Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins

Jörg Christian Kirchhof
Software Engineering
RWTH Aachen University
Aachen, Germany
https://se-rwth.de

Lukas Malcher
Software Engineering
RWTH Aachen University
Aachen, Germany
https://se-rwth.de

Bernhard Rumpe
Software Engineering
RWTH Aachen University
Aachen, Germany
https://se-rwth.de

Abstract

Developers questioning why their system behaves differently than expected often have to rely on time-consuming and error-prone manual analysis of log files. Understanding the behavior of Internet of Things (IoT) applications is a challenging task because they are not only inherently hard-to-trace distributed systems, but their integration with the environment via sensors adds another layer of complexity. Related work proposes to record data during the execution of the system, which can later be replayed to analyze the system. We apply the model-driven development approach to this idea and leverage digital twins to collect the required data. We enable developers to replay and analyze the system's executions by applying model-to-model transformations. These transformations instrument component and connector (C&C) architecture models with components that reproduce the system's environment based on the data recorded by the system's digital twin. We validate and evaluate the feasibility of our approach using a heating, ventilation, and air conditioning (HVAC) case study. By facilitating the reproduction of the system's behavior, our method lowers the barrier to understanding the behavior of model-driven IoT systems.

CCS Concepts: • **Computer systems organization → Distributed architectures**; *Embedded and cyber-physical systems*; • **Software and its engineering → Software testing and debugging**; *Architecture description languages.*

Keywords: Internet of Things, Architecture Description Languages, Model-Driven Development, Debugging

ACM Reference Format:
Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. 2021. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3486609.3487210

1 Introduction

Motivation. A major challenge in developing distributed Internet of Things (IoT) systems is that these systems are often exposed to harsh environmental conditions. This makes the development of robust systems more difficult, since environmental influences must be taken into account in addition to software errors [10]. The ways in which a system can be affected by the environment range from implausible sensor readings [5, 21], to network failures [22, 34], to hardware failures [22]. If users notice unexpected behavior of the system, the developers are often asked afterwards how this error could have occurred. The most prominent examples are those in which the misbehavior of the system leads to high property damage or loss of life, *e.g.,* in cases in which autonomous cars are involved in accidents [25]. Even when analyzing a system purely based on the incomplete system view of log files is possible, it requires very precise knowledge of the code sections that produce the respective log messages. Moreover, it is time-consuming and error-prone due to the often large size of the logs [20] combined with a low degree of automation.

Approach. To improve this situation, related work suggests to record data during the runtime of the system, which allows to reproduce the behavior of the system retrospectively [12, 17, 20], *e.g.,* in a simulation. This allows the developers not only to observe the system behavior again, but also to examine variables that were not captured by log messages, *e.g.,* by using debuggers and setting breakpoints. Furthermore, this approach offers the advantage that developers can retrospectively pose what-if questions to the system by manipulating variables or environmental influences. Ultimately, complex test cases for future versions of a system that were not previously considered in development can also be derived from such records.

Contribution and Opportunities. We apply this idea of recording and replaying data to model-driven IoT applications. Due to the distributed nature of the system and sometimes difficult to reproduce environmental influences, the behavior of IoT applications is often complex to understand. There are several advantages to using model-driven development here. First, it is possible to capture a lot of relevant data about a system at clearly defined interfaces of the system, *i.e.*, at specific model elements. Many model-driven frameworks for developing IoT applications are based on component and connector (C&C) architectures, *e.g.*, ThingML [14, 24], Ericsson's Calvin [2, 28], Node-RED [1], or CapeCode [6]. Accordingly, our approach also focuses on C&C architectures. As described in [15], the data that components exchange via ports can be captured to create digital twins of the system. Using the digital twin's data we extend models through automated model-to-model transformations to include elements that represent real-world influences, such as network delays or sensor values. This automation lowers the barrier of applying such a technique compared to a manual extension of the code base. Since agile processes are expected to improve IoT development [19], we integrate our method into an iterative process.

Hence, the contributions presented in this work are:

1. a method for replaying and analyzing the behavior of C&C architecture models of the MontiThings architecture description language (ADL) with information recorded by digital twins during execution of the system, and
2. a workflow for integrating the reproduction and analysis of IoT systems in iterative development processes, and
3. model-to-model transformations that automatically extend the architecture models with components enabling the retrospective reproduction of the system's execution.

Structure of the paper. The remainder of this paper is organized as follows. Sec. 2 introduces the MontiThings ADL and digital twins. Sec. 3 introduces a motivating example. Sec. 4 outlines our requirements. Our concept and realization are detailed in Sec. 5 and Sec. 6. We evaluate our concept in Sec. 7, followed by a discussion in Sec. 8. Related work is presented in Sec. 9. Finally, Sec. 10 concludes the paper.

2 Preliminaries

This section introduces the MontiThings ADL, used for the implementation of our concept, and defines digital twins.

2.1 MontiArc and MontiThings

MontiArc is a C&C-based ADL for the specification, analysis, and simulation of cyber physical systems [8, 13]. As many other ADLs, MontiArc describes systems using components which exchange data via ports. These ports are connected using explicit connectors. MontiArc's ports are directed and typed. The data types used by the ports can be defined using class diagrams. Components can either be composed, *i.e.*, contain subcomponents defining their behavior, or atomic, *i.e.*, not contain subcomponents. Atomic components define their behavior, *e.g.*, using automata [8], a Java-like behavior language [8], or handwritten code. MontiArc is based on the FOCUS calculus [7, 30]. The FOCUS calculus treats understands components as stream processing functions. FOCUS streams can include *ticks* that represent time progress. From a theoretical point of view, MontiArc's (composed) components are (composed) stream processing functions.

The MontiThings ADL [16] used in this work is an IoT-focused extension of MontiArc. MontiThings mainly extends MontiArc with a C++ generator and an IoT-focussed runtime that, *e.g.*, utilizes message brokers to implement connectors. Moreover, MontiThings extends the MontiArc graphical syntax with sensor and actuator ports identified by black fill. In the textual representation of the model, these sensor and actuator ports are indistinguishable from regular ports. The only difference is that developers have provided code that specifies how these ports can interact with the connected hardware.

Fig. 1 demonstrates MontiThings' graphical syntax using a smart home application, adapted from the MontiArc example in [15]. At the core of the application is the `SmartAssistant` component, which processes sensor data to control the actuators of the smart home. For example, a `Microphone` together with a `SpeechRecog` component can be used to evaluate voice commands. These voice commands are input to a heating, ventilation, and air conditioning (`HVAC`) component, which we use as our case study in Sec. 7.1. The `LawnWatering` component will be discussed in more detail in Sec. 3 as a motivating example for our approach. A more detailed overview of MontiThings can be found in [16].

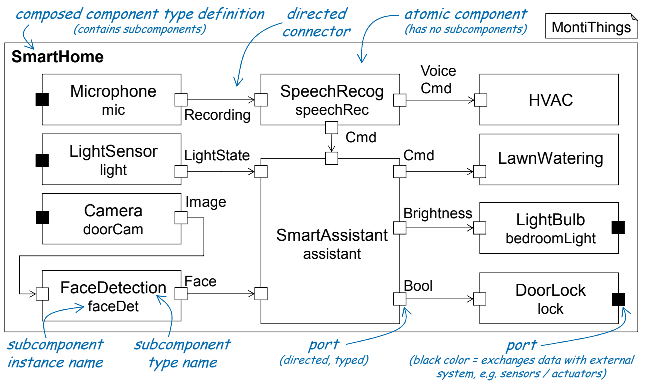


Figure 1. Smart home application demonstrating the graphical syntax of the MontiThings ADL (adapted from [15]).

