future internet

MDPI

Article

A Finite State Automaton for Green Data Validation in a Real-World Smart Manufacturing Environment with Special Regard to Time-Outs and Overtaking

Simon Paasche [1,*] and Sven Groppe [2,*]

[1] Automotive Electronics, Robert Bosch Elektronik GmbH, John.-F.-Kennedy-Strasse 43-53, 38228 Salzgitter, Germany
[2] Institute of Information Systems, University of Lübeck, Ratzeburger Allee 160, 23562 Lübeck, Germany
* Correspondence: simon.paasche@de.bosch.com (S.P.); sven.groppe@uni-luebeck.de (S.G.)

Abstract: Since data are the gold of modern business, companies put a huge effort into collecting internal and external information, such as process, supply chain, or customer data. To leverage the full potential of gathered information, data have to be free of errors and corruptions. Thus, the impacts of data quality and data validation approaches become more and more relevant. At the same time, the impact of information and communication technologies has been increasing for several years. This leads to increasing energy consumption and the associated emission of climate-damaging gases such as carbon dioxide (CO_2). Since these gases cause serious problems (e.g., climate change) and lead to climate targets not being met, it is a major goal for companies to become climate neutral. Our work focuses on quality aspects in smart manufacturing lines and presents a finite automaton to validate an incoming stream of manufacturing data. Through this process, we aim to achieve a sustainable use of manufacturing resources. In the course of this work, we aim to investigate possibilities to implement data validation in resource-saving ways. Our automaton enables the detection of errors in a continuous data stream and reports discrepancies directly. By making inconsistencies visible and annotating affected data sets, we are able to increase the overall data quality. Further, we build up a fast feedback loop, allowing us to quickly intervene and remove sources of interference. Through this fast feedback, we expect a lower consumption of material resources on the one hand because we can intervene in case of error and optimize our processes. On the other hand, our automaton decreases the immaterial resources needed, such as the required energy consumption for data validation, due to more efficient validation steps. We achieve the more efficient validation steps by the already-mentioned automaton structure. Furthermore, we reduce the response time through additional recognition of overtaking data records. In addition, we implement an improved check for complex inconsistencies. Our experimental results show that we are able to significantly reduce memory usage and thus decrease the energy consumption for our data validation task.

Keywords: consistency checking; sustainable IT; green computing; big data streams; smart manufacturing

Citation: Paasche, S.; Groppe, S. A Finite State Automaton for Green Data Validation in a Real-World Smart Manufacturing Environment with Special Regard to Time-Outs and Overtaking. *Future Internet* **2023**, *15*, 349. https://doi.org/10.3390/fi15110349

Academic Editors: Michael Sheng and Paolo Bellavista

Received: 18 September 2023
Revised: 4 October 2023
Accepted: 21 October 2023
Published: 26 October 2023

1. Introduction

One aspect of industry 4.0 aims towards the interconnecting of devices and sensors (cf. Industrial Internet of Things, IIoT) to, e.g., increase efficiency by enhancing business models and identifying bottlenecks [1]. Therefore, data are a key factor. Using data-driven applications enables companies to optimize internal and external processes to decrease resource consumption or to gain competitive advantage [2]. However, to exploit the full potential, analyst require a high data quality which has to be ensured during acquisition and storing process-related information [2,3]. Simultaneously, information and communication technologies (ICT) lead to steadily increasing energy usage (between 1% and 3.2% of global consumption in 2020) and are prognosticated to represent up to 23% by 2030 [4]. In order

to reduce climate-damaging emissions and become climate-neutral, the IT landscape is therefore an important factor to consider.

In our work, we aim to combine the aspects of data quality in combination with green computing approaches to archive improvements in data pipelines for IoT environments. Since our work is in cooperation with Robert Bosch Elektronik GmbH, we mainly focus on smart manufacturing scenarios, but our results can be adapted to other areas like smart healthcare as well.

Figure 1 shows an example of a smart surface-mount technology (SMT) line. Such lines are used at Bosch among others to manufacture printed circuit boards (PCBs) for control units (e.g., for vehicles and e-bikes). SMT basically includes five processes: (1) solder paste printing (SPP) for printing solder paste on the circuit board, (2) solder paste inspection (SPI) to inspect the paste, (3) surface-mounted device (SMD) for placing components on the board, (4) reflow soldering (RFL), which is the actual soldering process, and (5) solder joint inspection (SJI) to inspect and test the final product. From a data point of view, SPI, SMD, and SJI provide the most important information; thus, the highest effort has to be on these processes.

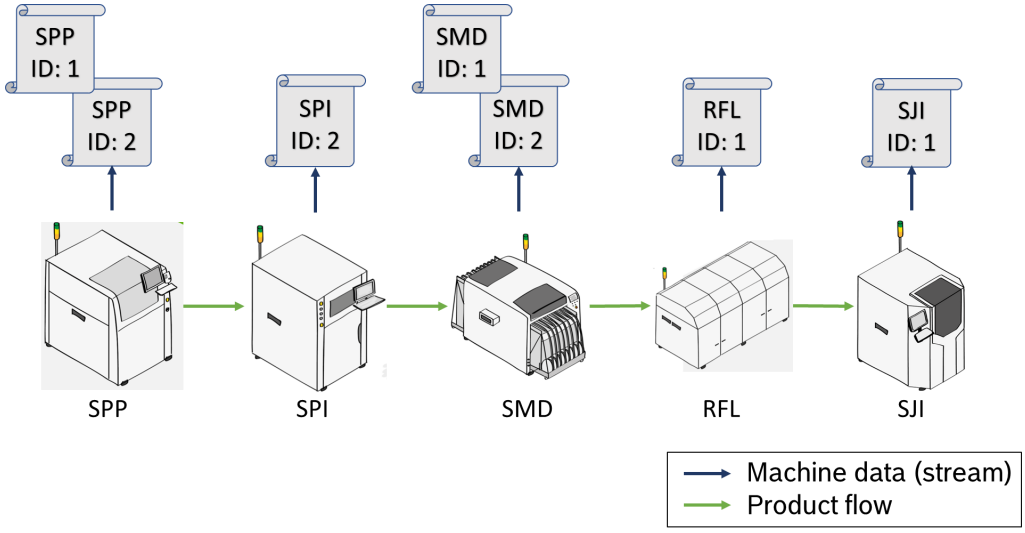

Figure 1. Run through a smart SMT line with data from SPP, SPI, SMD, and SJI. The machine pictures are provided by AE/MFT1 department.

During manufacturing, we noticed that in some cases, data and final product did not match. On a closer inspection, we observed that these mismatches are due to erroneous data, which are the result of a heterogeneous production landscape with various machines and software versions. We called these mismatches inconsistencies. Inconsistencies usually refer to a data set, which corresponds to all messages and information related to one manufactured product. To handle erroneous data, we categorized our inconsistencies into four main classes (cf. [5,6]):

1. Missing message. A missing message leads to incomplete data and thus, an information gap.
2. Multiple message. Multiple messages from one process may indicate problems with a machine.
3. Incorrect content. This category refers to the content of a single message.
4. With contradictions. By comparing the data of a data set, the matching has to be conflict-free.

The classification resulted from the study of our internal data. Categories 1 and 2 refer to entire messages that are either missing or duplicated (cf. Zhang et al. [7]). Since we validate complex JSON files, the content is of major interest in our use case.

Our examinations have shown that inconsistencies either concern the content of a single message (incorrect content) or constitute a mismatch between the information content of two or more messages (with contradictions). Although the categories (especially 3 and 4) evolved from a smart manufacturing use case, the manifestations of inconsistencies are similar in related IIoT and IoT domains. In smart healthcare, we can, for example, track the completeness of health records and identify discrepancies between two related parameters.

In order to ensure a high quality, in the past we presented a concept for a consistency checker (CC) [5]. Our CC enables the detection of known anomalies on a continuous data stream, making use of a domain ontology. According to [8], working on streams allows for fast feedback. In our scenario at Bosch, this domain ontology contains process flows and machine specifications. The CC utilizes this knowledge to map incoming data using SPARQL Protocol And RDF Query Language (SPARQL) (https://www.w3.org/TR/sparql11-query/, accessed on 14 September 2023) queries. These queries contain characteristics of relevant inconsistencies. We further adapted our CC to be more resource-efficient and energy-saving (GreenCC) [6]. The GreenCC consists of two units: (1) LightCC to predict the likelihood of inconsistencies and (2) FullCC to perform an accurate check using semantics. Experimental results showed that we can significantly reduce the energy consumption of data validation by adapting our architecture and optimizing the implementation. The GreenCC is currently running in an adapted version on real data. With the help of the CC, we were able to show that the inconsistencies affect between about 1% and 10% of the data, depending on the plant (larger plants are more likely to be affected). The small amount argues for an approach that is as efficient as possible to keep overhead to a minimum during monitoring. Although our previous CCs run stably on real data, we still encounter the following problems:

- We are able to detect known inconsistencies. If, for example, the characteristics of an inconsistency or the production environment change, these changes have to be entered manually. Similarly, new inconsistencies must be modeled accordingly in a machine-readable format. Nevertheless, the currently running version has its justification for existence, since newly modeled inconsistencies can be transferred to other plants and checked. Furthermore, the knowledge about possible causes can be transferred. Due to the heterogeneous landscape, it is not economical to prophylactically adjust all machines and process flows without knowing the problems.
- Inconsistencies in categories 1 and 2 can already be checked via an efficient procedure. For the other categories, it is so far only possible to identify periods in which they are more likely. A corresponding check must be carried out via transforming all gathered data from JSON into Resource Description Framework (RDF) (https://www.w3.org/RDF/, accessed on 14 September 2023).
- A final result is only available after the product has been manufactured. Depending on the production, this means that important time passes during which any errors cannot yet be corrected. Thus, in the worst case, many inconsistencies of the same kind are present.
- The longer the validation steps are, the longer large amounts of data are retained. This leads to increased memory and corresponding energy consumption of a CC.

We can transform these problems to four open challenges:

(C1) Detect inconsistencies of any kind and expression;
(C2) Increase efficiency of the checking process;
(C3) Shorten validation process;
(C4) Reduce the memory consumption during validation.

Challenge (C1) refers to quality aspects. To address this challenge, we propose an efficient matching using a message template in combination with an automaton structure. The machine specifies how the overall process to be checked is structured. The template is used to map the flow and steps of the sub-processes and thus to check complex inconsistencies. Deviations from the template are treated as anomalous.

