HiProX: Highly Efficient Process Execution on IoT and Edge Devices

Dominik Voigt, Lisa Podszun, Juergen Mangler, and Stefanie Rinderle-Ma

Chair of Information Systems and Business Process Management
TUM School of Computation, Information and Technology,
Technical University of Munich
85748 Garching, Germany
{firstname.lastname}@tum.de

Abstract The application of process technologies and systems has shifted from traditional business processes such as loan application to processes in domains such as manufacturing and healthcare. In these domains, Internet of Things (IoT) technology is widely used, requiring an integration of process and IoT technologies. While existing approaches focus on mining and analyzing IoT data in a process context, this work aims at providing the technology for highly efficient process execution on IoT and edge devices. This poses many challenges with respect to available computing resources that cannot be met by existing monolithic process management systems. Hence, this work presents HiProX, an transpilation-based implementation of the cloud process execution engine (CPEE) in RUST. It can be shown that HiProX meets the requirements of resource utilization and speed to run average process instances with full communication capabilities on suitable IoT platforms that provide a memory allocator using less than 8 MiB of memory per instance.

Keywords: Process Execution · IoT Data · Process Monitoring · Process Logging · Industry 4.0

1 Introduction

In recent years, Internet of Things (IoT) technology has found widespread adoption across many domains, such as manufacturing and healthcare [3, 20]. IoT offers several benefits such as improved data quantity, quality, and "freshness" for decision-making and analysis processes [16]. However, IoT integration also introduces many architectural and operational challenges, such as the heterogeneous nature of the different interfaces and capabilities of devices, scalability, unreliable power provisioning, and network connectivity [2, 20, 32].

The Version of Record is available online at: http://dx.doi.org/10.1007/978-3-032-02936-2_14. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms

Business Process Management (BPM) and Business Process Management Systems (BPMS) are nowadays the backbone of many enterprise-scale software [11]. From an architectural point of view, they provide service integration and orchestration, which has become paramount with the rise of service-oriented and microservice architectures in enterprise and specifically IoT-centric applications [17, 22, 24].

Hence, due to the symbiotic capabilities of IoT and BPM technologies [20], a wide variety of application scenarios have been discussed such as: BPM based Digital Twins for Smart Cities [27], BPM coordinated Cross-docking of Shipping Containers [13], or vertical integration in manufacturing companies, connecting shop-floor IoT systems to high level planning processes [26].

While progress was made, many challenges remain before all the envisioned advantages of IoT and BPM integration can be realized. In the classical integration approach, the Process Engine acts as an orchestration layer for the different software components and IoT devices [13] and is typically hosted within the cloud [9]. However this leads to several issues. For one, due to the latency between the Process Engine and the IoT devices, even for on-premise deployment, this approach is infeasible for latency critical applications [9, 24]. Secondly, this centralized approach can lead to a disconnect between sensor data and event logs [16, 20], as sensor data collection is often handled independently from the process enactment and just aggregated through the process engine. This makes the aggregation, preparation, and use of IoT data for decision mining and process mining challenging [16, 20]. Lastly, since the number of data sources in IoT scenarios is quite large, e.g., more than 10.000 IoT nodes in case of the Cross-docking example, network related issues due to data volume and velocity can become an issue [29, 32].

To address the issues of centralized components in IoT scenarios, decentralized approaches leveraging the Fog and Mist Computing Paradigms, have been proposed [3, 9, 23]. However none of these approaches moves the process execution out of the cloud, which would allow for a fully process oriented approach enabling easy correlation between IoT and process level events. Figure 1 depicts an example scenario for a fully process oriented approach in manufacturing. Here different process instances are directly deployed on the manufacturing machines. This minimizes latency for control and data access. Furthermore by local processing and filtering, as depicted in the expanded process instance (CNC Milling Machine), data volume and velocity over the network are greatly reduced. In this scenario the process engine routes data between instances and forwards it to standard process monitoring or logging software. Traditional IoT implementations would have a similar architecture, but rely on hard-coded components instead of easy to change process instances.

However executing processes on resource-constrained devices such as the manufacturing machines in our example, requires an efficient approach for process execution [33]. Hence, the research question (**RQ**) tackled in this work is how process instances can be deployed and executed on IoT devices.