2.2 Digital Twins

Digital twins (DTs) are digital representations of cyber-physical systems. They monitor systems to offer services to analyze, control, and optimize the system. Alas, there is no commonly agreed on definition of DTs [26]. Since many definitions of DTs do not take into account model-driven development, we adopt the following definition of DTs from [15]:

"*A digital twin (DT) of a system consists of a set of models of the system, a set of contextual data traces and/or their aggregation and abstraction collected from a system, and a set of services that allow using the data and models purposefully with respects to the original system.*" [15]

Many definitions relate a DT to simulations of the cyber-physical system or even understand the simulation itself as a DT [26]. Some definitions of DTs also address a synchronization between the DT and the cyber-physical system (*e.g.*, [35]). The most relevant part of the definition for our approach is that data is collected about the cyber-physical system, which is then processed for a specific purpose. Apart from that, it is not mandatory to agree with the definition chosen here in order to use our approach.

3 Motivating Example

Even if an application is carefully planned and specified, it may happen that a program generated from the architecture models behaves differently than desired. Using the example of a smart lawn watering system, we demonstrate how this might happen. The architecture of the application is shown in Fig. 2. The goal of the system is to water the lawn in the front yard of a house without soaking people on the walkway to the house.

Overall, the lawn irrigation is controlled by an `IrrigationController` component that evaluates sensor data to control the system's sprinklers. Lawn watering at night can contribute to disease development [33]. Therefore, the `WateringScheduler` component lets the controller know when it is a good time to water the lawn. An additional `MoistureSensor` component checks soil moisture. As recommended by [33], this allows the `IrrigationController` to make "smart" decisions instead of controlling sprinklers based on time of day alone. An undesired side effect of watering the lawn is that the sprinklers can wet people on the walkway. To prevent this, a `MovementSensor` component checks whether anyone is currently on or approaching the walkway. In addition, residents of the house can also control the lawn watering via voice commands.

During development, the system was tested and no errors were found. However, after deploying the system, users reported to the developers. They complain that the lawn was undesirably watered in the evening. In addition, a person on the sidewalk was wetted by the sprinklers. Moreover, the sprinklers do not all behave the same. The developers are now faced with the task of examining the system to discover

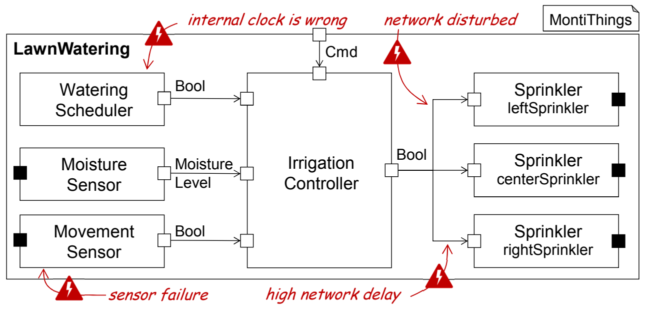


Figure 2. Smart lawn watering application. During the execution of the application, several problems occurred, which are shown in red.

possible sources of error. Upon re-testing the system, no errors can be detected. Without further measures for error detection, the developers would now have to evaluate any log data and, if necessary, ask the users to protocol how the system behaves.

The errors that have occurred are due to imperfections in the hardware used. The `WateringScheduler` component has an incorrectly set time and thus reports to the controller in the evening that the lawn may be watered. The motion sensor used by the `MovementSensor` component is damaged and detects movements on the walkway only unreliably. Also, the network connection of two sprinklers is poor. Messages to the sprinklers arrive with a high delay or are sometimes lost. Since the errors are not due to modeling or programming errors, they cannot be detected in the model itself.

Our approach is to utilize the system traces a digital twin records during the system's execution. This includes, *e.g.*, the data provided by the sensors. In the same way, data about the messages sent between components and non-deterministic behavior is recorded. If the developers are asked to analyze an error after the system has been executed, they can use this data to reproduce the behavior of the system.

For this purpose, we leverage model-to-model transformations. These transformations add components and connectors to the system that mimic environmental influences during recording. For this, the transformations utilize the data recorded during runtime. Developers can analyze the application generated from the transformed models to discover that the misbehavior is due to hardware and network errors. With this knowledge, they can now decide on the next development steps, such as using more robust hardware or implementing software-based countermeasures.

4 Requirements

Our overall goal is to understand the behavior of the components of the real system. We propose to capture the system behavior during its execution and then use the captured data for a reproduction of the execution of the real system while guaranteeing a deterministic environment. To achieve this we define a set of requirements for our system. First, we define the intended relationship between the real system and the reproduction in a high-level requirement:

(R1) **Reproduce behavior.** *Components from the original model shall behave in the reproduction like they did in the real world.* To understand the original system, it is indispensable that the behavior in the reproduction is as close as possible to the behavior in the real system. Otherwise, the error analysis of the reproduced system might not reflect the errors of the original system. Formally, in terms of the FOCUS calculus, this means that all streams of all ports in the reproduction shall be equal to those in the real system.

To achieve this requirement, we define more fine-grained requirements, that together help reaching **(R1)**.

(R2) **No modifications.** *Components from the original model may not be altered.* If the components were altered, they might behave differently in the reproduction than they did when executing the real system. This would violate **(R1)**.

Note that the reverse does not hold. Even if a component is unaltered, it may behave different in a reproduction because of external influences on the system. Therefore, we need to capture such influences:

(R3) **Reproduce environment.** *The reproduction shall reproduce environmental influences on the original system.* This includes all external inputs to the system, such as measured sensor values. As IoT devices often operate under harsh environmental conditions, such as vibrations or electromagnetic radiation, this also includes changes of the system itself. For example, the reproduction needs to account for failing components due to failed hardware.

(R4) **Reproduce timing.** *The reproduction shall reproduce the time consumed for executing component behavior and for exchanging messages between components.* The hardware running the real system and the host system of the reproduction can have different computing power. Thus, the execution times of the real system need to be recorded and reproduced. Also network delays that influence message exchanges between the components need to be captured and reproduced. This requirement ensures that messages exchanged by the components reach the target component in the original order.

(R5) **Reproduce non-deterministic behavior.** *The reproduction shall reproduce non-deterministic behavior of the original system.* Some components may rely on non-deterministic effects. We consider anything to be non-deterministic that could lead to different results if re-executed on a different device or at later time (*e.g.,* time accesses or file accesses). For example, a component may draw a random number to make a decision. If not captured, such non-deterministic decisions may lead the reproduction to behave differently than the original system. Hence, we need to capture non-deterministic behavior.

As for any software project, this list could be extended with non-functional performance requirements, such as keeping the network overhead low. Since these are not the main focus of this paper, they are not discussed in detail, although the performance is evaluated in Sec. 7.2. The following sections describe how we address the above requirements.

