ITEA 3

OPTIMUM

OPTimised Industrial IoT and Distributed Control Platform
for Manufacturing and Material Handling

Deliverable 3.5
User Guide for Distributed Control Platform

Deliverable type:	Document
Deliverable reference number:	ITEA 16043 | D3.5
Related Work Package:	WP 3
Due date:	2021-04-30
Actual submission date:	2021-05-31
Responsible organisation:	ifak
Editor:	M. Riedl
Dissemination level:	Public
Revision:	Final | Version 0.5

Abstract:	Short description about design principles of programs running on the DCP. The principles will be used in a small example program incl. a simple animation of the behavior.
Keywords:	DCP, design concepts for DPC programs, simple example, engineering tool

Table_head	Name 1 (partner)	Name 2 (partner)	Approval date (1 / 2)
Approval at WP level	K. Meisberger (NXP)	H. Borstel (THORSIS)	2021-05-05 / 2021-05-06
Veto Review by All Partners	No Vetos		2021-05-28

Editor
M. Riedl (ifak)

Contributors
D. L. Tran (ifak)
Tianzhe Yu (ifak)

Executive Summary

This document provides an overview how to use the Distributed Control Platform (DCP). Therefore, information about supported platforms and the way of deployment are given. In order to use a secured communication, the handling of certificates is shortly described.

In order to develop programs based on the DCP, application design principles are described. In order to split the program complexity of controller application into smaller parts, objects are used. An object is responsible for specific sub tasks of the overall program. Here especially the interfaces of the objects, their data and control flow as well as the synchronization of the objects is described.

Finally, a small example demonstrates how the DCP can be used. It enhances the pure demonstration of correctness of the functionality by means of a simple visualization of the dynamics in the control program. By means of this information, control programs can be designed, implemented and commissioned.

Table of Content

Figures

Tables

1 Introduction

The Distributed Control Platform (DCP) can be seen as a middleware. It combines specific services and makes them available in a parameterizable form. The intention of this document is to provide an overview about provided packages for the different platforms and describe the design principles from a developer's perspective. For beginners, it includes a small example inclusively animation of the program behavior.

2 Deployment

The runtime will be deployed for following platforms:

- Linux, Ubuntu x86 based as Debian-Package
- Linux, ARM V7, Rasbian OS, as Debian-Package
- Win10, x64 based, as part of the engineering tool "DOME Studio"

3 Preconditions

DCP is based on DOME [1] and uses for the first contact at a controller node a fixed Ethernet port. This port is defined as 39179. All connections established at runtime use dynamic assigned port numbers. The runtime uses freely available ports. This implies that the operating system has to provide a dedicated number of free Ethernet ports.

Furthermore, the runtime requires root privileges e.g. to access hardware interfaces.

4 Deployment of Application Programs

The automation objects building the application program for the distributed control application will be deployed in form of binary libraries. This binary libraries contain the compiled objects for the specific controller platform. The libraries can be deployed manually e.g. via command line access to the controller node or by means of the engineering tool DOME Studio as introduced in [2]. The part of the distributed application running at a specific node will be loaded dynamically during start-up of the distributed application by means of configuration files (see section 6.1). The DCP runtime loads the referenced libraries automatically and instantiates the needed automation objects.

5 Handling of Certificates

In order to realize authentication and secure communication between DCPs, certificates and credentials are required. The structure and usage of certificates are implemented as follows (see also Figure 1):

1. DOME Studio imports a Certificate Authority (CA) certificate retrieved from Optimum webservice / from file system and save it as Dome 'Root' CA Certificate. When there is no such service, DOME Studio will generate a self-signed Root CA Certificate.
2. Two Sub CAs will be generated in DOME Studio with Dome 'Root' CA Certificate. These two Sub CAs issue End Certificates to DCP. ReadWrite/ReadOnly access level can be distinguished by checking the certificate issuer.
3. A DCP will ask DOME Studio for an End Certificate issued by one of the Sub CAs. CA Certificates (Dome 'Root' CA Certificate + Dome ReadOnly Sub CA Certificate + Dome ReadWrite Sub CA Certificate) will also be sent along with End Certificate. These certificates will be used in the TLS communication.

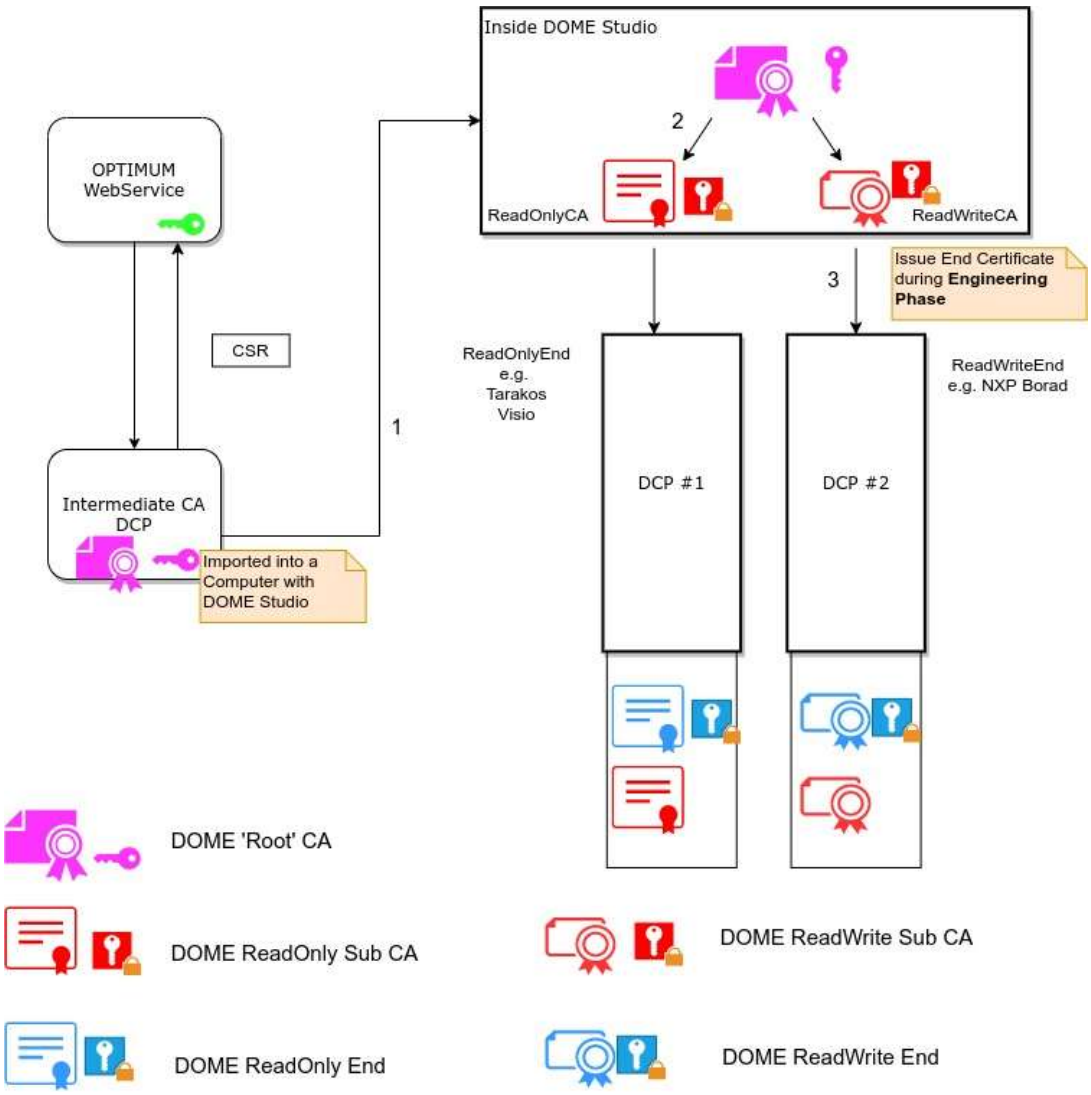

Figure 1: Structure of Certificates

6 Application Design

6.1 General Considerations

The application running in DCP is organized as network of interconnected objects. The interfaces of objects are called port. Only by means of such ports the information and invocation flow can be designed.

An object algorithm itself is implemented by means of an object-oriented language. C++ is used as low-level language, efficient and portable solution. For this programming language there are predefined toolchains for a wide area of hardware platforms.

The interconnection points, the ports and their types, are defined by the DOME-L language as introduced in [2].

Finally, the interactions are engineered as links. These connections can be expressed using either a special configuration language DOME-C (as introduced in [2]) or the generic scripting language Lua [3].

DCP follows the concept of Actor Pattern [4]. This design pattern defines that each object interacts with other objects by means of messages. Depending on the contents of a received message, the object can process its own business rules (internal algorithm). This means the object can e.g. interact with IO sub-systems or generate own messages send to other objects. Following this approach, each object has to be an actor.