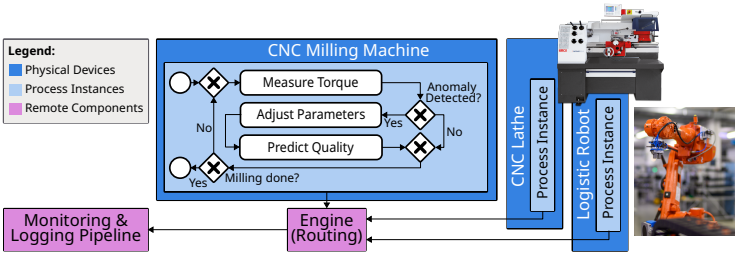

Fig. 1. Example Scenario of a Process Oriented Approach for Manufacturing

Existing process engines are often realized as custom interpreters. This makes their use in resource-constrained IoT fog and edge environments difficult. Hence, in this work, we present HiProX, a fast and resource-efficient general-purpose process engine that enables efficient process execution on IoT and edges devices. In detail, this work contributes to the field of IoT-enhanced BPM by (1) identifying existing resource-efficient approaches and deriving requirements for a resource-efficient process engine (2) presenting the HiProX process engine for process execution on resource-constrained devices, and (3) evaluating the engine regarding resource utilization and speed, which allows to identify suitable IoT platforms for instances to run on.

The remainder of the paper is structured as follows. Section 2 discusses execution scenarios and requirements in IoT-enhanced BPM. Section 3 presents HiProX, its architecture, and design decisions. Section 4 evaluates the proposed solution on two compute platforms. Section 5 discusses related work and Sect. 6 concludes the paper.

2 Requirements for Process Execution on IoT Devices

As integration of IoT into processes continues to increase, such as in IoT-enhanced BPM, the need to deploy and execute processes on IoT devices has been identified repeatedly [3,33]. Due to the resource constraints of IoT devices, the following requirements for developing a process engine for process execution on IoT devices, independently of the application domain, arise:

Requirement R1: While modern IoT devices offer increasing amounts of computing capabilities, the overall quantity of computing resources on these platforms is still constrained compared to desktop or server environments, making utilization of these computing resources more challenging (e.g., due to the general lack of an operating system or the restricted feature set of the provided OS) [2]. To enable the execution of business processes on such devices, resources must be utilized efficiently. Therefore, memory consumption and process execution time are used as quantitative measurements. Memory efficiency and execution time should be significantly better than an existing reference implementation, which has no focus on resource-efficiency.

Requirement R2: Logging of the process execution is an essential part of the process-oriented paradigm. This allows the BPMS to monitor, stop, and resume instances and freely modify the instance model, the execution context, and move the execution pointer(s) [34]. The resulting event log, containing classical process events and IoT collected data, furthermore enables meaningful data analysis as the collected IoT data is contextualized by the process events and other meta data [18].
Requirement R3: Although a variety of application protocols are commonly used in IoT scenarios (e.g. MQTT, CoAP, HTTP, ...), we decided to provide HTTPS support including multipart. This decision ensures that the process engine can be used in a wide variety of application scenarios (general-purpose), as HTTP is the de-facto standard application protocol of the Internet [35]. Nevertheless, supporting protocols would be feasible and easily implementable in our extensible architecture. Nonetheless this would required providing a protocol gateway to communicate with classical HTTP-based web services. Finally, while cryptographic protocols are expensive on a resource-constraint device, ensuring secure communication is essential. Thus we decided on supporting HTTPS (TLS).

3 HiProX Architecture

The HiProX architecture is depicted in Fig. 2. The implementation is based on the Cloud Process Execution Engine (CPEE) [19]. New contributions to the architecture are highlighted in green. The adapted architecture is discussed by providing a walk-through based on how a business process would be transpiled, deployed, and executed in Section 3.1. Section 3.2 discusses the advantages of a transpilation based approach that was chosen for the proposed solution instead of using model interpretation. Section 3.3 discusses the criteria considered when selecting a host language and why the Rust programming language was selected for this solution. Section 3.4 discusses the process DSL used for the model-to-code approach as an intermediate representation between graphical process model and binary code.

3.1 Architecture Walk-through

HiProX consists of two major components, i.e., the central **BPMS** component and the individual lightweight **Process Instance** component. The BPMS component is responsible for general features like logging, monitoring, and asynchronous callbacks. This component is the one proposed by Mangler [19]. This component was extended by providing a Transpilation Unit. For the enactment of a process model the BPMS spawns a process instance. These process instances are standalone programs that are semantically equivalent from the process model they have been generated from.

Execution of a process is initiated by the **Control Interface** of the BPMS which in turn passes the process model to the newly implemented **Instance**

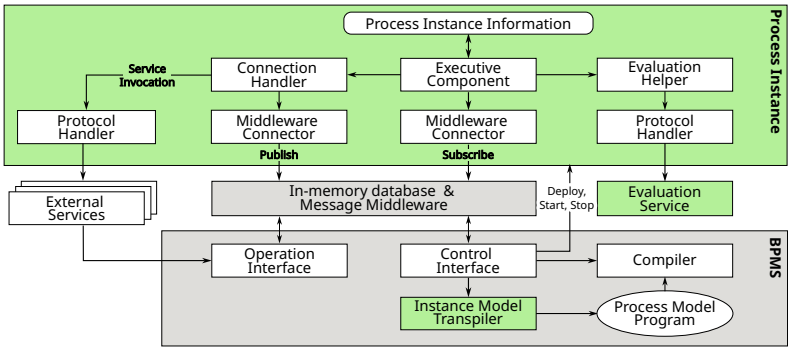

Fig. 2. HiProX Architecture (Contributions of This Paper Highlighted Green)

Model Transpiler that transpiles it to an equivalent program in the Process DSL described in Sect. 3.4. Thereafter, the **Control Interface** invokes the compiler to create the executable process instance for the target architecture at hand.

