# A Service-Oriented Domain Specific Language Programming Approach for Batch Processes

Martin Melik Merkumians, Matthias Baierling, Georg Schitter
Automation and Control Institute, Technische Universität Wien
Gußhausstraße 27-29/E376, Austria, Vienna
Email: {melik-merkumians, baierling, schitter}@acin.tuwien.ac.at

***Abstract*—The demand for flexible automation architectures in industry is raising, due to the trend to individualized products and the shortening of product life-cycles in general. Current approaches in automation systems are either only suited for a small range of processes or are only reprogrammable by software and automation experts. The presented domain specific language based approach decouples production-domain concerns from the automation program, by separating the process description into a specialized domain language. This domain language can be used by plant owners and domain experts to define and execute vastly different processes, as long as the needed process steps are supported by the actual automation system. This contribution demonstrates the development of a domain-centric SOA-based batch process automation system, by creating a directly executable DSL based on domain terms found in ISA-88.**

## I. Introduction

Until recently it was sufficient to optimize a manufacturing system for just one or a small range of processes. The current trend towards individualized products and the associated demand for mass customization, as well as the shortening of product life-cycles, leads to a reduction in lot size and an increasing number of product variations. Rigid automation systems cannot cope with the actual market situation, and industry is in search for more flexible solutions and starts to abandon inflexible production lines [1], [2], [3]. Apart from the novelty of the requirements and the associated trend to modularization and flexibility, one reason for inflexible automation systems is that manufacturing plants are usually treated as one-of-a-kind products themselves [4]. In other words, you never build the same plant twice, and therefore, any investment in modularization and reusability was perceived as uneconomical. Currently, 55% of production system creation costs are spent on planning, engineering, programming, and system ramp-up phase of a manufacturing plant [5]. Industry is demanding a novel approach to system design and integration which is expressed by several industry-driven initiatives like Industry 4.0 in Germany or Industrial Ethernet in the USA [6], [7]. Flexible solutions, however, are introducing a lot of added complexity to automation systems, with the cause lying in the need to support a large range of processes, process variations, retooling, changes to process intra-logistics, and the associated plant reconfigurations, just to name a few.

Traditional development approaches are not well suited to support the development of such highly complex and flexible systems and novel approaches to this problem were proposed and analyzed by academia.

The application of Service-Oriented Architecture (SOA) concepts to the automation domain [2] has been first proposed, as automation systems nowadays are mainly networked and distributed systems, and with the rise of Ethernet-based field-buses the application of standard web technologies, namely web-services, was natural. The EU projects SIRENA [8] and SOCRADES [9] pioneered in using web-services for automation systems but were primarily concerned with the technical implementation of services and less with the overall design approach suited for automation systems.

Although web-services presents a possible technical solution, it is not a well established standard technology in the automation domain, like classic OPC or OPC Unified Architecture (OPC UA). The suitability of OPC UA for SOA-based automation systems is analyzed and an application example in the domain of process industry is used to demonstrate the IEC 61499 – Function blocks [10] standard and OPC UA can be combined in order to create an SOA system and expose the provided services in the address space of OPC UA [11]. The applicability of OPC UA as a middleware for the process industry, using the ISA-88 – Batch Control (ISA-88) reference model [12] has been demonstrated [13]. An orchestration engine based approach has been proposed, which allows simplifying Programmable Logic Controller (PLC) programming, by splitting the desired action sequence of a plant component into higher level abstractions [14]. In the presented application case of a pick-and-place crane these abstractions are *lift*, *lower*, and *turn left* and by sequencing them into a workflow development effort can be reduced.

Agent-based approaches were also considered as an alternative to traditional automation systems [15], presenting a three-tiered architecture separating an automation system into an agent-based control system for the High-Level Control (HLC), a real-time capable control layer for the Low-Level Control (LLC), and the physical system layer. Although the general idea is quite intriguing, this work does not clearly explain how the physical layer is separated from the LLC. Also, it is not well explained on how the HLC can be reused easily. Another agent-based control system for the process domain focuses on path finding and path recovering in the case of a fault [16]. An agent-based approach for controlling the patching process of wood panels [17] decomposes the

978-1-5090-1314-2/16/$31.00 © 2016 IEEE

plant into components, where each component is controlled by a dedicated agent, the so-called automation agent. Each agent exposes functions which can be used by other agents in the system too [18]. Tasks like process scheduling, which cannot be directly associated with a mechatronic component are represented by a special type of agent, the functional agent.

Although academia has seen agent-based control systems as a promising technology, such systems were never in widespread use in industry, due to the lack of real industrial applications, missing trust in the idea of delegating tasks to autonomous agents [19], and concerns regarding the stability, scalability, and survivability especially in unpredictable environments of attacks and system failures [20].

An alternative approach coping with the complexity of modern automation system are model-based development methods. Model Driven Architecture (MDA) approaches are capable of separating the logical application of an automation system, the Platform Independent Model (PIM), from its implementation details of the targeted goal-platform, the Platform Description Model (PDM). Both models are then combined to generate an executable automation program (the Implementation Specific Model (ISM)) [21]. Such and MDA approach is applied to automation systems, but with focus on the reusability of IEC 61499 programs and Function Blocks (FBs), whereas the PIM is defined by a Function Block Network (FBN). This FBN includes concrete implementation of the automation program, which can be expressed without any hardware-specific implementation details [22]. Hardware interfaces (e.g., a sensor, or a pick and place unit) are communicated with and to via messages, which abstract the concrete hardware access, but gives the needed implementation detail in order to execute a program step (e.g., the next desired pose of a pick and place unit). These messages are received by the abstract FB and forwarded via an Adapter Function Block (AFB) to a concrete implementation Service-Interface Function Block (SIFB) for execution.