5 Iterative Development Process

Iterative development processes are widely used by development teams. Thus, it is crucial for the acceptance of a new approach to be embeddable in such iterative development processes. Fig. 3 provides an overview of our process for reproducing model-driven systems. Since this process is also built on the iterative improvement of the system under development, it is easily embeddable into other iterative methodologies. At its core, the process consists of six steps:

Step (1): First, the developers create a set of architectural models as they usually do. The goal of these models, in addition to their potential use for documenting or analyzing the system at design time, is to be used for code generation. In addition to architecture models, class diagrams describing the data structures can also be developed, as usual in MontiThings. But only the architectural models are relevant for our process. If handwritten code is used to implement the behavior of components, non-deterministic function calls have to be marked to make the recorder aware of their non-deterministic nature.

Step (2): Next, the models created in step 1 are given to a code generator. The code generator creates code from the models in a GPL, in our case C++. In particular, the generator adds the necessary code to enable the components to exchange messages. For this, we use OpenDDS[1], an open source implementation of the OMG's data distribution service (DDS) [27] publish-subscribe communication middleware standard. The generated code is compiled to executable binaries. Optionally, these binaries can be packaged into container images for easier distribution. Our implementation uses Docker for this purpose.

Step (3): The executables created in the previous step are distributed to the devices on which they are to be executed. During the execution of the devices, a recorder collects the data exchange between the components. Moreover, the recorder records metadata about the system, *e.g.,* the network delay between sending and receiving a message. Components report non-deterministic behavior to the recorder, *e.g.,* random numbers or the current time. The recorder stores

[1]Project website (last checked 23.09.2021): https://opendds.org/

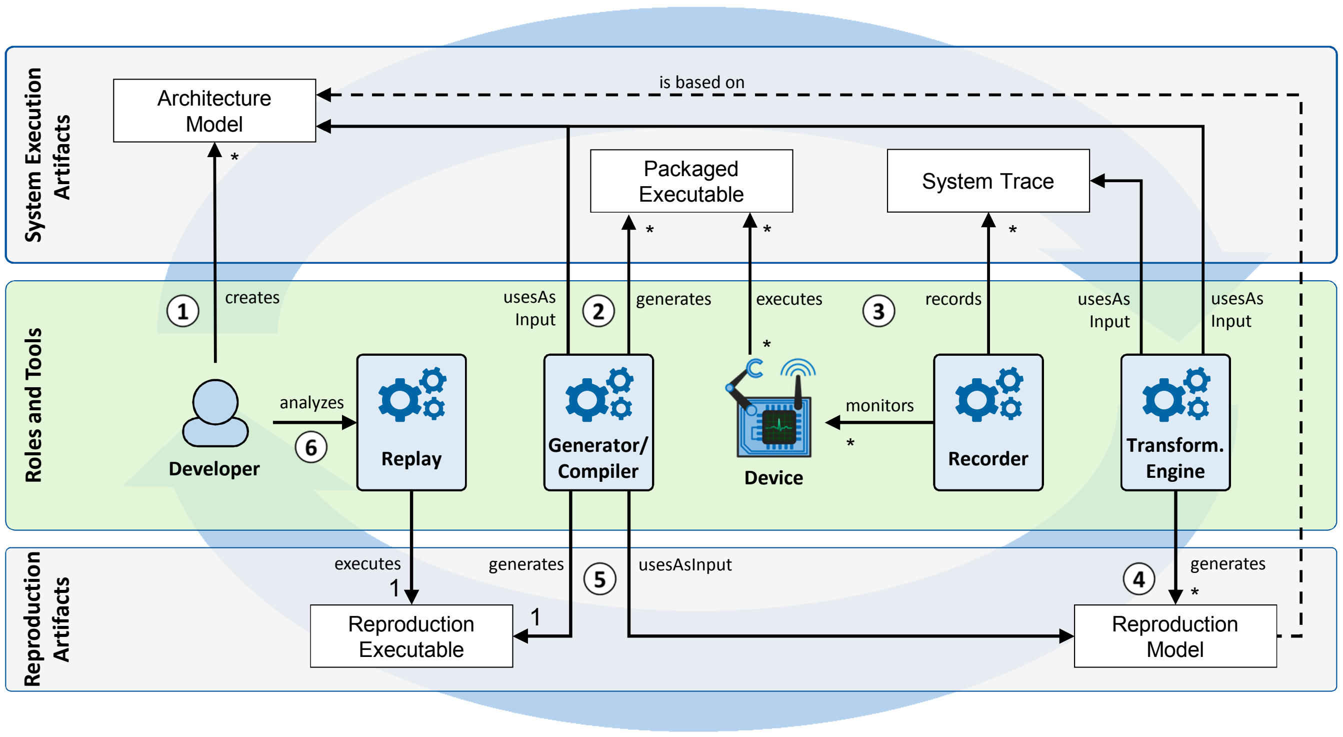


Figure 3. Iterative, model-driven process: Developers create models from which packaged executables are generated. These are executed by devices monitored by a recorder capturing real-world influences and non-deterministic behavior. A transformation engine generates a reproduction model from the original models and the system traces. Developers can use the reproduction model to retrospectively analyze the system and refine the architecture models for the next iteration.

the data recorded about the system in files we call *system traces*.

Step (4): Using the system traces and the original models, the transformation engine creates the reproduction models. These reproduction models are modified versions of the original models that contain components that mimic the environmental influences such as sensor data. Sec. 6 describes the individual transformations in more detail. In summary, the transformation engine contains transformations to reproduce certain influences of the real world.

Step (5): The models enriched with the real data in the previous step can now be used again for code generation. In our case, we can use the same code generator that we originally used for this purpose. Depending on the implementation of the code generator, however, it may be necessary for other languages to use a different code generator for the reproduction than for the real execution. For example, if the generated code relies on libraries that are only available on the IoT devices, it cannot be compiled as-is for a standard computer. It should be noted that the use of a different code generator may lead to a change in the behavior of the generated code and thus to a violation of requirement **(R1)**. Note that the code generator does not create multiple executables for the reproduction but only a single reproduction executable. This is possible since ports define clear communication interfaces for the components. Thus, the code generator can easily replace real communication over a network with local function calls. Having only one reproduction executable makes it easier to attach debuggers and inspect the application.

Step (6): Developers can use the reproduction executables created in step 5 to analyze the behavior of the system based on the recorded execution on the real devices. The reproduction is executed by a regular computer instead of a distributed system or a resource-constrained embedded device. Unlike distributed or embedded systems, using a debugger on a normal computer is not a problem. Thus, developers can attach a debugger to the reproduction to observe certain properties of the system or to influence its behavior.

Step 6 concludes this iteration of the process. The results of the analysis of step 6 can be used in the next iteration of the process to improve the models. Agile processes are often based on continuously making small improvements to prototypes. Our tooling can also be used in these prototypes as long as the prototypes are generated from architecture models. When the finished product is rolled out en masse to customers, developers might consider turning off our instrumentation (step 2) to save compute and network resources.