This resulting instance binary contains the instance logic and all the operative code like logging and protocol handlers and can thus be deployed and executed on a different device than the BPMS component. Lastly, the binary, execution context (variables and endpoints), and configuration files are deployed on the target, and the instance is executed.

Instance execution on the target starts by loading the configuration. Then, the **Executive Component** is initialized. This component semantically represents the instance as a whole. All *Process Instance Information* is managed by this component, and the DSL is defined as methods on the **Executive Component** instance. Subsequently, a **Message Middleware Connector** is initiated to listen for events emitted by the management component. This is used for features such as callbacks for asynchronous tasks [19]. The interfacing of process instances with the BPMS component follows the solution proposed by Mangler [19]. Here any events emitted during the execution of the process engine are send to the **Message Middleware**. The **BPMS** then retrieves these events for every process instance for monitoring and logging. This allows seamless integration of the proposed solution into the existing BPMS component. Finally, the execution of the instance code begins.

From the point of view of the process instance, every task, such as sending an email, executing a script, starting a machine or assigning work to a human process participant, is implemented by an external service [19]. This architectural decision keeps the engine lightweight and easily extensible [19].

For every task a **Connection Handler** is initialized. The tasks of the Connection Handler consist of the emission of events, e.g., position changes or changes in the Execution Context, such as changes to variables, through

https://cpee.org/ last access: 2025-05-29

Handlers, for logging (**R2**), and the creation and use of a **Protocol Handler** instance to invoke external services.

For the execution of scripts, e.g., to prepare data for service invocations or changing the execution context based on the result of a service call, an **Evaluation Helper** is instantiated. This component handles the invocation of the corresponding **Evaluation Service** via a **Protocol Handler** instance. The Evaluation Service executed the script against the execution context and the Evaluation Helper subsequently handles the result. Such results can include the manipulation of endpoints or data elements, signaling to retry execution or to stop the process instance execution.

The core motivation behind the architecture was to ensure easy extensibility and resource-efficient process execution while providing the features of a general-purpose process engine. Each of the following sections discusses a specific architectural decision and the resulting properties.

The communication between the BPMS and an instance is facilitated completely via the Message Middleware and In-Memory-Database component (excluding deployment, start, and stop of each instance).

This decision also decouples the **Process Instance** from the **BPMS** and thus allows the deployment of the individual components on possibly different tiers of the computing hierarchy [2]. Therefore, deploying instances directly on the IoT devices, while hosting the **Middleware** and the **BPMS** on the middle- or inner-edge tiers is possible. Furthermore this still allows to model, deploy and monitor the process instances via the BPMS component. And thus enabling Mist Computing architectures for IoT-enhanced BPM applications. Additionally this decision allows for horizontal scalability of certain services of the BPMS, such as the logging service by decoupling the event source (instance) and sink (BPMS).

Furthermore utilizing an in-memory database as a middleware component, combined with an appropriate DSL implementation, allows the complete application state of every instance and the corresponding execution logs to be automatically persisted during execution, satisfying the logging requirement (**R2**) (cf. Sect. 2). This feature, combined with the existing **BPMS** implementation in [19], to stop instances during execution and adapting the control flow, execution context, and even the instance model to handle errors occurring during instance execution, e.g. due to divergence of the instance model from reality [16].

The execution of scripts via the Evaluation Helper by invocation of the Evaluation Service differs from how scripts are handled in the architectures proposed by [19,34] that both support direct evaluation by the runtime (Ruby VM). Each approach has certain advantages and disadvantages. While using an external service increases latency and the potential for network-related issues, it provides significant advantages. First and foremost, this allows the formulation of scripts in arbitrary programming languages that do not have to be supported on the deployment target. Simplifying development of processes by Process Designers. Secondly, the possibility for handler code to rely on arbitrary execution environment requirements, e.g., access to a GPU. Lastly, it allows to either link the code directly into the *Instance Binary* or to deploy the execution service on the IoT

device by replacing the Protocol Handler component for scenarios where latency and network concerns are predominant. Thus, this approach offers flexibility and ease of use depending on the application scenario.