Object-Oriented (OO) design approaches are suited and used to hide the implementation-specific details from the user of the system via abstract classes or interfaces. An application of this approach on automation systems combines intelligent sub-plant components, so-called *mechatronic objects*, which represent a combined hard- and software component, in an OO manner in order to create automation solutions [18]. Each component contains a scheduler, a selector, and a synchronizer. These mechatronic objects are only controlled via a set of functions which the component provides. The programmer also provides the schedule of operations to the scheduler for execution. Although an interesting approach, mainly the theoretical foundation and almost no implementation details are presented.

Another MDA-based approach, which adapts the System Modeling Language (SysML) notation providing a specialized language profile SysML-AT, is capable to completely replace traditional IEC 61131 – Programmable controllers [23] languages [24]. These custom models describe the key facts of the desired automation solution and is used to generate IEC 61131 compliant code from the models, which enables the use of current technology. Results show a significant increase in software quality due to an extensive modeling of the problem domain but is not concerned to increase flexibility and reusability of plant components.

Similar ideas are also pursued in the MEDEIA approach, which defines several Domain Specific Languages (DSLs) for the different engineering aspects and by integrating these domain-specific views together in the MEDEIA model, automatic automation system code generation shall be enabled [4], [5], [25], [26].

All the presented approaches have in common, that a software/automation expert is needed to implement the changes required for implementing a new process or for variations of the plant. This is due to the resulting PLC code is usually being tightly interconnected with the desired process as well as the hardware-specific implementation (see [15], [18]), and the design process itself is designed to be handled by a software or automation engineer and not by the domain expert [21], [24], [25].

This contribution proposes a novel DSL based approach decoupling the desired production process from the automation code itself. Instead of augmenting traditional software engineering processes, it focuses on the view of the plant operators, by making the process description (or process recipe) the central element of the automation system design. Through a careful separation of domain concerns and implementation-dependent aspects, the process recipe becomes a first-class entity in the automation system, either being interpreted on the fly or being used to generate automation code. The domain concerns are then exposed via services in a SOA [27] approach via the responsible automation components to the system. This separation enables the enhancement of reusability of production processes and leads to an improved modularization of automation components, as single tasks and responsibilities are expressed via the process recipe. This will allow the plant operator to define and program his desired processes without the help of an automation engineer, as long as the plant and the DSL can cope with the new process. As the proposed approach is highly dependent on a particular application domain, the proposed approach is validated by an example in the process domain. The demonstrator for this approach is the Festo Didactic process tank model [28], as shown in Fig. 1.

The remainder of this contribution is structured as follows. In Section II the Process Recipe DSL is introduced, giving insight into the design process of such a DSL. Section III introduces a helper DSL, which is needed to describe the plant in order to perform recipe expansions for creating executable recipes. Section IV gives an implementation example on how to implement domain services in IEC 61499 and describes details on the used system configuration. Section V demonstrates the successful implementation via experiments, giving time charts of a recipe phase execution to demonstrate the functionality of the system. Section VI concludes the work presented in this paper.

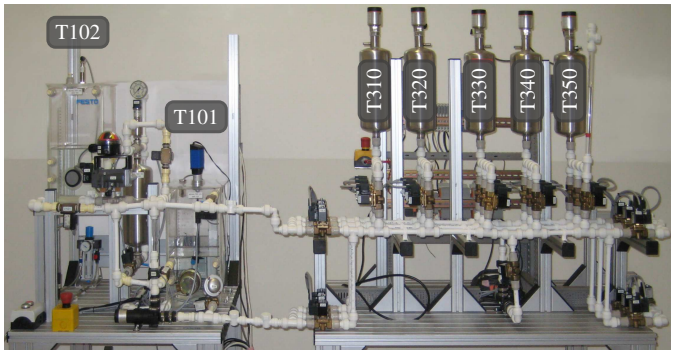


Fig. 1. Festo Didactic process industry demonstrator plant (left) and custom built storage tank system (right), showing the tank labels as used in Fig. 2

## II. Process Recipe DSL

The standard ISA-88 [12] is taken as a role model for the creation of the Process Recipe DSL (PRD). ISA-88 reflects the acknowledged state of practice in the batch process domain, as it is defined by an expert group of this domain, which makes it highly suitable as a basis for the PRD. Also, it defines and explains the batch process domain terms and concepts thoroughly, which makes it a very useful source of information for non-domain experts. As the ISA-88 defines terms and concepts from the Enterprise Resource Planning (ERP) layer to the machine layer, the remainder of this section will focus on the domain concepts and process sequence execution semantics needed for the definition of the PRD.

The first needed term is the *process reactor*, which is mentioned but not specifically defined in ISA-88. In principle, it is a single physical unit in a batch process plant providing material transformation capabilities, like heating, cooling, or similar capabilities, and is one of the basic building blocks of most batch process plants. ISA-88 names three concepts that are directly concerned with the production process, *process stages*, *process operations*, and *process actions*. *Process stages* which represent complex processing sequences (e.g., *polymerize*) and *process operations* which represent major processing activities (e.g., *Prepare reactor*) are high-level process abstractions. *process actions* low-abstraction, describing singular process steps together with its associated domain-relevant parameters (e.g., Heat the reactor to 55–60 °C), without giving details on implementation and execution details. ISA-88 names a handful of examples for *process actions*, e.g., *Add*, *Heat*, and *Hold*, with [29] and [30] expanding the set with *Agitate*. Also, ISA-88 indicates that *process actions* are executed on *process reactors* [12, p. 20] The ISA-88 process execution semantics is similar to Petri nets or Petri net like approaches, such as Sequential Function Chart (SFC).

SFC has been enhanced in order to integrate a batch execution engine into PLC code, by mapping the SFC to a tabular notation [31]. This enhancement introduces the concept of *dominant* and *non-dominant* phases, with dominant phases indicate that termination of the phase leads to recipe transition, whereas a non-dominant (or dependent) phase indicates that its execution shall only occur in parallel to a dominant phase. The analysis of ISA-88, [30], and [32] has shown, that process domain recipes consist of sequential processes, and parallel processes (*AND branching*) and conditional processes *OR branching*. Conditional processes are usually used to implement error handling routines or enable the execution of equivalent processes, based on certain plant states (e.g., available raw materials).

