ScienceDirect

IFAC PapersOnLine 53-3 (2020) 72–77

Towards Flexible, Fault Tolerant Hardware Service Wrappers for the Digital Manufacturing on a Shoestring Project [★]

Michael J. McNally [*] **Jack C. Chaplin** [*]
Giovanna Martinez-Arellano [*] **Svetan Ratchev** [*]

[*] *The Institute for Advanced Manufacturing*
Advanced Manufacturing Building
Jubilee Campus, University of Nottingham
UK, NG7 2RD
(correspondence e-mail: mike.mcnally@nottingham.ac.uk)

Abstract: The adoption of digital manufacturing in small to medium enterprises (SMEs) in the manufacturing sector in the UK is low, yet these technologies offer significant promise to boost productivity. Two major causes of this lack of uptake is the high upfront cost of digital technologies, and the skill gap preventing understanding and implementation. This paper describes the development of software wrappers to facilitate the simple and robust use of a range of sensors and data sources. These form part of a common architecture for data acquisition in the Digital Manufacturing on a Shoestring project. We explain the existing Shoestring demonstrator architecture, and discuss how a 'crash-only' microservices architecture would improve fault tolerance and adaptability of the system.

Keywords: Sensor Networks, Low-Cost Digital Manufacturing, Industry 4.0, Manufacturing Resilience, Microservices.

1. INTRODUCTION

The adoption of condition-based maintenance (CBM) in small and medium enterprises (SMEs) is shown to reduce the cost of both corrective and preventive maintenance along with lowering insurance premiums (Carnero (2012)), potentially saving adopters significant costs. However, Fumagalli et al. (2009) found that SME customers for CBM systems were dispersed in many sectors and so suffered from generic solutions on the market which did not have the specialisation for their needs, or products which were specialised for a different sector's needs.

Through the Digital Manufacturing on a Shoestring project ("Shoestring"), we aim to lower the cost barrier associated with implementation of digital manufacturing technologies within SMEs, including implementation of CBM. McFarlane et al. (2020) describes Shoestring as "*capturing core functions and services in technology building blocks*", which can be "*joined together in different configurations easily and repeatably*", presenting a path for low-cost yet tailored digital solutions.

This ease of use and 'plug-and-play' functionality is essential to lower the time investment and skill level required to create a digital solution for an SME's manufacturing system. Also, flexible, open-source building blocks will allow SMEs to rapidly achieve minimum working systems and lower the barrier to capturing their specific metrics for maintenance or other digital requirements.

Hawkridge et al. (2019) concluded that a service based architecture was the best way of capturing all of the requirements for a low-cost and flexible industrial communication system. The services are connected by implementing open communication standards such as Open Platform Communications Unified Architecture (OPC-UA) (OPC UA Foundation (2017)) or Data Distribution Service (DDS) (Object Management Group (2015)). Data is acquired through service wrappers on top of hardware platforms that gather data from sensors or are integrated with manufacturing hardware.

Low-cost microcomputers such as the Raspberry Pi (Upton and Halfacree (2014)) are widely available and have been successfully used in demonstrators for advanced manufacturing systems (e.g Chaplin et al. (2015) or Chen et al. (2018)). These can be readily integrated with cheap, off-the-shelf sensors or connected to legacy hardware platforms to provide a basis for simple and low cost monitoring. The combination of a micro-computer and appropriate software can function as service wrappers, and, within the Shoestring architecture, can be seen as elementary building blocks to provide specific services.

A key requirement for any additional IT infrastructure will be reliablity and self-corrective behaviour. Any benefits could be rapidly lost if the new infrastructure requires regular maintenance or user intervention.

[★] The research described in this paper was funded by EPSRC grant EP/R032777/1, for which the support is greatly appreciated. We'd like to thank our project partners at the University of Cambridge Institute for Manufacturing for their help and support.