Challenges (C2) to (C4) refer to sustainability aspects and complement each other. On the one hand, the explained template has the potential to check inconsistencies of Category 3 and 4 more efficiently (C2). On the other hand, our aforementioned automaton structure shortens the validation in case of detected inconsistencies (C3). Depending on the state, the automaton shows whether consistency is present or can still be achieved. Inconsistencies become directly visible in this way. We also take advantage of the fact that many production lines operate in a linear fashion. This means that the products pass through the machines one after the other and no overtaking is possible. If the data overtake each other, we can conclude that there is an error in the message transmission. Thus, we do not only consider the current data set as before, but include surrounding information and link further knowledge with it. In particular, the detection of overtaking data sets and drawing conclusions from an overtake in complex IoT environments are not addressed in existing studies. We want to achieve further sustainability benefits through clever internal handling of large amounts of data. In many IIoT scenarios, immense amounts of data of several gigabytes (GB) occur during, e.g., manufacturing. If these data volumes have to be held internally due to a long validation process, the storage requirements of a CC increase enormously. At this point, we also want to implement two optimizations: (1) perform validations as early as possible and release data that are no longer needed, and (2) store data smartly while they are in the checker. In this way, we address challenge (C4). Our present work includes an automaton that processes incoming messages, detects overtakes in the data stream, and handles time constraints. In this way, the automaton concept is useful for any IoT application where data quality is of high importance and at the same time limited computing resources are available or required in terms of energy consumption.

To present and explain our automaton, the remainder of this paper is as follows: Section 2 provides an overview of relevant work in the fields of data validation. We subdivide this into pure methods and efforts to use resource-saving technologies. Subsequently, we present our automaton concept in Section 3. In the course of this, we introduce an example automaton on which we visualize properties and algorithms. Section 4 shows possible applications based on two use cases. Afterwards, we evaluate our automaton with respect to the three identified challenges. We compare our new automaton with previous implementations in several experiments. The evaluation is followed by a discussion section. Finally, we conclude in Section 7.

2. Related Work

The literature provides a large body of work on the topic of data quality and validation. We first have a look at quality issues to work out how the use of the term relates to our view. Afterwards, we address the detection of inconsistent data in (I)IoT scenarios. We present semantic methods, complex event processing (CEP) systems, automatons, and machine learning methods. Furthermore, we investigate to what extent green computing has already found acceptance in the community. Additionally, our literature study includes related work with a focus on energy efficiency.

The work of Gao et al. [9] presents an overview of data quality and validation in big data environments. The authors point out that insufficient quality can lead to high costs for companies. In the literature, data quality often refers to aspects such as completeness, correctness, timeliness, accuracy, and availability (cf. [7,10–12]). The criteria *completeness* and *correctness* are similar to those in our work. Thus, we also try to detect incomplete data sets and aim to keep correct information in the data. With our categories 3 and 4, we also specifically address the fact that content correctness of data may depend on a single message or a set of messages. The arrival of the messages plays a minor role, especially in our manufacturing scenario, since we handle long delays via timers or detection of overtakes. Accuracy of the data is not a criterion, as we do not aim to create a trust score with our automaton but to detect and subsequently report inconsistencies. Karkouch et al. [10] also address *contextual anomalies*, which can only be detected with the addition of the

context. However, the focus of the work is on individual values and not on detection in complex documents.

The work of Haav et al. [13] refers to data validation in the timber industry. The authors use SHACL shapes to define constraints. The work describes their real example case in detail and gives descriptive examples. Nevertheless,the authors do not provide a concrete implementation of their approach. Furthermore, the proposal is not intended to process big data streams. Cortés et al. [14] introduce a practical data stream scenario from the medical sector. The authors explain data validation techniques to address IoT problems in healthcare, but mainly evaluate data throughput to identify challenges in the big data area. Concrete implementations and experiments are missing. In [3,15,16], further approaches are presented. In contrast to our work, the presented methods do either use fixed knowledge bases, work on static data instead of streams, or are not intended to be applied in real-world IoT environments. Furthermore, none of these validation approaches focus on green computing solutions.

The work of Maier et al. [17] and Hranislav et al. [18] present automatons for data validation. The authors refer to production plants and introduce time constraints. However, the authors do not mention detections of overtakes. This detection helps us to reduce the energy and resource consumption of the automaton.

In [19], the authors provide an overview of further anomaly detection approaches to detect outliers in time series. In general, the problem is very similar to our data validation. Time series, however, usually contain much simpler data sets. In our work, we present an approach that tests complex content issues.

Another possible solution to our data validation problems is complex event processing (CEP) systems. In preparation for this work, a literature search of existing CEP approaches has been performed. The literature search revealed that with Siddhi [20], Wihidum [21], and ETALIS [22], a large selection of CEPs exists. These systems allow the definition of simple constraints to detect patterns on a data stream. To do this, each of our consistency checks would have to be transformed into a pattern. For categories 1 and 2, this is possible. For inconsistencies of categories 3 and 4, however, the content is important. Since the content varies depending on the product, no uniform pattern is possible. Furthermore, the considered CEP systems do not offer different operating modes. We hope for an advantage in terms of energy consumption especially with the light mode.

Regarding energy efficiency, Ahmed et al. [23] present a blockchain-based approach with a focus on aggregation and protecting the IoT network. The system is used to validate edge servers in a hierarchical scenario. Our system is also suitable to run as a distributed application in large IoT scenarios. However, with our current application scenarios, we directly validate incoming data streams before data are stored. Furthermore, since we operate in protected environments and not in public networks, security and privacy play a subordinate role during validation. The work of Batmunkh et al. [24] provides an overview of carbon emissions emitted by social media platforms. The authors describe the effects of using these platforms, for example, by streaming videos and movies. The main energy consumption is due to the platform architectures consisting of large data centers. In our approach, we also consider carbon emissions due to the energy consumption of our system. However, our consistency checker does not consist of a large backend. We connect to an existing data pipeline to increase the usability of the collected data.

Other possibilities to increase the efficiency of IoT application are data prioritization techniques. According to Zahedina et al., data prioritization approaches focus on accessing correct data as quickly as possible. Therefore, IoT data are prioritized based on properties such as timeliness and significance to only use the most valuable data [25,26]. As a result, the approach of Zahedina et al. [25], for example, removes low-priority data from the cache. Sultana et al. [27] present an approach to implement patient monitoring in smart healthcare efficiently. Therefore, the authors classify patient data into critical and non-critical data packets to determine the urgency of the data. In general, we also aim to only store valuable data in our databases. In our scenarios, however, the focus is not on the real-time use

of data, but on monitoring processes and workflows. To achieve this, our automaton identifies deviations from the expected information. The detection and categorization of inconsistencies enable us to determine causes of errors to prevent them from recurring. How to deal with the detected anomalies is decided at a later stage (cf. [6]).

3. Concept of an Automaton for Data Validation

This section describes our concept to build automatons to handle stream data validation tasks in (I)IoT environments. The formal representation of our automaton is a tuple $A = (Q, \Sigma, \delta, Q_0, F)$ where:

- Q is a finite set, called the states of our automaton. In our case, an element of Q is a tuple $Q = (q, E)$ where q is the actual name of the state and E is a list of all current elements that are in the state. Each element of E is in turn a triple $e = (id, r(id), t)$, where id represents a unique identifier, $r(id)$ is the position of arrival, and t is a process-specific timer.
- Σ is the alphabet. A word of the alphabet consists of messages (m), timer expirations (ex), overtaking (o), and category 3 and 4 checks (cat_3 and cat_4).
- δ is a function $Q \times \Sigma \rightarrow Q$ (transition function).
- Q_0 represents a finite set ($Q_0 \subseteq Q$), called the initial states.
- F is a finite set with $F \subseteq Q$, which holds the accepting states.

We call our concept *AutomatonCC*. In the following, we give deeper insights into each element of the automaton and explain our concept step by step, building up an example automation. Therefore, we use the process described in Figure 2. Our process consists of the three steps S, M, and T, which are aligned linearly.

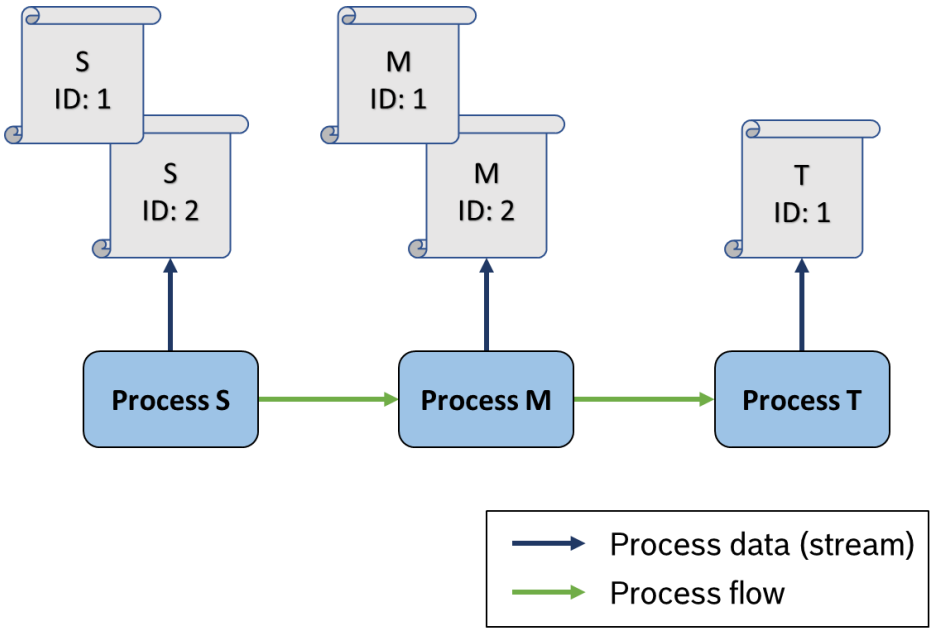

Figure 2. Example process to visualize states, transitions, and algorithms.

3.1. States

The first parameter we consider are the states Q. To create an automaton from the example process above, we determine the power set of the process steps: $\{\emptyset, \{S\}, \{M\}, \{T\}, \{SM\}, \{ST\}, \{MT\}, \{SMT\}\}$.

We use the elements of this set as states of our automaton. We ignore the empty set (see Figure 3). Sets containing only one element form the start states in each case. We characterize states that follow the process run as potentially consistent ($\{\{S\}, \{SM\}, \{SMT\}\}$). Potentially consistent means that it is ensured that the existing messages have arrived in the correct order, but it remains to be checked whether there are inconsistencies within the

messages (category 3 and 4). Furthermore, we add an inconsistent, a *cat4* state, in which a data set waits until it has been checked for category 4 inconsistencies, and a consistent state to our automaton. This gives us a total of ten states for our example process.

