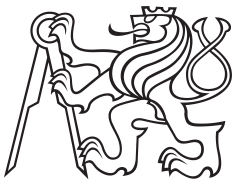
Bachelor Project



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Implementation of a distributed MES in Testbed for Industry 4.0

Petr Douda

Supervisor: RNDr. Jiří Vyskočil, Ph.D.
Field of study: Cybernetics and Robotics
May 2022

Acknowledgements

I would like to thank my supervisor RNDr. Jiří Vyskočil, Ph.D. for his guidance during my work on this thesis. I would also like to express my gratitude to all the members of Testbed for Industry 4.0 for allowing me to work with their state-of-the-art production line and all the indispensable assistance they provided during the testing of my MES.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 20. May 2022

Abstract

This bachelor thesis deals with the design and implementation of a distributed MES (Manufacturing Execution System) suitable for the control of an industrial production line in the Testbed for Industry 4.0 at CTU CIIRC. Its aim is to create a distributed MES, whose individual instances will be directly connected to the individual components of the production line, instead of traditional control (according to the automation pyramid: ERP → MES → SCADA → PLC → devices). The implementation is written in Python 3.10 programming language. It uses the OPC UA protocol to communicate with production line components. The implementation is fully integrated with other software components (Digital Twin, AI planner, ERP system) that have been developed for Testbed in the past. It also has a web interface that facilitates control and visualization of the manufacturing process (both the entire process and individual MES instances). Testing on the production line has shown that the system is fully functional and meets all specified requirements.

Keywords: MES, Industry 4.0, Python, OPC UA, HTTP/REST, AI, Digital Twin

Supervisor: RNDr. Jiří Vyskočil, Ph.D.
Intelligent Systems for Industry and Smart Distribution Networks
Czech Institute of Informatics, Robotics and Cybernetics
Czech Technical University in Prague

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací distribuovaného MESu (Manufacturing Execution System) vhodného pro řízení průmyslové výrobní linky v Testbedu pro Průmysl 4.0 při ČVUT CIIRC. Jejím cílem je vytvořit distribuovaný MES, jehož jednotlivé instance budou propojeny přímo s jednotlivými komponentami výrobní linky, namísto tradičního řízení (dle automatizační pyramidy: ERP → MES → SCADA → PLC → zařízení). Implementace je napsaná v programovacím jazyce Python 3.10. Ke komunikaci s komponentami výrobní linky používá protokol OPC UA. Implementace je plně integrovaná s dalšími softwarovými komponentami (Digitální dvojče, AI plánovač, ERP systém), které byly v minulosti pro Testbed vyvinuty. Dále má také webové rozhraní, které umožňuje systém ovládat a visualizovat stav výrobního procesu (jak celého procesu, tak jednotlivých instancí MESu). Testování na výrobní lince prokázalo, že systém je plně funkční a splňuje všechny specifikované požadavky.

Klíčová slova: MES, Průmysl 4.0, Python, OPC UA, HTTP/REST, AI, Digitální dvojče

Contents

Figures

Tables

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Douda Petr** Personal ID number: **483486**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Implementation of a distributed MES in Testbed for Industry 4.0

Bachelor’s thesis title in Czech:

Implementace distribuovaného MESu v Testbedu pro Průmysl 4.0

Guidelines:

1) Implement a Manufacturing Execution System (MES) in Testbed for Industry 4.0 at CTU using the programming language Python 3. Given the existing flexible Production Line at Testbed for Industry 4.0 at CTU, employ current advanced planning and scheduling methods, an already implemented Digital Twin, and OPC UA protocol for communication with industrial components. Consider LispPlan as an input.
2) Develop support for distributed processing/execution of LispPlan according to available access points for MES sub-units.
3) Implement a webserver for monitoring and control of such MES.

Bibliography / sources:

[1] Novák Petr, Vyskočil Jiří, Wally Bernhard - The Digital Twin as a Core Component for Industry 4.0 Smart Production Planning - IFAC-PapersOnLine, Germany, 2020.
[2] Mahnke Wolfhang, Leitner Stefan-Helmut, Damm Matthias - OPC Unified Architecture - Springer Science & Business Media, 2009
[3] Nguyen Quan - Mastering Concurrency in Python - Packt Publishing Ltd, 2018
[4] Larsen Kim G., Mikucionis Marius, Nielsen Brian - Online Testing of Real-time Systems Using UPPAAL - Linz Austria, 2004.
[5] McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M.; Weld, D., Wilkins, D. - PDDL---The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165 - Yale Center for Computational Vision and Control, New Haven, CT, 1998.

Name and workplace of bachelor’s thesis supervisor:

RNDr. Jiří Vyskočil, Ph.D. Department of Cybernetics FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **19.01.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until:
by the end of summer semester 2022/2023

RNDr. Jiří Vyskočil, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student's signature

Chapter 1

Introduction

"Currently, we are witnessing the 4^{th} industrial revolution that is related to the smart automation based on the use of Cyber-Physical Systems in industry, which are complemented with the Internet of Things and Artificial Intelligence (AI) technologies, that are customized for industrial application" [1]. This new Industry 4.0 (I40) requires development of new digitalized and highly flexible/modular systems in all areas of industry at all levels of the industrial pyramid[1]. One of the crucial systems is Manufacturing Execution System (MES) that is responsible for interpreting orders from the Enterprise Resource Planning (ERP) and executing their production. "The main functionalities of the MES are data acquisition and abstraction, detailed scheduling of operations, resource allocation and control, dispatching production to machines and workers, controlling product quality, and managing the maintenance of equipment and tools" [14].

A state-of-the-art MES should follow the I40 trends and comply with the I40 requirements [15] such as:

- Plan/schedule tasks in the production line according to its current state.
- Employ advanced AI planning.
- Implement advanced optimization algorithms for replanning/rescheduling the production and maintenance plans in case of failure
- Facilitate integration of autonomous equipment.

[1] https://www.researchgate.net/figure/The-automation-pyramid-according-to-the-ISA-95-model-The-five-levels-0-5-are-defined_fig2_326224890

- Use Open Platform Communication Unified Architecture (OPC UA) [6] for communication with industrial components.

The most prominent approach to designing a MES that would meet these requirements is the Multi-Agent or holonic paradigm. One of the proposed architectures is PROSA [4] which is based around the use of holons, autonomous units each having a specific function facilitating the production process using negotiation among themselves. An adaptation of this architecture is ADACOR [5] which has been utilized in several pilot studies. However this approach proved very difficult to implement in industrial practice as discussed in [7]. There is also an ongoing project in Testbed for Industry 4.0 at CIIRC CTU[2] trying to implement a Multi-Agent MES. This MES has been in development for about three years with many technological barriers still to overcome.

There are two main motivation factors for this thesis. Firstly, a new MES system is needed in Testbed for Industry 4.0 to make use of newly developed technologies such as central PyAutomationML[11] configuration and description file and Digital Twin. A MES which would enable better visualization of the production process and more control over it. Secondly, there is currently a gap in the market for MESes suitable for controlling I40 production lines.

Therefore the aim of this thesis is to design and implement a MES for the experimental production line located at Testbed. The MES should be written in Python 3, use OPC UA for communication, and make use of currently available systems at Testbed. It should be a distributed system to allow for greater flexibility and robustness and have a web interface to facilitate visualization and control. However, given the problems connected to development of Multi-Agent MESes so far, the MES should build on the concepts currently utilized in industry and try to extend them with the use of advanced AI planning, Digital Twins and decentralization instead of trying to change the paradigm entirely.

[2] https://www.ciirc.cvut.cz/cs/teams-labs/testbed/

Chapter 2

Testbed for Industry 4.0

In this chapter, all the components previously implemented in Testbed for Industry 4.0 at CIIRC CTU[1] are specified. These components and their respective interfaces/message types are depicted in Figure 2.1. This figure also shows how the proposed MES should fit into the existing system and interact with it. The individual components are further described in the following sections.

I have developed some of the systems described in this chapter. All such systems were developed during my previous work. They are related to this thesis, but they are not to be considered part of the thesis.

[1] https://www.ciirc.cvut.cz/cs/teams-labs/testbed/

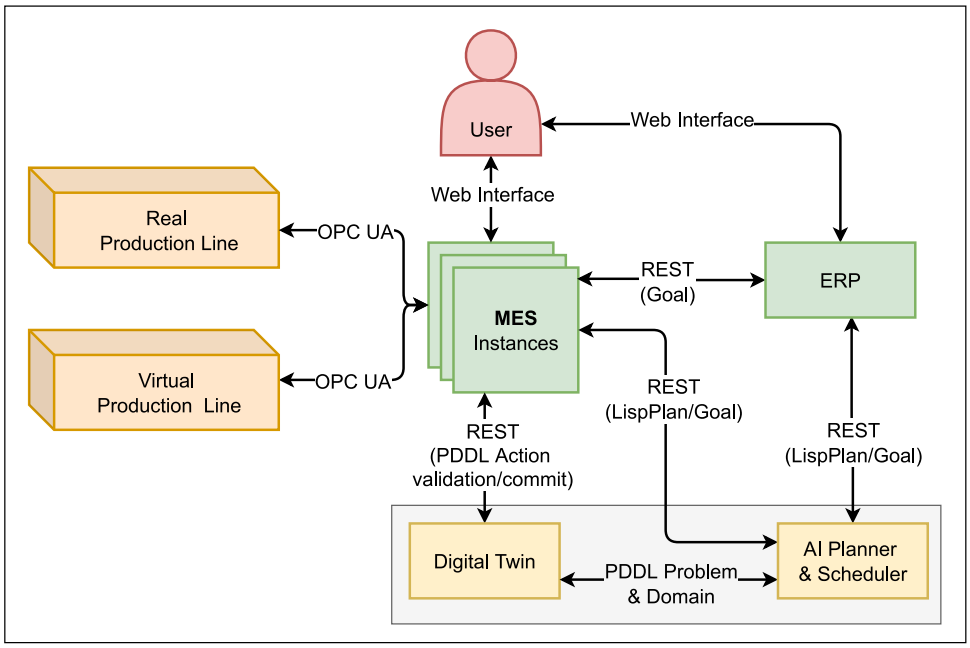


Figure 2.1: Testbed production line architecture

2.1 Testbed Production Line

The production line consists of three KUKA[2] Agilus robots, one KUKA Iiwa robot, and one KUKA Cybertech robot (which was added during my work on the thesis). A Montrac monorail transportation system (Montrac[3]) connects the robots and their workstations. There are currently four shuttles operating on Montrac, but the number of shuttles can be flexibly changed (Montrac can operate one to six shuttles). The physical structure of the line can be seen in Figure 2.2. Resources (robots and shuttles) can be controlled via their individual OPC UA[4] interfaces. A photo of the real production line can be seen in Figure 2.3.