Several reference architectures exist as guidelines for industrial digitalisation, such as the Reference Architectural Model Industry 4.0 (RAMI4.0) (Platform Industrie 4.0 (2016)), and the Industrial Internet Reference Architecture (IIRA) (Industrial Internet Consortium (2019)). Both of these are descriptions of a whole digital enterprise, ranging from the business operations to the factory floor. Examples of successful implementation of RAMI4.0 systems can be seen in case studies in an automotive foundry (Sipsas et al. (2016)), cork processing and footwear manufacturing (Hernández et al. (2020)). All of these involve the digitalisation being outsourced.

The Shoestring approach should be seen as a sub-set of RAMI4.0 / IIRA or similar reference architectures, specific to the challenges of upgrading an already existing production facility, complete with legacy assets, without requiring the expense of an external consultancy or provider.

Communication layers like OPC-UA have been used to integrate legacy devices into a network (Park and Wook Jeon (2019)), and to build service oriented architectures (Lam and Haugen (2019)). However, OPC-UA has many different implementations in software and, by its nature as a generic solution, is very complex (Cenedese et al. (2019)).

Our approach is to build much more constrained software wrappers which can be used to enable very rapid development of basic functionality for businesses who are highly resource constrained.

In this paper we will describe the development of communication wrappers for the Shoestring architecture, and their deployment on representative manufacturing equipment and demonstration platforms. We also propose a set of improvements for optimising the software for flexibility, reliability, and fault tolerance, whilst keeping within the Shoestring ethos of low-cost simplicity. These communication wrappers are a significant, necessary step towards bringing effective CBM to more manufacturing SMEs.

2. USE CASES

Three use cases based on common examples of manufacturing hardware were identified for implementation and validation of the communication wrapper approach. The particular solutions were chosen from some of the highest priority identified by Schönfuß et al. (2020) in structured workshops with UK manufacturing SMEs.

2.1 Job Status Monitoring

We collected data from a representative production line (An SMC HAS-200 training platform from SMC International Training) using MQTT communication to relay the status reports from programmable logic controllers. We used the state changes and reported product ID from the production line to provide a real-time dashboard showing work station jobs, production status and individual product ID history.

2.2 Condition Based Monitoring

Using a low-cost accelerometer we gathered vibration data on a Hermle 5-Axis machining centre. On a dashboard we displayed both the inferred machine state, cutting or not cutting, as well as the results from deep-learning inference of the current tool wear state.

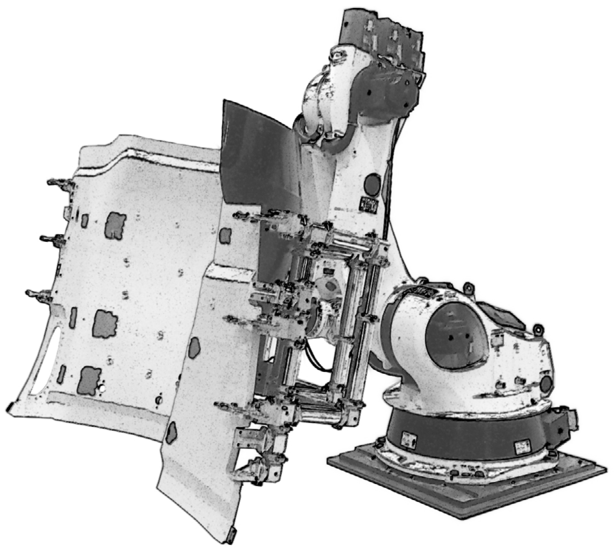
Fig. 1. The flexible fixturing held by this robot provides a platform for two sensor modules: motion sensing using accelerometers and gyroscopes, and panel temperature and position relative to the fixturing.

2.3 Robotic Process Monitoring

We attached sensors to a reconfigurable fixture designed for holding composite aerospace components and mounted on an industrial robot. Sensors measured motion of the fixture as the robot moves. To represent environmental quality markers we also recorded the vibration and temperature of the component (Fig. 1), as well as the ambient temperature. All of this was displayed on a live dashboard.