Based on these findings and the requirement that the DSL shall be executable (see Section I), the first step is to define the execution model for the PRD. As ISA-88 the PRD has a Petri-net like execution semantic similar to SFC, enabling sequential and parallel processes, with the exclusion of conditional processes. The reason behind this restriction is to keep the focus on the desired process. Alternative equivalent processes are expressed as separate processes, but this imposes not a real restriction to the overall system, as the concrete process to be executed can be selected, based on the current plant state, before the execution of the desired process starts. Also, the PRD includes the concept of dominant and non-dominant phases in order to provide simple means to describe dependent parallel processes.

With the execution mechanics now defined, the next step is to define the available steps for execution. Here the ISA-88 *process actions* directly translate to the available steps in the PRD.

- *Start* – As in SFC this is the starting point of a recipe.
- *Stop* – As in SFC the end point of a recipe.
- *Add* – The added substance is measured in <l>.
- *Heat* – The desired temperature given in <°C>.
- *Agitate* – No parameters, as it is designed to be a dependent phase.

Following the concepts of ISA-88 that *process actions* are executed on *process reactors*, all PRD steps have the unique identifier of a *process reactor* as a mandatory parameter. Additional parameters are based on the needed parameters associated with the represented activity itself. Listing 1 gives a shortened version of the Extended Backus–Naur Form (EBNF) of the *PRD* developed in Eclipse Xtext [33], a framework for developing DSLs. Xtext specific import and generate declarations in this and all other EBNF definitions are omitted for conciseness.

Listing 1
EBNF of the Process Recipe DSL

```
Model:
   {Model}
   'Tanks'
   tanks+=Tank*
   'Recipe'
   steps += Step*;
Tank:
   name=ID;
Step:
   Add | Heat | Agitate | Stop | Start;
Start:
   'start' name=ID ('after' formerStep+=[Step]* )?;
Stop:
```

```
    'stop' name=ID 'after' formerStep+=[Step]*;
Add:
    'add' name=ID 'amount' ((amount_l=INT 'l') | (
        amount_ml=INT 'ml')) 'from' source=[Tank] 'to
        ' target=[Tank] 'after' formerStep+=[Step]* (
        dominant='nondominant')?;
Heat:
    'heat' name=ID 'tank' tank=[Tank] 'up to' temp=
        DOUBLE '°C' 'after' formerStep+=[Step]* (
        dominant='nondominant')?;
Agitate:
    'agitate' name=ID 'tank' tank=[Tank] 'after'
        formerStep+=[Step]* (dominant='nondominant')
        ?;
```

This simple DSL enables to concisely express the desired process of a process plant, without the need of information on how a *process action* needs to be implemented by the control soft- and hardware but captures the necessary process detail needed to execute the process. The execution engine itself is also implemented via the Xtext framework. Xtext grammars are compiled into traversable meta-models, which can be used to implement a language interpreter [34].

## III. Process Domain System Modeling Language

The *PRD* given in Listing 1 is enough to describe and execute the desired process if everything would be in one *process reactor*. Even simple plant configurations in the process domain usually consist of more than one *process reactor* and includes storage tanks, pumps and so forth, which is why an additional DSL is needed to model the process plant itself. Analyzing the demonstrator plant, shown in Fig. 1, the following domain-relevant elements: tanks, level sensors, pipes, valves, heaters, mixers, and pumps are identified. In order to connect the above-mentioned domain elements, the concepts of *connection points* (junctions to tanks and pumps), and nodes (junctions between pipes) are needed. All elements are identified via unique IDs. *Heaters* and *Mixers* which can be added to *Tanks*, provide *Tanks* the capability to execute the *Heat* and *Agitate* services.

Piping and Instrumentation Diagrams (P&IDs) do not take physics and the actual physical position of plant components into account, for example, it is not possible to determine the flow direction between *T101* and *T102*. Considering our demonstrator plant, shown in Fig. 2, there are situations where the fluid can only flow unidirectional.

For example the connection between *T102* and *T101* has only valve *V102* in between, so due to gravity, a bidirectional flow of fluids is not possible in this case. By examining the actual plant in Fig. 1, it can be seen that *T102* is on a higher level than *T101*, which is why fluids will flow from *T102* to *T101*. In order to reflect such situations, the concept of flow direction was added to valves, whereas *bidirectional* means bidirectional flow is possible, and *1to2* and *2to1* signals a unidirectional flow from valve gate 1 to valve gate 2 (gates are marked with 1 and 2 in the P&ID) and vice versa, also the *tank level* concept is used to model the difference of mount height of tanks, so it can be verified that a fluid flow without a pump in between, is always directed from the higher level tank to the lower level tank. A similar concept of flow directions is used for tanks, which is necessary as some tank flanges on our demonstrator plant are positioned in such a way that a bidirectional flow is not possible (e.g. a flange mounted on the top of a tank), which further limits possible paths in the plant. Please note that the same direction concept for tanks and valves could have been used, but in contrast to valves tanks have clear *in* an *out* directions both concepts have been added to the DSL. Pipe lengths are also considered in the model, in order to select a path for the *Add* service via Dijkstra's shortest path algorithm [35].

Listing 2 gives the shortened form of the developed Xtext EBNF notation, which was used to describe the demonstrator plant.

