DyST: Dynamic Specification Mining for Heterogenous IoT Systems with WoT

Ege Korkan[1][0000−0003−4910−4962], Silvia Oliva Ramirez[2][0009−0001−4340−7594], and Sebastian Steinhorst[2][0000−0002−4096−2584]

[1] Siemens, Munich, Bavaria, DE ege.korkan@siemens.com
[2] Technical University of Munich, Munich, Bavaria, DE silvia.ramirez; sebastian.steinhorst@tum.de

Abstract. The comprehension of a distributed system and its verification is one of the most challenging problems in today's software engineering, commonly referred to as observability. The complexity increases when one cannot control all the components, like in IoT systems composed of third-party devices. The Web of Things standards by the W3C help with this by describing what one can do with an IoT device via network messages. However, no work has leveraged these standards to offer an observability solution that works with any set of IoT devices. This work addresses this gap by proposing a method to verify the correctness of the system by mining its specification from device interactions. Our approach can reverse engineer complex application logic in the form of UML Sequence Diagrams from the analysis of network messages of any protocol between the devices during system runtime, which can be used to programmatically assert the correctness of the mined specification. We have evaluated our approach with three case studies to assess our mining technique, the performance of our algorithms, and the applicability of our contributions to system verification in the IoT. Our results show that our approach can produce accurate Sequence Diagrams that help understand and verify the behavior of IoT systems.

Keywords: Internet of Things · Process Mining · Web of Things

1 Introduction

The Internet of Things (IoT) has become a key enabler of today's emerging technologies, entering our daily lives through Internet-connected devices. However, its exponential growth has spawned numerous IoT platform providers and manufacturers, each designing devices to function within distinct frameworks and technical environments, leading to interoperability challenges. The absence of best practices or standards has resulted in the proliferation of diverse technologies and implementation methods, causing significant fragmentation in the IoT domain and necessitating substantial integration and development efforts.

To tackle this interoperability challenge, the World Wide Web Consortium (W3C) introduced the Web of Things (WoT) [18], an architecture aimed at fostering interoperability across IoT platforms and application domains. Central to

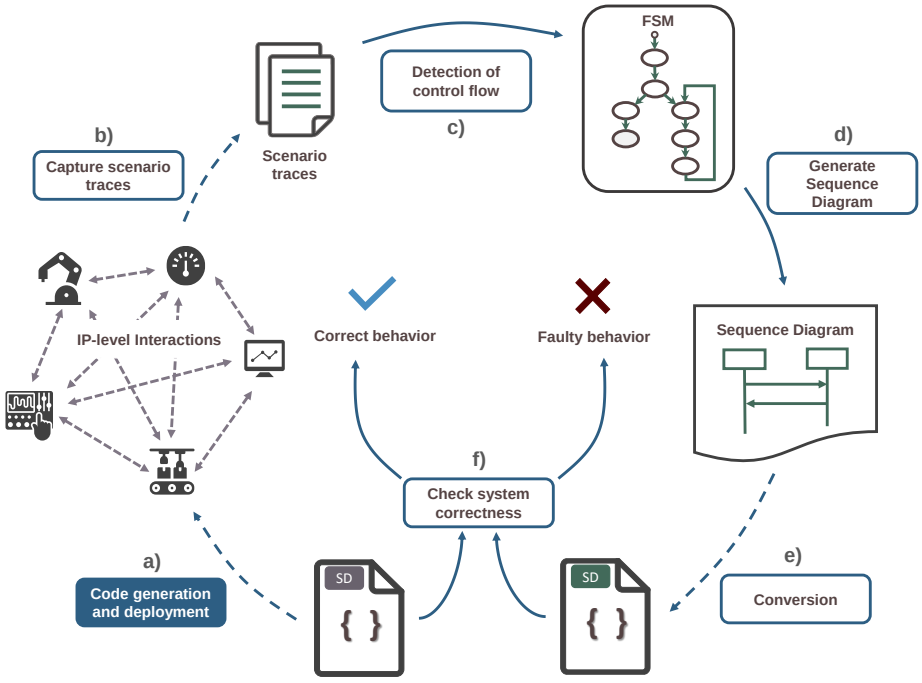


Fig. 1: Approach overview. State-of-the-art techniques are marked with dashed lines while our contributions are drawn with continuous lines. a) System design (with a System Description (SD)) and code deployment. b) During normal system execution, communication traces for various scenarios are obtained. c) Traces are analyzed, loops and branches detected, and a Finite State Machine (FSM) is constructed. d) Based on the FSM, a Sequence Diagram (SeqD) is generated. e) The SeqD is automatically converted to an SD. f) Both SD files are compared to verify system correctness.

this architecture are Thing Descriptions (TDs) [19], which provide a standardized format for uniformly describing the network-facing capabilities of a Thing. Given that individual IoT devices may lack complex functionalities, they are often combined to perform common tasks, known as Mashups. These can range from automatic irrigation systems in smart farms to item-sorting systems in factories or even turning on lights when a presence is detected in a smart home.