3. SOFTWARE DEVELOPMENT

A common requirement of these use cases was data capture. Data was logged on a cloud database common to all three platforms. We used an Amazon web services EC2 computing instance, running the time-series database influxDB. We chose to use a time-series database due to the strongly time ordered nature of the data we were recording. Every item uploaded to the database was time stamped, and we anticipate this as a requirement for all operational data in the manufacturing field. All data was accessible from a visualisation engine running in the cloud. A requirement for these use cases was for real-time data visualisation, so data captured needed to be uploaded to the cloud also in real-time. This was achieved with Raspberry Pi microcontrollers, which powered the sensors, logged the data, pre-processed the data, and uploaded the data to the cloud.

We use the term sensors to indicate data sources more generally. For example, this could be a sensor measuring a physical parameter such as temperature. Alternatively, it could be an MQTT client capturing messages between industrial programmable logic controllers.

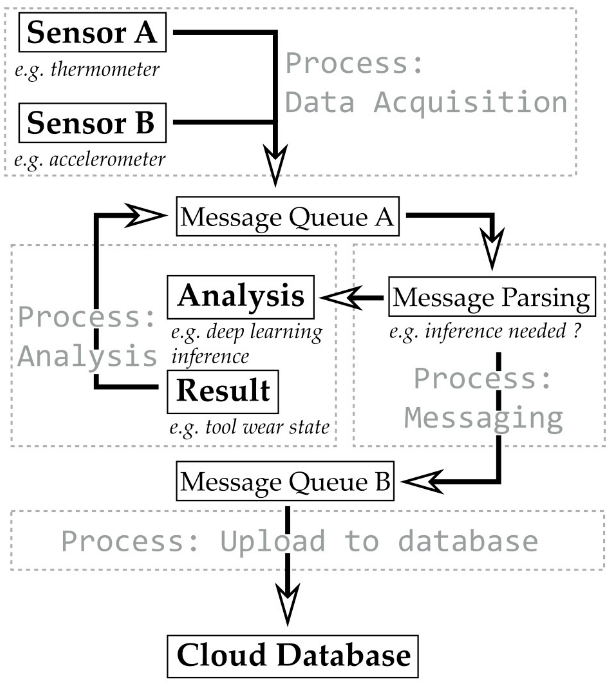

Fig. 2. The present monitoring software architecture, as realised in the implemented wrapper approach. A data acquisition process retrieves sensor readings and metadata from Sensors A & B and outputs to Message Queue A. A messaging process parses the metadata in each message, batches for analysis, reduces the volume of data for storage, and passes batches to Message Queue B, where it is read by a database upload process which performs data upload. Analysis is carried out by another process, results and are tagged and placed back onto Message Queue A.

All code was written in the Python programming language.

3.1 Program Flow

The main program code was broken up into functional blocks which ran in separate Python `process`es using the Python `multiprocess` library. This allows multiple instances of the Python interpreter to run simultaneously, preventing thread locking when waiting on external interfaces. We will refer to code running in it's own Python instance as a `process`. The Raspberry Pi 3B+ is a four core processor, and task distribution between cores was handled automatically (and effectively) by the operating system.

The Python `multiprocess` library implements message queues, enabling communication between separate asynchronous processes. The architecture implemented for data-logging programs is illustrated in Fig. 2.

In the data aquisition `process`, data is recorded from the sensor(s) and placed onto Message Queue A, together with some necessary metadata (timestamp, sensor name, measurement axis, etc).

Sensor $\rightarrow$ *MQ-A*

In a separate messaging `process`, message metadata is parsed from Queue A. The operations taken are preconfigured by a user, and based on simple logical operations on the metadata.

MQ-A $\rightarrow$ *Parser*

For example, for vibration data in the condition monitoring system, further analysis is needed. Data is batched and sent to the analysis `process` running the analysis routines.

Parser : metadata = vibration, Data $\rightarrow$ *Analysis*

Results from analysis are placed back onto Message Queue A for parsing again.