Listing 2
EBNF used to define the process domain plant system DSL

```
Model: entities += Entity*
      points += Point*;
Entity: Tank | Pipe | Valve | Pump ;
Tank: 'tank' name = ID '{'
      'tankLevel' tankLevel = INT
      ('heater' 'actuator' heaterActuator=ID 'sensor
          ' heaterSensor=ID)?
      ('mixer' mixer=ID)?
      ('levelSwitchTopAlarm' levelSwitchTopAlarm=ID)
          ?
      ('levelSwitchTopWarning' levelSwitchTopWarning
          =ID)?
      ('levelSwitchBottomWarning'
          levelSwitchBottomWarning=ID)?
      ('levelSwitchBottomAlarm'
          levelSwitchBottomAlarm=ID)?
      directedConnectionPoint +=
          DirectedConnectionPoint+
   '}';
DirectedConnectionPoint: 'directedConnectionPoint'
    '(' connectionPoint = [ConnectionPoint] ','
    direction=Direction ')';
ConnectionPoint: 'connectionPoint' name=ID;
Pump: 'pump' name=ID '{'
      'input' input = [ConnectionPoint]
      'output' output = [ConnectionPoint]
    '}';
Pipe: 'pipe' name = ID '{'
      'length' length=INT
      'endPoint1' point1 = [Point]
      'endPoint2' point2 = [Point]
    '}';
Direction: typeName=('in' | 'out' | 'bi');
Valve: 'valve' name=ID '{'
       'type' type = ValveType
       'connenctionPoint1' connectionPoint1 = [
           ConnectionPoint]
       'connenctionPoint2' connectionPoint2 = [
           ConnectionPoint]
       'flowDirection' flowDirection = FlowDirection
     '}';
ValveType: type = ('continuous' | 'OnOff');
FlowDirection: flowDirection = ('1to2' | '2to1' | '
    bi');
Point: Node | ConnectionPoint;
Node: 'node' name = ID;
```

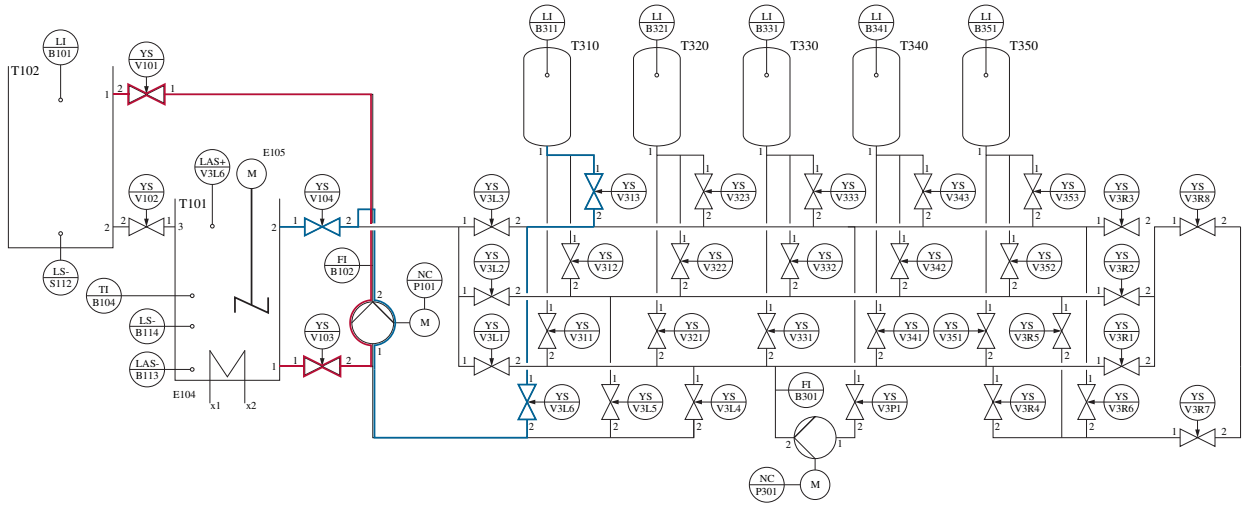


Fig. 2. The P&ID of the demonstrator plant. The red and blue paths are dynamically calculated from the recipe given in Listing 3, as a result from the *Add1* (blue) and *Add2* (red) recipe steps.

## IV. Implementation

The *process actions* identified and added to the *PRD* defined in Section II as well as a workflow execution engine needs to be implemented for the automation system. For the purpose of demonstration the open source IEC 61499 project *4DIAC* [36], [37] was chosen for controlling the PLCs, as the event-driven execution semantics of IEC 61499 based control systems suites the execution model of our DSL and SOA-based approach very well. Actions are triggered and the next step is initiated as soon as the requested action has finished, which can easily be modeled via events. The recipe is executed via a simple workflow engine, which is a piece of software that takes a process model as input, interprets the input, and performs actions based on this interpretation. In this case the workflow engine executes the process recipe by starting its execution at the *Start* node, and then stepping through the subsequent recipe phases, although the PRD step elements point to their former steps, in order to enable that the recipe steps occur in their execution order (see Listing 3). Each recipe phase is checked if a recipe expansion is necessary, and performed if it is. After the current step has been fully expanded the resulting service requests are sent from the PC to all PLCs, as no information is given in the recipe model which controller is responsible for a certain resource. This is necessary in order to exclude as much plant information from the recipe as possible. In order to synchronize the recipe execution with the plant, the workflow engine waits for confirmation signals from the plant before it further progresses through the recipe phases.

Since the recipe shall contain as little as possible information about the physical plant, a multicast mechanism has to be used to provide the capability to inform all available PLCs without the need to either know how many PLCs need to be informed or the hostnames or IP addresses of the devices. Instead of using the standard TCP/IP family multicast mechanism via the unreliable UDP protocol, which would force the implementation of both PLC code and the process recipe application to cope with lost, duplicated, or wrong-ordered messages, the Message Queue for Telemetry Transfer (MQTT) client implementation *Paho* [38] is used. The only information which needs to be provided to the systems components is the address of the message broker, the central message hub of the message queue and the topic to which the associated Publish/Subscribe FB is registered to. Another parameter is the device name, which is an arbitrary string used to identify a device in MQTT, but the actual value is of no consequence for the execution of the recipe engine.