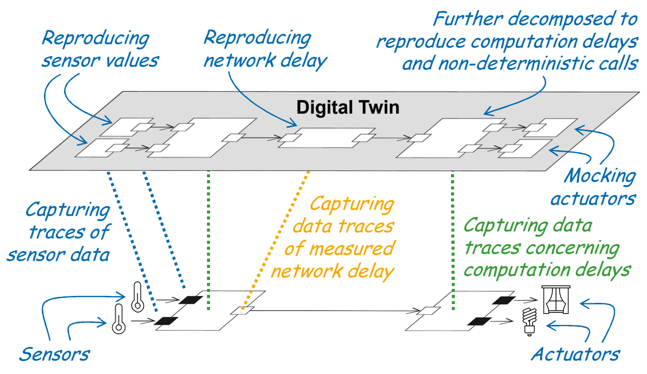

Reproducing sensor values
Reproducing network delay
Further decomposed to reproduce computation delays and non-deterministic calls
Digital Twin
Mocking actuators
Capturing traces of sensor data
Capturing data traces of measured network delay
Capturing data traces concerning computation delays
Sensors
Actuators

Figure 4. Our digital twin (top) uses data and metadata captured from the cyber-physical system (bottom) to reproduce its behavior with the purpose of finding potential problems in the system.

```
template <typename A>
A nd(A value)
{
    if (isRecording)
    {
        storage[index] = value;
    }
    else if (isReplaying)
    {
        value = storage[index].get<A>();
    }

    index++;
    return value;
}
```

Listing 1. Simplified method which can be used to wrap non-deterministic calls. In recording mode, the `value` input is stored and returned. In replay mode, the function returns the previously recorded values.

Fig. 4 summarizes the relation between cyber-physical systems and their DTs in our approach. Overall, we use a set of C&C models of the system to generate the code for a cyber-physical system. By instrumenting the generated code with recording modules, we collect system traces. The models and system traces are then used to reproduce the execution of the system with the goal of analyzing and improving the original system. Thus, applying the definition of DTs (Sec. 2.2) to our concept, these parts of our concept constitute a DT.

6 Recording and Reproduction

Our concept is to record data during the runtime of the system, which can later be used to reproduce the behavior of the system. As mentioned in Sec. 4, several aspects have to be taken into account when reproducing the system's behavior. This section describes the recording of the system traces and the transformation of the architecture in more detail (steps 3 and 4 from Sec. 5).

As already indicated by Fig. 3, our systems can operate in two different modes: Recording and replaying mode. When the recording mode is enabled, the code generator (step 2 from Sec. 5) will inject a recording module into each port of each component. The module subscribes to a special topic where our recorder publishes commands such as starting or stopping the recording phase. The recorder is a central entity that captures user data exchanged between components, non-deterministic behavior, and metadata such as network delay. Since recording is not done permanently, a snapshot including all variable assignments of the internal system state is taken upon recording and restored before replaying. The remainder of this section explains how our framework records and reproduces various aspects of the original system.

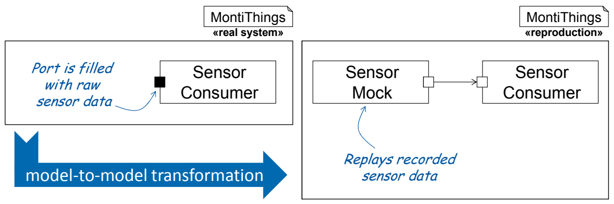
6.1 Record Message Exchange and Timing

MontiThings components are only allowed to communicate with the outside world via ports. This includes both communication with sensors and actuators as well as communication with other components. As explained in Sec. 2, sensor and actuator ports are indistinguishable from regular ports. To capture the communication, the code generator injects a recording module into each port (step 2 from Sec. 5). Thereby, non-deterministic external inputs, *i.e.,* sensor values, are recorded.

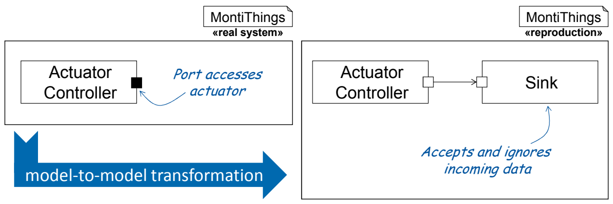
When sending or receiving a message, the injected recording module immediately informs the central recorder. When sending a message, it shares the current vector clock with the recorder. When receiving a message, it acknowledges the message and thus enables its communication partner to calculate the network delay. The partner in turn piggybacks the network delay onto subsequent messages to the recorder. Analogously, recorded data containing computation durations and system calls is also piggybacked as soon as possible.

Components instantly notify the recorder about the receiving or sending of a message. Thus, the recorder calculates the message's timestamp by subtracting the current network delay from the timestamp of the message's arrival at the recorder. Therefore, acknowledges are not limited to messages exchanged by components, but are also used by the recorder. This way we reduce relying on distributed time to the minimum. Similar as before, network delay information is piggybacked and sent to the recorder in subsequent messages.

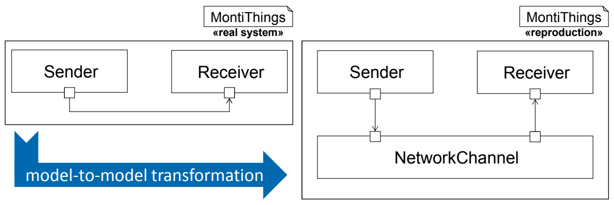
Similarly, injected recording modules capture information about how long computations take to execute. MontiThings uses run-to-completion semantics, *i.e.,* the computation of


MontiThings
«real system»
Port is filled with raw sensor data
Sensor Consumer
model-to-model transformation
MontiThings
«reproduction»
Sensor Mock
Sensor Consumer
Replays recorded sensor data

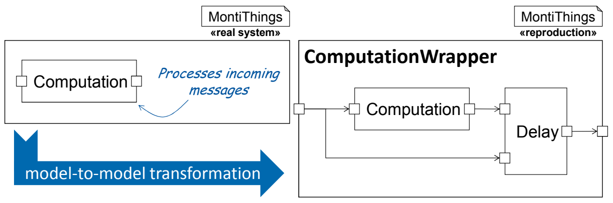
(a) Inject recorded sensor data from system traces using a mock that provides it.



(b) Accept commands to actuators and ignore them. The actuator's influence on the environment is represented using system traces.



(c) Reproduce communication channel imperfections (delay, loss, etc.) using a component reproducing the channel based on recorded meta-data.



(d) Reproduce computation delays. The `ComputationWrapper` has the same interface as the original `Computation` component.

Figure 5. Model-to-model transformations used to instrument MontiThings models of a real system with reproduction-specific model elements. All components from the models of the real system are still present in the transformed models used for reproduction **(R2)**.

an incoming message is not interrupted by subsequent messages. Thus, measurements can be done in sequence and are indexed in ascending order.

6.2 Record and Reproduce Non-deterministic Behavior (R5)

Capturing non-deterministic effects without knowing the semantics of the model implementation is difficult. Intercepting low-level system calls is possible but should be handled with care, as one cannot assure that all possible system calls are intercepted. Besides that, we aim to stay platform-independent.

Thus, our approach leaves it to developers to inform us which code statements introduce non-determinism. When developers inform us of a source of non-determinism, we record it so that we are able to reproduce its behavior. Developers have to mark non-deterministic function calls by wrapping them with a method provided by our framework. The result of the method call is then recorded during recording phase. During replay, the actual result of the non-deterministic call is replaced by the recorded data. Executing the non-deterministic methods in replay mode is safe, as methods used inside MontiThings components must be side-effect free. Note, that this approach does not violate **(R2)**, as the wrapper is present in both the original system and the reproduced system. Wrapping non-deterministic calls is the only manual modification required by the developers.