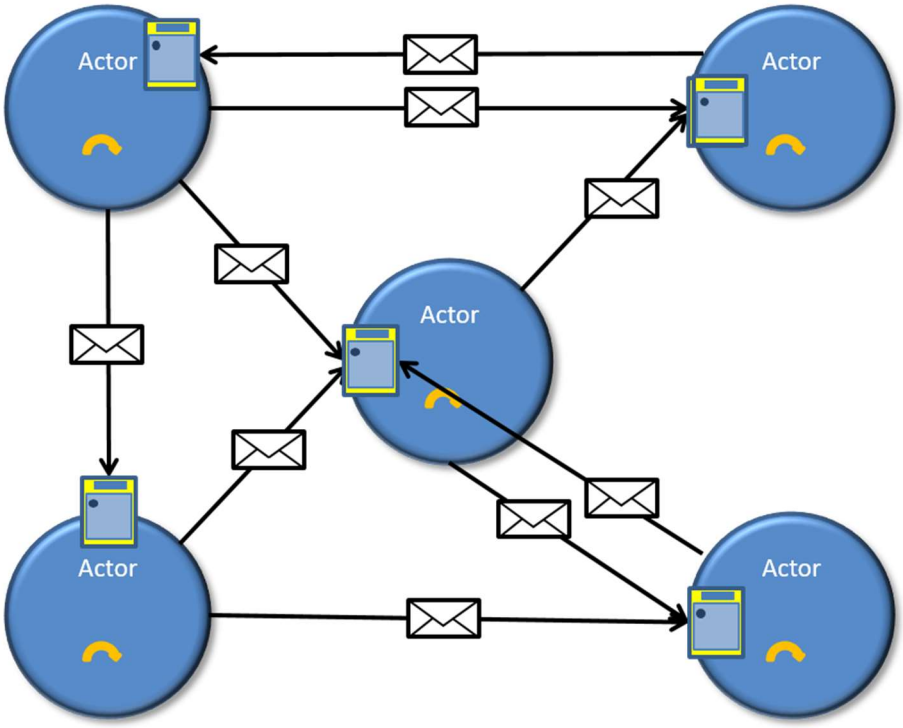

Figure 2: Actor Model

This concept is an ideally basis for concurrent or distributed computation, given that each object has its (own) execution context and there is a communication system transporting the messages.

6.2 Object Synchronization

Any kind of code and also objects according the object oriented programming paradigm will be executed in an execution context. Normally, a program is executed in a process environment managed by the operating system. Depending on the operating system, smaller environments such as threads are also available. The program execution starts at a specific entry point, e.g. the `main()` function in C/C++ or a specific function for a thread, that is defined by the program specifically. The sequence of instructions performed by the process or thread are defined inside the program or by means of control statements like loops, conditional jumps or subroutine invocations (function or method calls).

6.3 Kind of Objects

In DCP, two types of objects are known in principle: Active Objects and 'normal' Objects. The following subsections introduce these object types.

6.3.1 Active Objects

An Active Object will be instantiated by a special class `dome_active_class` defined in DOME-L. An Active Object has its own execution context, implemented as a thread. Furthermore, there is a predefined method, which will be called by the thread. It depends on

the task of the object, what kind of code is implemented there. For instance a timer would fire an event by a required port in a loop within a predefined period.

6.3.2 Objects

In contrast to an 'Active Object', a ‘normal’ Object requires an external execution context.

So, except the developer has not implemented its own concurrent execution of code, all data inside this object is thread safe, because synchronization is done from outside. Such an object will be instantiated by the class `dome_class` in DOME-L.

6.4 Interfaces of Objects

An object in DOME provides interfaces in form of ports. A port can be seen as a specific connection point for sending or receiving messages. A port is of a specific type that defines the data, the data types and the directions of the data flow. Following a service oriented approach, an object offers services. If an object consumes services from other objects, it requires such services for its own business logic. By means of this, the roles of ports are called service ports and required ports.

Messages are transmitted between ports. For compatible message types, each port is typed and only type compatible ports can be interconnected.

6.5 Synchronization of Objects

The synchronization between message handling or method invocations is done by means of message queues. Each message send by a remote object to one of the objects organized in the same execution context is put into a message queue of the execution context (Nb 1 in Figure 3). This means, the message is not transmitted directly to the receiving port of the object. If the operating system scheduler activates the execution context, a message is taken from the queue and handed over to the receiving port of the object (Nb 2 in Figure 3). The port unmarshalls the message parameters and invokes the associated method.

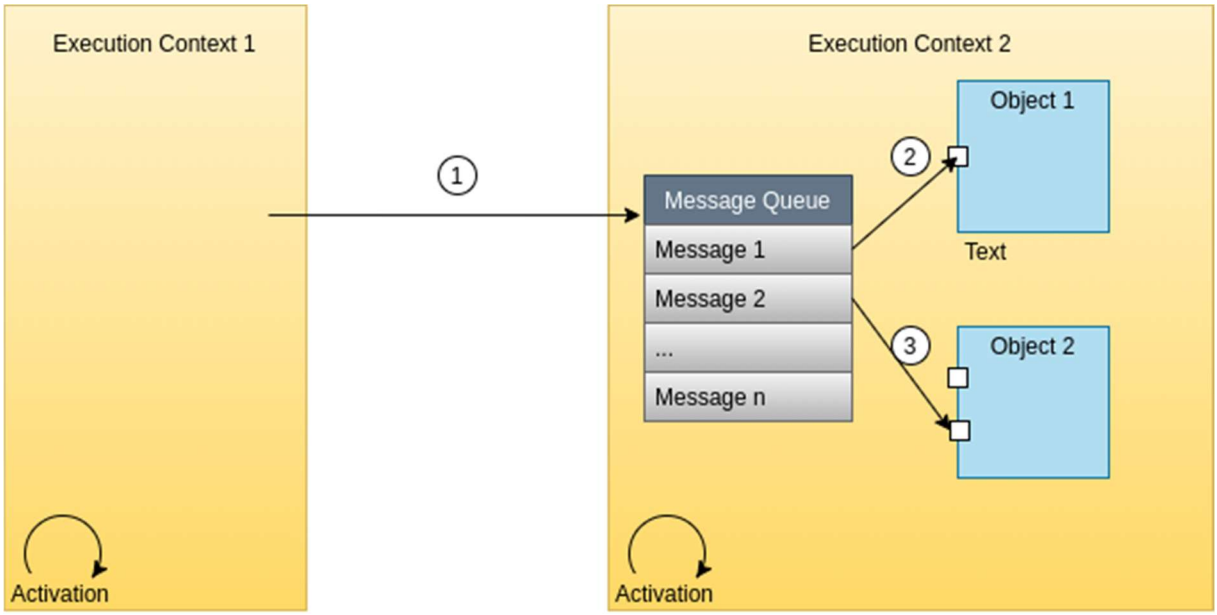

Figure 3: Synchronization between execution contexts

In case of higher priority tasks, the method execution may be interrupted by the operating system. When this method call is completed, the next message is taken from the queue, if the

execution context has the ability to do so or during next activation (Nb 3 in Figure 3). Figure 4 shows the sequence diagram of this scenario.

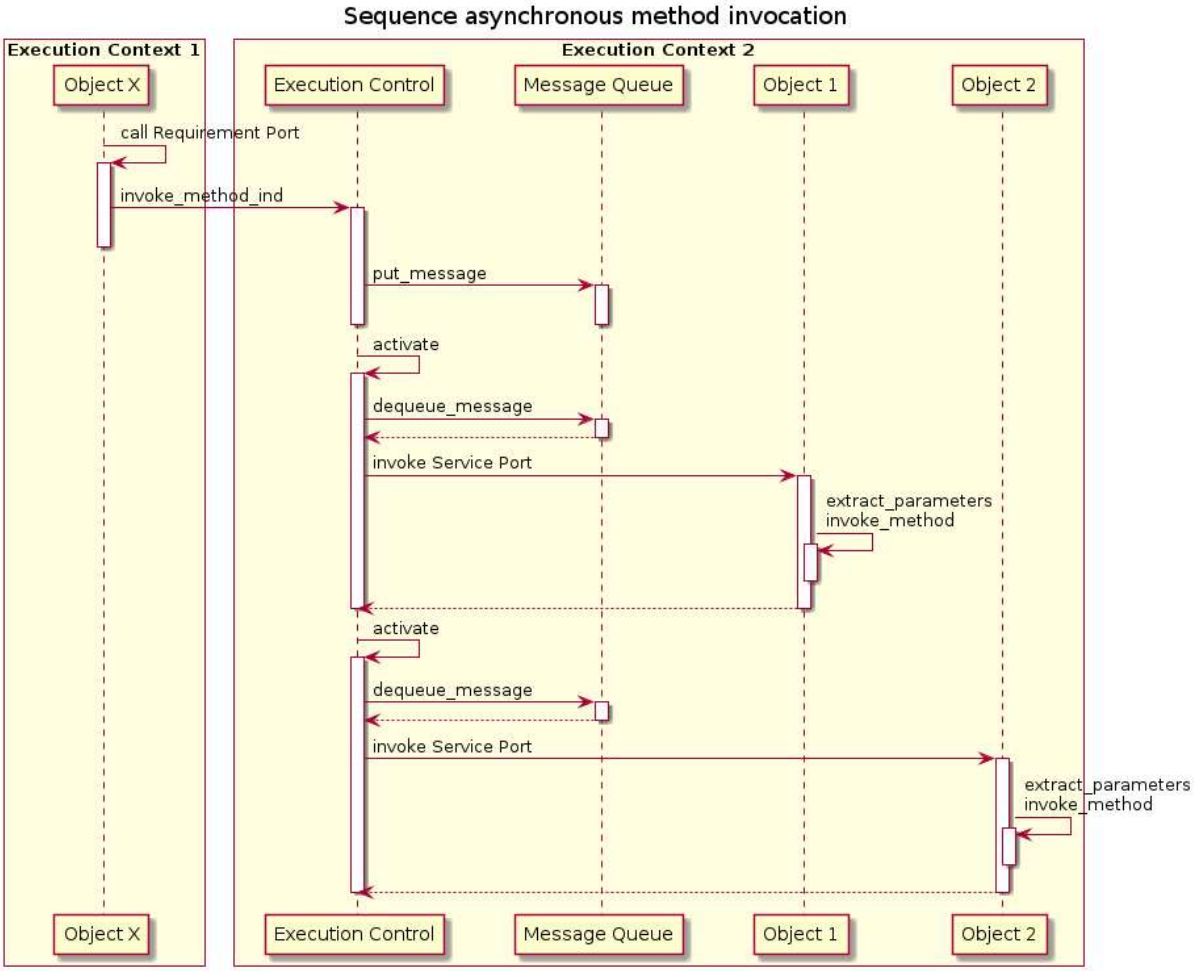

Figure 4: Sequence of asynchronous method invocation

The sequence of invocations is different, if a method invocation has to communicate with other objects of the same execution context. In this case, the requirement port of the object transmits without any interruption the invocation to the related object's port and the associated method is also invoked immediately (Nb 3 in Figure 5). Such behavior is analogous to method invocation from one object to another in normal object-oriented programming languages. This implies, that the next message coming from a remote execution context is handled after the sequence described above (Nb. 4).