In this demonstration, the message broker *Mosquitto* [39] is used. All messages are tagged with a so-called *topic* sent to the message broker, which relays them to all registered subscribers who registered for the respective *topic*. MQTT can guarantee that messages are delivered once and only once, and in the correct order, which is necessary for the recipe execution system.

In the case of our demonstrator, two Beckhoff CX5010 PLCs are used as the control hardware platform, with FORDIAC RunTime Environment (FORTE) as the IEC 61499 compliant runtime environment. One Beckhoff CX5010 controller is controlling the Festo Didactic process industry plant with all the associated valves, pumps, tanks, and tank equipment, the other one is in sole control of the components of custom built storage system, both shown in Fig. 1, forming reusable components with clear responsibilities. The EtherCAT bus terminal is controlled via the Beckhoff ADS interface. Each component's capabilities are exposed via their services, which are the *process actions* adopted in the proposed DSL (see Listing 1). The developed process recipe creation and execution software is deployed on a standard PC system.

Fig. 3 shows an example FBN how to implement the *Agitate* service in IEC 61499. Requests are received via the leftmost *Subscribe* FB, and buffered in a First-In–First-Out (FIFO) buffer for sequential execution.

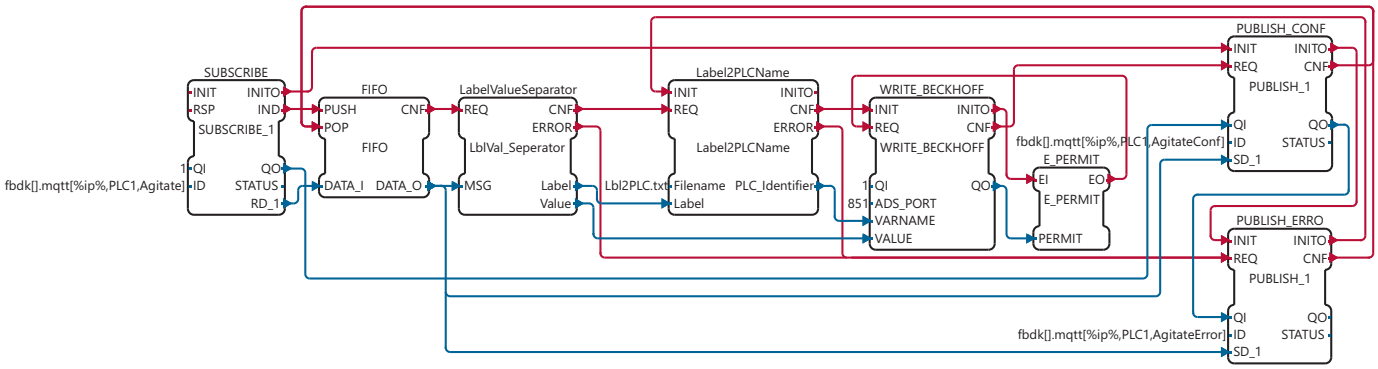


Fig. 3. The IEC 61499 Function Block network implementation of the *Agitate* service. The *Subscribe* and *Publish* FBs are configured to use MQTT, the first parameter is a placeholder for the *broker IP*, the second parameter is the *device name*, the third parameter is the *topic name*.

When a message is ready to be processed the message gets parsed and the PLC checks if it is responsible for the requested resource. The received message contains the resource name (e.g., *T101*) for which the corresponding service (selected via the MQTT topic) is requested. The PLC has a list of resources it shows responsible for, and if the PLC is not responsible the execution ends. If the PLC is responsible, the request is processed and a confirmation message is published to the confirmation topic via the *Publish* FB on the right side. As soon as the central PC received all confirmations, it proceeds the execution of the recipe.

## V. Demonstration

The proposed domain-centric design and SOA-based implementation and execution approach is demonstrated and evaluated on the 7-tank batch process plant setup, as shown in Fig. 1 and Fig. 2. The execution of a process is only defined by the sample recipe given in Listing 3 (a graphical representation is given in Fig. 4) and the plant model as given in Listing 4. Please note that only a reduced, but meaningful, model is given in Listing 4 for conciseness. Both the sample recipe and the plant model comply to their respective EBNFs.

Listing 3
Recipe Language Example

```
Tanks
  T101 T102 T310 T320 T330 T340 T350
Recipe
  start Start
  add Add1 amount 2000 ml from T310 to T101 after
      Start
  agitate Agitate1 tank T101 after Add1 nondominant
  heat Heat1 tank T101 up to 50.0 °C after Add1
  add Add2 amount 2000 ml from T101 to T102 after
      Heat1 Agitate1
  stop Stop after Add2
```

Listing 4
Tank System Language Example

```
tank T101 {
  tankLevel 1
  heater actuator E104 sensor B104
  mixer E105
  levelSwitchTopAlarm S111
  levelSwitchBottomWarning B114
  levelSwitchBottomAlarm B113
  directedConnectionPoint(T101C01,bi)
  directedConnectionPoint(T101C02,in)
  directedConnectionPoint(T101C03,in) }
tank T102 {
  tankLevel 2
  levelSwitchBottomWarning S112
  directedConnectionPoint(T102C01, bi)
  directedConnectionPoint(T102C02, bi) }
                    ⋮
tank T350 {
  tankLevel 2
  directedConnectionPoint(T350C01, bi) }
valve V101 { type OnOff
          connenctionPoint1 V101C01
          connenctionPoint2 V101C02
          flowDirection bi }
                    ⋮
valve V3R8{ type OnOff
          connenctionPoint1 V3R8C01
          connenctionPoint2 V3R8C02
          flowDirection bi }
pump P101 { input P101C01
          output P101C02 }
pump P301 { input P301C01
          output P301C02 }
pipe P1001 { length 6 endPoint1 T101C01 endPoint2
     V103C01 }
                    ⋮
pipe P3P04 { length 8 endPoint1 N342 endPoint2 N334
     }
```