Common non-deterministic calls are, for instance, `now()` for retrieving the current time. In our prototypical implementation, developers can mark `now()` as non-deterministic using `nd(now())`. The wrapping call is implemented as a single generic-typed method as shown in Listing 1. This way, the non-deterministic system call is executed as usual before executing our wrapper. When the recording mode is active, the wrapper indexes and stores the method's return value locally until it is sent to the recorder. During replay phase, the wrapper replaces the actual result of the non-deterministic call by the recorded value. Leveraging a simple FIFO queue for handling internal system calls is sufficient, as all inputs to the component are deterministic, including the timing, quantity, and actual values.

6.3 Reproduce Environmental Influences (R3), (R4)

Components use sensor and actuator ports to communicate with their environment. During reproduction, the sensors are usually not available. And even if they are, they will likely produce different values than during the recording. Thus, we use the system traces captured from these ports to replay the interaction with them. To this end, a model-to-model transformation as shown in Fig. 5(a) creates a new mock component and connects it to the original port. During reproduction, this new component acts as a black-box that replays external inputs and their timing based on the system traces.

Analogously to the sensor ports, actuators are mocked as well, as illustrated in Fig. 5(b). In contrast to sensors, the implementation details of mocked actuators are irrelevant for the system execution. Hence, the sinks serving as actuators do not have any behavior. This is possible since the influence actuators have on the system's environment is detected by the system using sensors. By recording sensor values, the recorder captures if an actuator influences the system's environment in a way that is relevant to the software. This, of course, does not cover cases in which an actuator directly modifies the system, *e.g.,* by physically destroying parts of the system.

Finally, **(R4)** includes the requirement that exchanged messages must reach the target in the correct order. If two events occur almost simultaneously (*i.e.,* approx. $1-2$ ms or less between them), an operating system's scheduler may decide to process them in a different order. Since we do not require a simulation environment where the timing can be precisely controlled, we need to ensure that the scheduler of the operating system of the developer's computer does not execute events that are very close to each other in a different order while replaying. To mitigate this, we put events that are very close together a little farther apart, making it less likely that the scheduler will change the order of events. The recorder uses the vector clock data to make small adjustments to the timing, which we refer to as *determinism spacing (DS)*. DSs are short delays that impose a minimum delay between two events on the same component. They affect only the simultaneous events and do not postpone later events, *i.e.,* DSs do not add up. Overall, DSs trade a less accurate timing for a more accurate order of the events.

6.4 Reproduce Network and Computation Delay (R4)

A common mistake in developing distributed systems is to fall for the (wrong) assumption that there is no network delay [34]. To enable developers to identify network-related errors, our tool must reproduce network delay. As mentioned above, components communicate only via ports and connectors. Replaying network delay is done by the model-to-model transformation shown in Fig. 5(c). This transformation replaces the connector between the sender and receiver by a new component which applies the recorded network delay. As explained in Sec. 6.1, this delay is derived based on the round-trip time of exchanged messages. Since the network delay is recorded with each exchanged message, this component can vary the delay over time should it not be constant. This also enables us to replay failing network connections, which cause an infinite delay.

The time consumed for computations is reproduced using the transformation described in Fig. 5(d). The original component (`Computation`) is wrapped by a new component (`ComputationWrapper`). The wrapper has the same interface as the original component, so the wrapper can replace the original component in all places where it is used. Besides the original component, the wrapper also includes a `Delay` component. Incoming messages are forwarded to both the original component and the `Delay`. The `Delay` component leverages incoming messages to keep track of the arrival

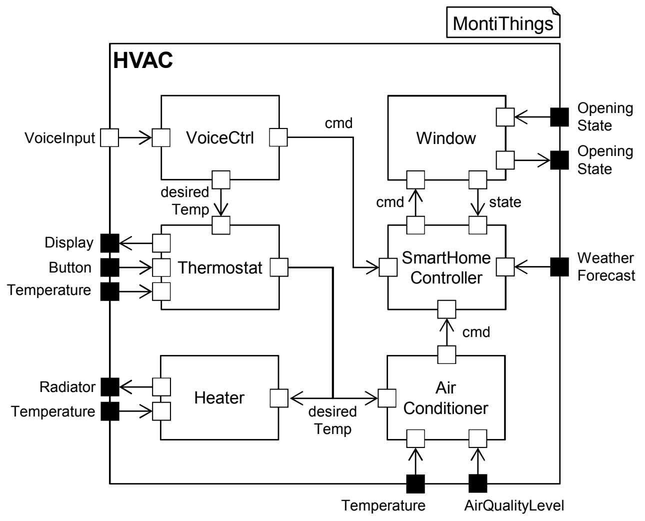


Figure 6. High level abstraction of the HVAC component managing the heating, ventilation, and air conditioning of a smart home (*cf.* Fig. 1)

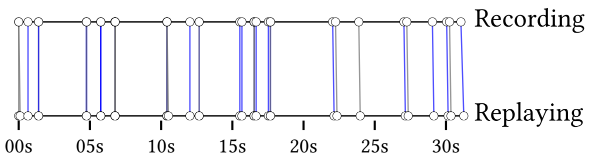
times. When the original component finishes its calculation, the outgoing messages are intercepted by the `Delay` component. The actual computation duration is measured, compared to the recorded computation duration, and, if necessary, a delay is added before sending the computation result on outgoing ports. To this end we assume that the computer executing the reproduction will carry out computations faster than the ressource-constraint IoT devices of the original system. We could, of course, also replay the recorded computation results should the reproduction take longer to execute than the recorded execution. However, this would mean that developers could no longer pose what-if questions to the system by modifying variables or behavior, since the system would always behave exactly like the original system.

7 Evaluation

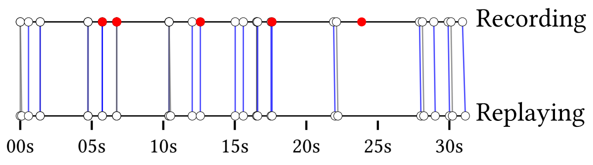
We validate the feasibility of our approach on using a case study specifically designed to produce difficult to comprehend executions in Sec. 7.1. In order to measure the accuracy of the reconstructed behavior we run predefined sensor input. This enables us to control the execution flow in the recording phase. By capturing specific events during the executions we are able to compare the differences between the original and the reconstructed behavior. Further, we evaluate the effect of DSs on the accuracy. Since record and replay approaches are expected to cause computational overhead, Sec. 7.2 focuses on the efficiency of our prototypical implementation.

7.1 Validation

The heating, ventilation, and air conditioning (HVAC) application (Fig. 6) decides on several actions such as opening the



(a) Timeline of an equal execution replay.



(b) Timeline of a faulty replay.

Figure 7. Timeline comparisons of two record and replay executions. Each circle marks an event at a specific time. Vertical connections indicate which event of the original execution corresponds to which event within the replayed execution. In a 100% accurate reproduction, all connecting lines run perfectly vertically. The more oblique a line is, the less accurate the reproduction. External input events are marked in blue. In erroneous replays, events might not occur in the time due to the different execution flow (red circles).

window if the air quality falls below a threshold, or turning on the heater if the room temperature is lower than desired. The corresponding decisions result from the environmental influences which the HVAC consumes over its external ports. For space reasons, subcomponents of the HVAC's subcomponents are omitted in Fig. 6. Overall, the case study consisted of 15 component types. We developed the system iteratively, using our tool to repeatedly track down errors in the system and then using these findings to improve the system.

