AGH University of Science and Technology

FIELD OF SCIENCE: Engineering and technology
SCIENTIFIC DISCIPLINE: Information and communication technology

DOCTORAL THESIS

Automating software development for distributed control systems with the OPC UA standard

AUTHOR: Piotr P. Nikiel
SUPERVISOR: dr hab. Krzysztof Korcyl (IFJ PAN)
COMPLETED IN: 1. CERN, Geneva, Switzerland
2. Institute of Computer Science,
Faculty of Computer Science, Electronics and Telecommunications,
AGH University of Science and Technology, Kraków, Poland

Kraków, 2022

To my parents – Barbara and Paweł – who sometimes enjoyed my childhood interest in computers.

To the memory of Jolanta Olszowska – without her my life's path (including this PhD) would have been so very different.

Contents

Chapter 1

Introduction

Rapid evolution in Information Technology and Electronics nowadays allows to construct control systems with unparalleled complexity in many dimensions. One of them is a huge number of parameters (e.g. process variables) available for control and monitoring of the order of millions or more. Design and implementation of such a complex system might require a decade if the system is very heterogeneous and made of many custom parts. During its operation, which might span over few decades, the system has to function faultlessly. In addition it should be maintainable and able to accept extensions, modifications and upgrades of both software and hardware.

One of the main problems in construction of such a system is in providing software development consistency. Every new (software or hardware) component added to the system at the beginning or ten years later has to cooperate with the system without jeopardizing its operation.

Components of such a system are interconnected. There are many different ways of interconnecting components of control systems. A relatively new addition to the family is the OPC Unified Architecture technology (abbreviated OPC UA) [1], also known as the international standard IEC-62541[2]. The technology offers modern solutions for machine-to-machine information exchange and unifies many recent technical developments attractive for applications in big and heterogeneous control systems. By its application, the interconnected components are guaranteed to maintain long-term interoperability, ease of migration and upgrades, security and network friendliness and portability to different computing platforms. The standard defines interfacing between components but leaves related details of software design and implementation open. The implementation aspects need to be proposed and standardized to fully profit from software development consistency when constructing systems based on the OPC UA standard.

This thesis proposes various techniques in the domain of software development automation to provide required consistency in such a complex system. The scale of the system presents a number of design, development, operation and maintenance challenges. The large complexity of the system refers here to the large number of heterogeneous components integrated into the control system together with large number of operational parameters (process variables)[1].

Such a system is designed for long-term operation (predicted to 20-30 years) due to the high investments (cost of the order of 1 billion euros) that need to be made. During operation, down-time must be minimized because of operational expenses. Therefore any improvement minimizing the down-time is advantageous if it applies to fault-related down-time as well as maintenance-related down-time or both.

A heterogeneous system of large size integrating various hardware and software technologies, applications and products requires relatively big group of contributors (order of hundreds or thousands) taking part in the system design as well as software development and maintenance. Due to complex nature of the system not every contributor is a seasoned control system software engineer, and many contributors might not even be computing professionals.

[1] In further chapters a representative example of a system of this kind is given where the number of heterogeneous components is of the order of thousands and the number of process variables is of the order of millions.

The briefly highlighted features of the large, complex and heterogeneous system lead the author to state the following **thesis**:

> Automating software development coherently with information modeling features of the OPC UA standard provides consistency and economy in multi-node, heterogeneous, complex control systems with a long life-time.

1.1 Contribution to the research domain

The thesis is an effect of about 5 years of author's work at Central Detector Control System group of ATLAS experiment at CERN in Geneva, Switzerland. A subset of author's work was published in a peer-reviewed journal [3]. Author's research work, including identifying software creation issues for methods based on plain OPC UA, and its initial solutions proposed by the author of this thesis, was recognized in an independently published IEEE paper[4]. The research work, supplemented by the proof of concept and test applications, accompanied by extensions from other contributors[2], resulted in a software framework called *quasar* (abbreviation from **Q**uick OPC-**UA** **s**erver gener**a**tion f**r**amework). The framework is in wide use at CERN[3] starting from 2015 for creation of software for LHC experiments control. Examples are outlined in chapter 5.