Analysis -> MQ-A

Any messages ready to be sent to the cloud database are placed onto Message Queue B.

Parser : metadata = analysis-result, Data $\rightarrow$ *MQ-B*

In a final upload `process`, messages are read from Message Queue B, reformatted for upload to the online database, batched, and sent via an HTTP POST request.

MQ-B $\rightarrow$ *Reformat* $\rightarrow$ *Batch* $\rightarrow$ *POST*

Note that data can be moved to several places at once, so data could be moved to analysis and message queue B for analysis:

Parser : metadata = vibration, Data $\rightarrow$ *Analysis*
Parser : metadata = vibration, Data $\rightarrow$ *MQ-B*

By splitting different functions into seperate python processes we avoid several pifalls:

- **Bottlenecking of maximum frequency.**

The HTTP POST operation (Fielding and Reschke (2014)) takes approximately 200 ms, leaving gaps in data processing even when batching data before sending.

- **Parallel I/O operations.**

Reading the Raspberry Pi Inter-integrated Circuit (i2c) bus can require continuous polling for new data. This essentially results in blocking execution in its thread.

- **Parallel resource intensive operation.**

The condition monitoring program required execution of a deep learning inference model with batches of recorded data. This is a resource intensive process which can block execution for hundreds of milliseconds to process one second of data.

3.2 Code Division

Splitting code into separate `process`es enabled multiple functions to occur simultaneously.

Similarly, splitting our programs into separate files not only simplifies practical coding, but enables us to separate commonly used functions from those which are specific to a particular sensors or analyses.

The breakdown of code into a main script and the device files and drivers is illustrated in Fig. 3.

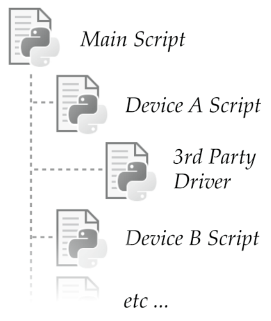

Fig. 3. Hierarchy of devices and drivers.

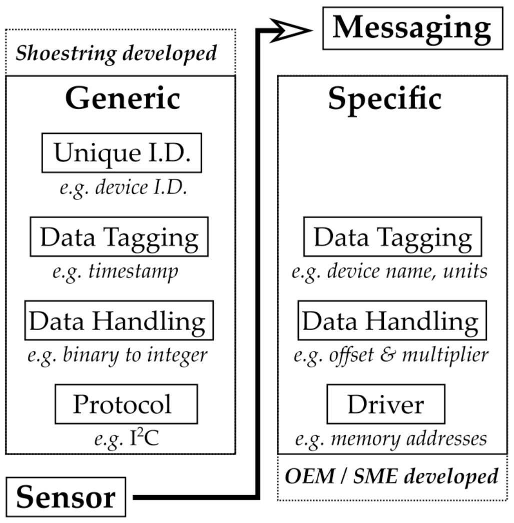

Fig. 4. Sensor data passes through protocol and drivers, through internal data handling, tagging and identification, before passing to message queues. These can be divided into generic and specific tasks. Generic tasks should be handled by shoestring standard architecture, whilst OEMs or SMEs can develop device specific drivers and configurations.

The main script handled core functionality, such as creating and closing message queues, interpreting metadata and handling the appropriate messaging, or controlling low level hardware interfaces like USB. This is common to all implementations of sensor data collection. Then, secondary modular sensor packages are included.

These secondary packages include device files that implement the functions for e.g. reading a value from a sensor, or configuring a device to start collecting data. These are called from the main script as required. In many cases a third party driver file is also required which encapsulates complex operating instructions such as the process for starting, configuring and calibrating the laser range-finder sensor. These are accessed as functions through the device file.

The division has the added effect of making it easier to integrate third party devices into the solution. We can formalise the division into the software features which are *generic*, and the features which are *specific* to a particular device.

So, for example, the infrared thermometer sensor attached to the robotic fixture demonstrator might communicate over i2c, which is a *generic* bus with native implementations in multiple microcomputer platforms.