We create deterministic execution flows by reproducing timed events on the outermost components' incoming ports, thus requiring a reaction from the system. We constructed a specific scenario that triggers a complex execution flow; a case where a developer would prefer to debug a replayed version rather than browse numerous log files. The scenario involves hard-to-trace behavior caused by sensor misreadings, network delay, and side effects. For instance, the window state may change, *e.g.*, due to one of the following events:

- the temperature is too far from the desired temperature
- the voice assistant receives a command to open or close the windows
- the air quality decreases
- the weather sensor forecasts stormy conditions
- a timer automatically closes the window at night

The specific sequence of events that caused the window's behavior becomes difficult to trace when multiple, possibly simultaneous, events or erroneous sensor measurements could be the cause.

To be able to compare replays with the original executions, we injected specific context-aware and time-stamped log entries within the application implementation. These log entries notify about system state changes, such as receiving a new voice command or activating the heater. By capturing these log entries and sorting them on a timeline, an accurate representation of the execution flow is achieved.

Table 1. Results from 100 runs with different DSs. Disabling DS led to different replayed executions half of the time.

Det. Spacing (DS)	disabled	1 ms	3 ms	5 ms
non-equal replays	49 %	15 %	5 %	0 %
avg. latency error	1.9 ms	2.6 ms	2.5 ms	3.0 ms
median latency error	1.2 ms	1.5 ms	1.5 ms	2.0 ms
standard deviation σ	2.4 ms	3.5 ms	3.3 ms	3.2 ms

During the recording phase, we run the HVAC application on a single machine[2] using separate Docker containers for each component. This allows us to control network delay while ensuring that timestamps are generated only by a single source. After the execution, logs are collected, merged, and parsed into a timeline. We then can test if the replay was equal to the original execution. Since components are stream processing functions in FOCUS, we consider the replay to be equal to the original execution if all streams of all ports are equal in the original execution and the replay. Especially, if the replayed execution is equal to the original one, the corresponding timelines should be equal as well.

As described above, the scheduler of the operating system of the developer's computer can incorrectly change the order of replayed events that are very close together. For instance, at the beginning of our scenario the thermostat is configured. A desired temperature is typed in and the button is pressed to apply the change. Internally, the user-interface component receives both inputs nearly simultaneously with a delay of less than a microsecond. As a consequence, in nearly half of 100 repetitions, the heater never turns on which can be observed at about second 24 in Fig. 7(b) where an event is missing during replay. Table 1 shows how DS can mitigate this problem by trading a less accurate timing for a more accurate order of events. Since FOCUS considers components

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

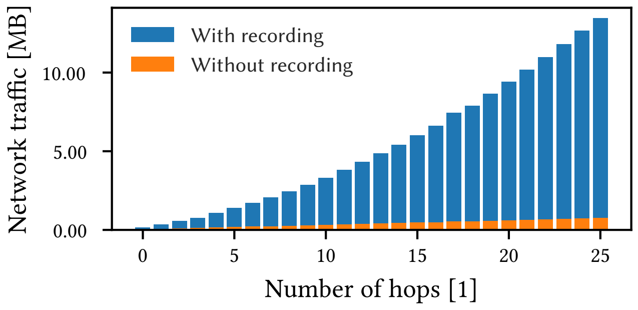


Figure 8. Total network overhead increases exponentially with the number of hops. This is expected, as messages include a copy of the vector clock.

to be stream processing functions, we consider a replay "non-equal" if at least one stream of messages at any incoming or outgoing port is not equal to the recorded stream.

7.2 Evaluation

To identify possible bottlenecks and potential room for optimization, we construct an application as follows that evaluates our approach in terms of worst-case overhead. A source component sends 100 messages to a sink component. In between, a certain number of components intercept the communication. These forwarding components, which we call *hops*, simply forward the message. This considerably increases the size of the vector clocks exchanged and the recorder must acquire data from all instances. During execution, we leverage *cAdvisor*[3] to capture resource usage metrics from all participating Docker containers.

Network Traffic Overhead. Network overhead is shown in Fig. 8. The additional traffic introduced while recording is justified as follows. For each message exchanged between model components, the participating instances have to send their action to the recorder, together with network delay information, a list of computation delays which have been accumulated since the previous message, vector clock information, and intercepted system call data. Since we have vector clocks, every node needs to have an entry for every other node. If we add a new node to the system all nodes need to get a new entry for the newly added node in their vector clocks. Additionally, the newly added node also has a vector clock. Thus, we have a $O(n^2)$ total network traffic. Further, each message is acknowledged, whereas acknowledgments are not limited only to messages between components but also include recorded samples which are sent to the recorder.

Since all data is serialized in JSON and the vector clocks include fully qualified instance names instead of smaller identifiers, our prototypical implementation has room for

[3]Project website (last checked 23.09.2021): https://github.com/google/cadvisor

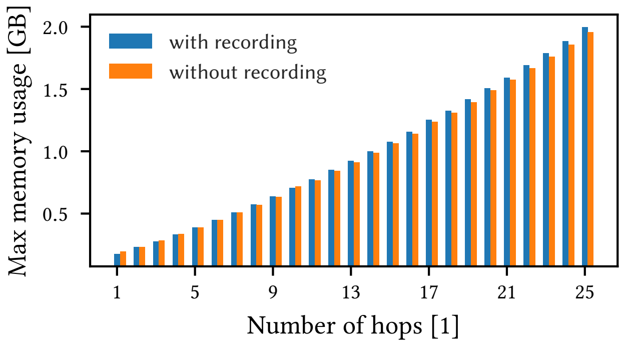


Figure 9. Maximum cumulated memory consumption of participating docker containers. Recording only adds little overhead and scales proportionally.

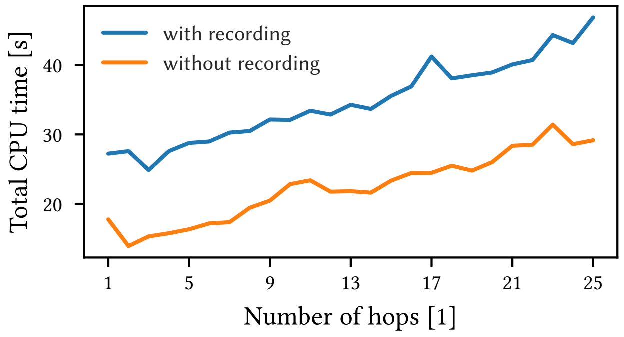


Figure 10. Total CPU time spent in the kernel and user space by the sink component. Recording scales proportionally.

optimization. As a further improvement one could track acknowledgements on the transport layer instead of exchanging additional acknowledgements on a higher level. However, as such methods require very low level kernel readings and are bound to a specific transport implementation like TCP, we decided against doing so to stay platform independent.

Memory Usage and CPU Time. As indicated by Fig. 9 memory consumption during recording increases linearly with the number of hops. Compared to the execution without recording, our approach causes only low overhead. Part of this overhead can be attributed to the additional resource allocations of the recording module. As shown by Fig. 10, CPU consumption is noticeably increased. This is to be expected because a lot of expensive JSON parsing is performed internally instead of using optimized data structures.