[2]https://www.kuka.com/
[3]https://www.montratec.de/en/
[4]https://opcfoundation.org/about/opc-technologies/opc-ua/

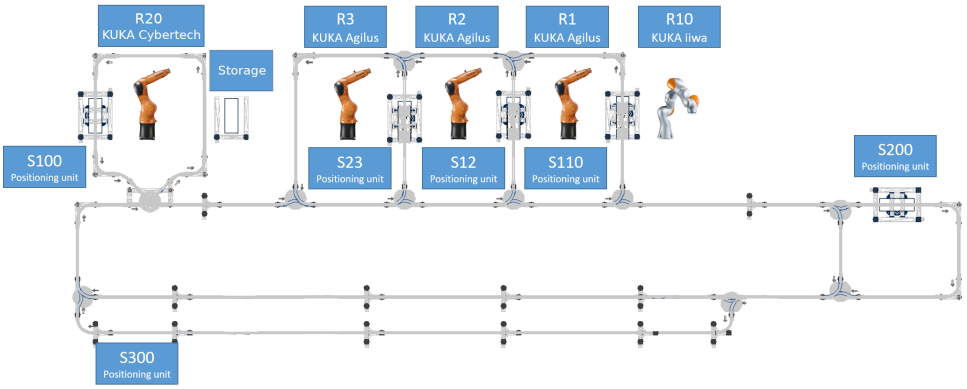


Figure 2.2: Testbed production line structure

2.2 Digital Twin and Planner

"Digital Twin (DT) refers to the virtual copy or model of any physical entity (physical twin) both of which are interconnected via exchange of data in real time. Conceptually, a DT mimics the state of its physical twin in real time and vice versa. Application of DT includes real-time monitoring, designing/planning, optimization, maintenance, remote access, etc." [16]

In Testbed a digital twin together with an AI planner [9] [12] is also implemented. The digital twin contains a specification of the current state of the production line in PDDL (Problem Domain Definition Language[5]) format. Along with the state, the digital twin has a domain specification which forms a mathematical model of the production line written in PDDL. It contains declarations of types, constants and predicates, and definitions of all PDDL actions that can be performed by the digital twin/production line. Each action is parameterized and specified by its pre-conditions and effects. An example of domain and state/problem specification can be seen in Figure 2.4. The digital twin contains an interpreter that can apply these PDDL actions (usually committed by MES) on the specified PDDL state, thus changing it. In this way a real-time state synchronized with the real production line can be kept by the digital twin. Using the interpreter the digital twin can also validate if a PDDL action can be safely executed on the real production line in a given PDDL state. Given a feasible PDDL goal (see Fig. 2.5) with current PDDL state the planner generates a production plan to reach this goal. The production plan (LispPlan [19] e.g. Fig. 2.6) consists of individual PDDL actions comprising a DAG (Directed Acyclic Graph) represented in Lisp syntax[6]. In this DAG, vertices are tasks/actions and edges are dependencies among these tasks/actions where the start of

[5]https://en.wikipedia.org/wiki/Planning_Domain_Definition_Language
[6]https://lisp-lang.org/

the edge is the requirement that needs to be satisfied before the target of the edge can be executed. Visualization of such DAG can be seen in Figure 4.3. This plan can then be interpreted by a MES. The services of the digital twin and the planner described above are accessible using HTTP/REST [7] GET/POST requests [19].

2.3 ERP

I have also implemented an Enterprise Resource Planning (ERP) system for the Testbed production line. In this context, it serves only to manage products, orders, and resources. It contains a database of products available to be made on the production line. This database can be updated with new products during run-time. The ERP also uses the planner to visualize which products can be produced/are available on the production line in its current state. The availability of products is updated using the planner whenever the state of the production line/digital twin changes. Currently, only 3D printed cars can be assembled on the production line. These cars are assembled from three parts, a cabin, a body and a chassis (part variants can be seen in Fig. 2.7). Each part can have different colors and the selection of these colors and also the part shapes can be easily extended. The cars in different color combinations can be ordered from the ERP systems website (see Fig. 2.8). These orders are queued for production on the Testbed production line. Next order to be processed by the production line can be obtained via a HTTP/GET request. The ERP returns an order in the form of a PDDL goal (e.g. Fig. 2.5). This goal can be directly passed on to the planner. After completing production of an order, the ERP can be notified using a HTTP/GET request. The ERP keeps records of all orders (queued, in production, and already produced). Another feature of the ERP is that it has an interface for communication with the digital twin and enabling/disabling resources (robots, shuttles) that can be used in planning/production.

2.4 PyAutomationML configuration

All the resources and related software components located at the Testbed production line are configured in a central PyAutomationML file [11]. PyAutomationML is an extension of AutomationML open format. “AutomationML is a comprehensive XML based object-oriented data modeling language. It

[7] https://restfulapi.net/

allows the modelling, storage and exchange of engineering models covering a multitude of relevant aspects of engineering. AML provides a comprehensive set of flexible mechanisms and innovations to model today's engineering aspects as well as future engineering aspects to come. Its language characteristics allow to model existing or new domain models." [2]

Each robot and shuttle are described in the PyAutomationML configuration file along with the specification of OPC UA servers controlling them according to the best-practice recommendation [3]. This means that there is a one-to-one correspondence to any interface that controls a resource in the PyAutomationML file. Furthermore, translations between PDDL actions and variable changes in the resources (via OPC UA interface) are also defined in the configuration file. These translation descriptions would not be possible without the Python extension provided by PyAutomationML.

The URLs and interfaces of the digital twin, planner, ERP and virtualization can be also found in the configuration file.

2.5 Virtual Production Line

During my previous work I have also implemented a virtualized version of the Testbed production line. Its purpose is to aid the development of new systems and make the debugging of new software easier. This virtualization has the same OPC UA interface as the real production line which makes it easily interchangeable with the real line from the perspective of a control system. This is achieved using the central PyAutomationML configuration file. It also automatically checks if the actions submitted via the OCP UA interface are feasible/valid using the digital twin. This makes it perfect for debugging.