However, the actual process of reading from that sensor requires polling a memory address continuously for a 'data-ready' bit, and then reading two memory addresses to acquire the two data bytes. These operations are *specific* to that particular sensor.

These are divided into two files - one, containing generic methods, the other, a file based on a simple template, containing the procedure for acquiring data. The benefit is that the *generic* methods can be shared between a very wide number of *specific* implementations.

An example is illustrated in Fig. 4. Following this example, *generic* methods can be used to convert binary data to integer representation, but *specific* information is needed to handle the offset and multiplier to turn the raw data into a floating point value for temperature.

Third party drivers are included where complex procedures have already been coded. The laser rangefinder module used in the robotics demonstrator has a highly complicated process for initialisation. However, this could be carried out by calling a single function from a driver file. Once the device was initialised and configured, it returned a 16-bit data word at a specific memory address to be read out regularly, exactly the same as the accelerometer, thermometer, or colour sensor used in the demonstrators, and could be read out with a common, *generic* function.

The specific modules only need to provide functions for discrete methods such as retrieving data, initialising hardware or collecting offset and multiplier information from the sensors. For this reason, they could be relatively easy for third parties such as SMEs and original equipment manufacturers (OEMs) to write compatible drivers. Furthermore this could easily be used to provide data acquisition from legacy devices without an excessive workload.

4. FUTURE ARCHITECTURE

Our development of the prototype demonstrators illustrated the core flaw within the software model were using. Although the python `multiprocess` library allows multiple instances of the interpreter to run, any errors rapidly propagate and crash the entire program.

To counter this behaviour, there are several steps which we will take towards reliable and fault tolerant behaviour.

- **Divide the generic and specific function into a separate handler and driver.**

We can split the services into a generic 'handler' program, which simply pulls 'handles' from a specific driver software as illustrated in Fig. 5. This will insulate the generic software from a crash in driver software or unexpected behaviour from the connected device.

- **Implement the different processes as microservices**.

Generic functions such as communication with a cloud server, handling message queues, or low level hardware

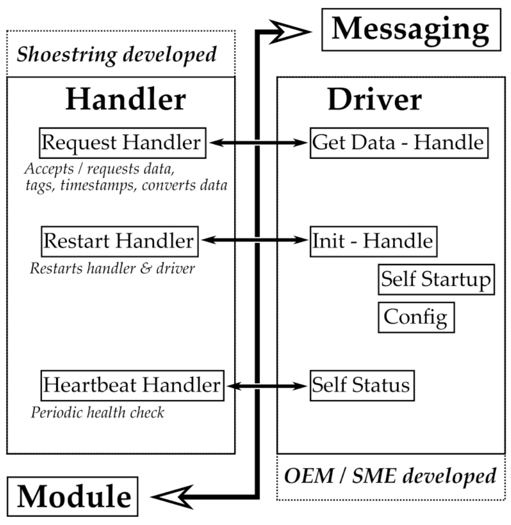

Fig. 5. Separation of module (sensor/analysis) into generic handlers and driver data.

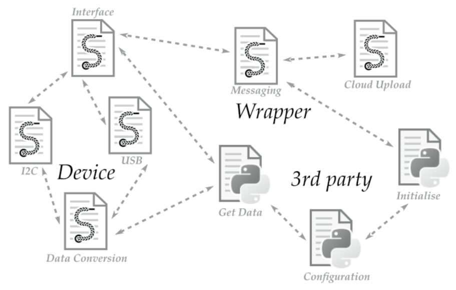

Fig. 6. Concurrent architecture and file structure proposed for wrapper software.

interfacing, illustrated in Fig. 2,Fig. 4 and Fig. 5 can be broken off as separate programs (see e.g. Porrmann et al. (2017), Thramboulidis et al. (2018) for a discussion of microservice architectures in manufacturing).

- **Implement 'crash-only' error handling**.