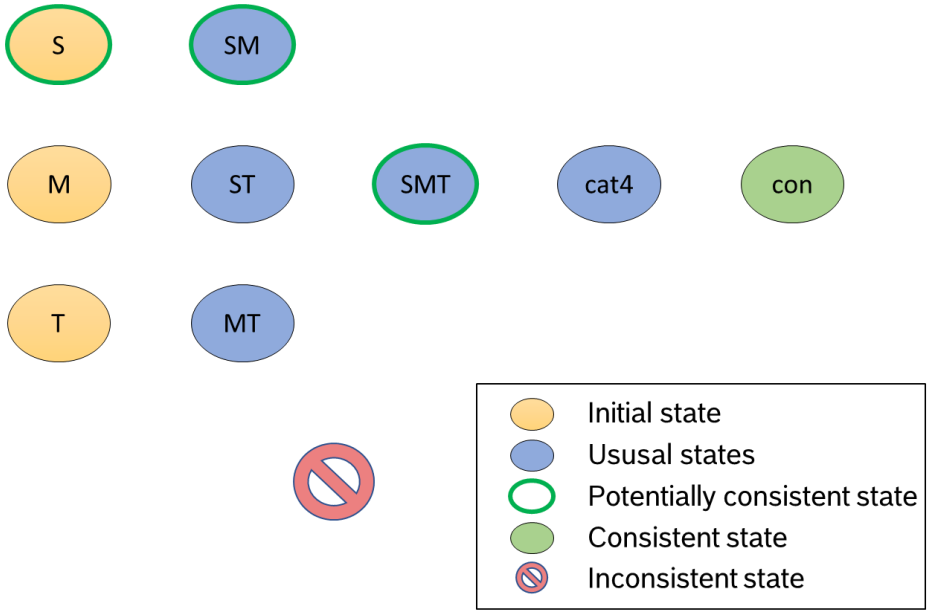

Figure 3. Resulting states from example process. To determine states, we refer to the powerset of the process steps.

As mentioned, we define a state as a tuple $Q = (q, E)$. With q, we denote the unique name of a state. E is a list consisting of all products, waiting in the corresponding state. An element of E is a triple $e = (id, r(id), t)$. The parameters of this triple are a unique product identifier (id) to be able to track each product and group related data, a rank ($rk(id)$), to detect overtakings, and a process-specific timer (t), to be able to terminate in the case of a missing message. Depending on the current manufacturing conditions, these timers can be extended or shortened.

3.2. Transitions

The second parameter is the transitions. In our automaton, state transitions are triggered by one of the following four types:

1. Incoming message. For each incoming message, an identifier switches its state.
2. Expiring timer. In case of an expiring timer, we terminate the run for the affected identifier.
3. Overtaking of a subsequent identifier. In case of an overtake, we terminate the run for all elements with a smaller rank than the actual identifier.
4. Content violation (Category 3 & 4). Content violations indicate an inconsistency in the considered data set.

Figure 4 provides an overview of all automaton transitions of our example process from Figure 2. As we develop a finite ω automaton, we have a transition function, mapping each possible input to exactly one output. Our function δ takes as input the current state $q \in Q$ and the trigger $\sigma \in \Sigma$ and maps to a new state from Q ($\delta = Q \times \Sigma \rightarrow Q$) Since our alphabet Σ consists of four different types of triggers, our σ is of the form of m, for an incoming message, ex, for an expired timer, o, for a detected overtake of a subsequent product, or cat_3/cat_4, which denotes a content violation in our gathered data.

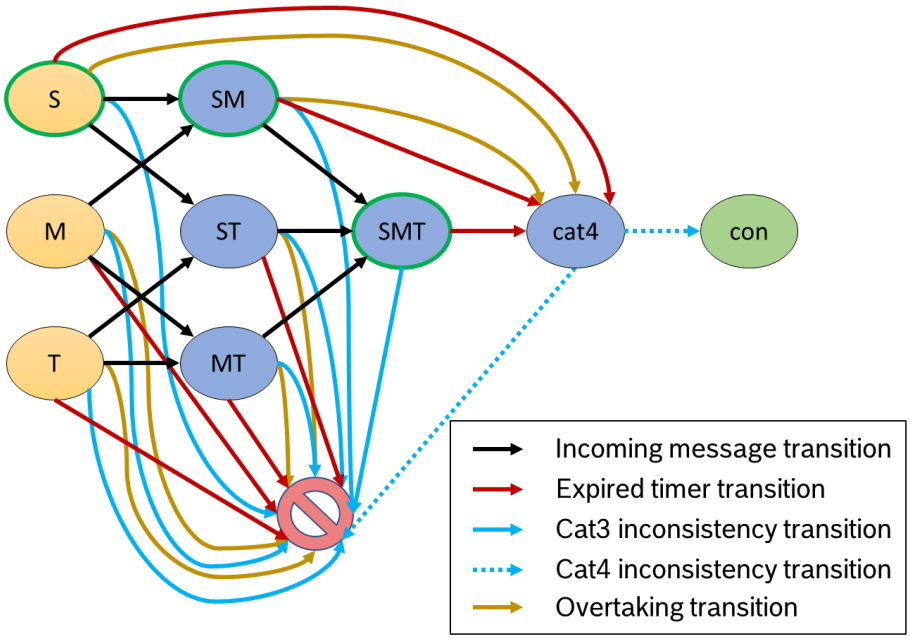

Figure 4. All transitions of our automaton.

3.2.1. Incoming Message

An incoming message can be assigned to a product. It thus leads to a data record being extended. For each message, we therefore check whether it contains new information or already exists. New information leads to a new state. If a piece of information is already present, we classify this as a *multiple message* inconsistency. The dataset under consideration is thus inconsistent and is moved to the *inconsistent state*. Further messages with the identifier of an inconsistent record do not change the state. Usually, all redundant messages lead to inconsistency.

Figure 5 shows the transition as an example for our S-M-T process. The regular pass in our example is the sequence S, SM, SMT. Since it is possible that messages are delayed, we can not build up a linear automaton but have various paths. For overview purposes, the state transitions to the *inconsistent state* have been omitted from the figure.

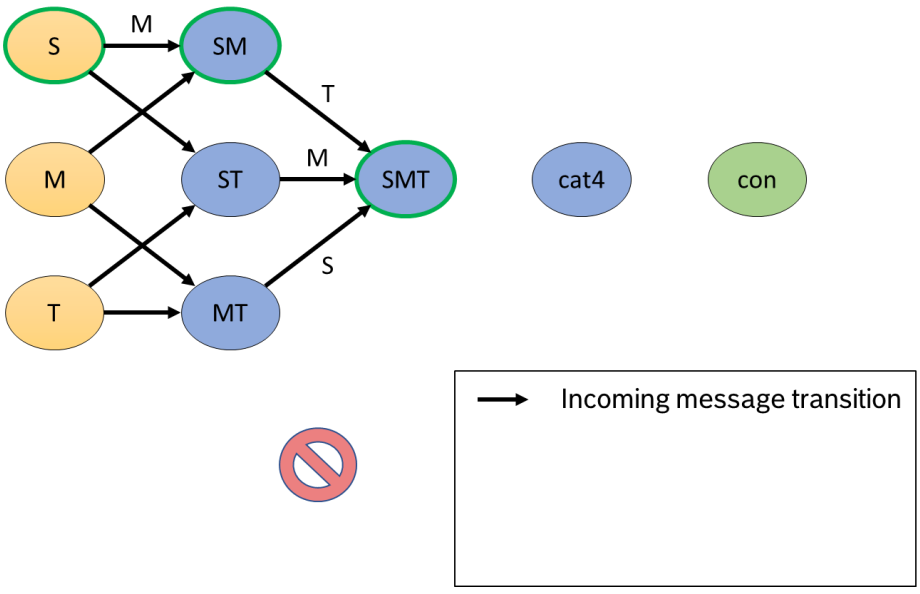

Figure 5. Transitions for incoming messages.

3.2.2. Expired Timer

If a timer expires, we terminate the run through the automaton for the affected identifier. A timer expiration indicates that we did not receive any information belonging to this data set for a longer period. To save resources and to prevent running out of memory, we stop tracking this product. If our product is in a *potentially consistent state* at this time, we transfer the identifier to the *cat4* state. Otherwise, the records will end up in the *inconsistent state* because not all expected information has arrived in the time limit (see example in Figure 6). Timers should be set accordingly, depending on the domain. Furthermore, we decided to automatically adjust the timers to the process flow at regular intervals, for example, to take into account traffic jams in a manufacturing line. Since in our manufacturing use case we sometimes receive operator messages regarding the final inspection, we also have a timer in the last process state (*SMT* in the example).

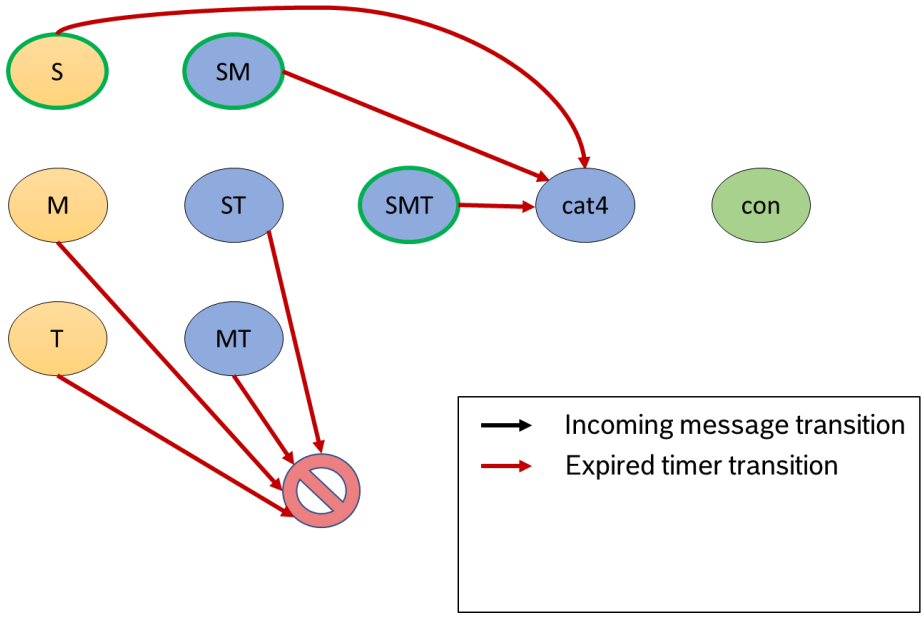

Figure 6. Transitions for expired timer.