1.1 Problem Statement

As distributed IoT systems like Mashups grow larger and more complex, the need to understand their behavior and to support their verification becomes more important. Verification techniques that target a distributed system before its full operation can only be partially representative of the executions in the field, which results in a large difference between the assumed behavior during the design and the actual behavior of the system.

So far this has been addressed by specification mining techniques, which are commonly used to derive the specification of a program using examples of cor-

rect usage to aid program understanding. Besides helping with comprehension, mined specifications are useful for supporting quality assurance and for examining the relationships between the actual behavior of a system and its designed specification. However, state-of-the-art specification mining techniques are unable to address the various protocols and platforms that are present in the IoT. As a result, such techniques are not applied to real IoT systems and IoT systems lack the benefits of well-established specification mining techniques to allow their verification.

1.2 Approach and Contributions

To improve the management of WoT Mashups, [9] introduced the WoT System Description (SD), which can interchangeably represent a Mashup in a well-defined text format or a graphical manner with a subset of the Unified Modeling Language (UML) Sequence Diagram [5]. This work builds on top of the System Description, which represents the system behavior description our approach tries to mine. More specifically, we introduce DyST, a method and a corresponding implementation as a solution to automatically observe IoT systems by analyzing the communication traces produced during the normal execution of the system and building a corresponding System Description. Fig. 1 shows an overview of the approach, where state-of-the-art techniques are combined with our contributions. In particular, we make the following contributions:

- We propose a new protocol-independent format for representing communication traces between devices in the WoT that focuses on application semantics and operations.
- We present a method and its open-source implementation[3] that takes multiple communication traces from different behavioral scenarios as input and generates UML-compliant Sequence Diagrams and System Descriptions (SDs).
- We perform a full evaluation of our method with transformations from communication traces to SDs using three different case studies to prove its applicability for checking a system's correctness as well as its time performance.

The rest of the paper starts with highlighting the background information and related work in Section 2. Section 3 explains the scientific contributions, as well as the implementation. Section 4 presents the evaluation methodology and results, and Section 5 concludes and proposes future work directions.

2 Background and Related Work

To target distributed systems with third-party devices to mine their behavior, this work builds on top of well-established methods found within the WoT, Specification Mining, and Specification and Modeling of Distributed Systems.

[3] Also referred to as our repository, it is available at `https://github.com/tum-esi/dyst-wot-miner`

2.1 Web of Things

The WoT encompasses standards at the W3C, focusing on Things that expose Interaction Affordances through well-described network interfaces. This enables Consumers such as applications, cloud services, or browsers to interact with them.

The TD [19] defines a standardized metadata format and vocabulary to describe functionality and interactions across networked Things. The TD is the main building block of the WoT and is extended with various **Binding Templates** [13] to describe various IoT protocols so that a Consumer can send the correct protocol messages to the Thing. The Interaction Affordances are categorized into three:

- *Properties* can be read, written, and observed, and represent a state of the Thing, such as a sensor value.
- *Actions* can be invoked and execute a function of the Thing, which might manipulate its state, e.g. executing a movement.
- *Events* can be subscribed to and result in a notification each time the event occurs, for example, notifying when a person is detected.

WoT System Description: The SD introduced in [9] extends the capabilities of a TD by providing additional keywords to describe the composition of WoT Things. To this end, SD introduces a means to specify the execution of interactions and to represent application logic that consists of programming structures, e.g. if statements, for loops, and wait statements.

2.2 Specification Mining

Specification mining is a program analysis method deriving program specifications from correct usage examples. [16] summarizes works in this field, ranging from early papers like [3] to recent ones. Specification mining techniques are categorized based on the input data source such as parsing program source files, analyzing execution traces obtained during runtime, or using a combination of both approaches. These are called Static Specification Mining, Dynamic Specification Mining, and Hybrid Specification Mining, respectively.

IoT systems, influenced by both environment and device programs, are highly dynamic. They comprise devices from diverse manufacturers, often inaccessible in terms of source code. With these aspects in mind, we focus on Dynamic Specification mining techniques in this work.