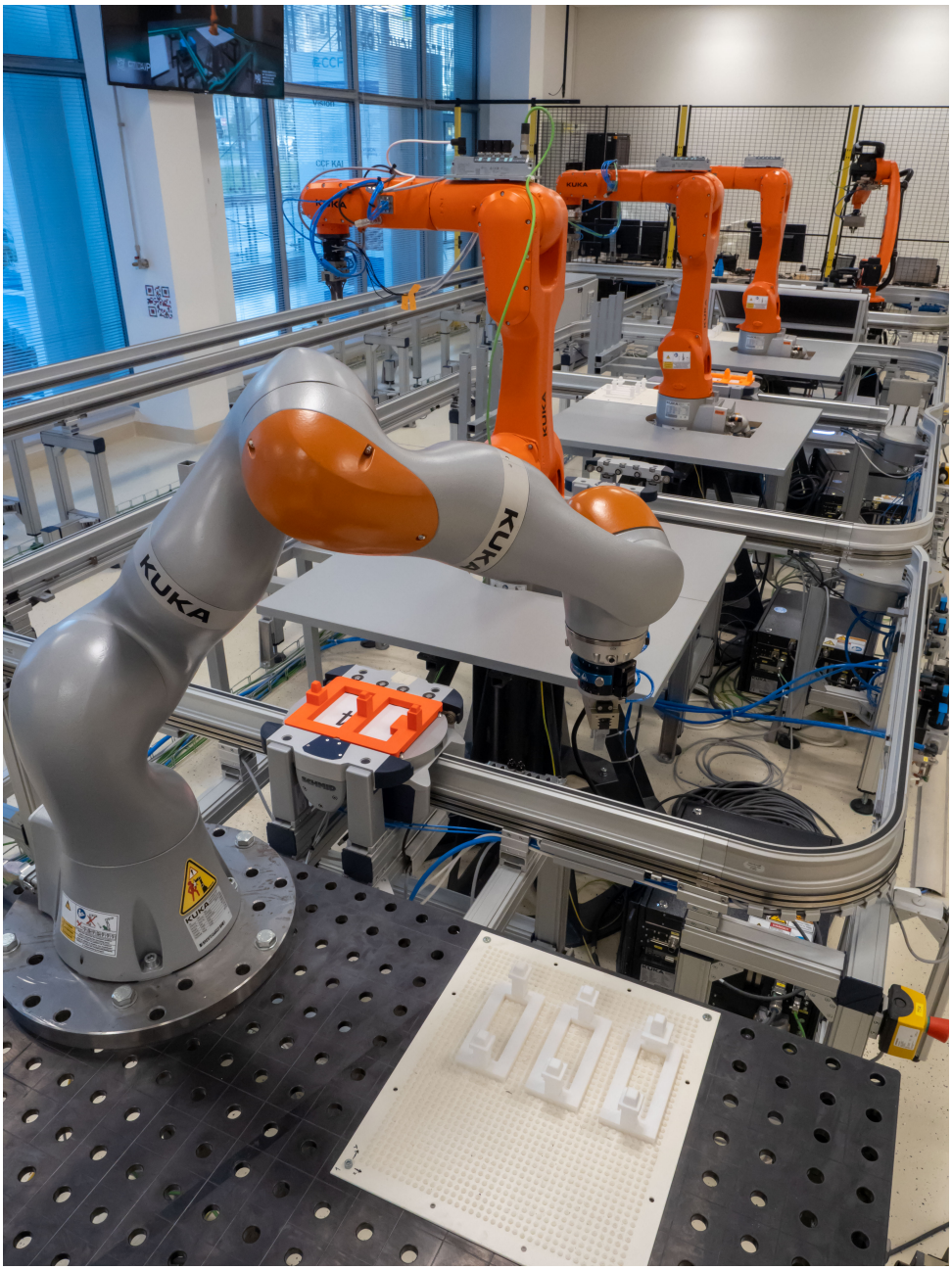


Figure 2.3: Testbed production line

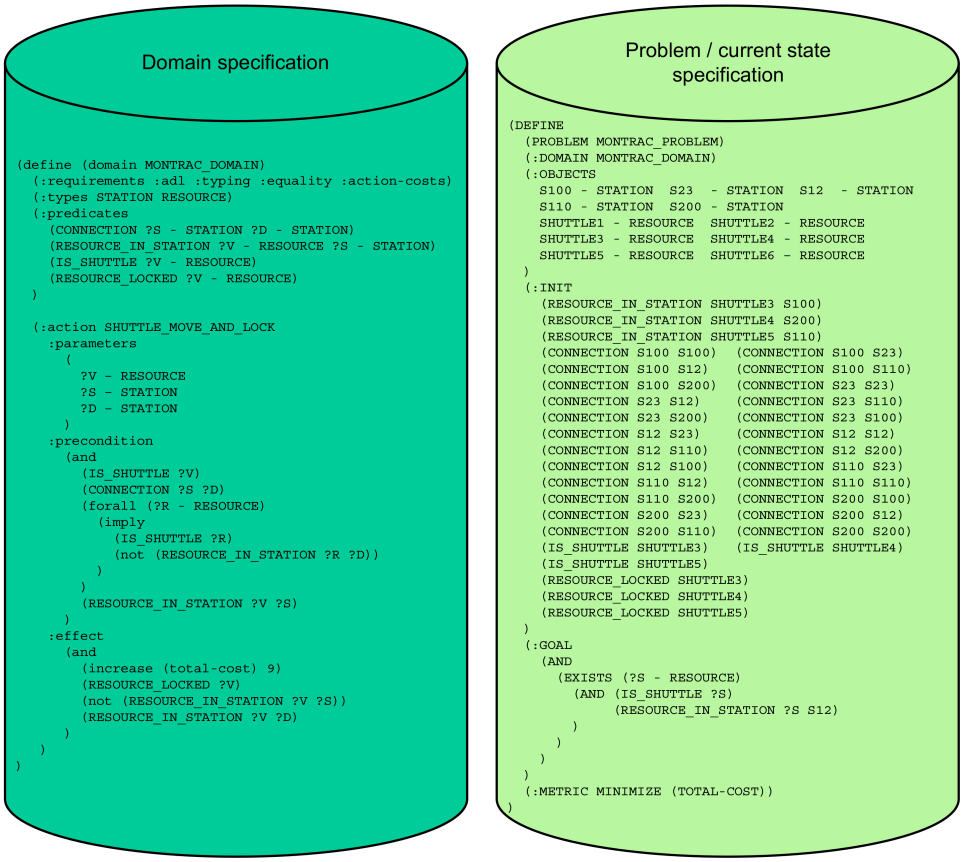


Figure 2.4: An example of a PDDL domain and state adopted from [12]

```
(:goal
    (and
        (exists
            (?P1 - POSITION ?P2 - POSITION ?P3 - POSITION ?S -
                RESOURCE)
            (and
                (RESOURCE_CONTAINS_PART_AT ?S
                    PART-TYPE-T-TRANSPORT
                    PART-BLACK-CHASSIS ?P1)
                (RESOURCE_CONTAINS_PART_AT ?S
                    PART-TYPE-T-TRANSPORT
                    PART-YELLOW-DUMPER ?P2)
                (RESOURCE_CONTAINS_PART_AT ?S
                    PART-TYPE-T-TRANSPORT PART-SILVER-CABIN
                    ?P3)
                (IS_SHUTTLE ?S)
                (RESOURCE_IN_STATION ?S S200)
            )
        )
    )
)
```

Figure 2.5: PDDL goal example

```
(define
    (task building_truck)
    (:location testbed.ciirc.cvut.cz)
    (define
        (task 0)
        (:location montrac)
        (:action (SHUTTLE_SWAP_AND_LOCK SHUTTLE2 SHUTTLE5
            S200 S23)))
    (define
        (task 1)
        (:requirements 0)
        (:location R3)
        (:action (ROBOTIC_PICK R3 SHUTTLE2 S23 X0_Y5_Z7_R0
            PART-WHITE-STAKEBED PART-TYPE-T-TRANSPORT))
    )
    (define
        (task 2)
        (:requirements 1)
        (:location R3)
        (:action (ROBOTIC_PLACE R3 R3_TABLE SVR3 X2_Y6_Z7_R0
            PART-WHITE-STAKEBED PART-TYPE-T-STORAGE)))
    (define
        (task 3)
        (:requirements 1 2)
        (:location R3)
        (:action (ROBOTIC_PICK R3 SHUTTLE2 S23 X0_Y21_Z7_R90
            PART-BLUE-CABIN PART-TYPE-T-TRANSPORT))
    )
    (define
        (task 4)
        (:requirements 2 3)
        (:location R3)
        (:action (ROBOTIC_PLACE R3 R3_TABLE SVR3 X-9_Y2_Z9_R90
            PART-BLUE-CABIN PART-TYPE-T-STORAGE))
    )
    (define
        (task 5)
        (:requirements 0)
        (:location montrac)
        (:action (SHUTTLE_SWAP_AND_LOCK SHUTTLE3 SHUTTLE5
            S100 S200)))
    (define
        (task 6)
        (:requirements 5)
        (:location R20)
        (:action (ROBOTIC_PLACE R20 SHUTTLE5 S100 X0_Y5_Z7_R0
            PART-SILVER-TANK PART-TYPE-T-TRANSPORT))
    )
)
```

Figure 2.6: LispPlan example (corresponds to the LispPlan visualized in Fig. 4.3, 4.2)