3.2.3. Detected Cat_3 and Cat_4 Inconsistency

As in our previous validation approaches, we have to check for content violations in our data [5,6]. By validating categories 3 and 4, our automaton has a larger range of functions than our LightCC (cf. [6]), which only tests for the first two categories. In addition, we do not use semantic SPARQL queries for the validation step as before, but match the received data with a template. Expert knowledge is still involved via the template as well as through the actual transitions and states. In the automaton, this on the one hand eliminates the transformation step from JSON to RDF, which we expect to result in more efficient resource usage. On the other hand, unknown discrepancies can be detected as well. This possibility did not exist in the previous GreenCC (cf. [6]). Overall, the automaton thus offers an extended range of functions.

In our machine, we follow the approach of detecting inconsistencies as soon as possible. As a result, we check as soon as something is testable. Category 3 Inconsistencies refer to the content of a single message. The validation takes place after each incoming piece of machine information (see Figure 7). Violations result in the *inconsistent state*. If the test is successful, the data set remains in the current state. In the *cat4* state, validation for category 4 inconsistencies takes place. This step is only possible when all expected data are available, as discrepancies between messages are validated.

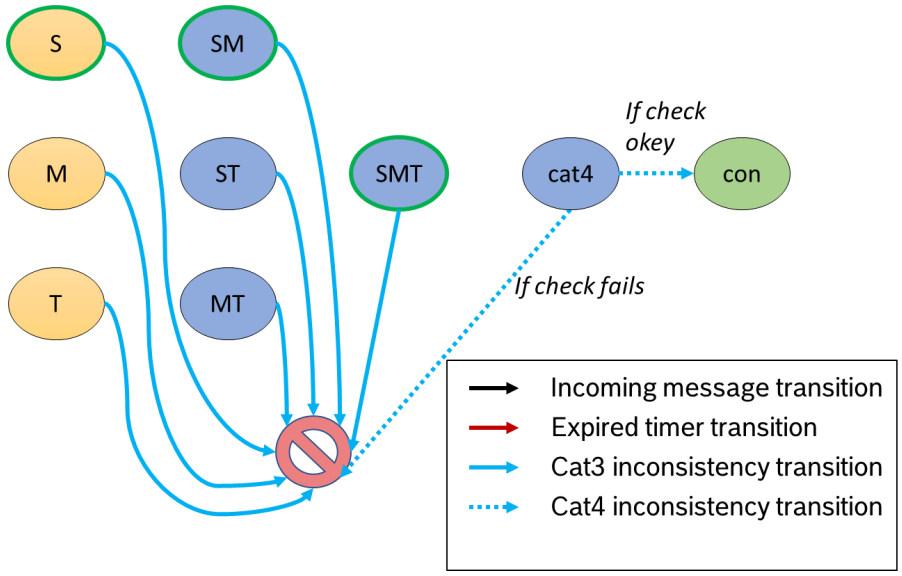

Figure 7. Transitions for complex inconsistencies (Category 3 and 4).

3.2.4. Detected Overtake

In our special case of a manufacturing scenario, it is often the case that the individual machines are arranged linearly. This means that it is not possible to overtake another product. If overtaken identifiers are in a normal state, an overtake leads to inconsistency (see Figure 8). In this case, we assume that messages have been lost and thus incomplete information is available (*missing message* inconsistency). In contrast, an overtake in a *potentially consistent state* does not automatically lead to inconsistency. In this scenario, the first assumption is that the overtaken identifiers were taken out of the manufacturing line prematurely. For this reason, the state transitions in Figure 8 of *S* and *SM* each lead to *cat4*. As described previously, category 4 is validated in this state. In the last regular state (in our example *SMT*), an overtake is no longer possible. As described in Section 3.2.2, in this state we wait only for the timer end.

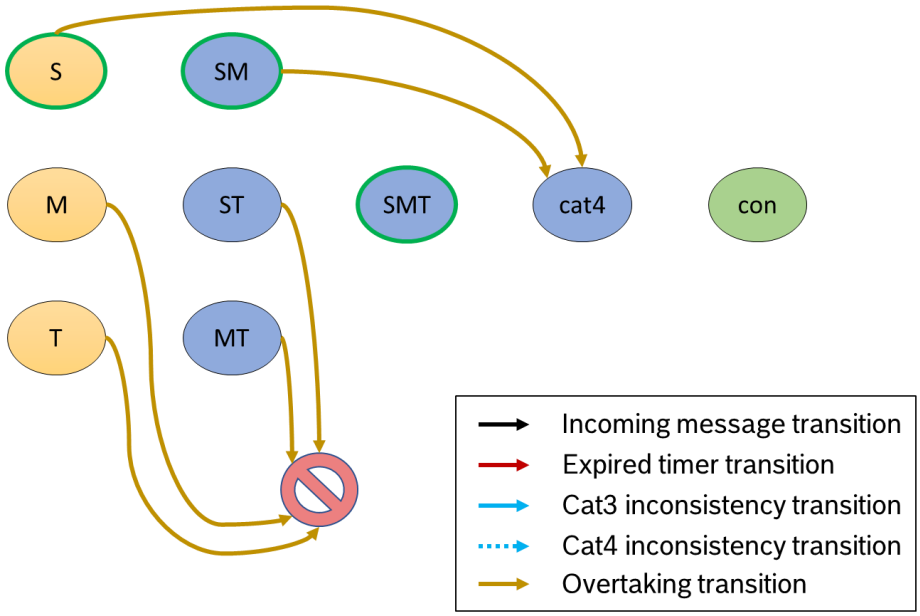

Figure 8. Transitions for overtaking messages.

3.3. Algorithms

To implement our automaton, we split up the functionality in three main algorithms running in parallel. These are: (1) *handleMessage(m)*, (2) *expiringTimer(t)*, and (3) *validateContent(m)*. In the following, we will go into more detail about these three algorithms and describe how they work in general.

3.3.1. Handle Incoming Messages and Detect Overtakings

The most important task is to handle incoming data from the IoT environment. Algorithm 1 provides an overview of the steps to perform. For incoming information, we first have to identify the data set it belongs to. If a corresponding data set does not yet exist and the identifier is thus unknown, we proceed as follows: Depending on the message type, we enter the associated initial state. The new identifier is then assigned with a rank and a process-specific timer is started.

Algorithm 1 Process incoming messages and detect overtakings

Require: $A \leftarrow new\ ValidationAutomaton()$
1: **procedure** $handleMessage(m)$ ▷ For each incoming message
2: $id \leftarrow m.getId()$
3: **if** $\neg A.find(id)$ **then** ▷ Start of a new manufacturing process
4: $q_{new} \leftarrow \delta(Null, m)$ ▷ Enter initial state
5: $A.set(q_{new}, id)$
6: $A.startTimer(id, m)$ ▷ Process-specific timer per product
7: $A.determineRank(q_{new})$ ▷ Compute rank considering time of arrival
8: **else**
9: $q \leftarrow A.getCurrentState(id)$
10: $r \leftarrow q.getRank(id)$
11: $A.clearTimer(id)$
12: **if** $r > 1$ **then** ▷ Overtake detected
13: $A.transferToCat4(\{[rank(all) < r] \sqcap [consistentState(id)]\})$
14: $report(\{[rank(all) < r] \sqcap [\neg consistentState(id)]\})$
15: **end if**
16: $q_{new} \leftarrow \delta(q, m)$
17: $A.set(q_{new}, id)$
18: **if** q_{new} *is con* **then** ▷ Reached final consistent state
19: $validateDataSet(M)$ ▷ M contains all messages related to the current id
20: **else if** q_{new} *is valid* **then**
21: $A.startTimer(id, m)$ ▷ Start timer considering next process
22: $A.determineRank(q_{new})$ ▷ Compute new rank
23: **else**
24: $report(id)$ ▷ When entering invalid state, report inconsistency
25: **end if**
26: **end if**
27: **end procedure**

Otherwise, we clear the active timer to prevent an expiring timer from triggering unwanted actions. Afterwards, we check the rank of our identified set. If $rank > 1$, we detect an overtake, because the state contains older data sets. An overtake is not problematic for the current data set, but is for older ones. If the overtake happens in a *potentially consistent state*, we can transfer all older elements into the *cat4* state. This is possible because we assume that the identifier has been removed from the IoT scenario. An overtake in a usual state means that older data sets are not currently consistent. We conclude that information was lost. The affected data sets are classified as inconsistent according to the *missing message* criterion.

Thereafter, we can switch the state and proceed. First, the algorithm checks in what type of state the identifier is. In the *cat4* state, the entire data set is checked for category

4 inconsistencies. In a valid state, a new process-dependent timer is started and the new rank of the identifier is determined. As we already mentioned in Section 3.1, each process has its own time restrictions. The rank is derived from the timestamp of a message. The recalculation of the rank is necessary in order to further identify overtaking items.

If we are in an inconsistent state, there is no longer the possibility to receive an anomalous-free data set. As a result, we can report an identified inconsistency directly.

3.3.2. Handle Expiring Timer

Algorithm 2 lists the steps for the *expiringTimer(t)* procedure. In case of an expired timer, we check if the affected identifier currently is a *potentially consistent state*. If so, we assume that we received all expected messages and transfer the identifer to the last check (*cat4*). Otherwise, we report a *missing message* inconsistency.

Algorithm 2 Handle expiring timer

Require: $A \leftarrow new\ ValidationAutomaton()$
1: **procedure** $expiringTimer(t)$
2: $id \leftarrow t.getId()$
3: $q \leftarrow A.getCurrentState(id)$
4: **if** q *is potentiallyConsistent* **then**
5: $A.transferToCat4(id)$ ▷ Check for Category 4
6: **else**
7: $report(id)$
8: **end if**
9: **end procedure**

3.3.3. Inconsistencies of Category 3

Our last algorithm shows how to treat inconsistencies of category 3 (Algorithm 3). As already mentioned, we validate each incoming machine message directly to report discrepancies as early as possible. In contrast to earlier approaches, we check for these inconsistencies by matching a message with a consistent template. If the match is above a defined threshold, the message is consistent. Otherwise, an error will be reported.

Algorithm 3 Handle complex Inconsistencies

1: **procedure** $validateContent(m)$
2: $type \leftarrow m.getMessageType$
3: $template \leftarrow loadTemplate(type)$
4: $content \leftarrow m.getContent()$
5: **if** $\neg match(content, template)$ **then**
6: $report(id, m)$
7: **end if**
8: **end procedure**

3.4. Example Runs

To visualize our algorithms, we use three example data sets and present the runs through our automaton in Figures 9–11. The graphics each represent a time-dependent snapshot of the automaton.