The framework was open-sourced[5] in 2015 and significantly extended since then, under the author's leadership. It was presented in international conferences [6, 7]. The *quasar* framework was recognized by CERN as one of "Industry 4.0" technologies which CERN could offer to world-wide industry and enterprise under Technology Transfer Framework [8]. Basing on this agreement, European companies[4] started using *quasar* to offer OPC UA servers for equipment they manufacture [12].

From the initial objective of efficient development of OPC UA servers (the first of such servers entered production environment in 2015), *quasar* evolved into a software ecosystem which can be used to build advanced software-based OPC UA control systems made of multiple components. The author led the following activities in the ecosystem evolution:

- definition of software architecture, including architecture of OPC UA servers, OPC UA clients, SCADA integration layer etc. (see section 4.9),
- definition of notation of *quasar design* and its interchange format (see sections 4.5, 4.6, 4.6.1),
- implementing transformation and code generation engines (see section 4.8),
- server-client co-generation facilities and consequently chaining facilities (see sections 4.9.2 and 5.5),
- implementation of a build system with support for portability to native builds as well as cross-compilers (including the Yocto platform[13]),
- interfaces to SQL and No-SQL data storage for data archiving and historic data retrieval,
- design and implementation of integration with different programming languages which permitted embedding of OPC UA software components in software packages written in different programming languages, for instance in Python (see section 5.5.3),
- design and implementation of integration with Siemens WinCC OA industry-grade SCADA system (see section 4.9.3),
- design and implementation of various related software components like *open62541-compat*[14] and other.

[2] The exact description of contributions received from other collaborators is detailed in appendixB.
[3] CERN, the European Organization for Nuclear Research; the author's employer.
[4] At the time of writing, three companies were known to use *quasar*: ISEG[9], CAEN[10], W-IE-NE-R Power Electronics[11].

All aforementioned activities are detailed in chapters 4 and 5.

This thesis, unless explicitly noted, shows only the author's conceptual and implementation work, most of which became foundation ingredients of the *quasar* framework. Relevant contributions of other contributors are detailed in appendix B.

1.2 Prior work and acknowledgments

The author would like to acknowledge projects, individual persons and teams who inspired work described in this thesis.

The Front-End Software Architecture (FESA)[15] is "a comprehensive environment for equipment specialists to design, develop, test and deploy real-time control software written in C++ for front-end computers" [16]. FESA profits from model-driven engineering to provide different technical and economical advantages which let non-software-experts write control software for CERN control systems. The FESA project inspired the author to realize the extent of possible economical and organizational gain when such a systematic approach is used.

The author acknowledges the relevance of a project realized by his colleague, Viatcheslav Filimonov, called *CANopen OPC UA Server*[17]. Many problems observed in the software development progress of the project inspired the author to find better solutions and thus significantly improve future OPC UA server projects.

CORBA[18] was an important point of reference for the author and its certain characteristics (e.g. *location transparency*) were inspiring for certain concepts which were used in the *quasar ecosystem*, for example *UaObjects for C++* (see section 4.9.2.1) and consequently *chained servers* (see section 5.5).

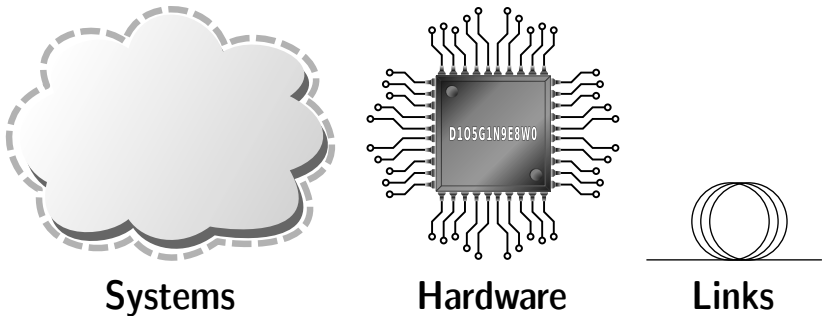
1.2.1 Typographical conventions

In the text, the following typographical conventions are chosen:

- proper names are written in italics, for example: *OPC Foundation*,
- names of concepts are written in italics, for example: *quasar design diagram*,
- source code listings are written in mono-space font, for example: `int someFunction(int x)`

1.3 Origin and acknowledgments of figures

All figures (except *quasar design diagrams*) used in this thesis were made with InkScape (`https://inkscape.org/`), the free and open-source vector graphics editor. In many figures, multiple commonly available graphical elements with permissive licenses were used, mostly from the Open Clipart repository. The author would like to list them in this section (including their typical meaning) and thank all the authors for making their work available to the general public.



1.4 Author's personal acknowledgments

I would like to thank Professor Krzysztof Korcyl for supervision of writing of this thesis. The special appreciation is expressed for help in planning, structuring and organizing the thesis which, taking into account its rather wide scope, was a daunting task for the author. I also appreciate a lot the motivation towards finishing it which taking into account different life circumstances was a rather fragile act.

I would like to thank Professor Janusz Chwastowski for his encouragement and hundreds of corrections that increased the scientific level of this thesis. It would probably never be complete without Professor Chwastowski encouragement.

I would like to thank my colleagues: Paul Thompson and Benjamin Farnham who, as native English speakers, helped me enormously with the quality of English used in the thesis.

I would like to thank Professor Piotr Malecki, Mrs Jolanta Olszowska and Dr Elżbieta Banaś for introducing me to the world of control and supervision software of the ATLAS detector and being my first guide there. Back then I'd never think that life would let me complete a PhD based on such complex software projects.

I would like to thank my supervisor, Dr Stefan Schlenker, for the trust and opportunity of introducing software techniques proposed in this thesis into 24/7 controls software of the ATLAS detector. I also acknowledge my colleagues, especially Viatcheslav Filimonov, and Paris Moschovakos, Ben Farnham and Michael Ludwig for many interesting discussions on OPC UA technologies and their possible application in the software for detector controls.

Last but not least, I would thank my parents – Barbara and Paweł – for getting me a PC back in the 1990s hoping it made sense.

Chapter 2

State of the art review

2.1 The technologies of information exchange

The aspects, problems, standards and architectures of information exchange between computer systems are well represented in scientific and technical literature. Generic concepts of information exchange over computer networks are well laid out for example in [19, 20, 21].

The application of such communication from software engineering point of view calls for software-centric architectures and approaches, especially when it is in relation to very heterogenuous software-driven systems. There the communication – either for information exchange (data flow) or for affecting program flow or both – is often organized as a separate layer in software systems engineering called middleware. According to [19], where middleware is a synonym for distributed system layer, it is said:

> The distributed system provides the means for components of a single distributed application to communicate with each other, but also to let different applications communicate. At the same time, it hides, as best and reasonable as possible, the differences in hardware and operating systems from each applications.

Overview of concepts and architectures employed in middleware technologies can be found e.g. in [21].

Middleware technologies are of varied complexity. They range from relatively simple technologies, for example covering only information serialization aspects, like ASN.1 [22] or passing data as XML[23] or JSON[24] over a stream (e.g. TCP/IP) or a datagram socket (e.g. using ZeroMQ[25]). Such relatively simple technologies are often insufficient to efficiently interconnect complex and heterogenuous computer systems, because they are not standardized enough to guarantee interoperability or long term availability of service. Also they might skip other relevant aspects such as security (for instance, for role-based access control to information exchange) or support for some specific communication patterns or long-term handling of backwards compatibility of interface changes.

In the realm of more advanced (but still generic) solutions one finds for instance different object request broker (ORB) technologies. A prime example of such architectures is for instance Common Object Request Broker Architecture (CORBA) [18] or Microsoft's Distributed Component Object Model (DCOM) [26], both of which are cited here for historic importance. These technologies, even though well established and standardized, turned out to be troublesome regarding their usage in modern global networking environments, involving complex routing and firewalls. With the growth of the Internet, the convergence to protocols able to easily pass such complex networking environment (for instance, anything that could be serialized over the HTTP[27] protocol in a simple server-client architecture) is clearly observed. That provided growth to Simple Object Access Protocol (SOAP) and related protocols.