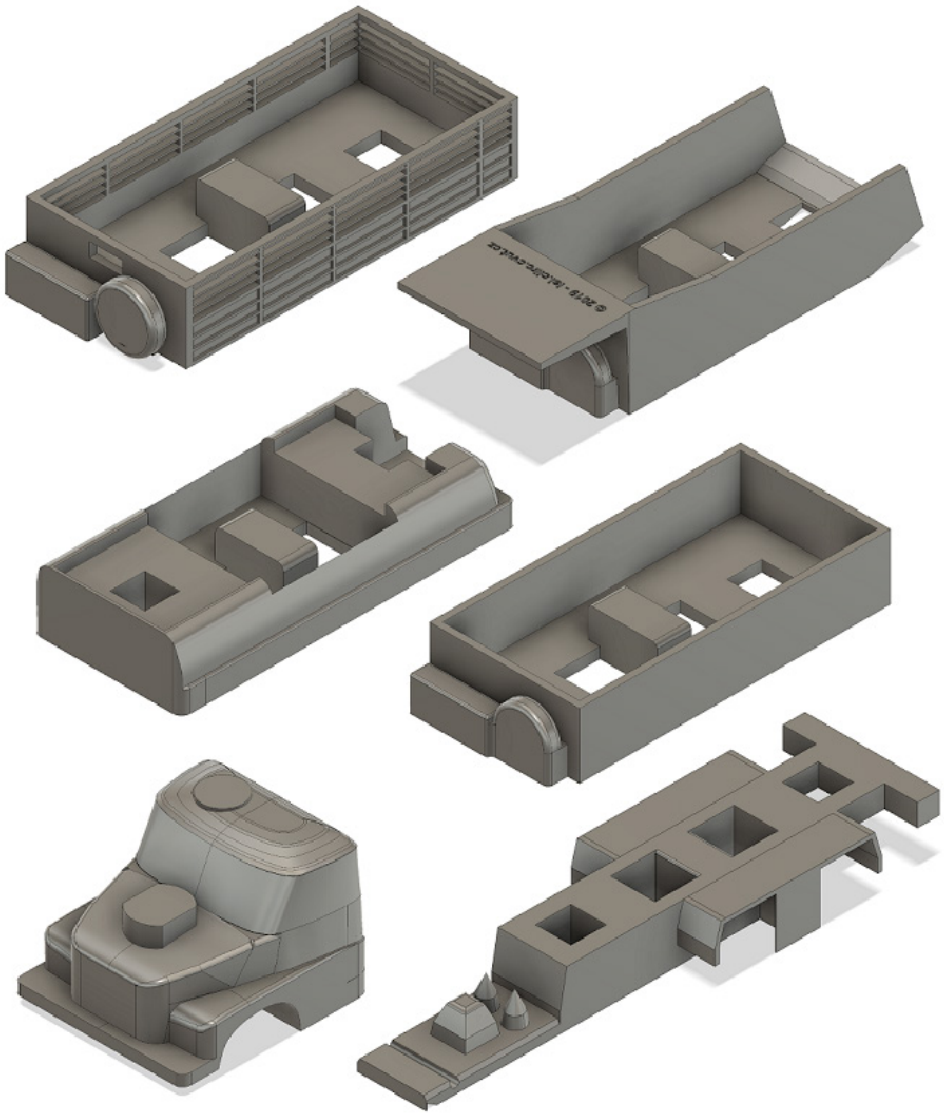


Figure 2.7: Parts that can be used to assemble the 3D printed car

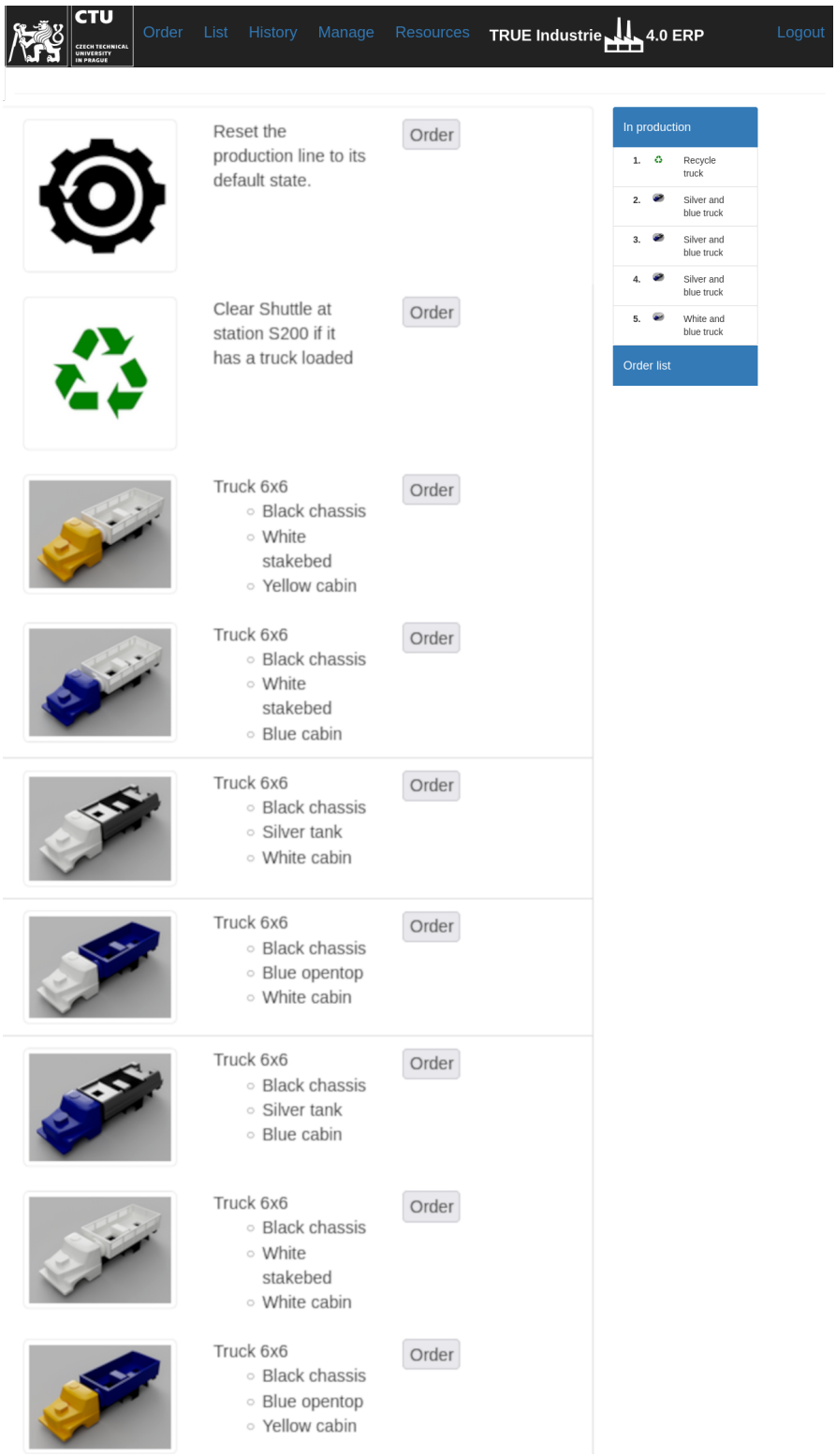


Figure 2.8: ERP order web page

Chapter 3

Algorithms

With regard to the requirements on an Industry 4.0 MES and the existing systems in Testbed, I designed two algorithms. First, we have the initialization Algorithm 1 used to extract the necessary configuration information from the central PyAutomationML configuration file and start the MES. The second is the algorithm to interpret and process a LispPlan using a distributed MES 2. This general algorithm is also used in an article [10] submitted for IEEE SMC 2022[1] that I am a co-author of.

3.1 Initailization

The *initialization* procedure described in Algorithm 1 has two arguments. The first argument, *location*, is the MES instance identifier in the form of a PDDL location [19]. The second argument, *config_file*, is a path to the central PyAutomationML configuration file. The file is then opened as *config* (line 2). After that, the configuration for the MES instance specified by *location* is found (line 3). Then all the specifications of resources (that the MES controls) are found (lines 6–12). Following this, the digital twin and ERP configurations are found. Using this information, an instance of MES is created. Finally, the web server controlling the MES instance is created and started.

[1] https://ieeesmc2022.org/

Each instance of MES (that is uniquely identified by its *location*) is started by Algorithm 1 at its specified domain. All instances together form a network of MESes – distributed MES. An example of such a network of MES instances and I40 devices can be found in Figure 5.1. In this figure, MES_0 is considered as a main MES instance and is the only instance that can accept a new LispPlan from the ERP and afterwards redistribute parts of it to subordinate MESes. As can be seen in the figure, this redistribution continues gradually on all instances of the network with each instance communicating only with its parent and direct children. After redistribution, each instance starts processing its part of the LispPlan using the devices available to it. The processing of a LispPlan by such a network of MESes is described in Algorithm 2. This algorithm is used by all instances in the network.

Algorithm 1: Initialize the MES instance

```
1  procedure initialization(location : string identifying MES, config_file : path
     to PyAutomationML);
2  config = Load config_file;
3  mes_config = Find the MES configuration in config using location;
4  mes_opcua_clients = ∅;
5  mes_resources = ∅;
6  foreach OPC UA server in config do
7      resources = Find all resources in config that belong to both location and
         OPC UA server;
8      if resources ≠ ∅ then
9          rc = OPC UA client for OPC UA server;
10         Add rc to mes_opcua_clients;
11         foreach resource in resources do
12             Add resource to mes_resources;
13 dt = Find Digital Twin specification in config;
14 erp = Find ERP specification in config;
15 mes = Instantiate MES with: mes_opcua_clients, mes_resources, config_file,
     dt;
16 Start the web server with: mes, config_file;
```

3.2 LispPlan execution

The procedure *MES* described in Algorithm 2 has two arguments: *plan* which is a LispPlan or its redistributed part and *source* which is URL of a MES where the LispPlan was delegated from or `None` if *plan* is the main LispPlan. First, all parts of *plan* that can be delegated to another MES instance are redistributed and the addresses of these MES instances are stored in *mes_pool* (lines 5–8). After that, all remaining tasks that were not delegated are processed by the

MES instance using the resources that are under its control (lines 9–33). The tasks are all processed concurrently in separate threads. First, the task state is initialized to `QUEUED`, then after all requirements for processing the task are met, the state is set to `IN_PRODUCTION` and the task is executed on the production line. If its processed successfully its state is set to `DONE` otherwise it is set to `FAILED`. Each change in task state is immediately communicated to all MES instances in *mes_pool*. Also all task updates received from other MES instances are reflected in the currently processed plan and redistributed to MES instances in *mes_pool*. The procedure ends if either all tasks are `DONE` or `FAILED`. If any of the tasks fail the procedure can be restarted after the problem that caused the failure is fixed.

3.2.1 Correctness and termination

First I will prove the partial correctness, then termination of Algorithm 2 and last the total correctness. For the sake of simplicity, let us assume that the plan is a finite DAG with states of the tasks as vertices and requirements as directed edges.

Definition: The requirements of vertex v are *met* if all vertices that have an edge directed to vertex v are in the `DONE` state.

Definition: DAG G is *consistent* if every vertex v from G is in one of the following states:

- `QUEUED` if the requirements of v are not *met*
- `IN_PRODUCTION` if the requirements of v are *met* and v has not yet been successfully processed
- `DONE` if the requirements of v are *met* and v has been successfully processed in state `IN_PRODUCTION`

Definition: *Step* is when Algorithm 2 changes a *consistent* DAG G into a *consistent* DAG G' where G' differs form G in at least one vertex's state.

Partial correctness means that *step* of Algorithm 2 is correct.

We prove partial correctness by exhaustion. From state `QUEUED` a vertex's state is changed to `IN_PRODUCTION` only if all the requirements are satisfied (line 13). From state `IN_PRODUCTION` the state is changed to `DONE` only if the processing of vertex was successful (lines 19, 23, 24). If the processing was not successful the problem is resolved manually and the whole algorithm is failed. Therefore, after doing a state change/step on a *consistent* DAG the resulting DAG is also *consistent* or the algorithm failed. □

To prove termination let us assume that the processing of a vertex is finite. Because there is only a finite number of possible combinations of vertex states for a finite DAG G and because transitions between vertex states are possible only in one direction (a reverse change of state is not possible) the number of *steps* (each will change G by definition) for G must be also finite (same combinations of vertex states in G cannot be reached by doing *steps*). □

To prove total correctness let us assume that the input DAG G has all vertices in state `QUEUED` or `IN_PRODUCTION` if they do not have any input edges. Such G is *consistent* by definition and is the result of parallel execution of code on lines 12–14. By induction, after doing a finite number of *steps* the algorithm will terminate with G' in a *consistent* state or failed. If it has not failed all vertices of G' must be in state `DONE`. This statement can be proved by contradiction. □

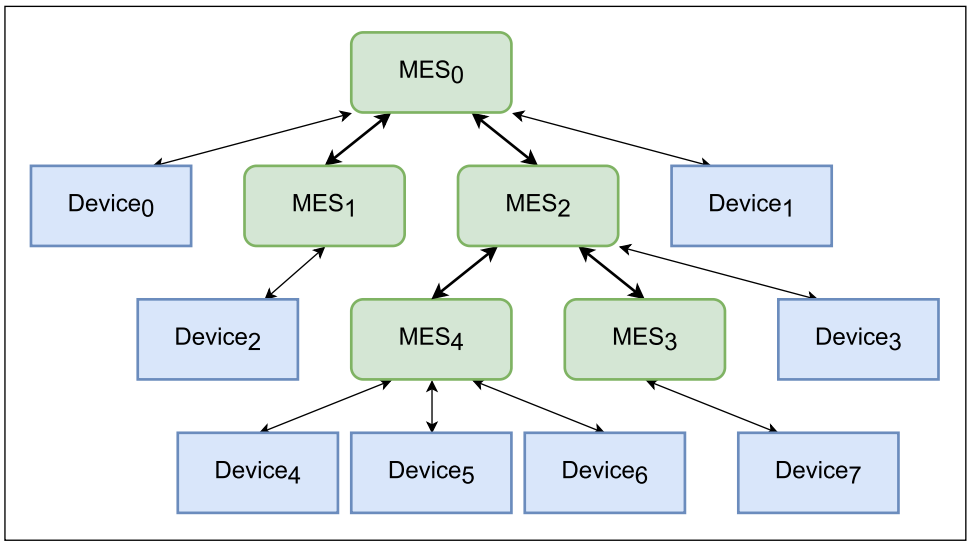


Figure 3.1: An example of distributed MES network with I40 devices

Algorithm 2: Processing of a LispPlan with distributed MESes.

```
1  procedure MES(plan: LispPlan, source[a]: identification of where the plan came
     from)
2  mes_pool = ∅ /*A set that can contain only MESes that are direct children or
     parent.*/;
3  if source is not None then
4      Add the parent MES from source to mes_pool;
5  foreach task in plan using DFS or BFS ordering[b] do
6      if task is not yet delegated and there is another ready MES m that is able[c] to
         process task then
7          Delegate the task and all its sub-tasks to MES m as a new standalone
             sub-LispPlan;
8          Add MES m to mes_pool;
9  foreach task in plan where task is not delegated to another MES do
10     if task can be processed by the current MES then
11         begin a new thread as t
12             Set the state of task to QUEUED and sync it with MESes from mes_pool;
13             wait until all task requirements are fulfilled[d];
14             Set the state of task to IN_PRODUCTION and sync it with MESes from
                 mes_pool;
15             if task contains an action then
16                 if the action is successfully validated by the Digital Twin then
17                     Process the action on the real hardware;
18                     if the action is processed successfully then
19                         Set the state of task to DONE and sync it with MESes from
                             mes_pool;
20                         Commit changes to the Digital Twin;
21                         terminate t;
22             else
23                 wait until all its sub-tasks are processed or some sub-task is failed;
24                 if all sub-tasks were processed successfully then
25                     Set the state of task to DONE and sync it with MESes from
                         mes_pool;
26                     terminate t;
27             Set the state of task to FAILED and sync it with MESes from mes_pool;
28             Report an error and resolve the problem with the Digital Twin;
29             terminate t;
30     else
31         /*task cannot be processed on the current MES infrastructure*/;
32         Set the state of task to FAILED and sync it with MESes from mes_pool;
33         Report an error and terminate the MES procedure.
34 repeat
35     if task state changes[e] then sync plan and redistribute the change to MESes in
         mes_pool;
36 until plan is completed[f] or MES is stopped;
```

[a] to distinguish between a main LispPlan (*source*==`None`) or a redistributed part of that LispPlan on a specific sub-MES.