The recipe execution is started at the *Start* node in Fig. 4, and as the start node execution is finished immediately the single following step *Add1* is triggered. As it can be seen in Listing 3, the *Add1* service step shall move 2000 ml from T310 to T101, without giving details on how this can be performed on the sample plant. The recipe execution system, therefore, triggers a recipe expansion procedure for *Add1*. In order to create this additional information a path from T310 to T101 is analyzed. In the first step, the pathfinding algorithm checks if there is a gravity based route from T310 to T101. As no such route exists, a route via the available pumps has to be found. Utilizing Dijkstra's shortest path algorithm [35], the shortest partial routes starting from the start point of the *Add1* step (T310) to both pumps (P101 and P301), and from both pumps to the goal tank (T101), of course without the

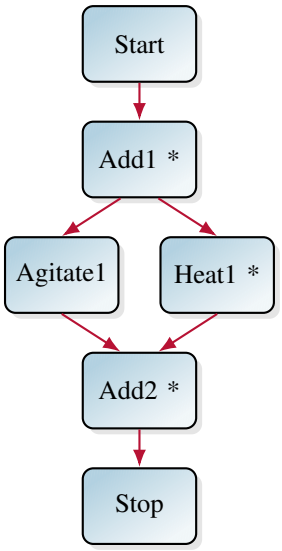


Fig. 4. The graphical representation of the recipe example given in Listing 3 (an asterisk[∗] indicates a dominant phase [31], as defined in Section II).

possibility to reuse the paths used in the first calculation step. The resulting path lengths are saved for the final construction of the route, which is derived by combining the respective partial routes (the shortest path from T310 to P101/P301 and from P101/P301 to T101) are summed up and choosing the path with the minimum length in the end. The path lengths are obtained from the plant model given in Listing 4. Based on the obtained final path plant components, which are participating in the path are identified via the plant model, and the needed activation commands for the components are generated. For the *Add1* process step, the participating components can be seen in Fig. 2, by examining the blue path, starting from T310 to V313, V3L6, P101, and over V104 finally into T101. The activational services for these components are generated as non-dominant phases, while the dominant phase in the generated recipe is the *Level Monitoring*. The generation rules for the *Level Monitoring* service as part of an *add* service are as follows:

1) The goal value is the desired value of the corresponding *add* service.
2) If the target tank has a level sensor, this sensor is used to measure the transported fluid.
3) If the target tank does not have a level sensor, but the source tank has one, then the source tank sensor is used to measure the transported fluid.
4) If both tanks do not have a sensor, then the transported fluid is estimated via the pump flow.

According to these rules, the *Level Monitoring* service is generated for T301.

After the *Add1* service is successfully executed, the non-dominant process step *Agitate1* and the dominant process step *Heat1* with a goal temperature of 50 °C on tank T101 is executed. As all services are available and localized in T101 the execution of both process steps start immediately.

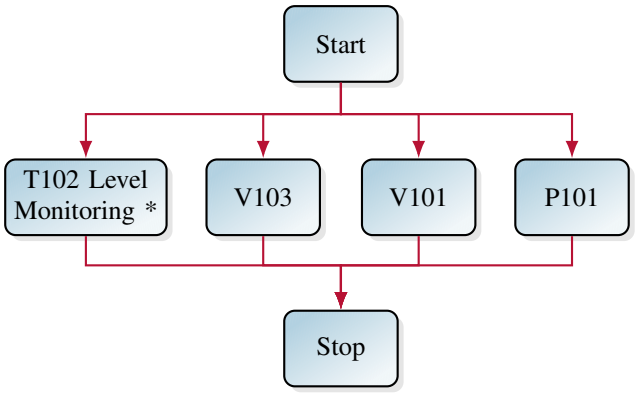


Fig. 5. The generated internal structure of the *Add2* service step, pumping the fluid from *T102* to *T101*, via the red-highlighted path in Fig. 2 (an asterisk indicates a dominant phase [31], as defined in Section II).

In the last to final process step *Add2*, another recipe expansion analogously to *Add1* is performed. After calculating the transportation path from T101 to T102, the participating plant components, V103, V101, and P101 can be identified. This is again shown in Fig. 2 by the red path, and the corresponding sub-recipe is given in Fig. 5.

The execution of the sub-recipe, as shown in Fig. 6, and the measured associated process variables of the sample plant are given in Fig. 6. The top chart shows the time response of the level sensor of tank *T102*. The second chart shows the state of the valves and pump, where *on* means that the valve is open and respectively the pump is active, and *off* means the valve is closed and the pump is inactive. The third chart shows the flow sensor (*B102*) value for pump *P101*. The lowermost charts show the message flow for initializing the *Add2* process as well as the end of the *Add2* process.

As it can be seen in Fig. 6, after issuing the commands for watching the tank level of tank *T102*, and to open the valves *V103* and *V101*, and the activation of pump *P101*, the flow through the pump and the tank level is rising. As soon as the tank level is increased by the requested 2.01 (see Listing 3), the PLC is sending a signal that the requested tank level has been reached, which prompts the recipe workflow engine to issue the requests to turn off the pump and to close the prior opened valves, therefore stopping the flow through pump *P101* and stabilizing the level of *T102*.

