[23] M. Dalibor, N. Jansen, J. Kästle, B. Rumpe, D. Schmalzing, L. Wachtmeister, and A. Wortmann, "Mind the Gap: Lessons Learned from Translating Grammars Between MontiCore and Xtext," in *Int. Workshop on Domain-Specific Modeling (DSM'19)*. ACM, 2019, pp. 40–49.