Dynamic Specification Mining: Dynamic specification mining can provide an accurate representation of a system's behavior because the analysis is performed entirely during runtime. In general, dynamic specification mining is based on the analysis of so-called execution traces. The simplest way to obtain execution traces is to instrument the source code [2, 6, 8] or rely on custom debuggers [22,23]. A smaller number of techniques [14] obtain the needed traces by analyzing the interactions between devices connected to the network. This can be accomplished by sniffing the network and collecting communication traces,

i.e. all necessary information about method calls, including callers and callees. This work uses communication traces since in WoT systems there is no guarantee to access the Things' source code or the Mashup application logic.

Scenario traces: A single program run generating a single trace may not capture the application behavior due to missing repetitions or conditional alternatives. To get a complete view of the program's behavior, it is necessary to obtain traces that represent different behavioral scenarios and thus cover all possible options. This collection of traces is called scenario traces in the rest of this work.

Testing of Things: To enable verification of individual WoT things, [12] proposes to do an affordance coverage test where their contributions interact with Things over the network and send well-planned requests to execute each affordance with different inputs. However, it influences Things and their physical environment, making it unsuitable for the normal working mode of a Thing.

2.3 Specification and Modeling of Distributed Systems

Specification languages and graphical representations are well-established methods that aid program documentation, development, and testing. The relevant methods for this work are briefly summarized below, together with how they relate to our method.

Finite State Machines (FSMs): FSM-derived specifications, based on [7], effectively model historical software behavior patterns. They are intuitive and powerful to represent recurring patterns of behavior, especially for sequencing, selection, and iteration. By an FSM, we mean a deterministic transition state machine, also known as a deterministic finite state machine or Deterministic Finite Automata (DFA).

The simplicity and power of FSMs have spurred techniques to reverse-engineer software applications, generating FSMs as final or intermediate representations for complex behavioral models like Regular Expressions. Here, FSMs are employed to infer software behavior, detecting loops and branches before transforming them into more complex models such as UML Sequence Diagrams.

UML Sequence Diagrams: UML Sequence Diagrams visually represent program behavior by illustrating interactions among software system objects. They depict system flow using incoming and outgoing messages, following a standardized representation defined in the UML specification [5]. In this work, we employ UML Sequence Diagrams, specifically the subset defined in [9], to depict behavior extracted from communication traces.

3 DyST Approach

In this work, we consider dynamic specification mining of IoT systems by reverse engineering Sequence Diagrams from execution traces. Our goal is to extract a Sequence Diagram that is WoT-compliant and can be further transformed into a System Description for system verification. Fig. 2 shows a more detailed overview of our approach, which is composed of the following steps that are further elaborated in the subsequent sections:

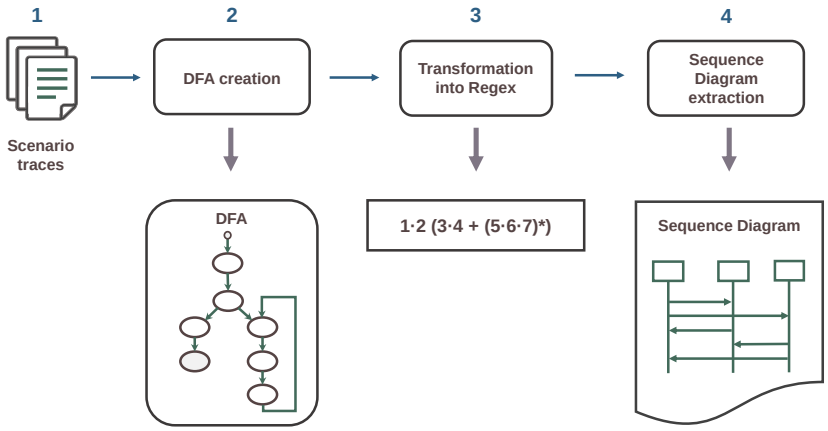


Fig. 2: Detailed approach overview. 1) The scenario traces collected during the program execution are the input to the algorithm that creates the DFA (2). 3) Based on the DFA of the system, a Regular expression is obtained. 3) A mapping from the Regular Expression to the Sequence Diagram is implemented in the last step.

- In step 1, the scenario traces obtained during normal system execution are parsed, sorted, and modified to follow a format we propose for representing communication traces between devices in the WoT.
- For control flow detection, in step 2 we create a DFA to obtain an overall representation of the system behavior. This creation is based on model inference algorithms and the minimization of the state transition diagram using transition manipulation formulas.
- With step 3 we create a Regular Expression following the Transitive Closure Method, which is a recursive technique that builds a collection of Regular Expressions that gradually describe bigger groups of pathways in the DFA's transition diagram. This intermediate step helps to extract Sequence Diagrams from the minimized DFA and ensures that they are trace equivalent.
- The resulting Regular Expression is transformed into a UML Sequence Diagram in the last step in Fig. 2 by performing its tokenization and applying a recursive algorithm to build the application logic.