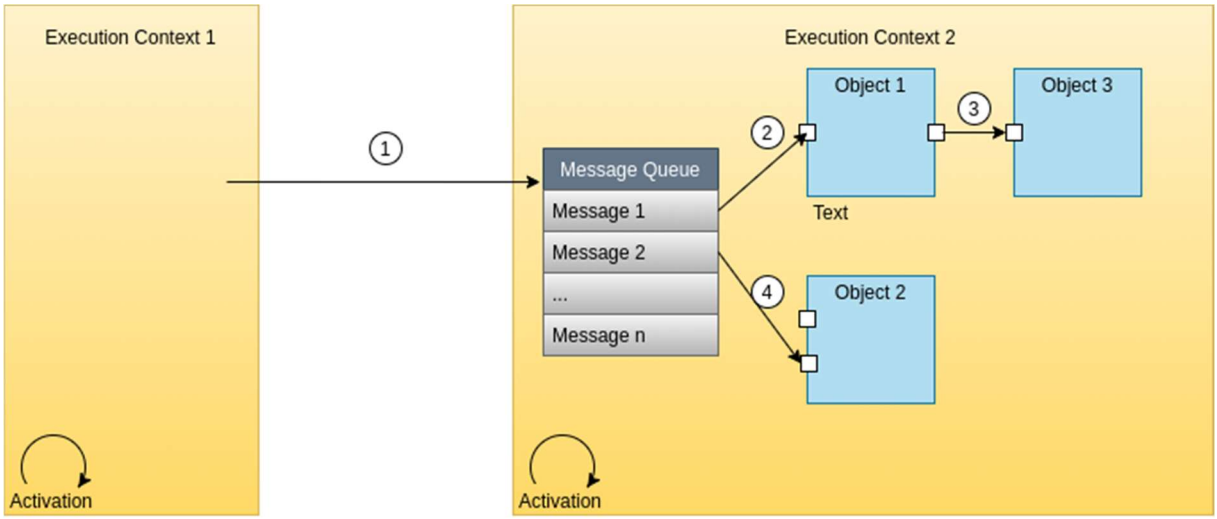

Figure 5: Synchronization inside same execution contexts

Figure 6 shows the sequence diagram of this scenario.

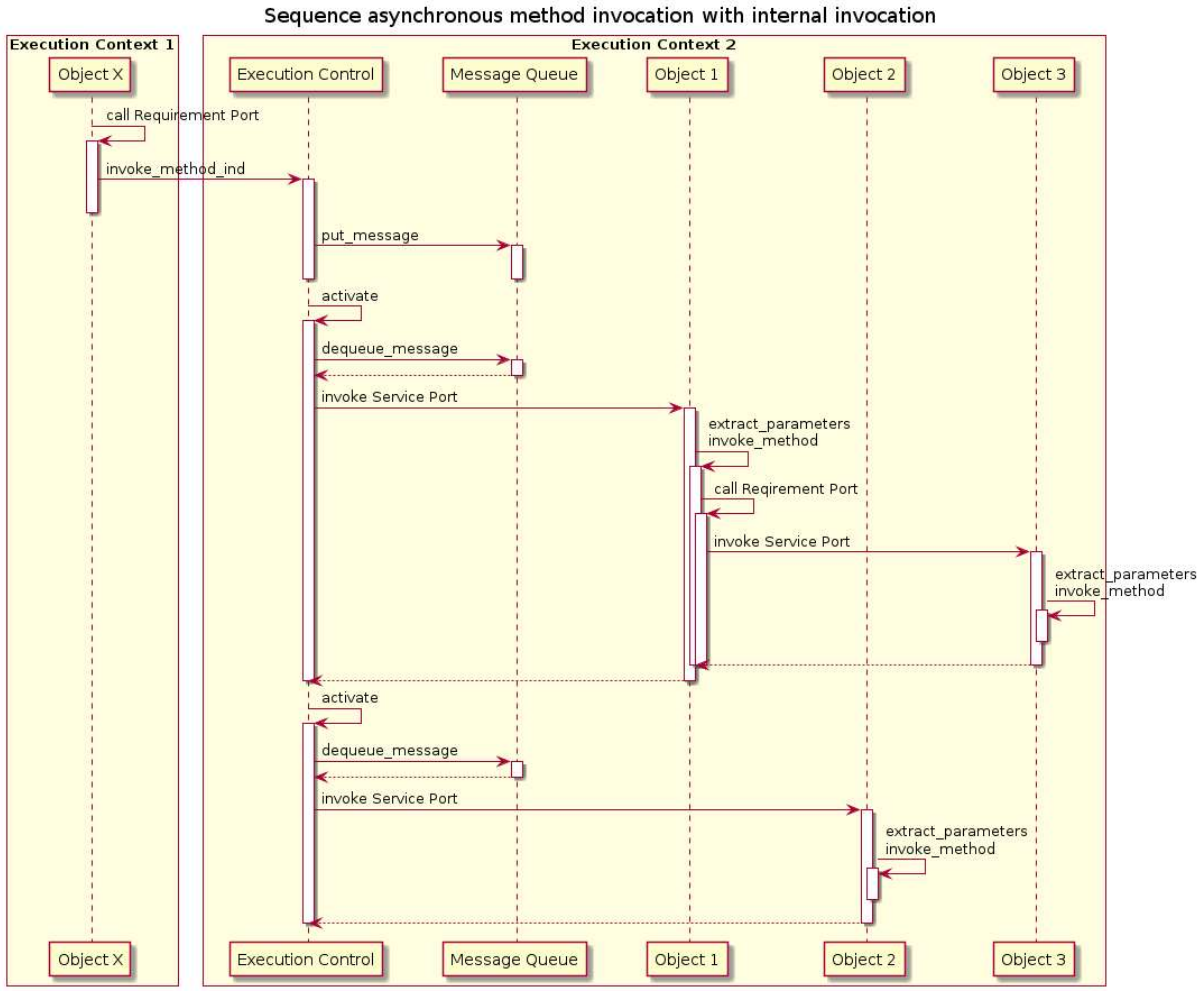

Figure 6: Sequence of asynchronous method invocation with internal invocation

6.5.1 Asynchronous Call

Asynchronous calls can be performed between objects of different execution contexts, if the port definition does not require any return parameter (out parameter or return values).

In order to ensure, that the DOME middleware has not to wait for responses of the port invocation, the port shall be annotated with `nonblocking` flag.

However, this special asynchronous port annotation is not affecting port invocations between objects running inside the same execution context. Such invocations are always synchronous.

6.5.2 Synchronous Call

Synchronous calls will always be performed between objects of the same execution context. In addition, if the port is annotated as `blocking`, the port invocation between objects of different execution contexts is done synchronous. In this case, output values, return values or exceptions are responded to the invoking instance.

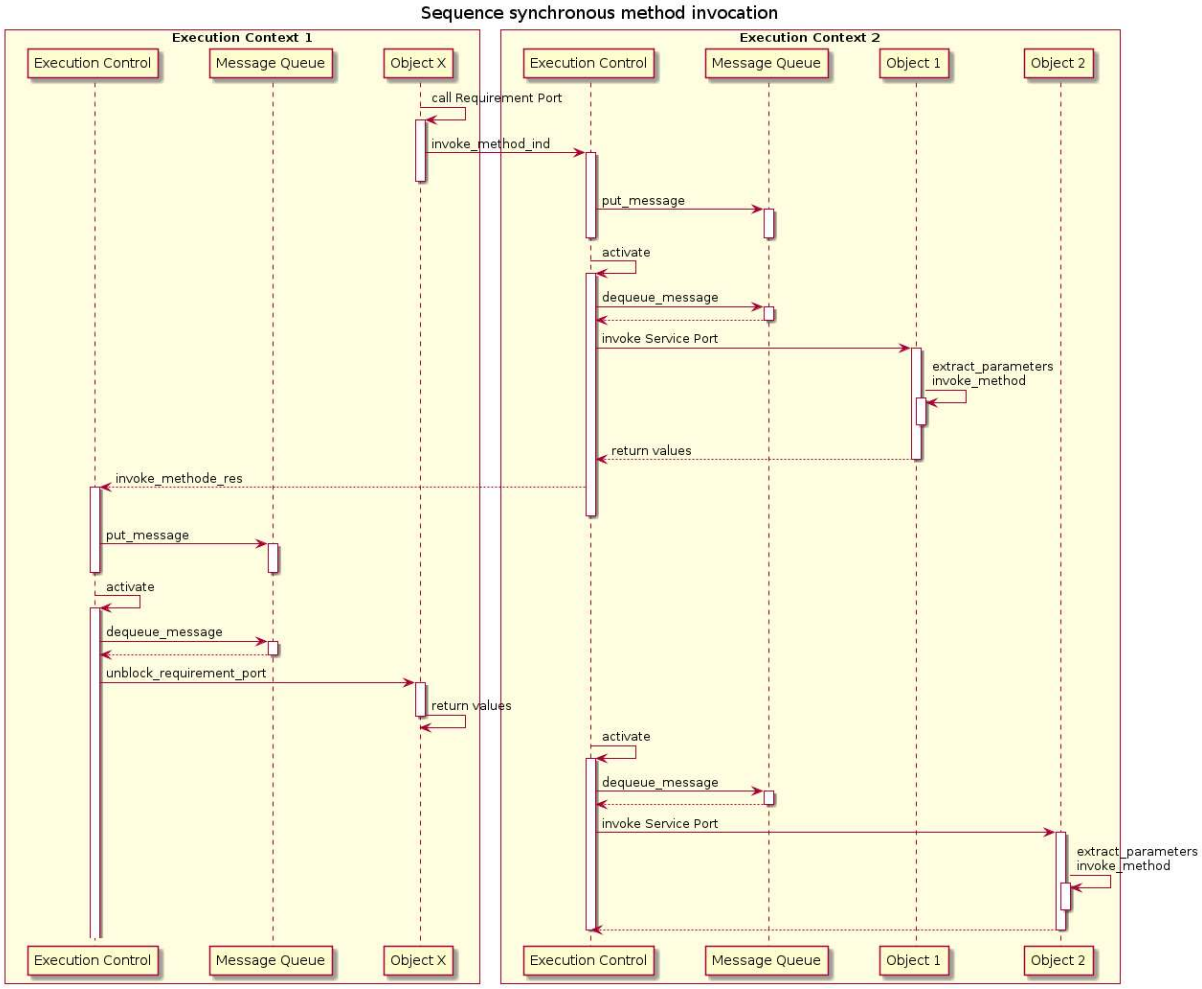

Figure 7: Sequence of synchronous method invocation

The requirement port has to wait for the response, i.e. the complete processing of the remote service. The transport of the response message works analogous to the invocation, except the encoding in the message queue detects the message type and unblocks the waiting requirement port.