Figure 9 shows the automaton in case of clean data. As one can see in the Figure, our automaton runs step by step through the *potentially consistent* states, starting with state *S* (timestep $t = 0$). In each step, a check for category 3 inconsistencies is performed. After the last message has arrived, the data set remains in state *SMT* until the last timer expires (timesteps $t = 3$ and $t = 4$). The automaton switches into state *cat4*, where the whole data set is checked for category 4. As the data set is consistent, the automaton switches into the final state *con*. For the last two states, we do neither need to start a new timer nor calculate the new rank.

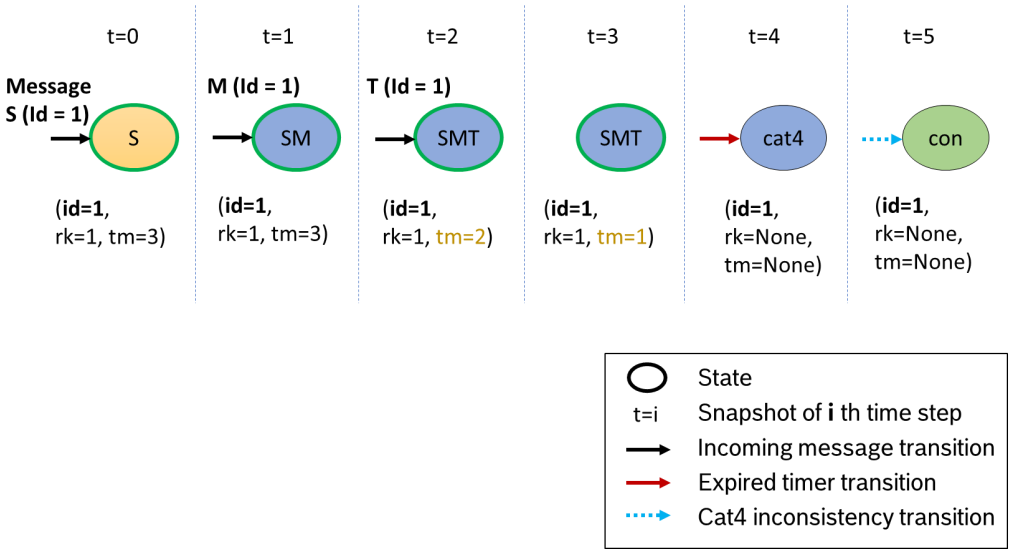

Figure 9. Example run for a consistent data set with messages arriving in order.

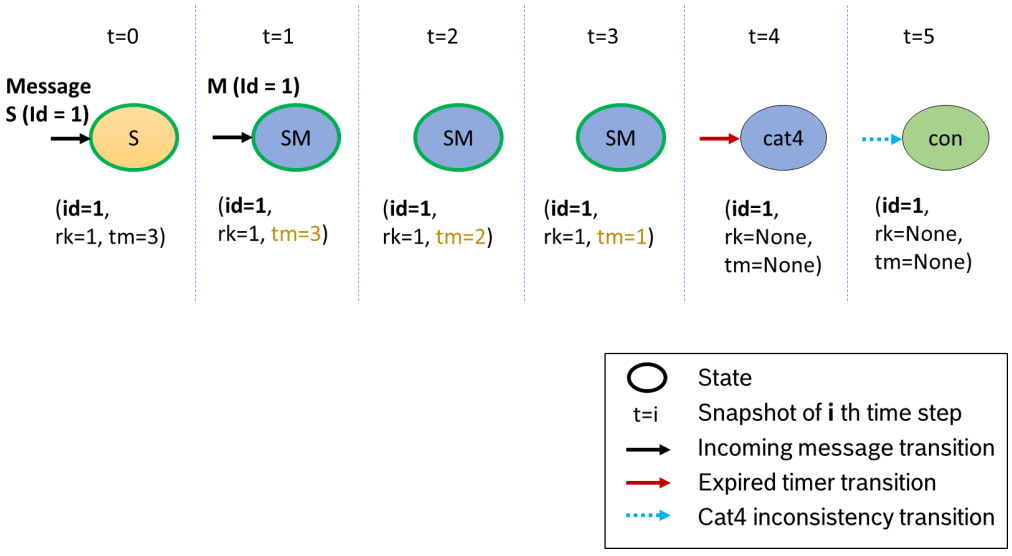

Figure 10. Example run for an expired timer.

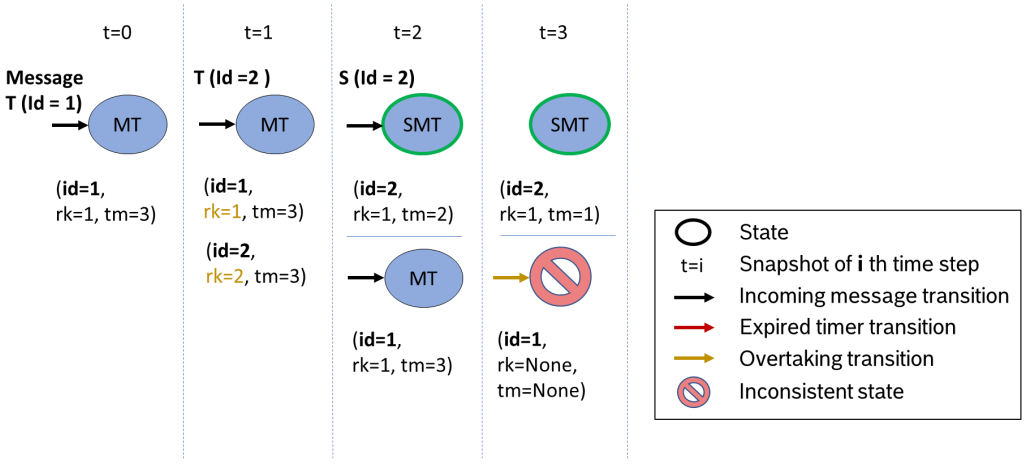

Figure 11. Example run for a detected overtake.

Figure 10 illustrates the run of a data set with a potentially missing message. Again, we start in state *S*, then change into *SM*. At this point, no more messages arrive. When the timer expires (time step $t = 4$), we switch into state *cat4*. Since *SM* is a *potentially consistent* state and we did not yet recognize any inconsistent content, we assume that the process has been stopped after step M. Our data are thus consistent and can be processed further.

Figure 11 exemplifies a detected overtake in $t = 3$. The data set with $id = 1$ has rank $r = 1$ in state *MT*. Data with $id = 2$ enter state *MT* and receive the rank $r = 2$. In $t = 5$, data with $id = 2$ switch into state *SMT*. At this point, our automaton detects an overtake due to the higher rank of $r = 2$. According to our algorithm, each data set with smaller rank has to leave the current state. Since the overtake takes place in a normal state, the overtaken data set ($id = 1$) is moved into the trash state.

4. Use Cases for the Automaton

In the following, we look at how to map different (I)IoT scenarios to our automaton. First, we consider a smart manufacturing environment. We will refer to such an environment in our evaluation (Section 5). Furthermore, we describe a smart healthcare scenario. Although these two domains are far apart, the underlying IoT structure has many similarities. This also results in similar problems. Inconsistencies within the collected data represent one of these problems. By comparing both scenarios, we want to show that our automaton is suitable for stream validation in both domains.

4.1. Smart Manufacturing

In smart manufacturing, machines are equipped with additional sensors and programs to monitor and track the production processes remotely. In our production lines at Bosch (see Figure 1), the individual machines document the tasks performed in each case. Using this documentation, process optimization can take place afterwards and error rates can be minimized. Each individual machine can be regarded as an IoT device or a composition of IoT devices.

In the first step, we map the machines to states in our automaton. According to the rules from Section 3.1, we create the power set of the set of machines and add a *cat4*, a final consistent, and an inconsistent state to it. Assuming that each process of Figure 1 is implemented by one machine, the total number of states is 34 ($2^5 - 1 + 3$, since we ignore the empty set). In our scenario, there is also the special case that different products are manufactured on the lines. So, it can happen that even if there are several SMD placement machines, not all of them perform a production step. For this reason, we cannot assume the number of machines for the SMD states but must check completeness of the data at the end by including additional domain knowledge. On our automaton, this has the effect that there is only one state for SMD.

Afterwards, we determine the set of potentially consistent states. In our case, this is the path corresponding to the regular process flow. If all messages arrive in the correct order, there are complete data on this path up to the respective state.

In the next step, we create the transitions. The *message* transitions can be derived from the process flow. Since we classify our states among others into *usual* and *potentially consistent* ones, the transitions for *timer expired*, *overtaking*, and $cat_3 + cat_4$ remain the same as in the example above (see Section 3.2). The records are grouped based on a unique product identifier. In real operations, we use one automaton per manufacturing line to simplify the detection of overtakes.

4.2. Smart Healthcare

In smart healthcare, patient-related measurement data, such as blood pressure measurements, electrocardiogram (ECG) data, or X-ray data are recorded by smart devices and stored for later assessment. The measurements are taken according to a schedule so that they take place at defined intervals. The schedule created by the intervals can be seen as a process. In this way, a process can be defined for each patient or a group of patients.

Patient identifiers can be used to identify related data. Our automaton can detect missing messages, duplicates, and contextual discrepancies in these processes. In the medical field, many medical treatments are customized, resulting in a process that is structured according to the type of treatment or specialty. As increasingly large amounts of data, such as the aforementioned ECG or X-ray pictures, have to be sent and managed, data retention during validation also plays a role. Appropriate timers as well as the detection of outdated data can help to reduce memory consumption. The general workflow is thus comparable to the manufacturing scenario above. Transitions for an automaton are again derived from the process flow. The timer lengths must initially be adapted to the environment.

4.3. Smart Parking Scenario

Furthermore, scenarios such as those described with the IoT simulator of Warnke et al. (cf. Github repository (https://github.com/luposdate3000/SIMORA, accessed on 14 September 2023) and [28]) can be tested. Among other things, the authors describe a smart parking scenario. Parking also creates a kind of process sequence in which data are expected from the various sensors. If obstacles are detected, the involved sensors start sending distance values. This behavior should remain unless the vehicle moves away from obstacle or an obstacle (e.g., a person) moves. Data are timestamped so that multiple and missing information can be determined. Complex inconsistencies can then arise, for example, by including driving direction and steering movements.

4.4. Universal IoT Scenario

An IoT scenario is characterized by information being collected and tracked via diverse distributed sensors, actuators, and embedded systems [29,30]. Using our concept, we can map an arbitrary (I)IoT scenario to create an automaton for data validation. The procedure in any smart scenario looks like this:

1. Determine the IoT Devices.
2. Form the power set of the determined devices and add a *cat4*, and a final and an inconsistent state.
3. Add *incoming message* transitions according to the process flow.
4. Take over the remaining transitions and define corresponding timers.