3.2 Transpilation vs. Interpretation

The execution of process models can be realized by interpretation or transpilation (model-to-code). The approach used by most process engines in literature and industry, is interpretation of the process model. While interpretation is intuitive, it has performance implications. Wherever an instance is executed an instance of the interpreter needs to be executed. This can require a significant amount of compute resources and is thus not viable for many IoT devices (e.g. on devices with less than 32 MiB RAM). Furthermore, since the model is interpreted, there is no automatic optimization performed, and thus either the model is executed as is, or a separate optimizer needs to be run prior to deployment which needs to be maintained. A transpilation based approach, which we refer to as model-to-code, transforms the process model into semantically and structurally equivalent code in a target programming language. This is the approach used by HiProX to realize process execution on a separate device from the BPMS.

Although this model-to-code approach generates code in a general purpose programming language, it has two advantages over manually writing the application code. First, changes in the business logic can easily be realized by updating the model. Second, this approach, given the actual DSL implementation is tested rigorously, the generated code is less likely to contain errors.

To ensure the maintainability of the produced instance code artifacts [10], an internal DSL within a host programming language is used, as proposed by Stürmer et al. [34]. The DSL, thus, should be easy to understand and extend and be powerful enough to express common workflow patterns. This has several advantages, depending on the programming language which we will discuss further in Sect. 3.3.

3.3 Language Selection Criteria

Selecting an appropriate DSL host programming language is vital to ensure that the instance binary (transpiled process model) is easily maintainable and is efficient to execute. Hence, we formulate the following Criteria C1–C3. The first criterion is whether the language requires a runtime or is compiled natively (**C1**), since runtime environments require additional memory. Furthermore, most popular languages of this category (e.g. Java, C#, Ruby, Python) use a garbage collector mechanism to manage memory. This makes memory-consumption less deterministic and requires additional compute resources. This is the main reason why the original Ruby Process DSL proposed by Stürmer et al. [34] is not suitable for IoT scenarios. The second criterion is the capability of the language to define a DSL that is easy to understand, and the resulting code fragment is easy to modify (**C2**), as an intermediate step between a graphical and a textual representation. Since process execution can be highly concurrent, the third criterion refers to how

easy it is to write memory-safe and race-condition free code to ensure stability and maturity of the implementation (**C3**).

Based on these criteria, we select Rust as programming language. Rust provides support for easily readable DSLs via macros and multi-line closures. Furthermore, there are examples of successful Rust DSLs such as SeaQuery for formulating SQL queries in native Rust (C2). Additionally, in contrast to other performance-oriented languages such as C and C++, Rust uses the Ownership memory management model instead of relying on the programmer to manage memory manually. Thus, using safe Rust (and thoroughly tested unsafe blocks where necessary) prevents a large class of memory-related issues such as data races and most memory leaks. Additionally, safe Rust provides a memory safety guarantee (C3).

3.4 Process Domain Specific Language

To ensure that the HiProX engine supports the modeling and execution of general-purpose processes, i.e., independent of any application domain, the process DSL has to be defined such that every component of the control flow has a corresponding DSL element. Since HiProX utilizes the CPEE for modeling, its DSL is based on the existing CPEE DSL in Ruby [21]. This design decision bears the advantage that the CPEE DSL has been already evaluated w.r.t. its expressiveness: it supports all basic control flow patterns and a wide range of more advanced patterns [21], thus satisfying requirement **R3** by design.

4 Evaluation

We evaluate our implementation based on a set of process models. Each process model represents a basic control flow pattern. We compare our novel Rust Process DSL implementation against the original Ruby Process DSL, currently used in the CPEE, based on memory consumption and run time.

4.1 Evaluation Setup

For the evaluation, we created and executed a process model for each of the following basic control flow patterns:

https://github.com/SeaQL/sea-query, last access: 2025-05-29
https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html, last access: 2025-05-29
https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html last access: 2025-05-29
https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html, last access: 2025-05-29
https://doc.rust-lang.org/book/ch15-06-reference-cycles.html, last access: 2025-05-29
https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html, last access: 2025-05-29

- **Sequence**: A sequence of 3 tasks
- **Parallel**: Parallel execution of 2 tasks
- **Choose**: Alternative execution of 2 tasks (only 1 is executed)
- **Loop**: A loop that executes a task a total of 3 times

While these patterns do not cover the full range of features supported by the DSL, they represent a foundational set of process components. We measured the execution time and memory usage of our Rust Process DSL against the original Ruby Process DSL proposed by Stürmer et al. [34] for each model which, in turn, have been evaluated against the van der Aalst Workflow Pattern [1] which compares and summarize the feature set of a multitude of process engines. We decided to limit our comparison to this reference implementation due to the following reasons. (1) Most of the engines we will discuss in Section 5 did not provide any code. (2) While some implementations provide memory consumption figures, none documented how these values are measured (Heap allocations vs. RSS). (3) Solutions that provided no network connectivity are not applicable for IoT scenarios. For more details see Section 5

For each model, tasks are implemented as a call to an external service that delays its response for a single second and then responds. For the proposed implementation, a dynamically (to minimize memory and disk usage on nodes with multiple running instances) and a statically linked version were evaluated. Memory and execution time are measured for all patterns. For the memory analysis, residual set size (RSS) and heap memory are considered. RSS is measured for both implementations using the `time`. To measure the heap allocations `heaptrack` is used for the proposed implementation in Rust and `MemoryProfiler` is used for the reference implementation in Ruby. For time measurements `time` is used for both implementations. For all measurements the time delay of the external service was eliminated from the reported time. Evaluation is done on two different platforms. To represent an unconstrained computing environment, a desktop computer with an AMD Ryzen 7 8745H and 32 GB of RAM (furthermore refered to as **Echo**) is used. To represent a constrained computing device an Orange Pi Zero 2W (furthermore called **Orange**) is used. While this device is still a powerful IoT device, we show in the following subsection that the instance binary could be executed on a device with significantly less memory.