[b] any parent must be processed prior to its children

[c] determined from *task* location

[d] all tasks specified in requirements are in `DONE` state and the parent task is in `IN_PRODUCTION` state

[e] sync is received from another MES

[f] All tasks including all sub-tasks in *plan* are in `DONE` state.

Chapter 4

PyMES Implementation

I implemented a distributed MES, called PyMES, using the algorithms described in Chapter 3. This chapter describes how I implemented PyMES. It contains information on the overall functionality of the system and specific solutions and classes that I programmed, and their relation to the algorithms described in Chapter 3. A high level overview of classes described in this chapter and their relations can be seen in UML[1] notation in Figure 4.1.

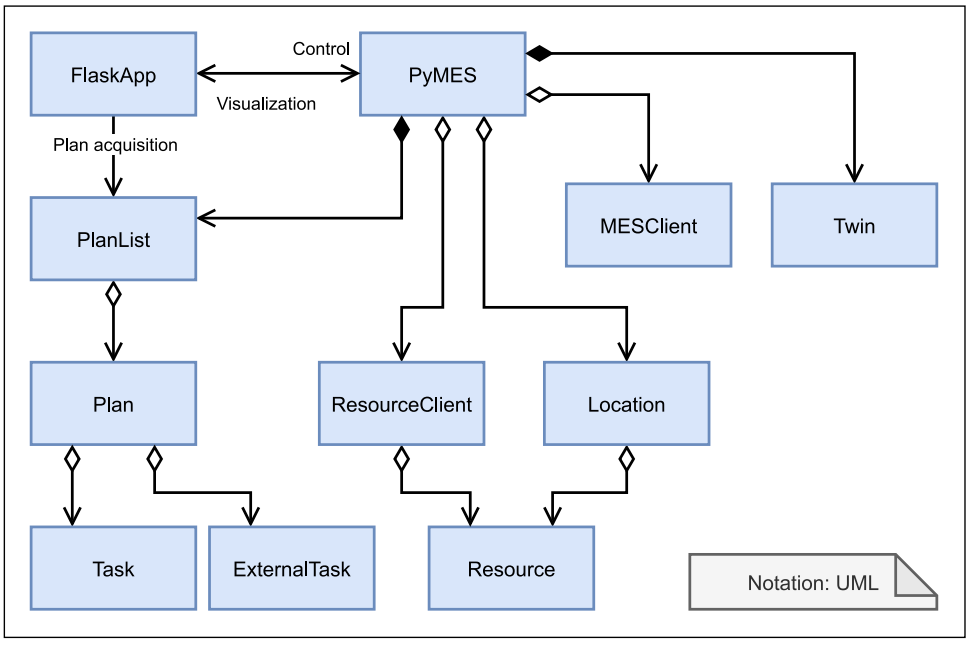


Figure 4.1: Class diagram of the PyMES in UML notation

[1] https://www.uml.org/

4.1 Parallelization

PyMES takes a LispPlan (e.g. Fig. 2.6) and controls the production line according to it. Because the resources on the production line can be operated in parallel, PyMES must allow parallel execution and supervision of tasks. Natural ways to deal with asynchronous code execution in Python 3 are either Multithreading or Multiprocessing [8]. I chose to use Multithreading[2] because it makes access to shared variables easier and safer (due to GIL[3]). Also, the application is not very computation intensive and is very I/O intensive, which means that the overall performance of the application will not be negatively affected by the sequential nature of Python *Thread* implementation (due to GIL). Because of this, most functions in this application are made (thread safe) so they can be called from multiple threads without negative interference and side effects.

4.2 LispPlan

To work with LispPlans easily in Python 3 I needed a suitable data structure to access the LispPlan. I chose to implement a tree structure consisting of objects corresponding to individual tasks in the LispPlan. To accomplish this, I have created two classes *Task* and *Plan* that can be found in *planning.py*. To deal with the requirements that can be located in a different part of the LispPlan/PyMES instance, I implemented the *ExternalTask* class that has the basic properties of the *Task* class but does not contain any execution specific task data.

4.2.1 *Task* class

The *Task* class inherits from the *NodeMixin* class from the publicly available Python *Anytree* module[4]. This allows us to construct a *Task* tree structure that is one-to-one with the structure of tasks in a LispPlan. Each *Task* instance represents one task in the plan and contains information about both – its parent task and children tasks. This means that from any given *Task*

[2] https://docs.python.org/3/library/threading.html
[3] Global Interpreter Lock
[4] https://anytree.readthedocs.io/en/2.8.0/

instance, all its subtasks are also available. *Task* class has the following attributes:

__init__*(task_id, plan, location, requirements, action, parent, children, virtual)*
1 Store arguments to their respective attributes;
2 Create local identifiers (with prefix *local_*)from the absolute identifiers passed as arguments;

- *virtual* which is set to `True` if *Task* is not to be rendered during visualization otherwise `False`.
- *task_id* contains the tasks *absolute identifier*[5] string from the LispPlan and needs to be passed as an argument on initialization of the object.
- *plan* which contains the *Plan* class instance to which the task belongs and needs to be passed as an argument on initialization of the class.
- *location* contains the *absolute location* string from the LispPlan and can be passed as an optional argument on initialization of the class.
- *requirements* containing a list of the *Task*/*ExternalTask* class instances corresponding to the *requirements* 5 of the LispPlan and can be passed as an optional argument (default is empty list) on the initialization of the class.
- *action* which contains the task's PDDL action and can be passed as an optional argument on initialization (default is *None*) of the class.
- *parent* which contains the parent task and can be passed as an optional argument (default is *None*) on initialization of the class.
- *children* which contains the list of child tasks and can be passed as an optional argument (default is empty list) on the class initialization.
- *state* contains the information about tasks status. Possible values are "`queued`", "`in_production`", "`done`" or "`failed`". At initialization of the class the value is set to "`queued`".
- *delegated* which is set to `True` if *Task* has been delegated to another PyMES otherwise `False`.
- *mes* which is set to `True` if *Task* is the same as the location of the PyMES instance, otherwise `False`.

[5] See specification of LispPlan [9]

As tasks in a LispPlan support both *local* or *absolute identifiers Task* class, it also has its local identifier that is stored in attributes with *local_* prefix in its name (e.g., let us assume that a class instance has an attribute *location* containing `'R2.TESTBED.CIIRC.CVUT.CZ'` and then it also has the corresponding attribute *local_location* containing `'R2'`). The following thread-safe methods are available for this class:

- *set_state(state: String)* method is used to set the state attribute of the task in a thread safe manner[6].
- *get_state()* → *String* method is used to retrieve the state attribute of the task in a thread-safe manner.
- *can_be_processed()* → *String* method which returns "`y`" if all the requirements for the task to start processing have been satisfied; otherwise *String* name of the first requirement that has not been satisfied.
- *view(order_by: String)* method displays a picture of the plan using *Graphviz*[7] tool in a new window. It has an optional argument *order_by* (the default is `tasks`). A detailed description can be found in the text below.
- *render(order_by: String)* → *String* method returns PNG 7 as *String*. It contains a picture of the plan rendered by the *Graphviz* tool. It has an optional argument *order_by* (the default is `tasks`). A detailed description can be found in the text below.
- *get_lisp()* → *String* method returns a new LispPlan representing *Task* and all its sub-*Tasks*, thus allowing us to split the LispPlan.

Because LispPlan has a recurrent structure, a conventional directed graph[8] is not suitable to represent it without a loss of information about task inheritance. Therefore, I used Supergraphs [13] to display LispPlans. To create SuperGraphs in Python, I used a custom module that can be found in the file *supergrapher.py*. It converts a supergraph represented by a Python dictionary into an image using *Graphviz* 7. The SuperGraph dictionary is constructed recursively when calling the method *view* or *render* from the *Task* class on all its subtasks. Two options are available for plan visualization. Either the graph edges represent the parent-child connections between tasks (*order_by* argument value is "`tasks`") or they link the task with its requirements (*order_by* argument value is "`requirements`"). The difference can be seen in Figures 4.2 and 4.3. Using the second option allows us to easily see the

[6] Using a *Threading Lock*
[7] https://graphviz.org/
[8] https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

sequence in which the tasks will be processed. In these images, the tasks are also color-coded. Gray tasks are yet to be processed, yellow are in processing, green are already processed, and red are failed. For better readability, these colors are picked from a palette suitable for color-blind people.

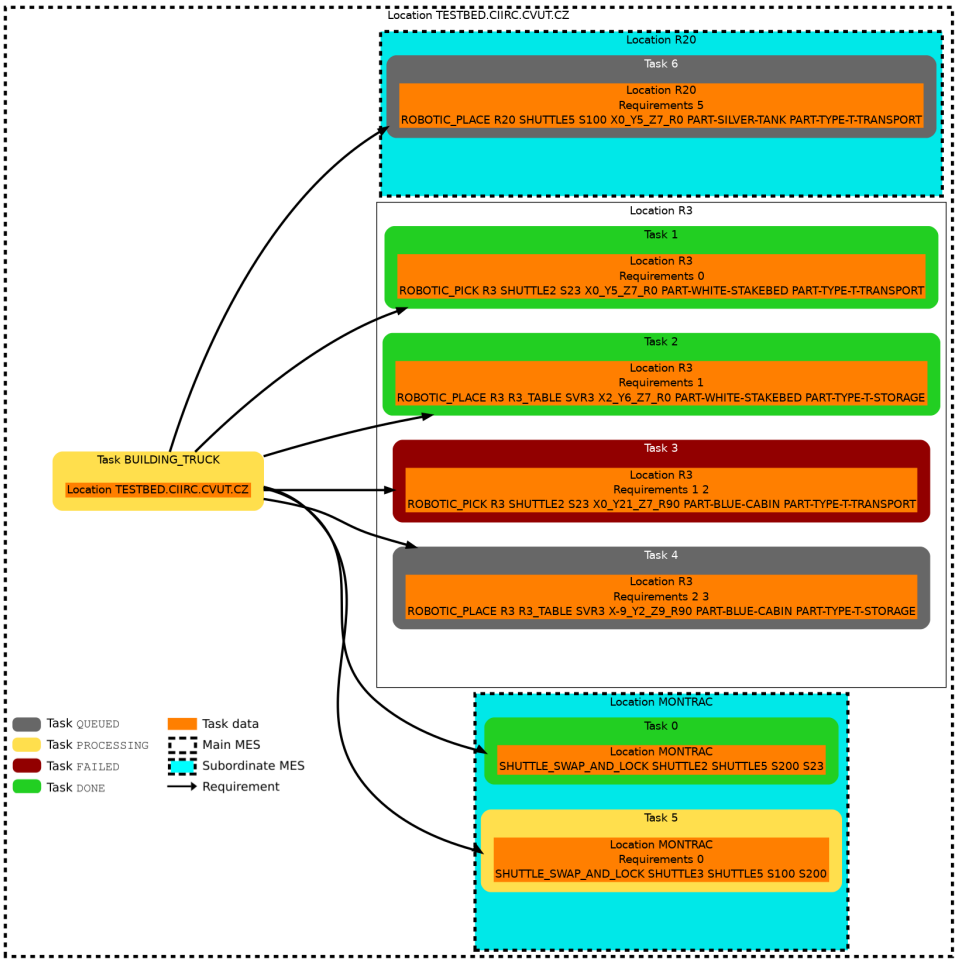


Figure 4.2: Supergraph example parent-child (visualization of LispPlan 2.6)