For each scenario, specify what happens to inconsistent records. In [31], the authors present an approach to handle corrupt data. Depending on the degree of damage, the authors propose to either discard or clean.

5. Evaluation

For our experimental results, we refer to an automaton in SMT environments. Our automaton and our reference programs are implemented in Python. The work of Pereira et al. [32] shows that Python in general has a larger overhead in contrast to, e.g., C language. However, in our current experiments, the focus is on evaluating different techniques for validating data streams. Future work may investigate the impact of the programming language on the overhead.

We use the paho mqtt package (https://pypi.org/project/paho-mqtt/, accessed on 14 September 2023) to connect our system to real manufacturing data. In case of an ideal scenario, we load a previously generated data set. As an automaton is developed to validate one line at the same time, in our evaluation we also limited the connection to one manufacturing line (important in the last evaluation). Furthermore, each of the approaches only listens to the specific line topics to leverage the capabilities of message brokers. By doing so, we reduce the internal data traffic and move this task to the broker management.

5.1. Experimental Results

The evaluations are performed on a computer with Intel i5-1145G7 processor and 16 GB RAM. For calculating the climate footprint in carbon-dioxide equivalents (CO_2e), we use the Python framework *CodeCarbon* (https://codecarbon.io/, accessed on 14 September

2023). We record the memory usage with *tracemalloc* (https://docs.python.org/3/library/tracemalloc.html, accessed on 14 September 2023). To determine the CPU load, we use *psutil* (https://github.com/giampaolo/psutil, accessed on 14 September 2023).

The focus of our experiments is on comparing previous approaches with our newly developed automaton. Previous consistency checkers are only able to detect already known inconsistencies on a data stream. By using template matching to check categories 3 and 4, our automaton is at least as powerful in detection as semantic methods. The main measurable difference between our systems is the direct energy consumption. In the first evaluation, we examine at the energy consumption of our approaches developed so far. The evaluation builds on the results from [6]. We then compare the memory consumption for validating complex inconsistencies (category 3 and 4) of our automaton with the previously developed FullCC. Both evaluations take place in an ideal scenario. Possible waiting times and thus additional energy consumption for data storage over a longer period are not taken into account. For this reason, we compare the memory and CPU utilization of the automaton in a real scenario with the aforementioned GreenCC in a third evaluation.

5.1.1. Energy Consumption in Ideal Environment

The first evaluation compares the total energy consumption when applying our systems in ideal manufacturing plants of different sizes (small, medium, and large). For small plants, we consider 400 k machine messages each day, in medium plants about 1.5 Mio, and in large plants around 2.5 Mio machine messages per day (cf. [6]).

In the evaluation, we compared the modules of our GreenCC (LightCC and FullCC) with Apache Flink (https://flink.apache.org/, accessed on 14 September 2023) and our newly developed automaton. For this we kept the concepts of GreenCC and used the Flink framework for stream handling. Table 1 presents consumption in kilowatt-hours (kWh) and costs in cent. Table 2 shows our evaluation results of the climate footprint in grams of carbon-dioxide equivalents (gCO_2e) . We have already presented part of the evaluation in [6], therefore, we will not go into detail on each point. The main results from [6] are that energy and costs can be saved by using more efficient processes. In a medium-sized plant, energy consumption can be reduced by a factor of more than 0.3 in this way. (cf. Flink (all) vs. FullCC (all)). This reduction does not affect the accuracy of the detection. The reduced consumption also lowers operating costs and the climate footprint. (medium-size plant in EU about 676 gCO_2e less). In the past evaluation, we also observed that the deciding factor between the approaches is not due to the semantic transformation and validation, but to the overhead that a framework like Apache Flink brings with it.

The consumption values of our automatons are highlighted in blue. The results show that the automaton even outperforms the FullCC with a larger range of functions. The difference in a medium-sized plant is 1.116 kWh, which corresponds to a factor of 0.24. For the climate footprint, this difference results in a reduction of 293 gCO_2e (assuming an EU average of 262 gCO_2e/kWh).

Overall, the first evaluation shows that the automaton structure has a positive influence on the computing resources required. One main reason for this is the adapted internal data management through which data are validated as quickly as possible in order to forget parts that are no longer needed. In our next analysis, we will look at the memory consumption during the analysis of a data set.

Table 1. Energy consumption in kilowatt-hours (kWh) and operating costs for small, medium, and large manufacturing plant on a daily basis.

Approach	Small Plant per Day (Costs\Day)	Medium Plant per Day (Costs\Day)	Large Plant per Day (Costs\Day)
Flink (1&2)	1.898 kWh (24.04 Cent)	7.116 kWh (90.16 Cent)	11.860 kWh (150.27 Cent)

Table 1. *Cont.*

Approach	Small Plant per Day (Costs\Day)	Medium Plant per Day (Costs\Day)	Large Plant per Day (Costs\Day)
Flink (all)	1.949 kWh (24.69 Cent)	7.308 kWh (92.59 Cent)	12.180 kWh (154.32 Cent)
Flink heuristic	1.856 kWh (23.52 Cent)	6.960 kWh (88.18 Cent)	11.600 kWh (146.97)
SPARQL	0.090 kWh + e_s (1.14 Cent) + c_s	0.336 kWh + e_m (4.26 Cent) + c_m	0.560 kWh + e_l (7.10 Cent) + c_l
LightCC	1.226 kWh (15.53 Cent)	4.596 kWh (58.23 Cent)	7.660 kWh (97.05 Cent)
LightCC with Change Detection	1.229 kWh (15.57 Cent)	4.608 kWh (58.38 Cent)	7.680 kWh (97.31 Cent)
FullCC (1&2)	1.251 kWh (15.85 Cent)	4.692 kWh (59.45 Cent)	7.820 kWh (99.08 Cent)
FullCC (all)	1.261 kWh (15.97 Cent)	4.728 kWh (59.90 Cent)	7.880 kWh (99.84 Cent)
Automaton	0.963 kWh (12.20 Cent)	3.612 kWh (45.76 Cent)	6.020 kWh (76.27 Cent)

Legend: e_s: ~0.049 kWh RDF transform in small plant. e_m: ~0.183 kWh RDF transform in medium plant. e_l: ~0.305 kWh RDF transform in large plant. c_s: ~0.61 Cent RDF transform in small plant. c_m: ~2.32 Cent RDF transform in medium plant. c_l: ~3.86 Cent RDF transform in large plant.

Table 2. Carbon-dioxide equivalents (CO_2e) in gram per kWh for daily operation in small, medium, and large plants.

Approach	*Plant Size*	**Germany 366 gCO_2e/kWh**	**EU 262 gCO_2e/kWh**	**USA 379 gCO_2e/kWh**	**World 441 gCO_2e/kWh**
Flink (1&2)	small:	695 g	497 g	719 g	837 g
	medium:	2604 g	1864 g	2697 g	3138 g
	large:	4341 g	3107 g	4495 g	5230 g
Flink heuristic	small:	679 g	486 g	703 g	818 g
	medium:	2547 g	1824 g	2638 g	3069 g
	large:	4246 g	3039 g	4396 g	5116 g
Flink (all)	small:	713 g	511 g	739 g	859 g
	medium:	2675 g	1915 g	2770 g	3223 g
	large:	4458 g	3191 g	4616 g	5371 g
SPARQL	small:	33 g + ge_s	23 g + eu_s	34 g + us_s	40 g + w_s
	medium:	123 g + ge_m	88 g + eu_m	127 g + us_m	148 g + w_m
	large:	205 g + ge_l	147 g + eu_l	212 g + us_l	247 g + w_l
LightCC	small:	449 g	321 g	465 g	540 g
	medium:	1682 g	1204 g	1742 g	2027 g
	large:	2804 g	2007 g	2903 g	3378 g
LightCC with Change Detection	small:	450 g	322 g	466 g	542 g
	medium:	1687 g	1207 g	1746 g	2032 g
	large:	2811 g	2012 g	2911 g	3387 g
FullCC (1&2)	small:	458 g	328 g	474 g	552 g
	medium:	1717 g	1229 g	1778 g	2069 g
	large:	2826 g	2049 g	2964 g	3449 g

Table 2. *Cont.*

Approach	*Plant Size*	**Germany 366 gCO_2e/kWh**	**EU 262 gCO_2e/kWh**	**USA 379 gCO_2e/kWh**	**World 441 gCO_2e/kWh**
FullCC (all)	small:	461 g	330 g	478 g	556 g
	medium:	1730 g	1239 g	1792 g	2085 g
	large:	2884 g	2065 g	2987 g	3475 g
Automaton	small:	353 g	252 g	365 g	425 g
	medium:	1322 g	946 g	1369 g	1593 g
	large:	2203 g	1577 g	2282 g	2655 g

Legend: ge_x: Additional CO_2e in plant of size **s**mall (~17.9 g), **m**edium (~67.0 g), **l**arge (~111.6 g) in Germany. eu_x: Additional CO_2e in plant of size **s**mall (~12.8 g), **m**edium (~47.9 g), **l**arge (~79.9 g) in EU. us_x: Additional CO_2e in plant of size **s**mall (~18.6 g), **m**edium (~69.4 g), **l**arge (~115.6 g) in USA. w_x: Additional CO_2e in plant of size **s**mall (~21.6 g), **m**edium (~80.7 g), **l**arge (~134.5 g) worldwide.

5.1.2. Validating Complex Inconsistencies

We have seen in the previous evaluation that our Automaton has lower energy consumption in an ideal environment compared to previous approaches. This results in lower operating costs as well as an improved climate footprint. In our next experiment, we want to look at the causes of the reduced consumption. To do this, we look at the memory consumption of our FullCC (best full verification approach) for checking complex inconsistencies with that of the automaton. We focus our evaluation on inconsistencies of categories 3 and 4, because for validation of these content requirements, large amounts of data have to be stored and, if necessary, transformed in the respective approaches. We have seen in the previous evaluation that our Automaton has lower energy consumption in an ideal environment compared to previous approaches. This results in lower operating costs as well as an improved climate footprint. In our next experiment, we want to look at the causes of the reduced consumption. To do this, we look at the memory consumption of our FullCC (best full verification approach) for checking complex inconsistencies with that of the automaton. We focus our evaluation on inconsistencies of categories 3 and 4, because for validation of these content requirements, large amounts of data have to be stored and, if necessary, transformed in the respective approaches. This is not required for categories 1 and 2, resulting in a non-significant additional memory usage. Figure 12 depicts our results.