Overall, our approach scales linearly. In its prototypical implementation, it incurs a notable overhead, which we attribute to the factors mentioned above. We expect our ch to incur limited overhead in optimized implementations.

8 Discussion

Overall, our approach enables us to reproduce error situations in retrospect. Our model transformations cover common errors such as sensor or network errors. One of the advantages of our model-driven approach is that there is no manual work involved when recording or replaying system traces. Developers only need to toggle a switch that activates the transformations. We would like to emphasize that we do not envision our approach as a substitute for conducting appropriate simulation and testing prior to deployment. We see the approach as an additional opportunity to analyze errors that were not uncovered by such pre-deployment analysis. We expect that with the increasing popularity of DTs, the system traces required by our approach will be available for more and more IoT systems with less additional effort, thus reducing the effort required to implement our approach. Extending our approach, this data could also be used for further (automated) artificial intelligence (AI)-based analyses and improvements. This would allow AI to further support human developers in iterative development processes. However, there are a few limitations to our approach that we would like to discuss.

First of all, any technique used to gather required data for the reproduction may already influence the behavior of the system. The unintended but inevitable alteration is known as *probe effect* [11]. Thus, our recording infrastructure may lead to an unintended change in the behavior of the system.

Moreover, since simulations always abstract from the real world, even when using recorded system traces, some deviations from the real world are to be expected. Currently, the sequence of events with an interval of less than one millisecond can be recorded but cannot be deterministically reproduced by our prototypical implementation. We expect the accuracy of the reproduction could be increased by dropping **(R2)**. This would allow us to force components to process specific events in the recorded order. Doing so would, however, limit the developers' ability to modify components for the reproduction. The benefit of **(R2)** is that it enables developers to modify the components' behavior and test out how the system would have behaved in the same situation if it was implemented differently. Of course, structural modifications, *e.g.,* adding new sensor ports, are not supported because the replayer does not have the necessary data to reproduce them.

Adding fully *simulated* components, rather than *replayed* components, or a simulation environment that controls time could potentially fill this gap and further improve timing accuracy. For example, network simulators such as ns-3 could simulate appropriate network channels for new connectors. However, if manual changes to the model are not covered by system traces, these changes will cause simulations to deviate further and further from the real world. As a result, the insights derived from these simulations become less and less valid. Therefore, we think that limiting the ways in which developers can modify the system when replaying system traces will lead to the most useful results. Moreover, the parameterization of such simulated network channels would require manual effort.

We are aware that MontiThings is not a widely used framework. Therefore, we have deliberately limited ourselves to common model elements—ports and components—that are also found in many other IoT frameworks such as ThingML, Calvin, or Node-RED. Since the other toolchains are very similar to MontiThings in terms of core modeling elements, we think that the concept is well transferable. While our context is the IoT, we think this approach is particularly useful when a system depends on many external influences and may be subject to hard-to-reproduce hardware faults.

9 Related Work

Using transformations for adding functionality to architecture models has been previously proposed. In [23], transformations are used to add loggers to annotated architecture elements. Thereby, the need for adapting multiple code generators is avoided. [15] uses transformations in combination with a tagging language to add model elements that keep an executed architecture model synchronized with its DT.

The record and replay approach is widely adopted in many domains, ranging from interprocessor communication to highly abstracted distributed systems [9]. The most closely related work is done in the field of actor models which share some key characteristics of C&C architectures, namely the isolation of entities and the communication via well defined interfaces. In Actoverse [32], each time an actor processes an incoming message, its state is saved as a snapshot. This allows rollbacks to past states, enabling reverse debugging capabilities. Message flow causality is ensured by additional Lamport timestamps which allow to replay the recorded execution deterministically. Other than Actorverse, the work of [3] also supports capturing non-deterministic system calls. It is more focused on the efficiency aspect of keeping the overhead as small as possible. However, both approaches replay the execution on the original actors, whereas our solution generates a local application that reproduces the original system.

WiDS Checker [20] records traces about distributed systems built using the WiDS toolkit. Like our approach, WiDS Checker uses these traces to reproduce executions of the distributed system in a local application that developers can use for debugging. In contrast to our approach, the WiDS Checker does not focus on model-driven systems or on the environmental influences to which IoT systems are exposed.

A model-driven approach was taken in [4], where traces of a system generated from an architectural model were recorded with the goal of replaying these traces. Their architecture models relied on state charts for modeling the behavior of the components. This enabled them to find implausible traces that could not have resulted from a correct execution of the state chart. By resorting the traces to a valid sequence, these errors could be repaired. Our work is different in that we apply transformations to create the model used for reproduction. Thereby, we can reproduce external inputs and meta-data about the system, *e.g.,* network delays, that is neglected by [4]. As future work, combining both approaches could improve the reliability of each one.

The robot operating system (ROS) [18, 29] ships a collection of libraries and tools for the robot automation domain. Recently, ROS2[4] emerged as the successor of ROS, integrating various DDS implementations as its middleware. Their tool rosbag enables the recording and replaying of ROS topics. The method of recording by subscribing to the available topics is very similar to our approach, although the network delay between the components and the recorder is neglected in rosbag. This, however, can be a crucial aspect in IoT applications where network delays may exceed the range of a few milliseconds, especially via 3G or 4G cellular networks. Replaying is done by sending the recorded data to the corresponding topics which differs from our approach where the replayed data is integrated within the model transformations. Therefore, the reproduction does not depend on DDS anymore which is desired, for instance, if the reproduction is packaged into a single binary. Packaging the application into a single binary for the replay has many advantages. Most importantly, this provides developers extended debugging capabilities. For example, they can set breakpoints, pause the application, and inspect its state by, *e.g.,* attaching debuggers like gdb. In contrast, when debugging a distributed application, pausing the application as a whole or inspecting the whole state is usually not possible. As future work, a more matured version of *rosbag* for ROS2 could benefit our recording tool, as they focus on the scalability and efficiency of collecting data.

The more general problem of debugging distributed systems is also examined, *e.g.,* in [31]. The network simulator ns-3 is extended to be able to execute the same code as the real devices. Thereby, developers can analyze and debug their software in a controlled environment with popular debuggers such as gdb. However, the system in [31] does not rely on recorded data from a real system. Hence, it can not retrospectively answer the question why the system behaved in a certain way without manually reproducing this situation in a simulator. Depending on the amount of needed sensor data, reproducing scenarios is time-consuming and error-prone.

10 Conclusion

The behavior of IoT applications is often difficult to analyze. It depends not only on the application's models and code, but also on external influences such as (errors in) sensor data and network communication. As a result, the behavior is challenging and error-prone to understand from log files, and suspected error situations are difficult to recreate. Using system traces recorded by a DT, the system's execution can be replayed later to better understand the system. We use model-to-model transformations to add reproduction components to architecture models. This enables us to recreate the behavior of the application in the real world retrospectively, facilitating developer debugging. Future work includes using this data for deriving test cases for future versions of a system. Moreover, applying the transformations only to parts of the system could enable developers to debug their application in a mixed environment where parts of the system operate on live data and other parts operate on replayed data.

Source Code

The source code for 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] [n.d.]. Node-RED—Low-code programming for event-driven applications. [Online]. Available: https://nodered.org. Last checked 27. September 2021.