Echo uses an evaluation service instance that is hosted on the machine. **Orange** utilizes an external evaluation service hosted within the internet and thus suffers from increased network delays. Echo is connected to the network via an Ethernet cable. Orange is connected via 2.4 GHz Wifi.

---

https://www.man7.org/linux/man-pages/man1/time.1.html last access: 2025-05-29
https://github.com/KDE/heaptrack last access: 2025-05-29
https://github.com/SamSaffron/memory_profiler last access: 2025-05-29
http://www.Orangepi.org/html/hardWare/computerAndMicrocontrollers/details/Orange-Pi-Zero-2W.html last access: 2025-05-29

4.2 Results

Table 1 shows the execution time measured for both implementations on both platforms. Here we can see that the proposed implementation in Rust performs better for every pattern on both platforms. For Echo, the proposed solution is more than 4 times faster for every pattern. This effect is even more pronounced on the low-compute Orange platform where the proposed implementation is faster by a factor of 5-13 times. The execution time measured for the proposed implementation increased by about one second when switching from the Echo to the Orange platform. For the Ruby implementation it increased by more than three seconds. This demonstrates that a compiled approach shows significant performance improvements, especially on resource-constrained platforms.

Table 1. Time Measurements

Pattern	Rust				Ruby	
	Time **Echo**		Time **Orange**		Time **Echo**	Time **Orange**
	Static	Dynamic	Static	Dynamic		
Sequence	0.25 s	0.26 s	1.12 s	1.42 s	1.35 s	6.27 s
Parallel	0.09 s	0.10 s	0.40 s	0.42 s	1.16 s	5.42 s
Choose	0.09 s	0.10 s	0.73 s	0.51 s	1.32 s	5.59 s
Loop	0.27 s	0.30 s	1.31 s	1.47 s	1.49 s	6.69 s

The results of memory profiling for the experiments is depicted in Tables 2 and 3. Table 2 shows RSS of both implementations across different patterns. Table 3 shows the maximum heap size of both implementations during the execution of the different patterns. These measurements show that the proposed implementation uses significantly less heap, by a factor of about 6–10 depending on the pattern across all platforms.

Table 2. Memory Measurements (RSS)

Pattern	Rust				Ruby	
	Memory **Echo**		Memory **Orange**		Memory **Echo**	Memory **Orange**
	Static	Dynamic	Static	Dynamic		
Sequence	6.78 MiB	14.00 MiB	5.82 MiB	13.60 MiB	94.5 MiB	85.0 MiB
Parallel	7.86 MiB	15.33 MiB	7.10 MiB	14.82 MiB	94.5 MiB	85.3 MiB
Choose	6.67 MiB	14.25 MiB	6.18 MiB	13.87 MiB	94.0 MiB	85.3 MiB
Loop	6.88 MiB	14.53 MiB	6.42 MiB	14.28 MiB	94.3 MiB	86.4 MiB

RSS of the proposed solution is also significantly smaller than for the reference solution (see Table 2). The statically linked approach has an RSS that is smaller by a factor of more than 10 across both platforms. The dynamically linked approach by a factor of about 5. However, here it is important to mention

that these factors are a lower bound if multiple instances of the same program are executed. The reason for this is that for each instance an individual ruby interpreter has to be instantiated and loaded into memory, consuming a significant amount of memory. For the dynamically linked version of the proposed solution this does not happen, the standard library code and the complete application code are linked dynamically, and thus only need to be kept in memory once which makes each process instance very lightweight. Therefore, given that multiple instances are executed on the same platform, significantly less memory is used by the proposed Rust implementation.

When comparing the memory consumption of the proposed solution with the implementations discussed in the related work (see Sect. 5), it becomes clear that the implementation uses comparative amounts of memory to the most resource-efficient approaches [7, 8] while providing essential features for general purpose process engines like network connectivity. Thus the proposed solution shows higher memory efficiency compared to related engines and compared to the reference implementation (**R1**).