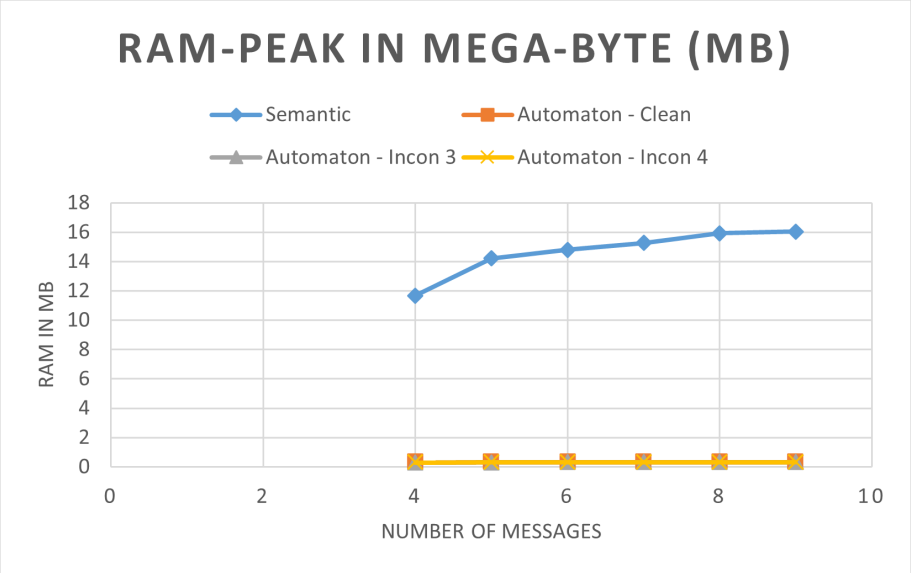

Figure 12. Memory usage during validation of category 3 and 4 inconsistencies.

The table shows the RAM usage in mega-bytes (MB) in three scenarios with increasing data set size: (1) No inconsistencies, (2) Category 3 inconsistency, and (3) Category 4 inconsistency. As can be seen in the table, the memory consumption of the automated runs is very close to each other. The maximum value here is 0.33 MB for no inconsistency and nine messages. This value differs by almost 0.01 MB compared to four messages. The result shows that RAM usage can be kept almost constant by early validation. If a category 3 inconsistency occurs, the run can be terminated early. This is reflected in a peak of 0.31 MB for nine messages. Compared to the semantic approach, the automaton beats it by far. Even with four messages, there is a difference of about 11 MB. In addition, the usage increases with every incoming message. The growth suggests a logarithmic curve which is positive for operation on large data sets in real-world environments (e.g., compared to linear increase).

As a result, we can state that storing large amounts of data as a graph structure increases the overhead of a program and thus has a large impact on memory usage. By omitting this step, the improved performance from Section 5.1.1 can be explained. In the next evaluation, we study the RAM increase over a longer period of time.

5.1.3. AutomatonCC vs. GreenCC in Real Manufacturing Scenario

The RAM increase detected in the previous experiment (especially the GreenCC) is the focus of this analysis. In this way, we want to identify if there exists a limit of the ramp up. This is important for a stable operation, because otherwise one risks running out of memory. During our next experiment, both systems have been connected to a real manufacturing stream during a time period of two hours.

Figure 13 compares the memory peaks of the automaton and the GreenCC. Both graphs show a slightly oscillating behavior at the beginning. This is due to the fact that memory is released and allocated again and again during the checking period. The usage of the GreenCC increases continuously up to about 200 MB. After that, there is a fluctuation around this value. The sharp increase is the result of a long intermediate storage of machine data over a longer period. With the automaton, on the other hand, you can see that memory consumption is significantly reduced. The maximum peak is about 25 MB. The usage of the automaton remains almost constant over the period.

The CPU comparison shows a relatively similar behavior for both systems. These are shown in Figure 14. Slight oscillations indicate times when the actual validation steps take place. Higher outliers can be explained by several expiring timers at similar time periods.

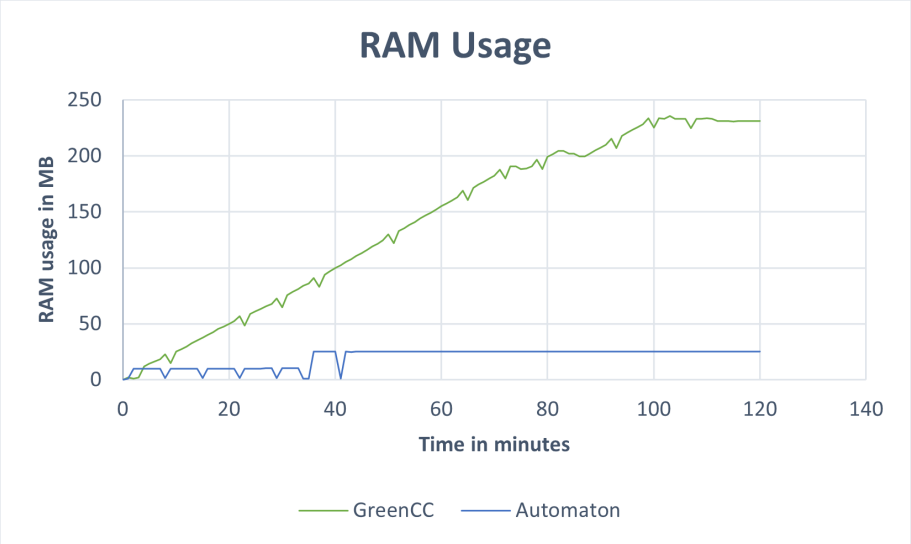

Figure 13. Memory usage during 2 h of operation on real manufacturing data.

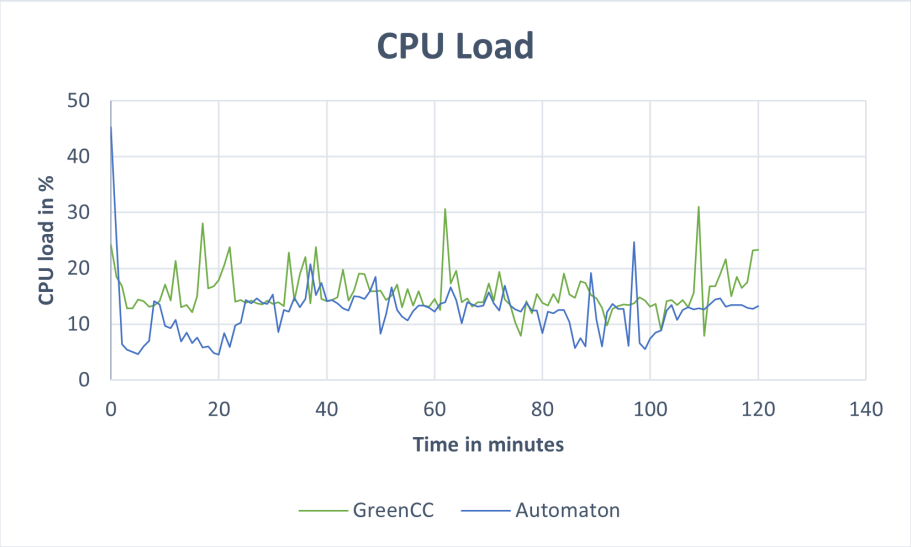

Figure 14. CPU load during 2 h of operation on real manufacturing data.

Overall, the results reinforce the findings from Sections 5.1.2 and 5.1.3. The higher energy consumption is explained by a higher memory overhead.

6. Discussion

This section discusses the advantages and disadvantages of the automaton in comparison with our previous semantic approaches. To this end, we first discuss the general structure and the extensibility of the methods. Afterwards, we will explicitly consider the checking for content inconsistencies (*complex inconsistencies*).

6.1. Automaton vs. Semantic Approach

The major difference between our *AutomatonCC* and our *GreenCC* is the response to incoming messages. As explained in the previous sections, in *AutomatonCC* we use an automaton structure for information management and processing, in which a state can change based on incoming information. From our point of view, this has the advantage that we know at any point in time whether all previously expected information is available or not. The check for inconsistencies of categories 1 and 2 therefore happens in the automaton almost without explicit checks (exception: SMD; see Section 4.1). In *GreenCC*, we had to perform a SPARQL test after timer expiration for this information.

However, the automaton structure shifts many tasks from the SPARQL queries to the automaton. This increases the complexity of the overall system. The program is developed modularly; the higher complexity has the consequence that fundamental changes bring more expenditure. In comparison, the *GreenCC* has a module for handling the data stream. This module does nothing but categorize incoming messages and store them until validation. Changes to the checking process take place in the SPARQL script.

Depending on the scope of the constraint to be checked, changes in SPARQL also become messy and difficult. For this, we consider the following section.

6.2. Checking Complex Inconsistencies via Template

In previous work, we performed the checks via SPARQL queries. For this purpose, the manufacturing environment had to be modeled with ontologies in advance. For the validation step, incoming process and machine data were converted from JSON to RDF format using these ontologies. RDF creates a graph structure that can be traversed via SPARQL queries. Validation via ontologies and SPARQL has the advantage that changes can be incorporated in a simple way. In addition, the modeling languages are easy to understand. These are two properties that argue for its use in a heterogeneous, continuously changing environment. A serious disadvantage of semantic modeling in our scenario is

that every discrepancy to be checked must be modeled and thus known. This makes it impossible to cover unknown errors. In addition, many models (including queries and ontology) have to be adapted even for small changes.

Therefore, we have decided to no longer perform checks of Categories 3 and 4 semantically, but by means of template matching. We receive the templates directly when changes are made. The line structure and the general process flow are relatively fixed. Thus, we can still refer to the respective process ontology when creating an automaton.

Template matching also offers further scope for optimization. Currently, we use an accurate method to detect inconsistencies with very high probability. As the authors have indicated in [6], relevant savings are recognizable in energy consumption when small measurement errors are taken into account. The consideration and evaluation of approximate approaches will be part of future work.

In summary, both concepts have their advantages and disadvantages. If changes in the checking process of complex inconsistencies are necessary, the adaptation of the template is easier than revising ontology and SPARQL queries. If the process changes, complex changes in the automaton are necessary. The GreenCC, on the other hand, remains unaffected. The stream handling controls the message grouping; at the end, a check is performed.

7. Summary and Conclusions

Our paper presented an automaton for data validation tasks in (I)IoT scenarios. At the beginning, we presented the problem of invalid data, especially in smart manufacturing environments, and first prototypes to cope with the task. Subsequently, we discussed the novel AutomatonCC concept, in which we particularly dealt with aspects of increasing efficiency. By direct validation of incoming messages as well as detection of overtakes, the automaton terminates the validation earlier in many cases and stores less data during the validation than previous methods. This is also reflected in the evaluation results, in which we have demonstrated the lower energy consumption due to reduced memory usage. Since our automaton concept is adaptable to related domains, it can be applied in each (I)IoT scenario where quality of data is of importance.