6.6 Object Development

6.6.1 Module Information and Usage of Other Sources

To create an object for DCP the source code has to be written in textual form with a text editor or by means of the engineering tool. One or more objects can be put in same library or module.

For documentation reasons extensive concepts in the form of comments are available. Helpful information about the module like version number and a description of the module is possible to define.

```
version = 0#1;
description = "Short description of the OPTIMUM example module.";
help = "Author: Matthias Riedl"
       "Vendor: ifak e.V., Magdeburg, Germany"
       "Definition of objects need for small example";
```

Figure 8: Meta information of a module / library

The `version` of the module is handled by the developer on its own. This number is composed by two sub numbers - the main version and the sub version. Changes in the main version document incompatible modifications according to earlier versions, changes in the sub version explain bug fixes respectively modifications, which do not effect interface definitions.

The `description` has the purpose of a short description of the functionality inside the module. The developer is responsible for the contents.

More detailed information about the module like vendor/author information can be lodged in the `help` entry. The developer is responsible for the contents.

Furthermore, the usage of source code of the target language is prepared (see A.1). The following features are supported:

- `IMPORT` - content of code (e.g. include files) will be copied before the namespace of the type declarations
- `EXTERN` - content of code (e.g. `extern`-definitions) will be copied inside the namespace before the type declarations
- `EXPORT` - content of code will be copied inside the namespace behind the type declarations
- `GLOBAL` - content of code will be copied before the namespace of the implementation
- `LOCAL` - content of code will be copied behind the namespace of the implementation

6.6.2 Ports

An optional special documentation (see above) can be defined before the port description.

```
/**
 * Source documentation area, e.g. for the port definition
 */
dome_port angle_port [nonblocking]
<
  description = "Transport of an uint16 value";
  help = "The value may be interpreted as an angle, range 0-360";
  return = void;
  args = uint16 angle;
>
```

Figure 9: Code of port type 'angle_port'

The example defines a port for the transport of a `dome::int16` value. Following the keyword `dome_port` the unique name of the port is required (unique inside the module).

After the name of the port follows an optional flag about the blocking/non-blocking functionality of the port. If this flag is omitted, the default `blocking` behavior is used. The `non-blocking` behavior is required for the asynchronous communication between automation objects in different groups.

Each port can have its own optional `description` entry about the idea of this interface and should explain the signature and return value of the port. For more detailed information, the optional `help` entry can be used. The developer is responsible for both contents.

The entry of `return` defines the type of return of the port. If the port has set the `non-blocking` flag, a return type of `void` is required (this will be checked by the DOME-L compiler).

For each port an unrestricted number of formal parameters can be defined. This will be done with the `args` entry. The formal parameter definition consists of the type and the name of the parameter, the parameters are separated by ',' .

6.6.3 Classes

An automation object of DCP will be defined in DOME-L by the keyword `dome_class` or `dome_active_class`. Before the class description an optional special documentation (see above) can be defined. Each class has a unique name inside the module and can inherit from another class of an automation object. If no base class is referenced, the class `dome::ClassObject` of the DCP runtime will be used.

```
/**
 * Automation object documentation
 */
dome_class angle_per_tick
<
      help = "Calculates a new values of the angle per tick";
      description = "Calculates a new values of the angle per tick, range
0...360 degree";
                  "The different of the angel is set by property
_delta_angle.";

      requires dome_port recent_angle: angle_port;

      property _delta_angle[type = uint16, access = readwrite];

      attributes
      {
            uint16 _angle;
      }

      prepare_init
      {
        _angle = 0;
      }

      construct
      {
        _delta_angle = 10;
      }

      service dome_port tick: tick_port
      {
        _angle = (_angle + _delta_angle) % 360;
        //debug("tick, new angle: %d\n", _angle);
        recent_angle(_angle);
      }
>
```

Figure 10: Code of class 'angle_per_tick'

Each class can have its own optional `description` entry about the idea of this class. For more detailed information the optional `help` entry can be used. The developer is responsible for both contents.

After these entries, the definition of the service and requirement ports of the class can be done. These port definitions determine the interface of the class. The beginning keywords `service` and `require` define the role of a port type in relation to the class. Then the class internal name of the port and the port type must be defined.

The ports of the role `services` implement an algorithm. Therefore the source code of the target language shall be placed inside the curly braces, e.g. the source code in C++ as shown in the example.

The ports of the role `requires` have no implementation. They can be used inside the algorithms as a function call (with actual parameters and return values, if defined).

Furthermore, following optional, special methods can be defined:

- `construct` – the constructor of the class, values of attributes and properties should be initialized
- `destruct` – the destructor of the class
- `init` – this method will be called, when the state machine is set to RUN with the INIT command; if the class supports the functionality of reset to default values, this should be done here
- `save` – method for saving the object state into a persistent stream
- `restore` – method for restoring the object state from a persistent stream

The section `attributes` offers the possibility to define class internal attributes. By default, the visibility of the attributes is private. Other access modifiers are possible. Inside this section, the target language specific syntax must be used, e.g. C++.

The keyword `property` starts a definition for a public attribute, called property. A property value can be read/set with the help of the runtime. Following the keyword, the name of the property is required. After that, the definition of the data type of the property and its access right shall be defined. If none access right is defined, the property value can be read and written.

The last possibility inside the `dome_class` declaration is the definition of internal methods. Such methods have the access right of protected. Before the method description an optional special documentation (see above) can be defined. The start of a method definition is initialized by the keyword method. After that, the data type of the return value must be defined in the target language, enclosed in ` ` signs. The same syntax has to be used for the definition of the formal parameters.

6.6.4 Active Classes

In contrast to normal classes, that show activity only when triggered by a service, active classes have their own behavior. Active classes and normal classes share most features. The list below shows the differences.

- Active classes do not reside in groups
- Active classes must implement its behavior in a `work()` method. It is executed by a unique thread.
- Active classes may be decorated by an attribute delay.
- The keyword `dome_active_class` is used to create active classes (instead of `dome_class`).

If `delay` is specified, the value of the attribute shall contain the time Δt microseconds. If specified, work is called every Δt microseconds. If `delay` is not provided, `work()` is executed as often as possible. Active classes may not inherit classes. Classes may not inherit active classes. If an active class inherits another active class, the base classes `work()` and `delay` definitions are ignored for the inheriting active class.

6.7 Predefined Streams for Text Messages

6.7.1 Aim

Text messages help during development and commissioning of the objects or the overall application. The runtime provides three streams to push test messages which can be used by the developers. The stream can be connected to different targets, depending on the specific needs.

All streams expect a format string parameter following the syntax of `printf()` [5] of programming language C.

6.7.2 Stream audit()

The stream `audit()` is used to continuously record all changes to an object. Such logbooks are required, for example, in the food or chemical industry to prove the quality of products.

6.7.3 Stream info()

The stream `info()` is intended to provide information about an object, a method invocation etc. Default is to print to `stdout` of a shell, if this stream target is available.

6.7.4 Stream debug()

The stream `debug()` is intended to provide information about an object, a method invocation etc. during development. The code has not to be changed. If the object code is compiled with no debugging information, such stream invocations are filtered out. Default is to print to `stdout` of a shell, if this stream target is available.

7 Application Example

7.1 Introduction

A small example should demonstrate the functionality. The example does not require any hardware specifics input / output. It shows the executions and flexibility of the DCP concepts and the feature of the engineering tool. In principle the example can be created also without the engineering tool. In this case, the configuration has to do manually.

The example emulates a motor engine of a four stroke engine (Otto or Diesel). Therefore one needs a crankshaft and a cylinder. These objects have to be modelled. In addition we need a stimulation for the rotation of the crankshaft. The stimulation is done by means of a trigger to move the crankshaft for a predefined angle.

In the end, the results should be visualized. This will be done via the streams explained above and via the graphical visualization integrated in engineering tool.

7.2 Create the Crankshaft

The source code of the crankshaft is already shown in Figure 10. The behavior is emulated by a conversion from time tick into new angle, were the delta is defined by the property `_delta_angle`. The need time tick is provided by another object, what is shown later. The class defines a service port `tick`, receiving the time ticks. The definition of this port does not require any formal parameter. Such port type is equivalent to an event. The code snipped shows the shortest definition of a port:

```
dome_port tick_port [nonblocking]
<
>
```

Figure 11: Code of port type 'tick_port'

Inside the implementation of `tick`, the new angle is calculated and passed to the required port `recent_angle`. Where this new value is used in a control application, is out of the scope of the class `'angle_per_tick'`.