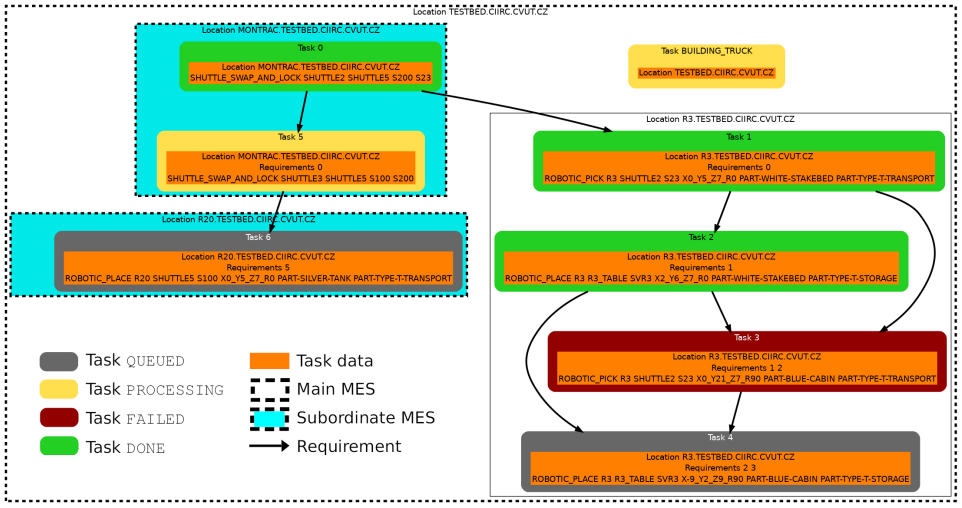


Figure 4.3: Supergraph example requirements (visualization of LispPlan 2.6)

4.2.2 *ExternalTask* class

Each *ExternalTask* instance represents one requirement of a task in a plan and contains only information about its id and state. This class is used for requirements from a different plan that cannot be instantiated as a *Task* class. *ExternalTask* class has attributes:

__init__(*task_id: String*)
1 *virtual* = `True`;
2 *task_id* = task_id;
3 *local_task_id* = task_id;
4 *state* = "`queued`";

- *virtual* which is set to `True` and means that the task will not be rendered during visualization.
- *task_id* contains the tasks *absolute identifier*[9] string from the LispPlan and needs to be passed as an argument on the initialization of the object.
- *local_task_id* also contains the tasks *absolute identifier* same as *task_id*.
- *state* contains the information about tasks status. Possible values are "`queued`", "`in_production`", "`done`" or "`failed`". At initialization of the class the value is set to "`queued`".

[9] See specification of LispPlan [9]

4.2.3 *Plan* class

The *Plan* class encapsulates the tree structure created from the *Task* class instances and provides some additional information and methods relevant to the LispPlan as a whole. I decided to implement this class to save memory on data that are common to all tasks. It also makes it clear whether the whole plan or just a subset of tasks are being referenced and, therefore, it makes the code more readable. It has the following additional attributes and methods:

__init__*(plan: String, plan_id: String, update_callback: Function)*

```
1 Store arguments to their respective attributes;
2 root = construct_tree(plan);
3 link_requirements();
```

- *external_tasks* which is a dictionary that is filled with instances of the *ExternalTask* class for all the requirements that were not found in the LispPlan on initialization.
- *root* stores the tree constructed by *construct_tree()*.
- *plan_id* attribute which contains the unique identifier of a plan and needs to be passed as an argument on initialization of the class. The identifier is obtained from the ERP along with the PDDL goal.
- *update_callback* attribute which can contain a function to be called every time a task belonging to the plan changes its state. It can, for example, display the plan on screen using the *view* method. It can be passed as an optional argument on initialization of the class.
- *iter_pre_order()* $\rightarrow$ *List* method which returns a list of all tasks and subtasks when traversing the task tree depth-first from left to right.
- *construct_tree(plan: String)* $\rightarrow$ *Task* method is automatically called on initialization of a *Plan* class. It takes a LispPlan string as an argument and returns the initialized task tree. The task tree is constructed recursively.
- *link_requirements()* method is automatically called on initialization of a *Plan* class. It links the corresponding *Task* or *ExternalTask* instances to the *requirements* attribute of the tasks in the task tree stored in *root*.

4.3 *PlanList* class

When a LispPlan is delegated (see Alg. 2 line 7) to another PyMES instance (according to the locations) it can be split into multiple parts (several tasks can have same location as can be seen in Fig. 2.6 tasks 0 and 5). To handle collection of these LispPlan parts and their recombination into *Plan* class instances I have implemented a thread-safe class called *PlanList* that can be found in *planning.py*. It has following methods and attributes:

___init___()
1 Initialize *list*;
2 Initialize *lock*;

- *list* is a Python *List* that stores the individual parts of LispPlan as *Tuples* (*String* part of the LispPlan, *String* its unique identifier from obtained from the ERP order). The unique identifier is useful for debugging and would be necessary for handling multiple LispPlans.
- *lock* is a Python *Threading Lock* used to make the list thread-safe.
- *push((plan: LispPlan, id: String): Tuple)* method is used to add a new LispPlan along with its unique identifier to *list*.
- *pop()* $\rightarrow$ *Plan* method takes all LispPlans with the same identifier from *list*, combines them into one LispPlan *String l*, removes them from *list*, creates a *Plan* class instance from *l* and returns it. If there are more parts with different identifiers, the first part/identifier is used.

4.4 Resources

Communication with the resources is done via OPC UA using the module *opcua*[10]. To interact with the production line easily, I have implemented three classes *ResourceClient*, *Location*, and *Resource*. These can be found in the files *resource_client.py*, *locations.py*, and *resource.py*, respectively.

[10] https://python-opcua.readthedocs.io/en/latest/

4.4.1 *ResourceClient* class

The *ResourceClient* class encapsulates *opcua* client which handles the connection to the resources. It also takes care of creating, removing and servicing subscriptions of *opcua* variables. Each class instance corresponds to one OPC UA client connected to only one server. Currently, the client connects to the server anonymously. Each client services one or more *Resource* class instances. The class has a Python *Logger*[11] built in to record communication via the OPC UA interface. The class has the following attributes:

__init__*(aml_root: AmlElement)*
1 *aml_root* = aml_root;
2 Load attributes from *aml_root*;
3 Initialize *opcua_client*;
4 Start *queue_worker_thread*;

- *aml_root* which contains the client identifier and needs to be passed as an argument on initialization of the class.
- *name* which contains the client identifier and is found automatically on initialization using *aml_root*.
- *endpoint* which contains the endpoint of the OPC UA server to which the client connects and is found automatically on initialization using *aml_root*.
- *opcua_client* which contains the opcua[12] client instance with *endpoint* used for communication over OPC UA.
- *client_queue* an asynchronous Python *Queue*[13] class instance where all *opcua* subscription events from the server are pushed. This is necessary because only simple operations, such as pushing an event to the queue, can be done in the *opcua* subscription handler. Each *ResourceClient* has its own *client_queue*.
- *queue_worker_thread* is a Python *Thread* that continuously takes the events from *client_queue* and pushes them to the *resource_queue* of the *Resource* class instance to which the event belongs to. It is automatically started upon initialization of the class.

[11] https://docs.python.org/3/library/logging.html
[12] https://python-opcua.readthedocs.io/en/latest/
[13] https://docs.python.org/3/library/queue.html

4.4.2 *Resource* class

The *Resource* class represents the robots and shuttles available at the production line. Each *Resource* class instance corresponds to one of the production lines resources. This class is responsible for task execution. It has a Python *Logger* built in to record the operations performed by the resource. The *Resource* class has the following attributes:

__init__*(client: ResourceClient, aml_root: AmlElement)*
1 Store arguments to their respective attributes;
2 Initialize *variables*;
3 Start *queue_worker_thread*;

- *client* which is the *ResourceClient* class instance that controls the resource via the OPC UA and needs to be passed as an argument on initialization of the class.
- *aml_root* which is the PyAutomationML element that stores all the necessary information about the resource and needs to be passed as an argument on initialization of the class.
- *state* which stores the current state of the resources (`available` or `busy`).
- *resource_queue* an asynchronous Python *Queue* object where all *opcua* subscription events relevant to this specific resource are pushed by the corresponding *ResourceClient* instance via the *queue_worker_thread*. Each resource has its own *resource_queue*.
- *variables* which is a list of all the *opcua* variable class instances only (not object or folder nodes).

The main feature of this class is the *process_task* method. This method takes a *Task* class instance as an argument and handles its execution on the production line. First, the PDDL action is translated into a dictionary containing *opcua* variable identifiers and values that need to be written to the respective variables. This is done via the translation methods provided in the PyAutomationML configuration file. After that, all the values are written to the corresponding OPC UA server variables, then the operation is started via setting the DataReady variable to `True`. After that the resource waits until all events signaling that the resource has finished the operation are sent or until a timeout is reached. Last, the method returns `True` if the operation finished successfully and `False` otherwise. this method implements the code on line 17 in Algorithm 2.

4.4.3 *Location* class

The *Location* class instances represent the individual PDDL locations[19]. This class is an intermediate step (in task execution) necessary to handle compound operations that use more than one resource. An example of operation with two resources can be found in task 0 in LispPlan 2.6 that uses the resources SHUTTLE2 and SHUTTLE5. The class has a Python *Logger* built in for debugging. The class has the following attributes:

__init__*(aml_root: AmlElement, resources: List)*
1 Store arguments to their respective attributes;

- *aml_root* is the element of PyAutomationML that contains configuration information for the location. *aml_root* needs to be passed as an argument on initialization of the class.
- *name* which contains the PDDL identifier of the location and is set automatically on initialization using *aml_root*.
- *resources* which contains the *Resource* class instances available to the *Location* and needs to be passed as an argument on initialization of the class.

This class also has a *execute_task()* method that has one argument *task* of class *Task*. It implements processing of *task* as described in the code at lines 11–29 in Algorithm 2. The method processes *task*'s operation using the appropriate *Resource* class instances calling the method *process_task()* of the *Resource* class.