3.1 Communication Traces Format

In this work, we propose a new format for representing communication traces between devices in the WoT. It is based on how messages can be abstracted as explained in the WoT architecture standard of W3C [18], which defines the basis of communication as the union of two main entities: a Thing and a Consumer. Our format uses application semantics and operations which makes it protocol-independent. We define the atomic element that composes each communication trace as an *interaction*, which is a message captured during communication between two objects in a WoT system as seen in Listing 1.1[4].

[4] Our open-source repository has further examples of the format as well as a schema for the validation of traces.

```
[{
  interactionId: 1, messagePairId: 1,
  recipient: {type: "thing",thingId: "1", thingTitle: "
    MyCoffeeMaker"},
  operation: "readproperty",
  affordance: {type: "property",name: "state"},
  timeStamp: "2023-12-04T03:04:16.01",
},{
  interactionId: 2, messagePairId: 1,
  recipient: {type: "controller"},
  payload: "ready",
  timeStamp: "2023-12-04T03:04:16.02",
}]
```

Listing 1.1: Interaction format example with a trace of a request-response pair.

3.2 DFA Creation

A common practice in specification mining techniques is to produce a state machine using a model-inference algorithm, whose input is a collection of traces recorded during the execution of the system. The algorithm generates a model, often a finite automaton, that accurately represents the behavior of the system that generated the trace log. The model is intended to adopt a formal language derived from the input traces which is constrained by their temporal and structural properties. In this work, our main goal is to preserve the temporal and structural features of the trace log and derive the exact behavior of the system's application logic for further verification of the system. To this end, we adapt the model inference algorithm specified in [1] whose pseudo-code is shown in Algorithm 1.

The algorithm starts by adding an initial state to the DFA object and one state per unique interaction type in the trace log. Then, a transition from the initial state to each first interaction of each trace is added.

Algorithm 1 Creation of a DFA from a trace log

```
1:  procedure generateDFA
2:      dfa ← add_state (init)
3:      for trace ∈ traceLog do
4:          for uniqueInteraction ∈ trace do
5:              dfa ← add_state (uniqueInteraction)
6:      for trace ∈ traceLog do
7:          dfa ← add_initial_transition (trace[0])
8:          for uniqueInteraction ∈ trace do
9:          if new transition then
10:             add_transition (transition)
11:     dfaReduced ← reduce_dfa (dfa)
```

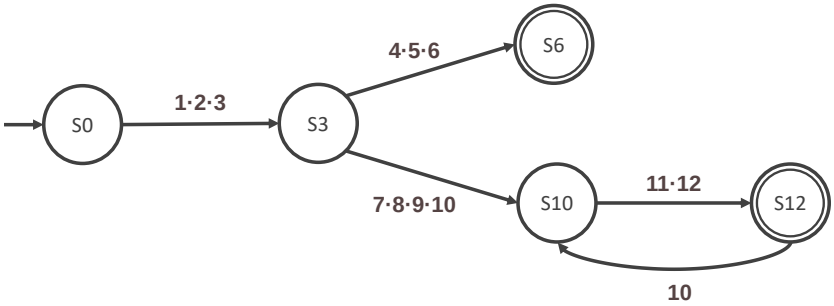


Fig. 3: Minimized DFA state transition diagram resulting from the traces.

After the initial DFA is created from the input traces, it is possible to minimize it according to one of the transition manipulation formulas defined in [21]. Applying this formula to a trace that conforms to the format in Listing 1.1, we get the diagram of the reduced DFA in Fig. 3, in which we have four states instead of twelve states if we were to not minimize it. The reduction of DFAs is especially useful in the next steps of the transformation into Sequence Diagrams.

3.3 Transformation into Regular Expressions

In this work, we use Regular Expressions as a declarative way to express Mashup application logic and to describe methods to obtain Sequence Diagrams from such Regular Expressions. The patterns of strings described by Regular Expressions show the same behavior as what can be described by finite automata, and therefore any formal language defined by any finite automata is also defined by a Regular Expression.

To construct a Regular Expression to define the language of any DFA we need to build a sequence of patterns or expressions that define a set of strings which in turn represent particular paths in the state transition diagram of the DFA. We adopt the Transitive Closure Method from [20], which is a recursive method that builds a collection of Regular Expressions that gradually describe bigger groups of pathways in the DFA's state transition diagram.