7.3 Create the cylinder

The cylinder shall emulate the four phases of the four stroke engine. Depending on the overall construction of a motor engine, a specific cylinder has a specific angle in reference to the crankshaft and a specific status of the phase. Such configuration will be set during initialization of the object.

```
dome_class cylinder
<
  description = "Emulation of a cylinder of a four-stroke engine";

  /**
   * Offset angle of the cylinder in relation to the crankshaft
   */
  property _offset[type = int16, access = readwrite];

  /**
   * Name of the cyclinder instance.
   */
  property _name[type = string, access = readwrite];

  /**
   * initial status of cylinder (0-3)
   */
  property _init_status[type = int16, access = readwrite];

  requires dome_port position : animation_position;

  requires dome_port status : animation_status;

  attributes
  {
    /**
     * angle of the cylinder
     */
    int16 _angle;

    /**
     * next expected angle with changes
     */
    int16 _next_expected;
```

```
  /**
   * status changed, so explosion happens once
   */
  bool _print_peng;

  /**
   * strokes of a 4-stroke engine
   */
  enum class stroke_type
  {
    SUCKING = 0,
    COMPENSING,
    WORKING,
    EJECTION
  };
  stroke_type _clock_stroke;
}

prepare_init
{
_offset = _offset % 360;
_angle = _offset;
_clock_stroke = static_cast<stroke_type>(_init_status());

  switch (_clock_stroke)
  {
    case stroke_type::SUCKING:
      _next_expected = 180;
      break;
     case stroke_type::COMPENSING:
      _next_expected = 0;
      break;
    case stroke_type::WORKING:
      _next_expected = 180;
      break;
    case stroke_type::EJECTION:
      _next_expected = 0;
      break;
  }
  _print_peng = false;
}

/**
 * Initials the angle by the offset
 */
construct
{
  _angle = 0;
  _offset = 0;
```

```
    _name = "Cyl_XYZ";
    _clock_stroke = COMPENSING;
    _print_peng = false;
  }

  /**
   * The new angle of the crankshaft is passed. The cyclinder calculates
new intenal angle and the new status
   */
  service dome_port set_angle: angle_port
  {
    int intern_angle_mod_360 = (_offset + angle) % 360;
    int intern_angle_mod_180 = intern_angle_mod_360 % 180;
    int last_angle_mod_180 = _angle % 180;
    bool change_state = false;

    switch (_clock_stroke)
    {
        case stroke_type::SUCKING:
        case stroke_type::WORKING:
            if ((intern_angle_mod_180 > _next_expected)||
                (last_angle_mod_180 > intern_angle_mod_180))
            {
                change_state = true;
                _next_expected = 0;
            }
            break;
        case stroke_type::COMPENSING:
        case stroke_type::EJECTION:
            if ((_next_expected > intern_angle_mod_180) ||
                (last_angle_mod_180 > intern_angle_mod_180))
            {
                change_state = true;
                _next_expected = 180;
            }
            break;
    }

    if (change_state)
    {
        switch (_clock_stroke)
        {
            case stroke_type::SUCKING:
            {
                _clock_stroke = stroke_type::COMPENSING;
                break;
            }
            case stroke_type::COMPENSING:
            {
                _print_peng = true;
```

```
                _clock_stroke = stroke_type::WORKING;
                break;
            }
            case stroke_type::WORKING:
            {
                _clock_stroke = stroke_type::EJECTION;
                break;
            }
            case stroke_type::EJECTION:
            {
                _clock_stroke = stroke_type::SUCKING;
                break;
            }
        }
    }

    _angle = intern_angle_mod_360;
    show_status();

    float32 val;
    if ((0 <= _angle) && (180 > _angle))
      val = (float32)_angle;
    else
      val = (float32)(360 - _angle);

    position(val*100/180);

    int32 color = cyl_color_val();
    status(color);
  }

  /**
   * Shows the status of the cylinder in 4-stroke engine in debug stream.
   */
  method "void"
  show_status()
  {
    if (_print_peng)
    {
        debug("%s  ! ! ! P E N G ! ! ! \n", _name().c_str());
        _print_peng = false;
    }

    std::string str_phase = phase();
    debug("%s: Angle: %d, Phase: %s\n", _name().c_str(), _angle,
str_phase.c_str());
  } // show_status

  /**
```

```
   * Provides a human readable string about the status of the cylinder in
4-stroke engine.
   * \return const string - status of the cyliner
   */
  method "const string"
  phase()
  {
    string ret = "phase";

    switch (_clock_stroke)
    {
        case stroke_type::SUCKING:
        {
            ret = "SUCKING_IN";
            break;
        }
        case stroke_type::COMPENSING:
        {
            ret = "COMPENSING";
            break;
        }
        case stroke_type::WORKING:
        {
            ret = "WORKING";
            break;
        }
        case stroke_type::EJECTION:
        {
            ret = "EJECTION";
            break;
        }
    }

    return ret;
  } // phase

  /**
   * Provides the coded color of the cylinder for the animation:
   * SUCKING, COMPENSING: 0
   * WORKING, EJECTION: 1
   * \return const int32: coded color
   */
  method "const int32"
  cyl_color_val()
  {
    int32 ret = 0;

    switch (_clock_stroke)
    {
        case stroke_type::SUCKING:
```

```
        {
            ret = 0;
            break;
        }
        case stroke_type::COMPENSING:
        {
            ret = 0;
            break;
        }
        case stroke_type::WORKING:
        {
            ret = 1;
            break;
        }
        case stroke_type::EJECTION:
        {
            ret = 1;
            break;
        }
    }

    return ret;
} // cyl_color_val
>
```

Figure 12 shows one example how the source code of the class could be implemented.

```
dome_class cylinder
<
  description = "Emulation of a cylinder of a four-stroke engine";

  /**
   * Offset angle of the cylinder in relation to the crankshaft
   */
  property _offset[type = int16, access = readwrite];

  /**
   * Name of the cyclinder instance.
   */
  property _name[type = string, access = readwrite];

  /**
   * initial status of cylinder (0-3)
   */
  property _init_status[type = int16, access = readwrite];

  requires dome_port position : animation_position;

  requires dome_port status : animation_status;

  attributes
```

```
{
  /**
   * angle of the cylinder
   */
  int16 _angle;

  /**
   * next expected angle with changes
   */
  int16 _next_expected;

  /**
   * status changed, so explosion happens once
   */
  bool _print_peng;

  /**
   * strokes of a 4-stroke engine
   */
  enum class stroke_type
  {
    SUCKING = 0,
    COMPENSING,
    WORKING,
    EJECTION
  };
  stroke_type _clock_stroke;
}

prepare_init
{
_offset = _offset % 360;
_angle = _offset;
_clock_stroke = static_cast<stroke_type>(_init_status());

  switch (_clock_stroke)
  {
    case stroke_type::SUCKING:
      _next_expected = 180;
      break;
     case stroke_type::COMPENSING:
      _next_expected = 0;
      break;
    case stroke_type::WORKING:
      _next_expected = 180;
      break;
    case stroke_type::EJECTION:
      _next_expected = 0;
      break;
```

```
    }
    _print_peng = false;
  }

  /**
   * Initials the angle by the offset
   */
  construct
  {
    _angle = 0;
    _offset = 0;
    _name = "Cyl_XYZ";
    _clock_stroke = COMPENSING;
    _print_peng = false;
  }

  /**
   * The new angle of the crankshaft is passed. The cyclinder calculates
new intenal angle and the new status
   */
  service dome_port set_angle: angle_port
  {
    int intern_angle_mod_360 = (_offset + angle) % 360;
    int intern_angle_mod_180 = intern_angle_mod_360 % 180;
    int last_angle_mod_180 = _angle % 180;
    bool change_state = false;

    switch (_clock_stroke)
    {
        case stroke_type::SUCKING:
        case stroke_type::WORKING:
            if ((intern_angle_mod_180 > _next_expected)||
                (last_angle_mod_180 > intern_angle_mod_180))
            {
                change_state = true;
                _next_expected = 0;
            }
            break;
        case stroke_type::COMPENSING:
        case stroke_type::EJECTION:
            if ((_next_expected > intern_angle_mod_180) ||
                (last_angle_mod_180 > intern_angle_mod_180))
            {
                change_state = true;
                _next_expected = 180;
            }
            break;
    }

    if (change_state)
```

```
    {
        switch (_clock_stroke)
        {
            case stroke_type::SUCKING:
            {
                _clock_stroke = stroke_type::COMPENSING;
                break;
            }
            case stroke_type::COMPENSING:
            {
                _print_peng = true;
                _clock_stroke = stroke_type::WORKING;
                break;
            }
            case stroke_type::WORKING:
            {
                _clock_stroke = stroke_type::EJECTION;
                break;
            }
            case stroke_type::EJECTION:
            {
                _clock_stroke = stroke_type::SUCKING;
                break;
            }
        }
    }

    _angle = intern_angle_mod_360;
    show_status();

    float32 val;
    if ((0 <= _angle) && (180 > _angle))
      val = (float32)_angle;
    else
      val = (float32)(360 - _angle);

    position(val*100/180);

    int32 color = cyl_color_val();
    status(color);
}

/**
 * Shows the status of the cylinder in 4-stroke engine in debug stream.
 */
method "void"
show_status()
{
    if (_print_peng)
```

```
    {
        debug("%s  ! ! ! P E N G ! ! ! \n", _name().c_str());
        _print_peng = false;
    }

    std::string str_phase = phase();
    debug("%s: Angle: %d, Phase: %s\n", _name().c_str(), _angle,
str_phase.c_str());
  } // show_status

  /**
   * Provides a human readable string about the status of the cylinder in
4-stroke engine.
   * \return const string - status of the cyliner
   */
  method "const string"
  phase()
  {
    string ret = "phase";

    switch (_clock_stroke)
    {
        case stroke_type::SUCKING:
        {
            ret = "SUCKING_IN";
            break;
        }
        case stroke_type::COMPENSING:
        {
            ret = "COMPENSING";
            break;
        }
        case stroke_type::WORKING:
        {
            ret = "WORKING";
            break;
        }
        case stroke_type::EJECTION:
        {
            ret = "EJECTION";
            break;
        }
    }

    return ret;
  } // phase

  /**
   * Provides the coded color of the cylinder for the animation:
   * SUCKING, COMPENSING: 0
```

```
   * WORKING, EJECTION: 1
   * \return const int32: coded color
   */
  method "const int32"
  cyl_color_val()
  {
    int32 ret = 0;

    switch (_clock_stroke)
    {
        case stroke_type::SUCKING:
        {
            ret = 0;
            break;
        }
        case stroke_type::COMPENSING:
        {
            ret = 0;
            break;
        }
        case stroke_type::WORKING:
        {
            ret = 1;
            break;
        }
        case stroke_type::EJECTION:
        {
            ret = 1;
            break;
        }
    }

    return ret;
} // cyl_color_val
>
```

Figure 12: Code of class 'cylinder'

The class defined the constructor as well as the initialization of the object by means of the properties. Depending on the given angle offset and the start status the internal parameters will be calculated.