Table 3. Memory Measurements (Heap)

Pattern	Rust				Ruby	
	Memory **Echo**		Memory **Orange**		Memory **Echo**	Memory **Orange**
	Static	Dynamic	Static	Dynamic		
Sequence	619.1 KiB	619.2 KiB	598.6 KiB	598.6 KiB	5790.0 KiB	5790.0 KiB
Parallel	966.1 KiB	977.4 KiB	946.5 KiB	799.1 KiB	5800.0 KiB	5810.0 KiB
Choose	949.9 KiB	950.0 KiB	982.3 KiB	982.4 KiB	6850.0 KiB	6850.0 KiB
Loop	998.5 KiB	998.3 KiB	1000.0 KiB	1000.0 KiB	6840.0 KiB	6850.0 KiB

5 Related Work

While there is a lot of work done on IoT-aware process execution, most of these works follow a similar approach by introducing some middleware layer that enables interoperability between the IoT and Process Engine, e.g. [30]. While these approaches certainly work, they utilize the Process Engine mainly as an orchestration layer, losing out on the correlation capabilties between IoT and Process level events a fully integrated approach of Process Engine and IoT could bring. Thus our research question was how process models could be deployed and efficiently executed on IoT devices. Therefor we discuss related work in the area of specialized process engines that are focused on process execution in a mobile or IoT setting as these engines would have the potential to execute processes on IoT devices enabling a fully integrated approach. Through a structured literature analysis [36], we identified 11 approaches that are summarized with their key characteristics in Tab. 4 and described in the following.

Hackmann et al. [14] implement Java-based BPEL workflow engine Sliver for mobile devices with focus on efficient implementation with SOAP support. While

Table 4. Existing process engines with key characteristics

	Language	Model	Connectivity	Mem.	Comp. or Interpr.	Execution
2006 [14]	Java	BPEL	SOAP	<22 MB	Interpreted	Centralized
2007 [25]	Java [3]	BPEL	SOAP	?	Interpreted	Centralized
2008 [31]	Java	BPEL	SOAP	?	Interpreted	Distributed
2009 [8]	C	SISARL-XPDL	-	1MB	Compiled	Centralized
2011 [7]	C	SISARL-XPDL	-	8 MB	Compiled	Centralized
2011 [28]	.Net	ADEPT	Standalone	?	Interpreted	Distributed
2011 [12]	C++	BPEL	SOAP	1 KB (???)	Compiled	Centralized
2012 [4]	Objective-C	BPMN	HTTP	?	Interpreted	Distributed
2014 [5]	Objective-C	BPEL	HTTP	?	Interpreted	Distributed
2015 [6]	?	BPEL	HTTP	?	Interpreted	Distributed
2016 [15]	Java	ADEPT2	?	?	Interpreted	Centralized

no information on the memory consumption is available, the authors state that their approach outperforms the memory consumption of ActiveBPEL of 22 MB. Pajunen et al. [25] follow a similar approach as [14]. However, they focus less on the resource-efficient implementation of the engine and more on support for local services and different communication protocols such as HTTP, SMS, and email. Sen et al. [31] leverage the Sliver engine for choreographic workflow execution in mobile ad-hoc networks. Both, Chou et al. [8] and Chen et al. [7] develop C-based embedded workflow engines for robotic applications supporting an augmented subset of XPDL called EMWF. In this approach, models are compiled into native code. However, both engines only provide support for local tasks and do not provide any form of network connectivity. While executing a workflow, the engines consume about 1 MB and 8 MB of memory respectively. Pryss et al. [28] propose a distributed interpreting workflow engine for a seemingly custom process definition language using .Net called MARPLE. Communication is facilitated through a custom protocol [3]. No resource utilization measures for MARPLE are available. Glombitza et al. [12] present a model-to-code approach for the execution of BPEL models on sensor nodes. Their approach facilitates network connectivity via LTP. Hence, realizing memory efficiency at the cost of compatibility or additional gateway services, as most of the Web utilizes HTTP-based communication. Additionally, their approach does not include an intermediary internal DSL that simplifies the maintainability of the generated code. Chang et al. [5] develop an application framework for the social private cloud called SPiCa. In their approach, individual devices host a SPiCa instance, including a BPEL workflow component. Conceptually similar works have been proposed by Chang et al in [4,6]. In all three approaches, resource-efficient process execution was not actively discussed and no resource utilization measurements were reported. Schobel et al. [15] develop a lightweight Java process engine to conduct medical surveys. Surveys are modeled within a graphical modeling language and then transpiled into a process model (ADEPT2). No resource utilization measurements are provided.