By adopting a 'crash-only' approach to the software, where we let code crash and then restart it, as described by Candea and Fox (2003), all we need to do is restart programs when our requests to them fail. This is the philosophy as implemented in the functional programming language Erlang ("Let it crash" Earle et al. (2005)) but has also been implemented as an architecture for object oriented, stateful, processes (Göri et al. (2014)).

The proposed concurrent, asynchronous architecture, broken up into microservices, is illustrated in Fig. 6.

Significantly the proposed approach to fault tolerance should enable iterative development on operational systems, without risking failure. This iterative approach has been shown to be highly effective in implementation case studies of RAMI4.0 (Hernández et al. (2020)), and we think is critical to reduce the capital and skills costs of incorporating digital technologies into a working factory.

5. CONCLUSION

We have developed software wrappers for the demonstrator platforms built to showcase the digital manufacturing on a shoestring project. We built software which was easy to re-use, parallelise, and modify, enabling users to connect many different sensors and data sources into a common platform. This was largely possible by the subdividing of the code into parallel modules which could easily be reused for reading sensor date, performing analysis or preparing data for upload to a database.

We propose further subdividing this code into smaller functional blocks, and running these as separate programs. This will allow us to encapsulate essential 'infrastructure' in stand-alone building blocks and provide isolation between process so that an error does not necessarily crash the entire program. This is critical to enable safe iterative development on a companies systems, so that digitalisation is not a one-off, upfront cost, but can be a gradual process.

This should enable SMEs to rapidly develop IT solutions for condition based maintenance and computer maintenance information systems, whilst not tying them into a closed, inflexible or inappropriately specialised solution.

Future work will implement the proposed architecture in software, test the reliability in a simulated environment and ideally be used in pilot studies.

This research was made possible by a grant from UK EPSRC, EP/R032777/1.

REFERENCES

Candea, G. and Fox, A. (2003). Crash-Only Software. In *9th Workshop on Hot Topics in Operating Systems*, May, 67–72. USENIX.

Carnero, M.C. (2012). Condition Based Maintenance in small industries. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, 45(31), 199–204. doi:10.3182/20121122-2-ES-4026.00010.

Cenedese, A., Frodella, M., Tramarin, F., and Vitturi, S. (2019). Comparative assessment of different opc ua open–source stacks for embedded systems. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 1127–1134.

Chaplin, J., Bakker, O., de Silva, L., Sanderson, D., Kelly, E., Logan, B., and Ratchev, S. (2015). Evolvable assembly systems: A distributed architecture for intelligent manufacturing. *IFAC-PapersOnLine*, 48(3), 2065 – 2070. doi:https://doi.org/10.1016/j.ifacol.2015.06.393. 15th IFAC Symposium on Information Control Problems inManufacturing.

Chen, B., Wan, J., Celesti, A., Li, D., Abbas, H., and Zhang, Q. (2018). Edge Computing in IoT-Based Manufacturing. *IEEE Communications Magazine*, 56(9), 103–109. doi:10.1109/MCOM.2018.1701231.

Earle, C.B., Fredlund, L.Å., and Derrick, J. (2005). Verifying fault-tolerant Erlang programs. *Erlang'05 - Proceedings of the ACM SIGPLAN 2005 Erlang Workshop*, 26–34. doi:10.1145/1088361.1088367.

Fielding, R. and Reschke, J. (2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. Internet Engineering Task Force Proposed Standards.

Fumagalli, L., Macchi, M., and Rapaccini, M. (2009). *Computerized maintenance management systems in SMEs: A survey in Italy and some remarks for the implementation of Condition Based Maintenance*, volume 13. IFAC. doi:10.3182/20090603-3-RU-2001.0416. URL http://dx.doi.org /10.3182 /20090603-3-RU-2001.0416.

Göri, G., Johnsen, E.B., Schlatte, R., and Stolz, V. (2014). Erlang-style error recovery for concurrent objects with cooperative scheduling. In T. Margaria and B. Steffen (eds.), *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, 5–21. Springer Berlin Heidelberg, Berlin, Heidelberg.