There are two requirement port used for the animation only. By means of these ports the movement of the cylinder and the phase can be visualized.

7.4 Use common File for Port Definitions

The separation of class definitions into separate files is common in object oriented programming. Following this approach, global type definitions or port definitions have to be declared more than once. This is error prone and should be avoided. In the simple example,

al port definitions are summarized in one file and will be included in the class definitions where needed.

```
import "port_definitions.dome";
```

Figure 13: Example to reuse code via import

7.5 Compilation of the classes

The example application shall run on the same Windows 10 computer as the program is developed on. E.g. the "Visual Studio Build Tools 2019" are available for this purpose, for example (free download). The following instructions are based on these compiler tools. If another compiler tool chain has been installed, the settings must be adjusted accordingly.

The engineering tool DOME Studio offers option to configure the tool chain to use. Via Configure/Settings/C++ the tool chain can be adjusted. Figure 14 shows the default settings that should work.

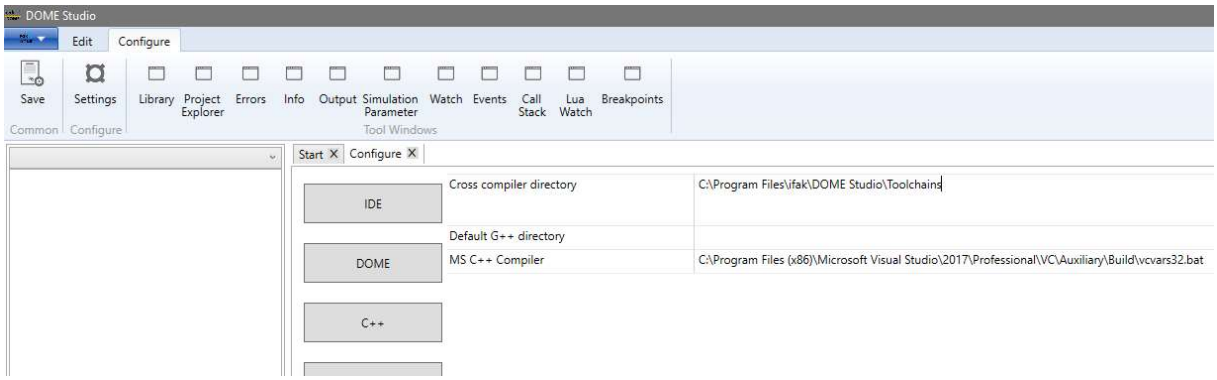

Figure 14: Settings for compiler tool chain

7.6 Design of the Application

The design of the application is also done by the engineering tool DOME Studio, see Figure 15. The classes can be defined in textual form. All files belonging to an application are grouped in a project, represented as a tree.

The class can be compiled by using the tool internal Build function, see button in Figure 15. Warnings or error messages are listed in an output window at the bottom of the tool.

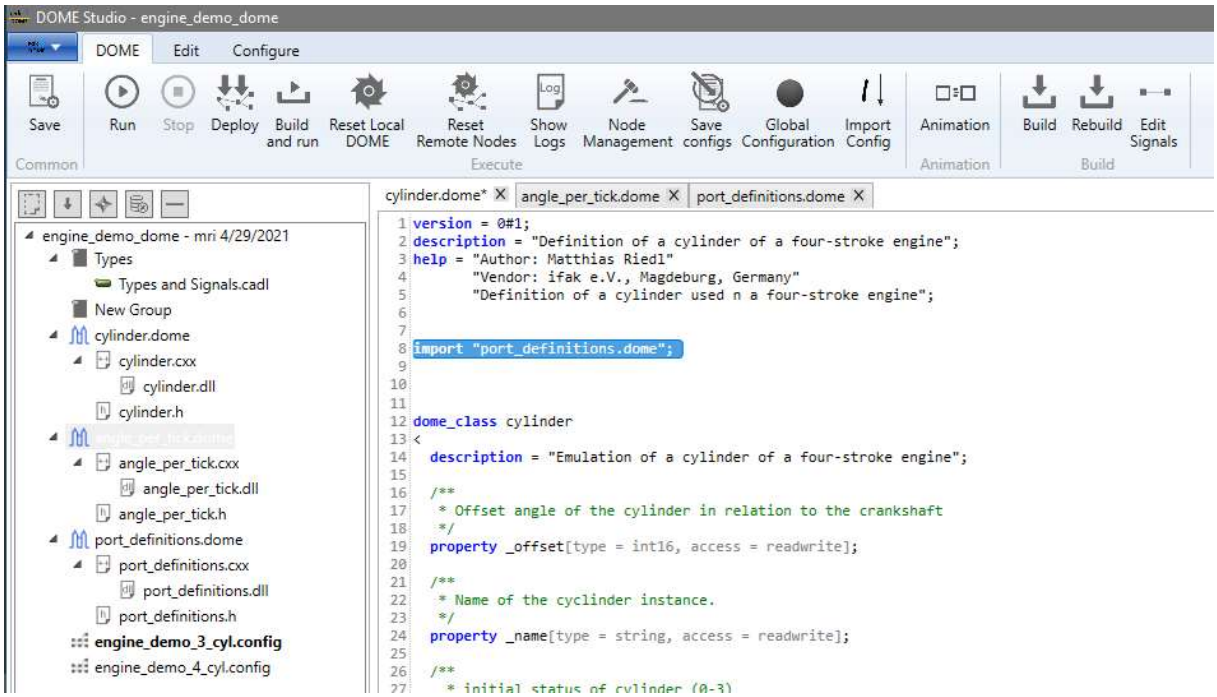

Figure 15: Class and port definitions of the example

The distributed application can be designed graphically. For this, each project provides at least one *.config file. By default, the name of this file is derived from the project name. We will create a new configuration file for using on cylinder in the engine. Via the main menu, section New File, select `DOME Configuration` and name the file `engine_demo_1_cyl.config`. The new file is activated automatically and an empty drawing area is shown. In order to instantiate objects in the application, use the tab `Library` in the project window. Now, the available classes can be seen. Besides the classes defined for the example, there are predefined classes `Delay` and `Timer`, which nearly each project needs.

Inside the library one selects the class to instantiate and drags this on the drawing area. After this, the object settings should be adapted: name the object correctly, assign the object to a process and to an execution context and finally set the properties. The name of the process can be adapted by a sub dialog, we will call it `engine`. The process shall run on the same machine (localhost). The execution context will be named `grp_angle`.

As shown above, the call provides one property `delta_angle`, the values will be adapted e.g. to 10, see Figure 16

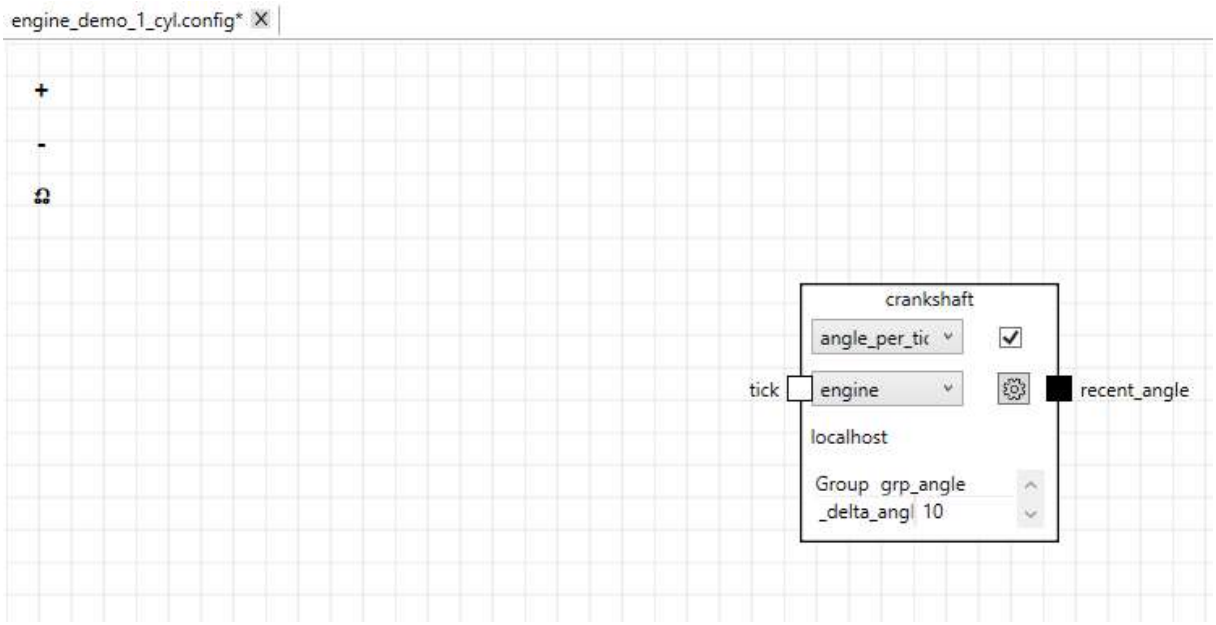

Figure 16: Crankshaft emulation by angle calculation

The next step is to introduce a timer object in order to get time ticks. The ticks shall be forwarded to the object named `crankshaft`. The instantiation of the time is analogous to the action described before. The instance will be renamed to `timer`, the execution context is set to `grp_active` and the property `cycle` is set to 200. This means, every 200ms the timer creates an event. To do this automatically, the property `autorun` has to be set to true. As next the connection between the `TimeEvent` port and the `tick` of the `crankshaft` we be established. Drawing a connection b pressing the left mouse button from requirement port of service port. That's it, see Figure 17.