Summary and Distinction from existing Research: Prior works focus on specific aspects of constrained process execution, e.g., Schobel et al. [15] supporting offline execution, Glombitza et al. [12] and Chou et al. [8] focusing on resource-efficient execution using a model-to-code approach. By contrast,

HiProX aims at proving comprehensive support by realizing the following three properties: (1) A model-to-code approach that translates the process model to executable code, specifically an internal DSL as in [21,34] that can be compiled and natively executed on devices with low compute capabilities. This contrasts approaches that require a runtime environment to execute the instance which needs to be supported therefore adding a further deployment dependency and leading to an resource overhead (e.g. in Java or C# based solutions [14, 15, 25, 28]). (2) Full HTTP support to communicate with external web services, including support for HTTP Multipart. This is essential since HTTP is one of the defacto standard communication protocols for IoT applications and the web [35]. This contrasts this work from prior solutions that provide non-standard or no connectivity, and thus cannot be used as general purpose process engines (e.g. solutions proposed by Glombitza et al. [12] that utilizes LTP or Chou et al. [8] and Chen et al. [7] that provide no network connectivity. (3) Full logging of the process execution [19] via a distributed engine architecture. Full logging of the process instance execution allows for monitoring, stopping, modifying (control flow, execution context, and even the instance model), and resuming the process instance [34] to recover from errors during execution.

The solution proposed in this paper is a full-featured process engine similar to the implementation by Mangler and Stürmer [21, 34], while improving the resource utilization by a factor of up to 13, thus making it viable option to use on IoT compute platforms.

6 Conclusion

To advance the integration of IoT and BPM technology, the aim of this paper was to develop general-purpose process engine HiProX that allows the deployment and execution of processes on IoT devices which are generally resource-constrained. For meeting the requirements of resource-constrained execution, the HiProX architecture includes a model-to-code approach for efficient process execution, logging capabilities, and, to complement the model-to-code approach, a powerful internal DSL. For the host language, Rust was chosen due its support fo easily readable DSLs and memory safety guarantee. As shown in the evaluation, HiProX shows significantly more efficient use of computing resources. The evaluation also shows that the engine is more memory efficient than most of the related engines, and consumes significantly less than the threshold described in the paper that proposed the Sliver engine [14] that provides comparable features. Future work will tackle the following challenges. While logging can provide fault tolerance, no mechanism was implemented to handle disconnects when talking to external services or the middleware component. Furthermore, the possibility of recovery after a crash, e.g., due to power supply issues [32], common in an IoT environment, is not yet implemented. Lastly, the implementation relies on

https://github.com/DominikVoigt/rusty_weel/blob/main/weel_lib/src/dsl.rs last access: 2025-05-29

the Rust Runtime for the `std` crate, and thus requires a memory allocator to function.

References

1. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases **14**(1), 5–51 (2003)
2. Buyya, R., Srirama, S.N.: Internet of Things (IoT) and New Computing Paradigms, pp. 1–23. Wiley (2019)
3. Chang, C., Srirama, S.N., Buyya, R.: Mobile cloud business process management system for the internet of things: A survey. ACM Computing Surveys **49**(4), 1–42 (Dec 2016)
4. Chang, C., Srirama, S.N., Ling, S.: An Adaptive Mediation Framework for Mobile P2P Social Content Sharing, pp. 374–388. Springer Berlin Heidelberg (2012)
5. Chang, C., Srirama, S.N., Ling, S.: Spica: a social private cloud computing application framework. In: Mobile and Ubiquitous Multimedia. pp. 30–39 (2014)
6. Chang, C., Srirama, S.N., Mass, J.: A middleware for discovering proximity-based service-oriented industrial internet of things. In: Services Computing (2015)
7. Chen, W.C., Shih, C.S.: ERWF: Embedded real-time workflow engine for user-centric cyber-physical systems. In: Conference on Parallel and Distributed Systems. pp. 713–720 (2011)
8. Chou, T., Chang, S., Lu, Y., Wang, Y., Ouyang, M., Shih, C., Kuo, T., Hu, J.S., Liu, J.: Emwf for flexible automation and assistive devices. In: Real-Time and Embedded Technology and Applications Symposium. pp. 243–252 (2009)
9. Del Gaudio, D., Hirmer, P.: A lightweight messaging engine for decentralized data processing in the internet of things. SICS Software-Intensive Cyber-Physical Systems **35**(1–2), 39–48 (Aug 2019)
10. Do Nascimento, L.M., Viana, D.L., Neto, P., Martins, D., Garcia, V.C., Meira, S.: A systematic mapping study on domain-specific languages. In: Software Engineering Advances (ICSEA 2012). pp. 179–187 (2012)
11. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer Berlin Heidelberg (2018)
12. Glombitza, N., Ebers, S., Pfisterer, D., Fischer, S.: Using BPEL to Realize Business Processes for an Internet of Things, pp. 294–307. Springer Berlin Heidelberg (2011)
13. Grefen, P., Brouns, N., Ludwig, H., Serral, E.: Co-location specification for iot-aware collaborative business processes. In: Information Systems Engineering in Responsible Information Systems. pp. 120–132 (2019)
14. Hackmann, G., Haitjema, M., Gill, C., Roman, G.C.: Sliver: A bpel workflow process execution engine for mobile devices. In: Service-Oriented Computing. pp. 503–508 (2006)
15. J. Schobel, R. Pryss, M.S., Reichert, M.: A lightweight process engine for enabling advanced mobile applications. In: Cooperative Information Systems. pp. 552–569 (2016)

https://doc.rust-lang.org/reference/runtime.html last access: 2025-05-29
https://doc.rust-lang.org/std/ last access: 2025-05-29