In future work, we want to further consider two limitations of our automaton: (1) Currently, an automaton has to be adapted manually to an (I)IoT scenario. Especially with many devices that are mapped to states, this process is time-consuming and error-prone. (2) With an increasing number of devices, the number of states increases nearly exponentially. An open question remains, up to which environment size our approach scales. For (1), we propose to automatically create automata based on an ontology of the environment to be validated. To address the second limitation, we propose to divide big IoT scenario into sub-scenarios. An automaton can be created for each sub-scenario. The resulting automata can exchange information at a higher level.

Further, we aim to incorporate data cleaning techniques into our automaton. Therewith, we want to consider in particular at what point cleaning should take place (e.g., directly after false validation or at the very end). Afterwards, we want to perform more tests with our automaton prototype on real manufacturing data to continuously increase the overall stability and performance of the consistency checker.

Our list also includes looking at machine learning techniques for validating data. These methods usually fall under the term *Anomaly Detection*. In analyses, we would like to find out whether machine learning can help us to efficiently detect outliers.

Author Contributions: Conceptualization, S.P. and S.G.; Software, S.P.; Supervision, S.G.; Writing—original draft, S.P.; Writing—review and editing, S.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The used datasets were provided by Robert Bosch Elektronik GmbH as part of a confidentiality agreement and are not publicly available.

Acknowledgments: The work of this paper has been supported by AE/MFT1 department of Robert Bosch Elektronik GmbH.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CC	Consistency Checker
CO_2e	Carbon-Dioxide Equivalents
ECG	Electrocardiogram
GB	Gigabyte
ICT	Information and Communication Technologies
IIoT	Industrial Internet of Things
IoT	Internet of Things
kg	Kilogram
kWh	Kilowatt-Hours
PCB	Printed Circuit Board
RDF	Resource Description Framework
RFL	Reflow Soldering
SJI	Solder Joint Inspection
SMD	Surface Mounted Devices
SMT	Surface-Mount Technology
SPARQL	SPARQL Protocol And RDF Query Language
SPI	Solder Paste Inspection
SPP	Solder Paste Printing
X-ray	Electromagnetic Radiation

References

1. Iftikhar, N.; Nordbjerg, F.E.; Baattrup-Andersen, T.; Jeppesen, K. Industry 4.0: Sensor data analysis using machine learning. In Proceedings of the Data Management Technologies and Applications: 8th International Conference, DATA 2019, Prague, Czech Republic, 26–28 July 2019; Revised Selected Papers 8; Springer: Berlin/Heidelberg, Germany, 2020; pp. 37–58.
2. Tao, F.; Qi, Q.; Liu, A.; Kusiak, A. Data-driven smart manufacturing. *J. Manuf. Syst.* **2018**, *48*, 157–169. [CrossRef]
3. Tian, Y.; Michiardi, P.; Vukolić, M. Bleach: A distributed stream data cleaning system. In Proceedings of the 2017 IEEE International Congress on Big Data (BigData Congress), Honolulu, HI, USA, 25–30 June 2017; pp. 113–120.
4. Geiger, L.; Hopf, T.; Loring, J.; Renner, M.; Rudolph, J.; Scharf, A.; Schmidt, M.; Termer, F. Ressourceneffiziente Programmierung, 2021. Available online: https://www.bitkom.org/sites/default/files/2021-03/210329_lf_ressourceneffiziente-programmierung.pdf (accessed on 14 September 2023).
5. Paasche, S.; Groppe, S. Enhancing Data Quality and Process Optimization for Smart Manufacturing Lines in Industry 4.0 Scenarios. In Proceedings of the International Workshop on Big Data in Emergent Distributed Environments, BiDEDE '22, Philadelphia, PA, USA, 12 June 2022. [CrossRef]
6. Paasche, S.; Groppe, S. GreenCC: A Hybrid Approach to Sustainably Validate Manufacturing Data in Industry 4.0 Environments. In Proceedings of the 12th International Conference on Data Science, Technology and Applications (DATA), Rome, Italy, 11–13 July 2023.
7. Zhang, L.; Jeong, D.; Lee, S. Data quality management in the internet of things. *Sensors* **2021**, *21*, 5834. [CrossRef] [PubMed]
8. Groppe, S.; Groppe, J.; Kukulenz, D.; Linnemann, V. A SPARQL Engine for Streaming RDF Data. In Proceedings of the Third International IEEE Conference on Signal-Image Technologies and Internet-Based System (SITIS), Shanghai, China.
9. Gao, J.; Xie, C.; Tao, C. Big data validation and quality assurance–issuses, challenges, and needs. In Proceedings of the 2016 IEEE symposium on service-oriented system engineering (SOSE), Oxford, UK, 29 March–2 April 2016; pp. 433–441.
10. Karkouch, A.; Mousannif, H.; Al Moatassime, H.; Noel, T. Data quality in internet of things: A state-of-the-art survey. *J. Netw. Comput. Appl.* **2016**, *73*, 57–81. [CrossRef]
11. Mansouri, T.; Sadeghi Moghadam, M.R.; Monshizadeh, F.; Zareravasan, A. IoT data quality issues and potential solutions: A literature review. *Comput. J.* **2023**, *66*, 615–625. [CrossRef]
12. Song, S.; Zhang, A. IoT data quality. In Proceedings of the 29th ACM International Conference on Information & Knowledge Management, Virtual Event, 19–23 October 2020; pp. 3517–3518.
13. Haav, H.M.; Maigre, R.; Lupeikiene, A.; Vasilecas, O.; Dzemyda, G. A semantic model for product configuration in timber industry. In *Databases and Information Systems X*; IOS Press: Amsterdam, The Netherlands, 2019; Volume 315, pp. 143–158.
14. Cortés, R.; Bonnaire, X.; Marin, O.; Sens, P. Stream processing of healthcare sensor data: Studying user traces to identify challenges from a big data perspective. *Procedia Comput. Sci.* **2015**, *52*, 1004–1009. [CrossRef]

15. Gao, S.; Dell Aglio, D.; Pan, J.Z.; Bernstein, A. Distributed stream consistency checking. In *Proceedings of the International Conference on Web Engineering*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 387–403.
16. Xuanyuan, S.; Li, Y.; Patil, L.; Jiang, Z. Configuration semantics representation: A rule-based ontology for product configuration. In Proceedings of the 2016 SAI Computing Conference (SAI), London, UK, 13–15 July 2016; pp. 734–741.
17. Maier, A.; Vodencarevic, A.; Niggemann, O.; Just, R.; Jaeger, M. Anomaly detection in production plants using timed automata. In Proceedings of the 8th International Conference on Informatics in Control, Automation and Robotics (ICINCO), Noordwijkerhout, The Netherlands, 28–31 July 2011; pp. 363–369.
18. Hranisavljevic, N.; Niggemann, O.; Maier, A. A novel anomaly detection algorithm for hybrid production systems based on deep learning and timed automata. *arXiv* **2020**, arXiv:2010.15415.
19. Schmidl, S.; Wenig, P.; Papenbrock, T. Anomaly detection in time series: A comprehensive evaluation. *Proc. VLDB Endow.* **2022**, *15*, 1779–1797. [CrossRef]
20. Suhothayan, S.; Gajasinghe, K.; Loku Narangoda, I.; Chaturanga, S.; Perera, S.; Nanayakkara, V. Siddhi: A second look at complex event processing architectures. In Proceedings of the 2011 ACM Workshop on Gateway Computing Environments, Seattle, WA, USA, 18 November 2011; pp. 43–50.
21. Jayasekara, S.; Kannangara, S.; Dahanayakage, T.; Ranawaka, I.; Perera, S.; Nanayakkara, V. Wihidum: Distributed complex event processing. *J. Parallel Distrib. Comput.* **2015**, *79*, 42–51. [CrossRef]
22. Anicic, D.; Rudolph, S.; Fodor, P.; Stojanovic, N. Stream reasoning and complex event processing in ETALIS. *Semant. Web* **2012**, *3*, 397–407. [CrossRef]
23. Ahmed, A.; Abdullah, S.; Bukhsh, M.; Ahmad, I.; Mushtaq, Z. An energy-efficient data aggregation mechanism for IoT secured by blockchain. *IEEE Access* **2022**, *10*, 11404–11419. [CrossRef]
24. Batmunkh, A. Carbon footprint of the most popular social media platforms. *Sustainability* **2022**, *14*, 2195. [CrossRef]
25. Zahedinia, M.S.; Khayyambashi, M.R.; Bohlooli, A. Fog-based caching mechanism for IoT data in information centric network using prioritization. *Comput. Netw.* **2022**, *213*, 109082. [CrossRef]
26. Kiourtis, A.; Mavrogiorgou, A.; Kyriazis, D. A computer vision-based IoT data ingestion architecture supporting data prioritization. *Health Technol.* **2023**, *13*, 391–411. [CrossRef]
27. Sultana, N.; Huq, F.; Razzaque, M.A.; Rahman, M.M. User utility maximization in narrowband internet of things for prioritized healthcare applications. *Sensors* **2022**, *22*, 1192. [CrossRef] [PubMed]
28. Warnke, B.; Sehgelmeble, Y.C.; Mantler, J.; Groppe, S.; Fischer, S. SIMORA: SIMulating Open Routing protocols for Application interoperability on edge devices. In Proceedings of the 2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC), Messina, Italy, 16–19 May 2022; pp. 42–49.
29. Vijayakumar, K.; Dhanasekaran, C.; Pugazhenthi, R.; Sivaganesan, S. Digital Twin for factory system simulation. *Int. J. Recent Technol. Eng.* **2019**, *8*, 63–68.
30. Wingerath, W.; Gessert, F.; Friedrich, S.; Ritter, N. Real-time stream processing for Big Data. *it-Inf. Technol.* **2016**, *58*, 186–194. [CrossRef]
31. Paasche, S.; Groppe, S. Poster: Handling Inconsistent Data in Industry 4.0. In Proceedings of the 17th ACM International Conference on Distributed and Event-Based Systems, Neuchatel, Switzerland, 27–30 June 2023; pp. 180–181.
32. Pereira, R.; Couto, M.; Ribeiro, F.; Rua, R.; Cunha, J.; Fernandes, J.P.; Saraiva, J. Energy efficiency across programming languages: How do energy, time, and memory relate? In Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, Vancouver, BC, Canada, 23–24 October 2017; pp. 256–267.