The main drawback of the Transitive Closure Method is that it tends to generate very long Regular Expressions compared to those generated by other methods. The simplification of DFA in the previous section was designed to reduce the length of the final Regular Expression and the time needed to iterate and reach the result (fewer states, fewer iterations). Nevertheless, the final Regular Expression must be further minimized before proceeding with the transformation process. The simplification is done by iteratively transforming the Regular Expression using a set of known algebraic equivalences in the Kleene algebra [11]. The final Regular Expression obtained using the Transitive Closure Method of the DFA from the state diagram in Fig. 3 is the following:

$$RE_{final} = 1 \cdot 2 \cdot 3\,(\,7 \cdot 8 \cdot 9 \cdot 10 \cdot 11 \cdot 12\,(\,10 \cdot 11 \cdot 12)^* + 4 \cdot 5 \cdot 6\,) \qquad (1)$$

3.4 Sequence Diagram Extraction

To derive Sequence Diagrams from the obtained DFA, we employed Regular Expressions as an intermediate step. As outlined in [9], Sequence Diagram application logic is described by elements defining specific behaviors like par, loop, alternative, etc. These interactions can be composed using UML operators resembling those in Regular Expressions. This mapping ensures equivalence between the Sequence Diagram, DFA, and input traces. The mapping from Regular Expressions to Sequence Diagram is defined as follows in [23], where i is an interaction in the trace log:

1. $(i_1 \cdot i_2) = (i_1 \ \textbf{\textit{seq}} \ i_2)$.
2. $(i_1 + i_2) = (i_1 \ \textbf{\textit{alt}} \ i_2)$.
3. $(i)^* = \textbf{\textit{loop}} \ (i)$.

The Sequence Diagram can be therefore defined as:

$$\begin{aligned} RE_{final} = 1 \ \textbf{\textit{seq}} \ 2 \ \textbf{\textit{seq}} \ 3 \ \textbf{\textit{seq}} \ (\ 7 \ \textbf{\textit{seq}} \ 8 \ \textbf{\textit{seq}} \ 9 \ \textbf{\textit{seq}} \ 10 \ \textbf{\textit{seq}} \ 11 \ \textbf{\textit{seq}} \ 12 \\ \textbf{\textit{seq loop}} \ (\ 10 \ \textbf{\textit{seq}} \ 11 \ \textbf{\textit{seq}} \ 12) \ \textbf{\textit{alt}} \ 4 \ \textbf{\textit{seq}} \ 5 \ \textbf{\textit{seq}} \ 6 \ ) \end{aligned} \tag{2}$$

The resulting expression with UML operators can in turn be transformed into a UML Sequence Diagram by performing its tokenization and applying the recursive executable procedure from Algorithm 2. Algorithm 2 converts a Regular Expression into an intermediate description called Mashup Logic before its conversion to Sequence Diagrams. In the context of this work, a Mashup Logic is a tree-like structure consisting of each element of the application logic represented in the Regular Expression. It describes the control flow of the system, including nested elements and all the features of each message and each operator, e.g. how often a loop is repeated.

The creation of the Mashup logic using Algorithm 2 is generalized in the following way:

- The procedure *generateMashupLogic* starts by searching the tokens that are included in the Regular Expression.
- If the algorithm sees a token, i.e. the operators *seq*, *alt* and *loop*, a new logic element is added to the Mashup logic structure.
- If the new logic element has logic content, e.g. it contains nested elements such as other operators, the procedure *generateMashupLogic* is called again and the parsing process starts over.

Algorithm 2 Algorithm that implements a Mashup's application logic.

```
1: procedure GENERATEMASHUPLOGIC
2:     for token ∈ regularExpression do
3:         logicElement ← generate_logic_element(token)
4:         if logicElement has logicContent then
5:             generateMashupLogic(logicContent)
```

– The procedure finishes when the string to parse does not contain any logic content and is only composed of sequential elements.

Finally, to obtain a UML Sequence Diagram, we use a specific subset of PlantUML[5] defined by the authors in [9]. The resulting Sequence Diagram is omitted here for brevity reasons but can be seen in our repository[6]. As explained in [9], it can be further processed to generate a SD or code for the controller.

3.5 Implementation

To facilitate evaluation and encourage reuse, we offer a publicly accessible implementation of our method. While it's built on Node.js and the reference implementation of the WoT Scripting API [10], the algorithms can be implemented in other programming languages. Each step we've outlined previously can be used as separate functions or together as an end-to-end solution.

4 Evaluation

To evaluate our contributions, we use a setup to showcase three case studies with different characteristics to mine. All mashups considered in the evaluation consist of physical or virtual Things that are exposed to the Web via an industrial gateway and a Mashup controller that is hosted on a conventional laptop. While this allows the right setup for an evaluation, a real-life setup would have the mashup logic running in the cloud for non-real-time applications or in an industrial PC or controller for real-time applications.

The scenario traces between them have been collected through the gateway, as shown in Fig. 4. Every Mashup is composed of Things from one or several WoT setups.