16. Janiesch, C., Koschmider, A., Mecella, M., Weber, B., Burattin, A., Di Ciccio, C., Fortino, G., Gal, A., Kannengiesser, U., Leotta, F., Mannhardt, F., Marrella, A., Mendling, J., Oberweis, A., Reichert, M., Rinderle-Ma, S., Serral, E., Song, W., Su, J., Torres, V., Weidlich, M., Weske, M., Zhang, L.: The internet of things meets business process management: A manifesto. IEEE Systems, Man, and Cybernetics Magazine **6**(4), 34–44 (Oct 2020)
17. Loke, S.W.: Service-oriented device ecology workflows. In: Service-Oriented Computing. pp. 559–574 (2003)
18. Mangler, J., Grüger, J., Malburg, L., Ehrendorfer, M., Bertrand, Y., Benzin, J.V., Rinderle-Ma, S., Serral Asensio, E., Bergmann, R.: Datastream xes extension: Embedding iot sensor data into extensible event stream logs. Future Internet **15**(3), 109 (Mar 2023)
19. Mangler, J., Rinderle-Ma, S.: Cloud process execution engine: Architecture and interfaces. CoRR **abs/2208.12214** (2022)
20. Mangler, J., Seiger, R., Benzin, J., Grüger, J., Kirikkayis, Y., Gallik, F., Malburg, L., Ehrendorfer, M., Bertrand, Y., Franceschetti, M., Weber, B., Rinderle-Ma, S., Bergmann, R., Asensio, E.S., Reichert, M.: From internet of things data to business processes: Challenges and a framework. CoRR **abs/2405.08528** (2024)
21. Mangler, J., Stuermer, G., Schikuta, E.: Cloud process execution engine - evaluation of the core concepts. CoRR **abs/1003.3330** (2010)
22. Mass, J., Chang, C., Srirama, S.N.: Context-aware edge process management for mobile thing-to-fog environment. In: European Conference on Software Architecture: Companion Proceedings (2018)
23. Mass, J., Chang, C., Srirama, S.N.: Edge process management: A case study on adaptive task scheduling in mobile iot. Internet of Things **6**, 100051 (Jun 2019)
24. Mass, J., Srirama, S.N., Chang, C.: Step-one: Simulated testbed for edge-fog processes based on the opportunistic network environment simulator. Journal of Systems and Software **166**, 110587 (Aug 2020)
25. Pajunen, L., Chande, S.: Developing workflow engine for mobile devices. In: Enterprise Distributed Object Computing Conference. pp. 279–279 (2007)
26. Pauker, F., Mangler, J., Rinderle-Ma, S., Pollak, C.: centurio.work - modular secure manufacturing orchestration. In: Business Process Management. pp. 164–171 (2018)
27. Pańkowska, M., Żytniewski, M.: Digital twins for smart city. In: Smart Spaces. pp. 269–286 (2024)
28. Pryss, R., Tiedeken, J., Kreher, U., Reichert, M.: Towards Flexible Process Support on Mobile Devices, pp. 150–165. Springer Berlin Heidelberg (2011)
29. Schönig, S., Ackermann, L., Jablonski, S.: Internet of things meets bpm: A conceptual integration framework. In: Simulation and Modeling Methodologies, Technologies and Applications. pp. 307–314 (2018)
30. Schönig, S., Ackermann, L., Jablonski, S., Ermer, A.: Iot meets bpm: a bidirectional communication architecture for iot-aware process execution. Software and Systems Modeling **19**(6), 1443–1459 (Mar 2020)
31. Sen, R., Roman, G.C., Gill, C.: Cian: A workflow engine for manets. In: Coordination Models and Languages. pp. 280–295 (2008)
32. Sneps-Sneppe, M., Namiot, D.: On web-based domain-specific language for internet of things. In: Ultra Modern Telecommunications and Control Systems and Workshops. pp. 287–292. IEEE (2015)
33. Stoiber, C., Schönig, S.: Event-driven business process management enhancing iot – a systematic literature review and development of research agenda. In: Innovation Through Information Systems. pp. 645–661 (2021)

34. Stürmer, G., Mangler, J., Schikuta, E.: A domain specific language and workflow execution engine to enable dynamic workflows. In: Parallel and Distributed Processing with Applications. pp. 653–658 (2009)
35. Uckelmann, D., Harrison, M., Michahelles, F.: An architectural approach towards the future internet of things. In: Architecting the Internet of Things. pp. 1–24 (2011)
36. Voigt, D.I.: RUST Process Model DSL. Master's thesis, Technische Universität München (2025)