[2] Ola Angelsmark and Per Persson. 2017. Requirement-Based Deployment of Applications in Calvin. In *Interoperability and Open-Source Solutions for the Internet of Things*, Ivana Podnar Žarko, Arne Broering, Sergios Soursos, and Martin Serrano (Eds.). Springer International Publishing, Cham, 72–87.

[3] Dominik Aumayr, Stefan Marr, Clément Béra, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2018. Efficient and Deterministic Record & Replay for Actor Languages. In *Proceedings of the 15th International Conference on Managed Languages &amp; Runtimes (ManLang '18)*. Association for Computing Machinery, New York, NY, USA, Article 15. https://doi.org/10.1145/3237009.3237015

[4] Majid Babaei, Mojtaba Bagherzadeh, and Juergen Dingel. 2020. Efficient Reordering and Replay of Execution Traces of Distributed Reactive Systems in the Context of Model-Driven Development. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, New York, NY, USA, 285–296. https://doi.org/10.1145/3365438.3410939

[5] U. Barakkath Nisha, N. Uma Maheswari, R. Venkatesh, and R. Yasir Abdullah. 2014. Robust estimation of incorrect data using relative correlation clustering technique in wireless sensor networks. In *International Conference on Communication and Network Technologies*. 314–318. https://doi.org/10.1109/CNT.2014.7062776

[6] C. Brooks, C. Jerad, H. Kim, E. A. Lee, M. Lohstroh, V. Nouvelletz, B. Osyk, and M. Weber. 2018. A Component Architecture for the Internet of Things. *Proc. of the IEEE* 106, 9 (September 2018), 1527–1542.

[7] Manfred Broy and Ketil Stølen. 2001. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement.*

[4]Project website (last checked 23.09.2021): https://github.com/ros2/rosbag2

Springer Heidelberg.

[8] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)* (Athens, Greece). IARIA XPS Press, 213–218.

[9] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. 2015. Deterministic Replay: A Survey. *Comput. Surveys* 48, 2 (Nov. 2015), 1–47. https://doi.org/10.1145/2790077

[10] P. Eugster, V. Sundaram, and X. Zhang. 2015. Debugging the Internet of Things: The Case of Wireless Sensor Networks. *IEEE Software* 32, 1 (2015), 38–49. https://doi.org/10.1109/MS.2014.132

[11] Jason Gait. 1986. A Probe Effect in Concurrent Programs. *Software: Practice and Experience* 16, 3 (March 1986), 225–233. https://doi.org/10.1002/spe.4380160304

[12] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. 2006. Replay Debugging for Distributed Applications. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (Boston, MA) *(ATEC '06)*. USENIX Association, USA, 27.

[13] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. 2012. *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems*. Technical Report AIB-2012-03. RWTH Aachen University.

[14] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 2016. ThingML: A Language and Code Generation Framework for Heterogeneous Targets. In *Proc. of the ACM/IEEE 19th Int. Conf. on Model Driven Engineering Languages and Systems* (Saint-malo, France) *(MODELS '16)*. ACM, New York, NY, USA, 125–135.

[15] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. 2020. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, 90–101.

[16] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. 2022. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software* 183 (January 2022), 111087. https://doi.org/10.1016/j.jss.2021.111087

[17] R. Konuru, H. Srinivasan, and Jong-Deok Choi. 2000. Deterministic replay of distributed Java applications. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*. 219–227. https://doi.org/10.1109/IPDPS.2000.845988

[18] Anis Koubaa. 2016. *Robot Operating System (ROS): The Complete Reference (Volume 1)* (1st ed.). Springer Publishing Company, Incorporated.

[19] X. Larrucea, A. Combelles, J. Favaro, and K. Taneja. 2017. Software Engineering for the Internet of Things. *IEEE Software* 34, 1 (Jan 2017), 24–28. https://doi.org/10.1109/MS.2017.28

[20] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. 2007. WiDS Checker: Combating Bugs in Distributed Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation* (Cambridge, MA) *(NSDI'07)*. USENIX Association, USA, 19.

[21] Y. Liu, T. Dillon, W. Yu, W. Rahayu, and F. Mostafa. 2020. Noise Removal in the Presence of Significant Anomalies for Industrial IoT Sensor Data in Manufacturing. *IEEE Internet of Things Journal* 7, 8 (2020), 7084–7096. https://doi.org/10.1109/JIOT.2020.2981476

[22] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. 2012. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks* 10, 7 (2012), 1497 – 1516.

[23] B. Morin and N. Ferry. 2019. Model-Based, Platform-Independent Logging for Heterogeneous Targets. In *ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 172–182. https://doi.org/10.1109/MODELS.2019.000-4

[24] B. Morin, N. Harrand, and F. Fleurey. 2017. Model-Based Software Engineering to Tame the IoT Jungle. *IEEE Software* 34, 1 (January 2017), 30–36.

[25] National Transportation Safety Board, Public Meeting of February 25, 2020. [n.d.]. Collision Between a Sport Utility Vehicle Operating With Partial Driving Automation and a Crash Attenuator. Mountain View, California, March 23, 2018, HWY18FH011.

[26] Elisa Negri, Luca Fumagalli, and Marco Macchi. 2017. A Review of the Roles of Digital Twin in CPS-based Production Systems. *Procedia Manufacturing* 11 (2017), 939–948. 27th International Conference on Flexible Automation and Intelligent Manufacturing, FAIM2017, 27-30 June 2017, Modena, Italy.

[27] Object Management Group (OMG). 2015. Data Distribution Service (DDS), Version 1.4. [Online]. Available: https://www.omg.org/spec/DDS/1.4/PDF (Last checked 21.03.2021).

[28] Per Persson and Ola Angelsmark. 2015. Calvin – Merging Cloud and IoT. *Procedia Computer Science* 52 (2015), 210 – 217. 6th Int. Conf. on Ambient Systems, Networks and Technologies (ANT).

[29] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[30] Jan Oliver Ringert and Bernhard Rumpe. 2011. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics* (2011).

[31] Martin Serror, Jörg Christian Kirchhof, Mirko Stoffers, Klaus Wehrle, and James Gross. 2017. Code-transparent Discrete Event Simulation for Time-accurate Wireless Prototyping. In *Conference on Principles of Advanced Discrete Simulation* (Singapore, Republic of Singapore) *(SIGSIM-PADS '17)*. ACM, New York, NY, USA, 161–172. https://doi.org/10.1145/3064911.3064913

[32] Kazuhiro Shibanai and Takuo Watanabe. 2017. Actoverse: A Reversible Debugger for Actors. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2017*. ACM Press, Vancouver, BC, Canada, 50–57. https://doi.org/10.1145/3141834.3141840

[33] Doug Soldat and John Stier. 2011. Watering your lawn. Bulletin. A3950. University of Wisconsin-Extension, Cooperative Extension. [Online] http://learningstore.uwex.edu/Assets/pdfs/A3950.pdf (Last checked 23.09.2021).

[34] A. Taivalsaari and T. Mikkonen. 2017. A Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software* 34, 1 (Jan 2017), 72–80. https://doi.org/10.1109/MS.2017.26

[35] Behrang Ashtari Talkhestani, Nasser Jazdi, Wolfgang Schlögl, and Michael Weyrich. 2018. A concept in synchronization of virtual production system with real factory based on anchor-point method. *Procedia CIRP* 67 (2018), 13–17.