4.1 Evaluation Procedure

The evaluation aims to assess our method's accuracy in identifying and representing Mashup behavioral components as Sequence Diagrams and to verify our approach for system verification in the WoT. It's divided into two sub-objectives: approach evaluation and system verification evaluation.

Approach Evaluation In the first sub-objective, we evaluate the performance of our approach. We focus on determining whether our method can correctly and quickly mine the behavior exhibited in the scenario traces recorded during normal system execution. The steps performed in each case study are:

– Manual design and creation of the Mashup application logic in SD format.

[5] `https://plantuml.com/`

[6] `https://github.com/tum-esi/dyst-wot-miner/blob/main/paper-appendix/method-result-seqd.pdf`

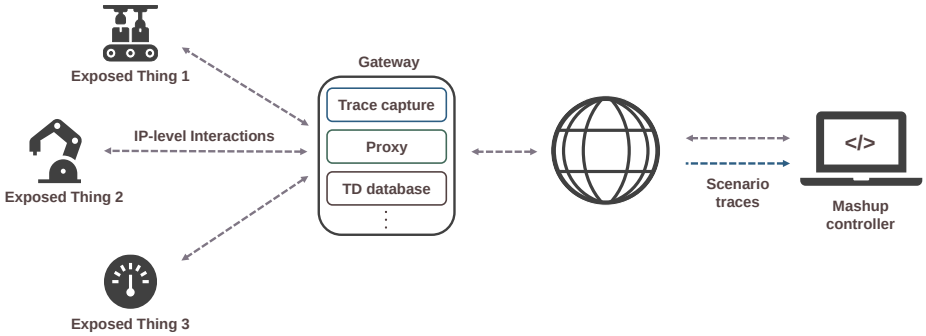


Fig. 4: Common device setup. The Things of a Mashup are proxied via a gateway that contains a trace capture to intercept and record messages exchanged between the Things and the Mashup controller.

- Automatic creation of Mashup controller code.
- Recording and formatting scenario traces.
- Automatic conversion from scenario traces to Sequence Diagram (approach from Chapter 3).
- Comparing the traces obtained with the mined Sequence Diagram to evaluate if our method has correctly mined the behavior recorded in the traces.

System Verification Evaluation The second sub-objective is to prove the effectiveness of our specification mining method for system verification. To do this, we evaluate the relationship between the actual behavior of the system and its designed specification to detect errors in the Mashup source code, design problems, or malfunctioning devices. After performing the previous steps, we proceed as follows:

- The previously obtained Sequence Diagram is further converted into SD.
- The designed SD is compared with the mined SD for system verification purposes.

4.2 Case Studies

We assess our method through three case studies with varying properties and levels of application logic complexity. The first, simpler case study involves inputs from external users, providing a large set of trace logs for testing our method across various correct and incorrect implementations. For the second and third case studies, we design, implement, and deploy them due to their inherent complexity. Though these cases yield fewer inputs, intentionally introduced bugs in the controller code create variation, allowing us to test the effectiveness of our method in mining faulty systems. More detailed descriptions and figures of each case study are available in our repository, but we summarize them here:

1. This Mashup is composed of one receive interaction and one send interaction. The controller reads the temperature of a Thing and then invokes an action to show the measured temperature in the device's display.

Case Study	Number of Devices	Number of recorded interactions	Number of Atomic Mashups	Number of Alternatives	Number of Loops
1	1	3	1	0	0
2	4	30	4	0	2
3	6	36	6	2	0

Table 1: Metrics characterizing the components of the different case studies.

Case Study	Number of Trace logs	Correctly mined DFA	Correctly mined SeqD	Correct implementation	Faulty implementation
1	19	19	8	8	11
2	4	4	4	1	3
3	4	4	4	1	3

Table 2: Metrics characterizing the evaluation results of the different case studies.

2. This system is characterized by two Things and by two loops. It starts with reading a property and invoking an action of a movable camera. After each loop, the property value is read again and displayed on the other Thing's display.
3. This system combines two sets of Things and aims to control the irrigation of a farm. Depending on the state of the sprinklers (ON or OFF) and the moisture of the soil, different actions are taken, including the subscription to an event that is triggered when the soil is too dry.

4.3 Evaluation Results

Table 1 provides an overview of each case study's characteristics, including the number of behavioral components and complexity. We conducted our evaluation using the setup and procedures detailed in previous sections, with results summarized in Table 2. The column labeled *Correctly mined DFA* indicates trace logs successfully processed, with their control application logic accurately mined and represented as a DFA. Conversely, the column *Correctly mined SeqD* denotes trace logs successfully transformed from DFA to Sequence Diagrams, evaluating the first sub-objective described in Section 4.1.