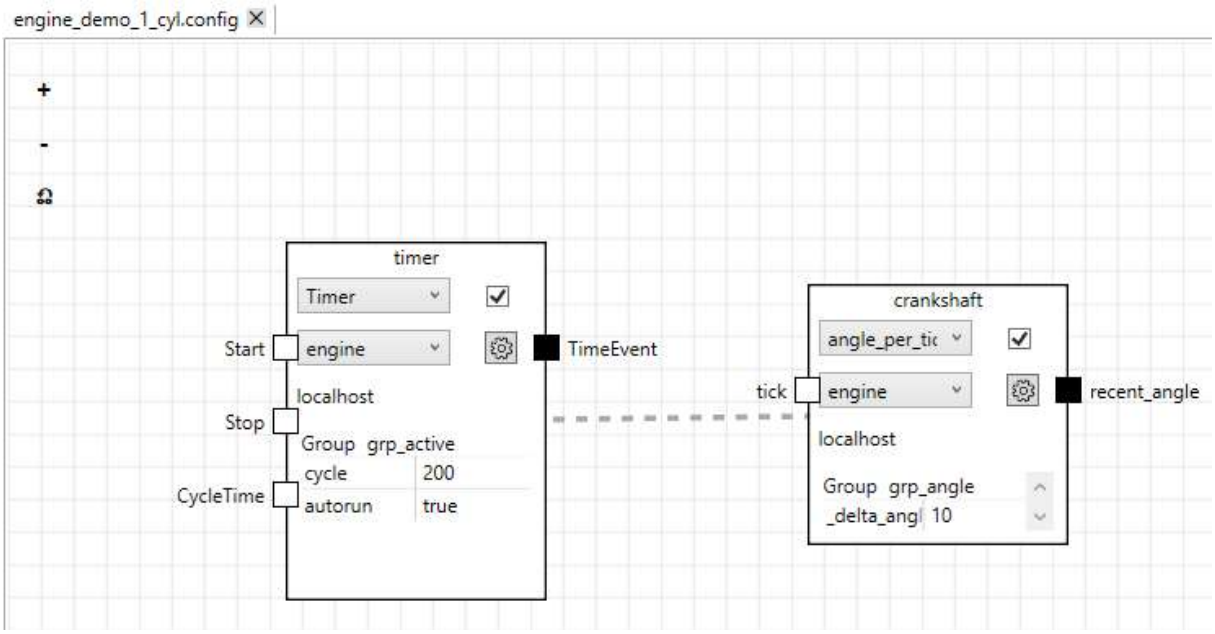

Figure 17: Connection between timer and crankshaft

The dashed line represents connections between the objects. If one of the related objects is selected, that the connected ports are visible, see Figure 18.

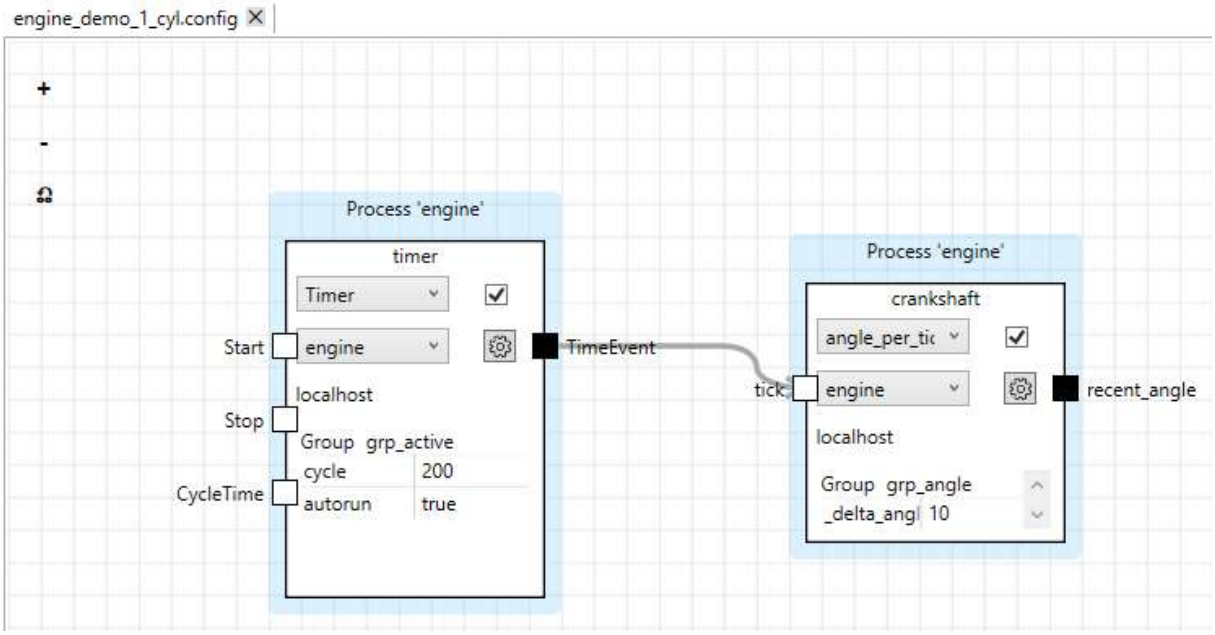

Figure 18: Port Connections in detail

Figure 19 shows the final program. Therefore the last object is instantiated. It is called `cyl1` and represents one cylinder of the engine. The properties are set with `_offset` to 0, `_name` of the instance to see in logging messages is set to "Cyl1" and finally the `_intit_status` is set to 0, meaning the phase "SUCKING". In this case, there is no offset between the crankshaft and the cylinder.

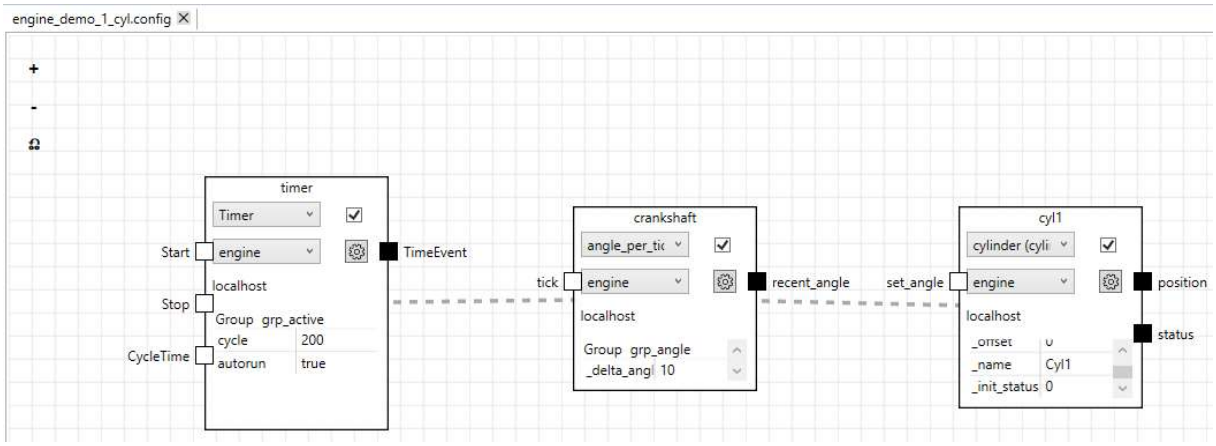

Figure 19: Complete example

After this is done, the application can be built via DOME/Build. The result is shown in the bottom, see Figure 20.

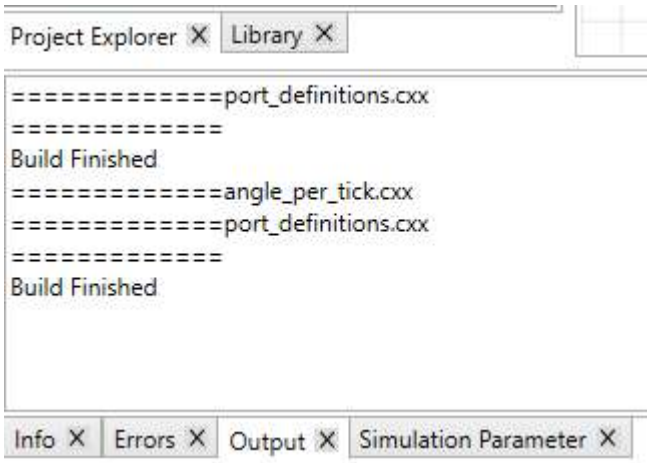

Figure 20: Build results

When the build process is finished successfully, the program can be started. The engineering tool transfers the needed libraries and the information about object instances and port connections to the involved nodes of the distributed application. The recent example creates an instance of the runtime and loads the program to it. It can be observed by viewing the command line window, see Figure 21.