## VI. Conclusion

The presented novel SOA-based DSL driven programming approach for batch process programming, enables plant operators to easily create and change process recipes without the support of automation experts, as the recipes themselves are based on domain expert terms. In addition, it allows the reuse of process recipes on different plants without adaption, and is able to cope with plant structure changes and faults during execution time due to its interpreted nature. In this contribution, we have successfully developed and demonstrated a DSL driven programming approach for the batch processing domain on a Festo Didactic batch process plant. By focusing

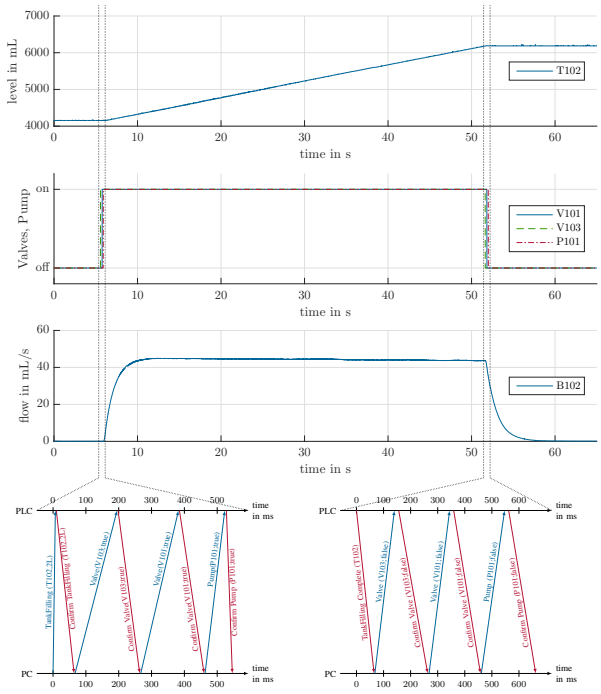


Fig. 6. Time charts showing the execution of the expanded *Add2* service as requested in Listing 3. The top graph shows the rising of T102's tank level. The next graph shows the actual on/off state of valves V101 and V10 3, and of pump P101. The third graph shows the flow-meter value B102 connected to P101. The bottom graphs show the communication flow of the SOA system in the time frame depicted with the dotted lines.

on the domain of use of an automation system, this approach relieves the plant operator from the burden of dealing with automation code, in order to modify production processes. The desired processes to be performed on an automation system are expressed exclusively by domain terms in the form of recipes, where automatic recipe expansion is performed in order to fill plant-specific execution gaps. The domain terms are then used to implement an SOA-based automation system and automation components are exposing their domain capabilities via services. The approach is validated on a demonstrator plant from Festo Didactic, using off-the-shelf Beckhoff CX5010 PLCs (Intel® Atom™ Z510 1.1 GHz, 512 MB) running Windows XP Embedded using 4DIAC/FORTE as IEC 61499 compliant runtime environment. The PLC responsible for the Festo Didactic process model is connected to four Beckhoff EL2008 8-channel 24 V digital output, one Beckhoff EL4004 0 V–10 V 4-channel analog output, and two Beckhoff EL3064 4-channel analog output modules. The PLC connected to the tank storage system features two Beckhoff EL2008 8-channel 24 V digital output, one Beckhoff EL4004 0 V–10 V 4-channel analog output, one Beckhoff EL3064 0 V–10 V 4-channel analog output and one Beckhoff EL1018 8-channel 24 V digital input module. The recipe execution program was run on a standard PC (Intel® Core™ i7 4800MQ 3.7 GHz, 16 GB) running Windows 7. The PLCs communicated with the recipe PC via standard 100 Mbit $s^{-1}$ Ethernet. Last but not least, an implementation example for a domain service using the open source IEC 61499 development environment Framework for Industrial Automation & Control (4DIAC) is given.

## References


[1] T. Cucinotta, A. Mancina, G. F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusina, "A real-time service-oriented architecture for industrial automation," *IEEE Trans. Ind. Informat.*, vol. 5, no. 3, pp. 267–277, Aug. 2009.

[2] F. Jammes and H. Smit, "Service-oriented paradigms in industrial automation," *IEEE Trans. Ind. Informat.*, vol. 1, no. 1, pp. 62–70, Feb. 2005.

[3] S. Karnouskos, D. Guinard, D. Savio, P. Spiess, O. Baecker, V. Trifa, and L. M. S. De Souza, "Towards the real-time enterprise: service-based integration of heterogeneous SOA-ready industrial devices with enterprise applications," in *IFAC Symp. on Information Control Problems in Manufacturing*, 2009, pp. 2127–2132.

[4] T. Strasser, M. Rooker, G. Ebenhofer, I. Hegny, M. Wenger, C. Sünder, A. Martel, and A. Valentini, "Multi-domain model-driven design of Industrial Automation and Control Systems," in *IEEE Int. Conf. on Emerging Technologies and Factory Automation*, Sep. 2008, pp. 1067–1071.

[5] T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sünder, A. Valentini, and A. Martel, "Structuring of large scale distributed control programs with IEC 61499 subapplications and a hierarchical plant structure model," in *IEEE Int. Conf. on Emerging Technologies and Factory Automation*, Sep. 2008, pp. 934–941.

[6] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," English, *Business & Information Systems Engineering*, vol. 6, no. 4, pp. 239–242, Aug. 2014.

[7] R. Drath and A. Horch, "Industrie 4.0: Hit or Hype? [Industry Forum]," *IEEE Ind. Electron. Mag.*, vol. 8, no. 2, pp. 56–58, Jun. 2014.

[8] H. Bohn, A. Bobek, and F. Golatowski, "SIRENA – Service Infrastructure for Real-time Embedded Networked Devices: A service oriented framework for different domains," in *Int. Conf. on Networking, Int. Conf. on Syst. and Int. Conf. on Mobile Commun. and Learning Technologies*, IEEE Computer Society, 2006, pp. 43–48.

[9] A. Cannata, M. Gerosa, and M. Taisch, "SOCRADES: A framework for developing intelligent systems in manufacturing," in *IEEE Int. Conf. on Ind. Eng. and Eng. Manage.*, Dec. 2008, pp. 1904–1908.

[10] IEC TC65/WG6, *IEC 61499: Function blocks for industrial-process measurement and control systems – Parts 1 to 4*. Geneva: International Electrotechnical Commission (IEC), 2004-2005.