4.5 *DigitalTwinClient* class

I have implemented a simple Python class called *DigitalTwinClinet*. It can be found in the file called *digital_twin.py*. *DigitalTwinClient* class handles the communication with the digital twin and planner using HTTP/REST. It takes the digital twin configuration PyAutomationML element as an initialization argument. It provides the following methods to interact with the digital twin and planner:

__init__(*aml_root: AMlElement*)
1 Load configuration from *aml_root)*

- *check_action()* returns `True` if the action passed as its argument is valid/executable in the current state of the digital twin and `False` otherwise. It is used in Algorithm 2 line 16.
- *get_state()* → *String* returns the current PDDL state.
- *do_action(action: String)* → *Bool* returns `True` if the action passed as its argument was successfully committed to the digital twin and `False` otherwise. It is used in Algorithm 2 line 20.
- *check_goal(goal: String)* → *Bool* takes a PDDL goal as an argument and returns `True` if the goal is reachable from the current PDDL state and `False` otherwise.
- *get_plan()* → *String* takes a PDDL goal as an argument and returns a LispPlan as *String* to fulfill this goal if such plan exists otherwise returns `None`.

4.6 *PyMESClient* class

For communication with other *PyMES* class instances i have written the *PyMESClient* class located in *py_mes_client.py*. Each instance of this class communicates with one instance of *PyMES*. It the has following methods and attributes:

__init__(*address: String*)
1 Store arguments to their respective attributes;

- *address* which is the URL of the *PyMES* instance and needs to be passed as an argument on initialization of the class.
- *get_mes_location()* → *String* method returns the location identifier of the *PyMES* instance at *address* if it can be reached otherwise returns `None`.

- *add_task(task: Task)* method takes argument *task* of class *Task*, constructs a LispPlan from *task* and its sub-*tasks* using the method *get_lisp()* and sends it via HTTP/REST to the the *PyMES* class instance running at *address* where it is pushed to *plan_list* of the *PyMES* class instance. It is used in Algorithm 2 line 7.
- *update_task(task_id: String, state: String)* method sends an update (task identifier and its state) passed as its argument to the *PyMES* instance running at *address*. It is used for plan sync in Algorithm 2 (lines 12, 14, ...).

4.7 *PyMES* class

Using all the classes described above, I have written a *PyMES* class which implements the main executive part of the system. This class is responsible for taking a LispPlan and executing it. It also contains a *Logger* for debugging. It can be found in the file *py_mes.py*. It has the following attributes and methods:

***__init__**(aml_config: AmlElement)*

1 Store arguments to their respective attributes;
2 Load configuration from *aml_config)* Initialize *plan_list* as *PlanList* instance;
3 Initialize *twin* as *DigitalTwinClient* instance;
4 *clients*, *resources* = *build_clients()*;
5 *locations* = *build_locations()*;
6 Start *task_worker_thread*;
7 Start *plan_worker_thread*;

- *name* attribute contains its arbitrary *String* identifier.
- *location* attribute contains the PDDL location of the *PyMES* instance.
- *state* attribute where the information about the current state of the MES is stored. Possible states are "`stopping`", "`stopped`", "`running`", "`pausing`" and "`paused`".
- *update_callback* attribute which can be set to a function that gets called upon a *PyMES* instance state update.
- *plan_list* is a *PlanList* instance where LispPlans are pushed using HTTP/REST by other *PyMES* instances.
- *clients* is a list that contains all the *ResourceClient* class instances.

- *resources* is a list that contains all the resources connected to the clients.
- *locations* is a list that contains all the locations under its supervision.
- *twin* contains the *DigitalTwinClient* class instance.
- *process_current_plan()* method is used to execute a LispPlan. It roughly implements the code in Algorithm 2.
- *task_update_queue* is a Python *Queue*[14] where all synchronization updates from other *PyMES* instances are pushed.
- *task_worker_thread* a Python *Thread* that takes the events from *task_update_queue* updates currently executed plan accordingly and redistributes the update to other *PyMES* instances using the *PyMESClient* class instances stored in *connected_meses* (corresponds to *mes_pool* from Alg. 2). It implements the code on lines 34–36 in Algorithm 2.
- *start_event* is a Python *Threading Event* that can be set by another *PyMES* instance to start plan execution.
- *plan_worker_thread* a Python *Thread* that pops a *Plan* from *plan_list* when *start_event* is set and starts its execution according to Algorithm 2. During the execution of a plan the MES can be paused and resumed by setting the *PyMES* state. In the same way the processing of the plan can be terminated by stopping the *PyMES*. This is done from a web interface.
- *cleanup()* method should be called upon exiting any program using the *PyMES* class. This method handles the correct disconnection of the clients and termination of running threads.
- *build_clients()* $\rightarrow$ *Tuple* method gets called automatically on initialization. It constructs and returns a *Tuple* containing a list of all the *ResourceClient* instances and a dictionary of all *Resource* instances currently available. This is done using the information about the available servers and resources obtained from the PyAutomationML configuration file.
- *build_locations()* $\rightarrow$ *List* method gets called automatically on initialization. It constructs and returns a *List* containing all the *Location* instances with their respective resources assigned. This is done using the information about the available servers and resources obtained from the PyAutomationML configuration file.

[14] https://docs.python.org/3/library/queue.html

4.8 Web interface

To control PyMES instances I have created a simple web server (each instance has its own web server). It has two pages (`<web server address/index>` and `<web server address/plan>`). From the first page (see Figure 4.4) the MES can be started, stopped, paused and resumed. When the `START` button is pressed a new goal/order is requested from the ERP, then a plan is retrieved from the planner and pushed to the *plan_list* and PyMES is started (→ *process_current_plan()* method is called). The other buttons can modify the state of *PyMES* accordingly. The second page (see Figure 4.5) displays the current state of the plan that is being processed. It displays the image created by the *view()* method provided by the *Plan*/*Task* and it is automatically updated.

The website back-end is implemented using the Python framework *Flask*[15] and *Flask-SocketIO*[16]. The front-end is created using the templating engine Jinja2[17] for HTML5[18] templates, Bootstrap 5[19] for styles and responsive design and Javascript[20]. Asynchronous communication and updating the pages in real time is achieved using WebSockets[21]. The control commands (buttons) and updates are passed to the *PyMES* class instance using WebSockets.

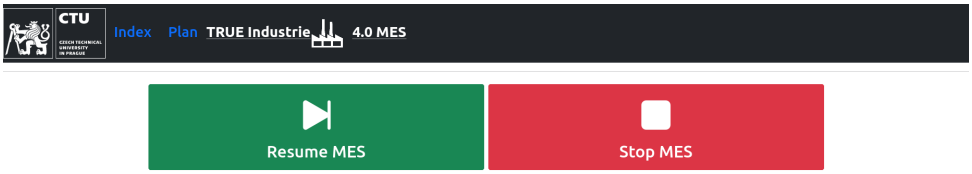


Figure 4.4: PyMES main web page (MES currently paused)

[15] https://flask.palletsprojects.com/en/2.0.x/
[16] https://flask-socketio.readthedocs.io/en/latest/
[17] https://palletsprojects.com/p/jinja/
[18] https://html.com/
[19] https://getbootstrap.com/docs/5.0/getting-started/introduction/
[20] https://www.javascript.com/
[21] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

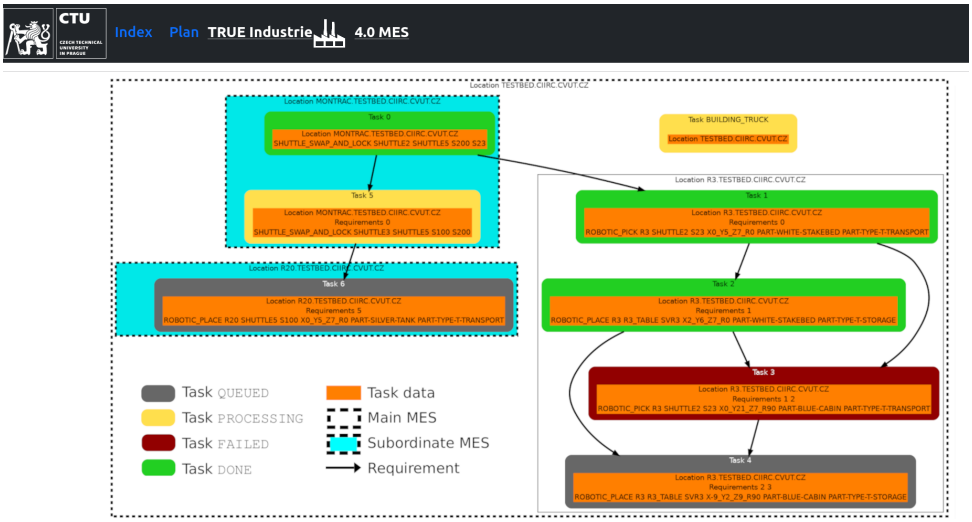


Figure 4.5: PyMES plan visualization web page (Plan currently being processed)

4.9 Initialization

Each instance of PyMES has to be started explicitly at the desired location. This is done using the function *init_mes* in *init_mes.py*. The function takes a single string argument *mes_location* that specifies the PDDL location[19] where the PyMES should be running. First, the PyAutomationML configuration file is loaded, and using *mes_location* the configuration of the PyMES instance is found. Using this configuration, a *PyMES* class is instantiated. Finally, a Flask-SocketIO web server is launched. This process corresponds to Algorithm 1.

Chapter 5

Testing and evaluation

5.1 Testing on the virtual production line

During the development of PyMES I have used the virtual production line extensively and it has proven to be an invaluable tool. Using the virtual line I was able to test various parts of the system during my work without depending on the real production line. This saved a lot of my time and allowed me to test even when the Testbed production line was under construction. It also allowed me to easily customize the testing setup in ways that would be very hard or even impossible to implement on the real production line. Thanks to that I was able to debug the algorithm of PyMES before the first test on the real production line. One of the PyMES configurations, that I was using, can be seen in Figure 5.1.