The functionality of the program can be seen by means of the debug stream. The engineering tool connects to this stream and prints the results in the output window, see Figure 22.

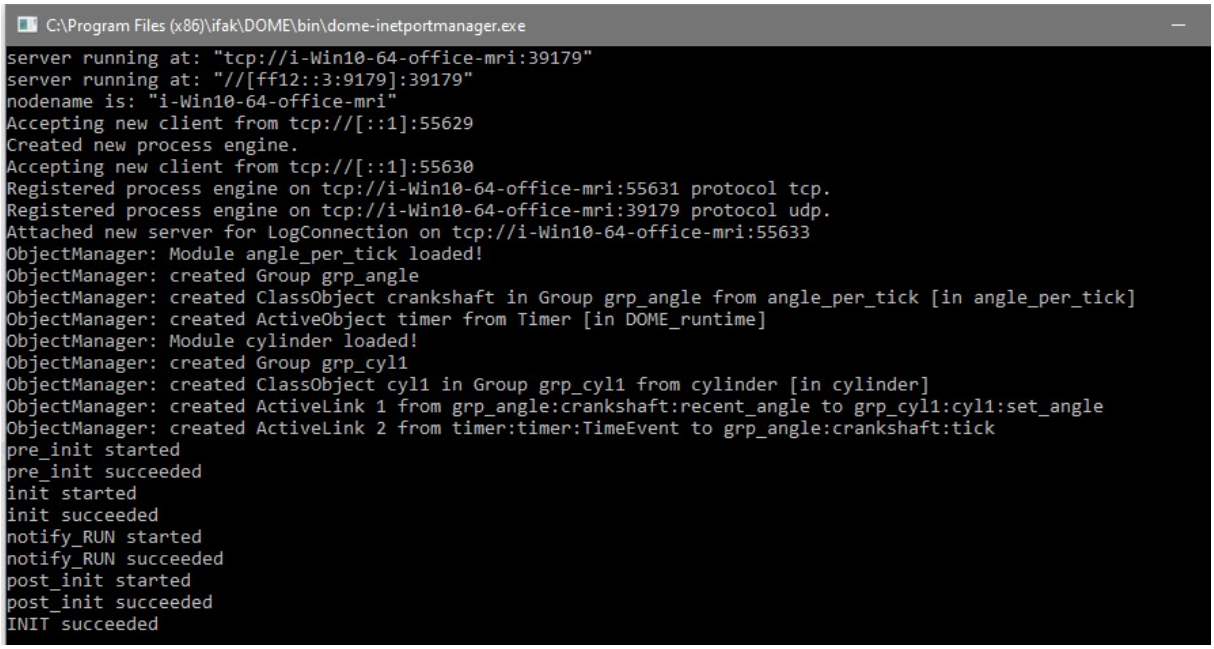

Figure 21: Successful initialization of the application

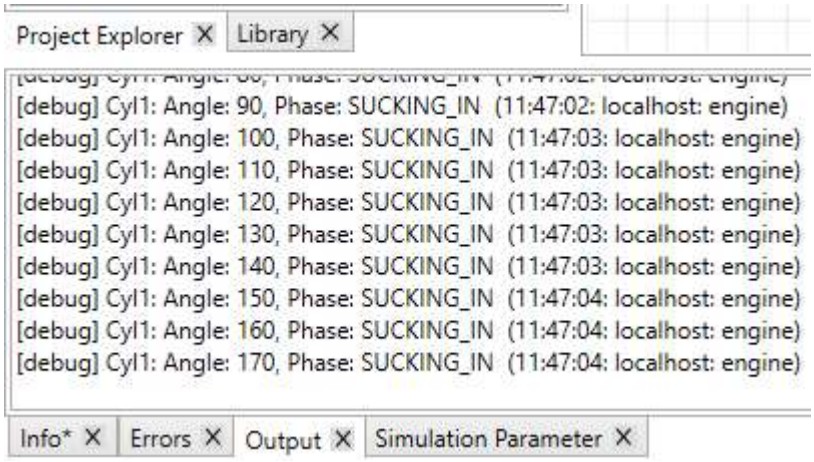

Figure 22: Showing the debug stream in engineering tool

7.7 Simple Visualization

For application developers such kind of presenting the program results may be acceptable, but the engineering tool offers also some simple kinds of visualization. The visualization allow to draw rectangles or circles. The movement of the piston in cylinder can be represented by means of the height of a rectangle, the “cold” phases SUCKING and COMPENSING by the color blue and the “hot” phases WORKING and EJECTION by the color orange, see Figure 23.

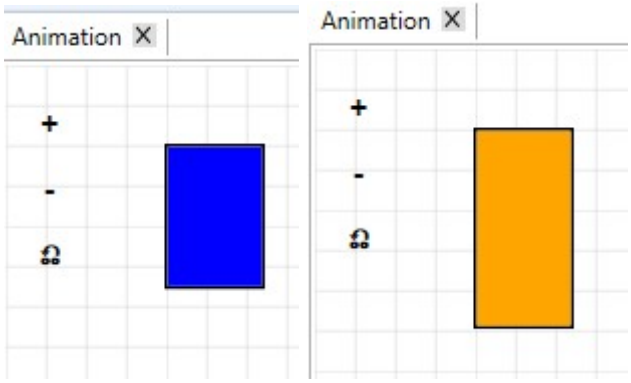

Figure 23: Representation of the working of the cylinder

The feature can be used via DOME/Animation. It opens a drawing area, where the graphical objects may be positioned. In order to animate such objects, some attributes can be set from the program. Therefore, objects should provide requirement ports and push recent values via these ports to the animation. In recent case, the class `cylinder` provides two ports `position` and `status`. They provide float or integer values which can be used in the animation by assigned the attribute of the graphical object with the port, see Figure 24.

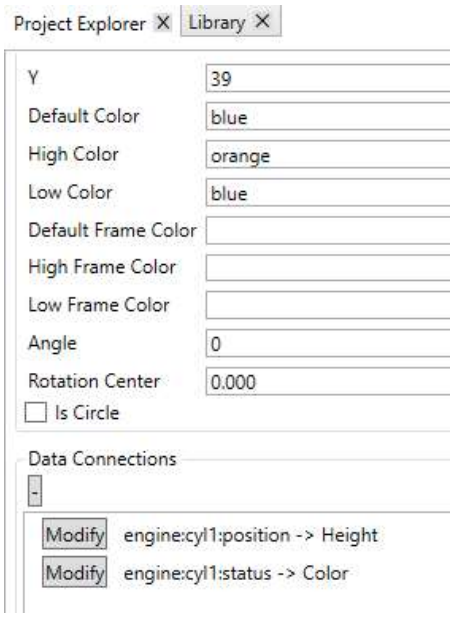

Figure 24: Assignment of ports the graphical attributes

7.8 How to distribute?

The example is running in one process of the Windows machine. Internally it uses three execution contexts. This means the objects run in concurrently. During instantiating the objects, the process where the objects will run, can be defined. At this place, additional processes can be added. Additional nodes can be added via DOME/Node Management. Figure 25 shows the options. It is possible to add nodes (name, IP-address or DNS-name) and to define the tool chain in order to create target code for these nodes.

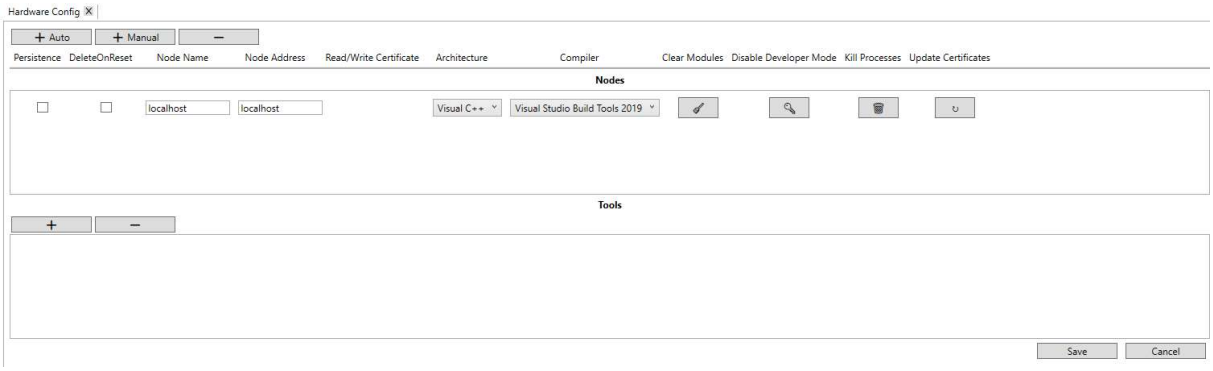

Figure 25: Definition of nodes and tool chains

8 Summary

The document presents a conceptual overview of the implementation of the Distributed Control Platform and its prepared packages for the deployment. In addition, it explains for the developers of distributed application the essentials of object synchronization and the language concepts to develop such objects. Finally, there is a small example demonstrating the process to develop a distributed control application using a simple (not distributed) example.

9 Abbreviations

DCP	Distributed Control Platform
DOME	Distributed Object Model Environment
DOME-C	DOME Configuration
DOME-L	DOME Language

10 References

[1] Riedl, M.: Distributed Object Model Environment: Ein objektorientiertes Softwaremodell für verteilte Automatisierungssysteme. Otto von Guericke University Magdeburg, ISBN 3-8322-4644-4, 2005

[2] OPTIMUM: Deliverable 4.6 - Final Implementation of control application development tools and App Store (2nd prototypes), OPTIMUM project, 2021-02-28

[3] Lua.org: The programming language Lua, https://www.lua.org

[4] Actor model: https://en.wikipedia.org/wiki/Actor_model

[5] ISO/IEC: ISO/IEC 9899:2018- C programming language, www.iso.org, 2018