[11] M. Melik-Merkumians, T. Baier, M. Steinegger, W. Lepuschitz, I. Hegny, and A. Zoitl, "Towards OPC UA as portable SOA middleware between control software and external added value applications," in *IEEE Int. Conf. on*

*Emerging Technologies and Factory Automation*, Sep. 2012, pp. 1–8.

[12] Int. Society of Automation, *ISA-88 – Batch Control*, 1995.

[13] J. Virta, I. Seilonen, A. Tuomi, and K. Koskinen, "SOA-Based integration for batch process management with OPC UA and ISA-88/95," in *IEEE Int. Conf. on Emerging Technologies and Factory Automation*, 2010, pp. 1–8.

[14] B. Vogel-Heuser, D. Schütz, T. Frank, and C. Legat, "Model-driven engineering of Manufacturing Automation Software Projects – A SysML-based approach," *Mechatronics*, vol. 24, no. 7, pp. 883–897, 2014.

[15] I. Hegny, O. Hummer, A. Zoitl, G. Koppensteiner, and M. Merdan, "Integrating software agents and IEC 61499 realtime control for reconfigurable distributed manufacturing systems," in *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*, Jun. 2008, pp. 249–252.

[16] W. Lepuschitz, B. Groessing, E. Axinia, and M. Merdan, "Phase Agents and Dynamic Routing for Batch Process Automation," in *Int. Conf. on Ind. Applicat. of Holonic and Multi-Agent Systems*, 2013, pp. 37–48.

[17] M. W. Hofmair, M. Melik-Merkumians, M. Böck, M. Merdan, G. Schitter, and A. Kugi, "Patching process optimization in an agent-controlled timber mill," *J. of Intell. Manufacturing*, vol. September, 2014.

[18] S. Panjaitan and G. Frey, "Functional Control Objects in Distributed Automation Systems," *IMS'07*, pp. 293–298, 2007.

[19] K. P. Sycara, "Multiagent Systems," *AI Magazine*, vol. 19, no. 2, pp. 79–92, 1998.

[20] A. Helsinger, M. Thome, and T. Wright, "Cougaar: a scalable, distributed multi-agent architecture," in *IEEE Int. Syst., Man and Cybern. Conf.*, vol. 2, 2004, pp. 1910–1917.

[21] M. Melik-Merkumians, M. Wenger, R. Hametner, and A. Zoitl, "Increasing Portability and Reuseability of Distributed Control Programs by I/O Access Abstraction," in *IEEE Int. Conf. on Emerging Technologies and Factory Automation*, 2010.

[22] M. Wenger, M. Melik-Merkumians, I. Hegny, R. Hametner, and A. Zoitl, "Utilizing IEC 61499 in an MDA Control Application Development Approach," in *IEEE Conf. on Automation, Sci. and Eng.*, 2011, pp. 495–500.

[23] IEC TC65/WG6, *IEC 61131: Standard - Programmable controllers – Parts 1 to 8*. International Electrical Commission, 2003.

[24] U. T. Bührer, C. Legat, and B. Vogel-Heuser, "Changeability of manufacturing automation systems using an orchestration engine for programmable logic controllers," *IFAC Symp. on Inform. Control Problems in Manufacturing*, vol. 48, no. 3, pp. 1573–1579, 2015.

[25] T. Strasser, G. Ebenhofer, M. Rooker, and I. Hegny, "Domain-Specific Design of Industrial Automation and Control Systems: The MEDEIA Approach," in *10th IFAC Workshop on Intelligent Manufacturing Systems*, 2010.

[26] T. Strasser, C. Sünder, and A. Valentini, "Model-driven embedded systems design environment for the industrial automation sector," in *IEEE Int. Conf. on Ind. Informatics*, Jul. 2008, pp. 1120–1125.

[27] N. Josuttis, *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007.

[28] Festo. (2016). MPS® PA Compact-Workstation, [Online]. Available: http://www.festo-didactic.com/.

[29] D. James, "Best of the batch [batch processing]," *Computing Control Eng. J.*, vol. 17, no. 4, pp. 30–35, Aug. 2006.

[30] D. Brandl, *Design Patterns for Flexible Manufacturing*. ISA, 2006.

[31] G. Godena, I. Steiner, J. Tancek, and M. Svetina, "Design of a Batch Process Control Tool on the Programmable Logic Controller Platform," in *THE WBF BOOK SERIES–ISA 88 IMPLEMENTATION EXPERIENCES*, ser. WBF book series, W. Hawkins, WBF, D. Brandl, and W. Boyes, Eds., Momentum Press, 2010, ch. 14, pp. 157–173.

[32] W. Hawkins, WBF, D. Brandl, and W. Boyes, *The WBF BOOK SERIES–ISA 88 Implementation Experiences*, ser. WBF book series. Momentum Press, 2010.

[33] Eclipse. (2016). Xtext Framework, [Online]. Available: https://eclipse.org/Xtext.

[34] L. Bettini, "A DSL for Writing Type Systems for Xtext Languages," in *Int. Conf. on Principles and Practice of Programming in Java*, ser. PPPJ '11, Kongens Lyngby, Denmark: ACM, 2011, pp. 31–40.

[35] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271,

[36] Eclipse. (2016). 4DIAC - Framework for Industrial Automation & Control, [Online]. Available: https://eclipse.org/4diac/.

[37] A. Zoitl, T. Strasser, and A. Valentini, "Open source initiatives as basis for the establishment of new technologies in industrial automation: 4DIAC a case study," in *2010 IEEE Int. Symp. on Ind. Electronics*, Jul. 2010, pp. 3817–3819.

[38] Eclipse. (). Paho - Open Source messaging for M2M, [Online]. Available: https://www.eclipse.org/paho/.

[39] ——, (). Mosquitto - An Open Source MQTT v3.1/v3.1.1 Broker, [Online]. Available: http://mosquitto.org/.