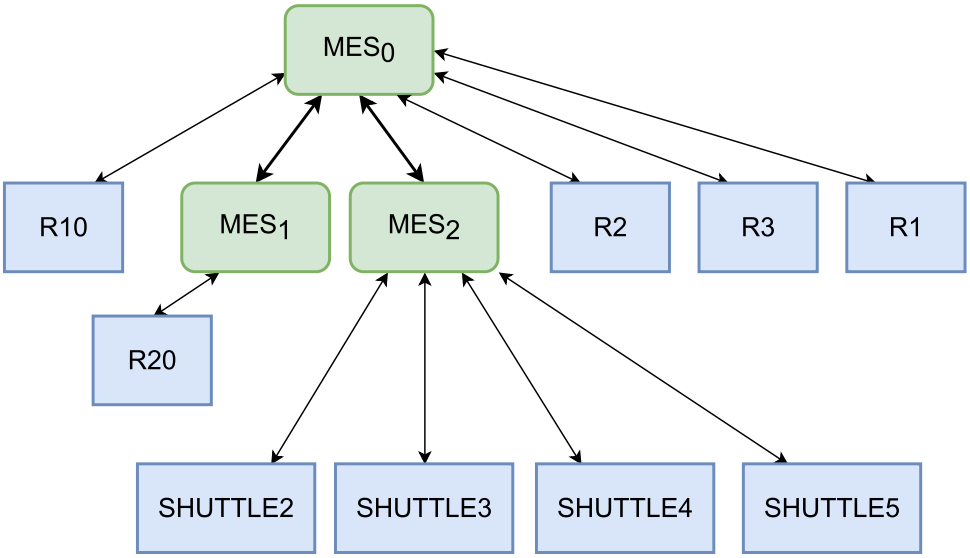


Figure 5.1: PyMES structure used in testing

5.2 Testing on the real production line

After completing my PyMES implementation and debugging it using virtualization, I moved to testing on the real production line. However, before I could start testing, I had to integrate a new KUKA Cybertech robot that was recently added to the production line. Due to the design of PyMES, I only needed to add the description of the new robot to the PyAutomationML configuration file and the system was functional.

Testing took an entire week and consisted of more than a hundred production runs. As expected, during my testing on the virtual line, I was not able to detect and fix all the bugs. The first major problem was that the handling of OPC UA variable data types was done incorrectly in my implementation. However, I was not able to debug this before, because on one hand the OPC UA servers running on the Siemens PLCs in Testbed strictly exact variable data types and on the other hand the Python OPC UA server implementation, I used in testing, does not provide type checking by default. Therefore, Python servers can even change the server variable data type to the data type of a variable that is written by OPC UA client.

Another bug I found was that when translating a swap shuttle operation (e.g., Task 0 in Fig. 2.6) I reversed their starting station with their destination. I would have expected to notice this during my virtual testing, however, the virtual line uses the same configuration file as PyMES and the actions were at the time committed to the digital twin from the virtualization. This meant

that PyMES switched the action during translation to OPC UA one way and the virtualization switched it back during translation from OPC UA so everything worked fine. There were some other bugs in the code, but none of them major and I was able to quickly fix all of them.

During testing, I also found that the color scheme I used was not suitable for colorblind people [1], so I updated it to one suitable for colorblind people, as can be seen in this thesis. Apart from that, the system worked without issues for three days during "ceremonious opening" [17] of new Testbed and during open days. I have used PyMES in three different instance configurations (one of them can be seen in Fig. 5.1) and with over forty different production plans. During this testing, there were 46 occasions when the line had to be stopped for various reasons (e.g. a visitor entered the safety area of the production line). In 30 cases PyMES recovered on its own after resuming the production line from the safety stop, in 16 cases the stop caused desynchronization of the digital twin which had to be manually resolved and the production had to be re-planned (without the need to reset PyMES), and in 0 cases PyMES had to be reset. This can be seen in Table 5.1. One of the factors that increased the number of cases in which manual resolution was necessary, was that PyMES currently does not have any information about the state of the emergency stop. This means that PyMES in some cases sends a new operation/action to a device while the production line is stopped (because of emergency) and then such operation/action cannot be processed. This could be resolved by some changes in the communication protocol between PyMES and the components of the production line.

Overall, PyMES proved to be a viable solution for controlling an Industry 4.0 production line. It is robust and allows for flexible problem resolution which, in many cases, can be done without human intervention. It also supports dynamic reconfiguration of the production line resources without the need to change the PyMES code.

Solution	Automatic recovery	Manual Recovery	Total system reset
Number of cases	30	16	0

Table 5.1: Emergency stop resolution with PyMES

[1] As was pointed out to me by one of my colleagues at Testbed, who is colorblind

5.3 Future work

As PyMES will continue to be used in Testbed, there are many improvements/additions that I would like to implement in PyMES. Some of them are, for example:

- Continue in the direction of decentralization and implement distributed Digital Twins and Planners.
- Use cytoscape[2] instead of graphviz for nicer and interactive plan visualization.
- Implement better visualization debugging.
- Implement automatic visualization of the distributed PyMES network (as can be seen in Fig. 5.1.

[2] https://cytoscape.org/

Chapter 6

Conclusion

In this thesis, I designed and implemented a distributed MES for the production line at Testbed for Industry 4.0 at CIIRC CTU. The implementation is written in Python 3.10 and is called PyMES. PyMES uses the protocol OPC UA to communicate with individual components of the production line. PyMES also has a web interface to facilitate visualization of the production process and control of the system. It is integrated with the existing digital twin and the planner available for the production line. This allows PyMES to track the state of the production line in real time, compute production plans (LispPlans) on-the-fly according to the current state of the production line and validate any operation before actually executing it. PyMES is designed to take production orders from the ERP (which I implemented prior to my thesis). It also uses a central PyAutomationML configuration file to facilitate changes in the configuration of the production line without touching the code of PyMES and related I40 components. PyMES is also distributed, so there can be multiple instances running at the same time, each controlling a part of the production line. Specifications of the individual instances of PyMES are written in the configuration file. This allows the production line to be extended with new production stations without the need to change the functionality of the rest of the line.

After implementation, I intensively tested PyMES on both the virtual production line and the real production line in Testbed. The first production runs revealed some bugs in the code, such as incorrect handling of OPC UA data types or incorrectly formatted synchronization messages. However, after fixing these bugs, PyMES worked correctly. Also, a new robot was added to the production line, which allowed me to verify the flexibility of PyMES and Testbed Industry 4.0 infrastructure. Due to the design of PyMES this change

only entailed adding the specification of this new robot to the aforementioned configuration file. After that, PyMES was deployed during open days at Testbed performing more than a hundred production runs without any failures on the side of PyMES, Digital Twin, Planner or ERP.

Appendix A

Bibliography

[1] Ajith Abraham, Edward Au, Alécio Binotto, Laura Garcia-Hernandez, Vladimir Marik, Felix Gomez Marmol, Vaclav Snasel, Thomas I. Strasser, and Wolfgang Wahlster. Industry 4.0: Quo vadis? *Engineering Applications of Artificial Intelligence*, 87:103324, 2020.

[2] Automationml. https://www.automationml.org/ Acessed 2022-05-15.

[3] AutomationML e.V. and OPC Foundation. AutomationML whitepaper: OPC Unified Architecture information model for AutomationML, 2016. https://www.automationml.org/o.red/uploads/dateien/1485865685-WP_OPCUAforAutomationML_V1.0.0.zip.

[4] Hendrik Van Brussel, Jo Wyns, Paul Valckenaers, Luc Bongaerts, and Patrick Peeters. Reference architecture for holonic manufacturing systems: Prosa. *Computers in Industry*, 37(3):255 – 274, 1998.

[5] Paulo Leitão and Francisco Restivo. Adacor: A holonic architecture for agile and adaptive manufacturing control. *Computers in Industry*, 57(2):121 – 130, 2006.

[6] W. Mahnke, S.H. Leitner, and M. Damm. *OPC Unified Architecture*. SpringerLink: Springer e-Books. Springer Berlin Heidelberg, 2009.

[7] Vladimír Mařík, Vladimir Gorodetsky, and Petr Skobelev. Multi-agent technology for industrial applications: Barriers and trends. In *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1980–1987, 2020.

[8] Q. Nguyen. *Mastering Concurrency in Python: Create faster programs using concurrency, asynchronous, multithreading, and parallel programming*. Packt Publishing, 2018.

[9] Petr Novak, Jiri Vyskocil, and Bernhard Wally. The digital twin as a core component for industry 4.0 smart production planning. *IFAC-PapersOnLine*, 53:10803–10809, 2020.

[10] Petr Novák, Petr Douda, and Jiří Vyskočil. Pymes: Distributed manufacturing execution system for flexible industry 4.0 cyber-physical production systems. unpublished, 2022.

[11] Petr Novák, Petr Douda, Jiří Vyskočil, and Bernhard Wally. Pyaml: Enhancing automationml for advanced virtualization of industry 4.0 cyber-physical production systems with python code injections. In *26th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2021, Vasteras, Sweden, September 7-10, 2021*, pages 1–8. IEEE, 2021.

[12] Petr Novák and Jiří Vyskočil. Digitalized automation engineering of industry 4.0 production systems and their tight cooperation with digital twins. *Processes*, 10(2), 2022.

[13] Jose F Rodrigues Jr, Agma JM Traina, Christos Faloutsos, and Caetano Traina Jr. Supergraph visualization. In *Eighth IEEE International Symposium on Multimedia (ISM'06)*, pages 227–234. IEEE, 2006.

[14] B Saenz de Ugarte, A Artiba, and R Pellerin. Manufacturing execution system–a literature review. *Production planning and control*, 20(6):525–539, 2009.

[15] Ardeshir Shojaeinasab, Todd Charter, Masoud Jalayer, Maziyar Khadivi, Oluwaseyi Ogunfowora, Nirav Raiyani, Marjan Yaghoubi, and Homayoun Najjaran. Intelligent manufacturing execution systems: A systematic review. *Journal of Manufacturing Systems*, 62:503–522, 2022.

[16] Maulshree Singh, Evert Fuenmayor, Eoin P Hinchy, Yuansong Qiao, Niall Murray, and Declan Devine. Digital twin: origin to future. *Applied System Innovation*, 4(2):36, 2021.

[17] Testbed opening. https://www.ciirc.cvut.cz/cs/grand-opening-ricaip-testbed/ Acessed 2022-05-15.

[18] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Style guide for Python code. PEP 8, Python Software Foundation, 2001.

[19] Jiří Vyskočil and Petr Novák. Draft - on specification of the production with the flexible production line. Technical report, Testbed for Industry 4.0 at CTU in Prague CIIRC, 10 2020.

Appendix B

PyMES code

The code of the PyMES implementation can be found attached to this thesis in the directory `/code`. The code is written in Python 3.10 and formatted according to PEP 8 [18]. Package requirements of the project can be found in the file `requirements.txt` in the `/code` directory. An example PyMES configuration can be run using the script `start.sh` from the `/code` directory. The requirements for running the example configuration are: Digital Twin and Planner running at the address specified in the PyAutomationML configuration file, ERP running at the address specified in the PyAutomationML configuration file, and the real or virtual production line as specified in the PyAutomationML configuration file.

Appendix C

Testbed specification

Because the Testbed specification [19] is not publicly available it can be found attached to this thesis as a PDF file *REST_interface_for_accessing_TWIN_and_Planner_DRAFT.pdf*.