The system in case study 3 presents a more complex control flow with a total of eighteen unique interaction types and two alternatives, as summarized in Table 2. We found that it is possible to mine simple to complex control flows, including loops and alternatives, and to represent them with a Sequence Diagram as shown by the previous examples and the results in Table 2.

On the other hand, the columns *Correct behavior* and *Faulty behavior* from Table 2 evaluate the second sub-objective described in Section 4.1 and we classify a result as follows:

- **Correct Implementation**: Same application logic, affordances, and Things involved.
- **Faulty Implementation**: Missing or extra messages, wrong order of messages, unnecessary loops, wrong Thing, wrong affordance, wrong branching.
- **False Negative:** Correct implementation that exhibits the issues of a negative result due to traces not reflecting the full range of possible behaviors. Even though this case can be observed[7], they are excluded in the tables above since they cannot be considered to evaluate our method, but only the lack of traces during capture.

All evaluation resources are accessible[8] to the reader so that they can view, understand, or reproduce our evaluation results.

4.4 Timing Evaluation

To conclude, we assessed[9] our algorithms' performance by measuring the time for each transformation from scenario traces to Sequence Diagrams. We evaluated four cases per study: one with correct implementation and three with faulty implementation, showcasing various error types relevant for time analysis due to their impact on performance. The faults found in case study 1 and the faults we introduced in case studies 2 and 3 are explained below with their detailed descriptions available in our repository.

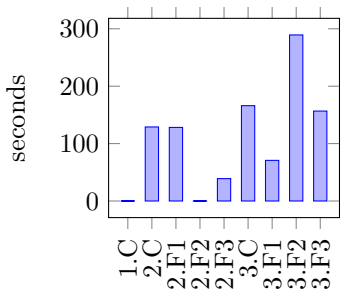
1. Case Study 1: Faulty 1 with extra loop (x6), Faulty 2 with Extra loop (x4), Faulty 3 with Extra loop (x2)
2. Case Study 2: Faulty 1 with Smaller loop count, Faulty 2 with No loop (x2), Faulty 3 with Missing interactions (x3)
3. Case Study 3: Faulty 1 with No *alt* (*Else*), Faulty 2 with Extra loop (x2), Faulty 3 with Wrong device

During our analysis, we observed an uneven distribution of time spent on each transformation. Fig. 5b illustrates this behavior, comparing a simple system (case study 1) with the most complex system analyzed (case study 3). Notably, the creation of Regular Expressions consumed the majority of time across all case studies, averaging 99.97%. This is attributed to the Transitive Closure Method (Section 3.3), where complexity in creating Regular Expressions is proportional to the number of states and control flow elements. Consequently, case studies 2 and 3 shown in Fig. 5b, being the most complex Mashups, exhibited extended processing times, as detailed in Table 1.

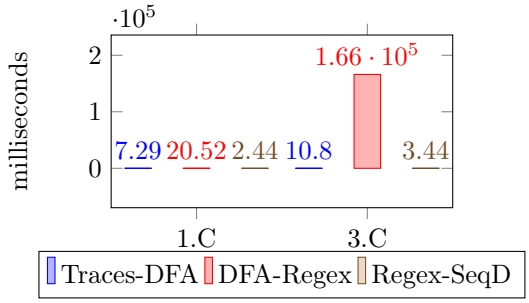
[7] An example can be found in our repository at `https://github.com/tum-esi/dyst-wot-miner/blob/main/paper-appendix/false-negative`

[8] Our repository contains the complete evaluation inputs, i.e. the communication traces of the systems, the generated Sequence Diagrams, System Descriptions, and the intervening output representations (DFA, Regular Expression and Mashup Logic).

[9] The hardware used for this evaluation is an Intel Core i7-6500U at 2.5GHz and 8GB of RAM.



(a) Total time needed to complete the mining procedure in each case study.



(b) Times taken to complete each transformation for case study 1 and 3 with correct behavior.

Case study 3 measurements seen in Fig. 5a show considerable differences for different complexities that occur when there are extra or missing items in the obtained traces:

- *Case Study 3 Faulty 2* is the slowest mining procedure (289.29s) of this analysis since it contains 3 loops, 2 branches, and 36 interactions.
- *Case Study 3 Faulty 1* contains fewer control flow elements (no loops and only 1 branch) which implies a reduction in the time needed by 75.53% (from 289.29s to 70.79s).

Overall, we can see that our method performs reasonably fast on an old laptop. Trivial Mashups can be evaluated as they happen whereas more complex ones need to be offloaded to another computer or to a cloud instance.