Hawkridge, G., Hernandez, M.P., De Silva, L., Terrazas, G., Tlegenov, Y., McFarlane, D., and Thorne, A. (2019). Tying Together Solutions for Digital Manufacturing: Assessment of Connectivity Technologies & Approaches. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2019-September, 1383–1387. doi:10.1109/ETFA.2019.8869411.

Hernández, E., Senna, P., Silva, D., Rebelo, R., Barros, A.C., and Toscano, C. (2020). Implementing rami4.0 in production - a multi-case study. In H.A. Almeida and J.C. Vasco (eds.), *Progress in Digital and Physical Manufacturing*, 49–56. Springer International Publishing, Cham.

Industrial Internet Consortium (2019). Industrial Internet Reference Architecture. URL https://www.iiconsortium.org/IIRA.htm. Retrieved 301/1/2020.

Lam, A.N. and Haugen, . (2019). Implementing opc-ua services for industrial cyber-physical systems in service-oriented architecture. In *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, volume 1, 5486–5492.

McFarlane, D., Ratchev, S., Thorne, A., Parlikad, A.K., de Silva, L., Schönfuß, B., Hawkridge, G., Terrazas, G., and Tlegenov, Y. (2020). Digital manufacturing on a shoestring: Low cost digital solutions for SMEs. *Studies in Computational Intelligence*, 853, 40–51. doi:10.1007/978-3-030-27477-1_4.

Object Management Group (2015). Data Distribution Service (DDS) formal specification version 1.4. URL https://www.omg.org/spec/DDS/1.4/PDF. Retrieved 30/1/2020.

OPC UA Foundation (2017). OPC UA Online Reference. URL https://reference.opcfoundation.org/v104/. Retrieved 30/1/2020.

Park, H.M. and Wook Jeon, J. (2019). Opc ua based universal edge gateway for legacy equipment. In *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, volume 1, 1002–1007.

Platform Industrie 4.0 (2016). RAMI4.0 - a reference framework for digitalisation. URL https://www.plattform-i40.de/PI40/Redaktion/EN/Downloads/Publikation/rami40-an-introduction.p-df. Retrieved 301/1/2020.

Porrmann, T., Essmann, R., and Colombo, A.W. (2017). Development of an event-oriented, cloud-based SCADA system using a microservice architecture under the RAMI4.0 specification: Lessons learned. *Proceedings IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, 2017-January, 3441–3448. doi:10.1109/IECON.2017.8216583.

Schönfuß, B., McFarlane, D., Athanassopoulou, N., Salter, L., de Silva, L., and Ratchev, S. (2020). Prioritising low cost digital solutions required by manufacturing SMEs: A shoestring approach. In *SOHOMA 2019: Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future*, volume 853, 290–300. doi: 10.1007/978-3-030-27477-1_22.

Sipsas, K., Alexopoulos, K., Xanthakis, V., and Chryssolouris, G. (2016). Collaborative maintenance in flow-line manufacturing environments: An industry 4.0 approach. *Procedia CIRP*, 55, 236 – 241. doi: https://doi.org/10.1016/j.procir.2016.09.013. URL http://www.sciencedirect.com/science/article/p-ii/S2212827116309477. 5th CIRP Global Web Conference - Research and Innovation for Future Production (CIRPe 2016).

SMC International Training (2020). HAS-200 Highly Automated System. URL www.smctraining.com /en /webpage /indexpage /517. Retrieved 30/1/2020.

Thramboulidis, K., Vachtsevanou, D.C., and Solanos, A. (2018). Cyber-physical microservices: An IoT-based framework for manufacturing systems. *Proceedings - 2018 IEEE Industrial Cyber-Physical Systems, ICPS 2018*, 232–239. doi:10.1109/ICPHYS.2018.8387665.

Upton, E. and Halfacree, G. (2014). *Raspberry Pi User Guide: 3rd Ed.* Wiley.