5 Conclusions and Future Work

In this work, we presented a novel method to mine specifications of WoT systems by reverse engineering Sequence Diagrams from communication traces obtained during the normal execution of the system, thus not requiring access to any source code. Through our evaluation process, we have proven the ability of our mining and conversion algorithms to quickly and accurately detect the application control flow of a system to allow comparison with the reference description. We envision our method to be used in real-life industrial applications that need accountability, traceability, and observability.

While further work can generalize our algorithm by integrating additional approaches based on neural networks, statistical methods and enhanced k-tail algorithms such as [4, 15, 17], our contributions establish the groundwork for further research in observability and dynamic specification mining for WoT to enable system verification.

References

1. Beschastnikh, I., Brun, Y., Abrahamson, J., Ernst, M.D., Krishnamurthy, A.: Unifying FSM-Inference Algorithms through Declarative Specification. In: Proc. of 35th ICSE. IEEE (2013)

2. Briand, L.C., Labiche, Y., Leduc, J.: Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. IEEE Transactions on Software Engineering **32**(9) (2006)
3. Cook, J.E., Wolf, A.L.: Automating Process Discovery through Event-Data Analysis. In: Proc. of 17th ICSE. IEEE (1995)
4. Cook, J.E., Wolf, A.L.: Discovering Models of Software Processes from Event-based Data. ACM TOSEM **7**(3) (1998)
5. Cook, S., Bock, C., Rivett, P., Rutt, T., Seidewitz, E., Selic, B., Tolbert, D.: Unified Modeling Language Version 2.5.1. Tech. rep., Object Management Group (2017)
6. Grati, H., Sahraoui, H., Poulin, P.: Extracting Sequence Diagrams from Execution Traces using Interactive Visualization. In: 17th Working Conference on Reverse Engineering. IEEE (2010)
7. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Acm Sigact News **32**(1) (2001)
8. Jiang, J., Koskinen, J., Ruokonen, A., Systa, T.: Constructing Usage Scenarios for API Redocumentation. In: Proc. of 15th IEEE ICPC. IEEE (2007)
9. Kast, A., Korkan, E., Käbisch, S., Steinhorst, S.: Web of Things System Description for Representation of Mashups. In: Proc. of COINS Conference. IEEE (2020)
10. Kis, Z., Aguzzi, C., Peintner, D., Hund, J., Nimura, K.: WoT Scripting API. W3C note, W3C (2023), https://www.w3.org/TR/2023/NOTE-wot-scripting-api-20231003/
11. Kleene, S.C.: Representation of Events in Nerve Nets and Finite Automata. Tech. rep., RAND PROJECT AIR FORCE SANTA MONICA CA (1951)
12. Korkan, E., Kaebisch, S., Steinhorst, S.: Streamlining IoT System Development With Open Standards. vol. 62. De Gruyter Oldenbourg (2020)
13. Koster, M., Korkan, E.: WoT Binding Templates. W3C note, W3C (Jan 2023), https://www.w3.org/TR/2023/NOTE-wot-binding-templates-20230928/
14. Kumar, S.: Specification Mining in Concurrent and Distributed Systems. In: Proc. of 33rd ICSE (2011)
15. Lo, D., Khoo, S.C.: QUARK: Empirical Assessment of Automaton-based Specification Miners. In: 13th Working Conference on Reverse Engineering. IEEE (2006)
16. Lo, D., Khoo, S.C., Han, J., Liu, C.: Mining Software Specifications: Methodologies and Applications. CRC Press (2011)
17. Lo, D., Mariani, L., Pezzè, M.: Automatic Steering of Behavioral Model Inference. In: Proc. of the 7th ESEC/FSE (2009)
18. Matsukura, R., McCool, M., Toumura, K., Lagally, M.: Wot architecture 1.1. W3C recommendation, W3C (Dec 2023), https://www.w3.org/TR/2023/REC-wot-architecture11-20231205/
19. McCool, M., Korkan, E., Käbisch, S.: WoT Thing Description 1.1. W3C recommendation, W3C (Dec 2023), https://www.w3.org/TR/2023/REC-wot-thing-description11-20231205/
20. Neumann, C.: Converting Deterministic Finite Automata to Regular Expressions (2005)
21. Nikiforova, O., Gusarovs, K., Ressin, A.: An approach to generation of the uml sequence diagram from the two-hemisphere model. ICSEA 2016 (2016)
22. Souder, T., Mancoridis, S., Salah, M.: Form: A Framework for Creating Views of Program Executions. In: Proc. of IEEE ICSM. IEEE (2001)
23. Ziadi, T., Da Silva, M.A.A., Hillah, L.M., Ziane, M.: A Fully Dynamic Approach to the Reverse Engineering of UML Sequence Diagrams. In: Proc. of 16th IEEE ICECCS. IEEE (2011)