Network friendliness was not the only aspect driving for change though. The microservice architecture called for organizing information exchange in distributed applications by decomposing

them into loosely coupled and well constrained individual components. In that decomposition the crucial dividing line was laid according to high level capabilities of individual software components, rather than on particular implementation technologies or interconnects. That way of looking at software distribution was certainly a strong inspiration to the family of OPC UA technologies, central to this thesis and introduced later in this chapter.

From a perspective, looking at a large number of existing architectures and technologies of information exchange that were invented and were being developed since the dawn of the digital computer era, one sees universal patterns and trends: preference of modularization and well defined coupling, avoidance of technological overcomplication, deprecation of closed-source and/or vendor-locked technologies. Also, scalability, extensibility and design for long-term use stick out as features that help specific architectures and technologies secure their safe place in the evolution of information exchange technologies.

2.2 Software modeling for control systems

Software modeling is a mature domain in software engineering spanning:

- modeling languages (for instance the UML[28] and the SysML[29]),
- psychological research on optimizing software modeling for humans (see e.g. [30]),
- processes and methodologies (for example, based on *Rational Software Architect*, a common environment for modeling-based software development),
- technical means to transform models info factual computer programs from the models (see e.g.[31] for a comprehensive reference).

Software modeling can augment the engineering of software for any business, thus it also covers control systems software. While the languages and processes listed above are applicable, there exist methods, projects and tools specific for the control systems software creation.

A project of particular interest to the author is the Front End Software Architecture (FESA) project [15] which originated at CERN (where it is still in wide use, as of the writing). FESA was also used in other big control systems in European research facilities [32].

Many goals and constraints of FESA are similar to these researched in this thesis (see chapter 1). Thus, FESA as such is a very attractive prior art to refer to. Some most important points of reference are:

- both projects are tailored to fit relatively big, industrial software-based control system with high reliability requitements,
- both projects should serve the timespan of decades, both for operational use (i.e. control systems realized using them) as well as for development (being able to extend or modify the system),
- both projects are expected to support personnel who might not necessarily be experts in respective middleware technologies or control system architectures. In both projects the economic outcome of control system software creation is very relevant.

Among the notable differences between FESA and the research of this thesis one servers:

- entirely different middleware technologies (custom protocols for FESA vs industry standard for this thesis),
- significantly different software architectures.

The domain of software modeling brings in an alternative way of looking at software engineering problems: less oriented at implementation and closer to the organic way of thinking which humans represent. The optimal spot is achieved when a modeling language can be both human-friendly (so the organic way of thinking in which humans excel can be applied) and machine-friendly (so technology can help to transform models into actual computer programs).

2.3 Optimizing software making for OPCUA

Authors of particular software frameworks providing OPC UA connectivity seem to recognize the gap between pure OPC UA information models and the practical aspects of OPC UA software components implementation.

In the OPC UA domain, *node sets* (see e.g. [33]) is the standardized format of storage and interchange of OPC UA information models. Many notable OPC UA connectivity providers support *node sets*, for instance open62541[34] and Unified Automation UA-SDK[35]. Such toolkits provide so-called *node set compilers* which enable some automation (support for software developer) to generate part of source code from the nodesets, for instance to map custom data structures defined in OPC UA to equivalent types or classes in the chosen implementation programming language.

The UaModeler tool [36] (a commercial product) goes some steps beyond *node set compilers*: it generates source code for far bigger scope than just custom structures and data mapping. However, since its only input is OPC UA information model, many practically relevant aspects get missing and software developer could clearly benefit from much higher level of automation if additional sources of information were used to supplement such code generation.

The problem at hand – i.e. the gap between pure OPC UA information models and respective software implementation – is also brought up by many other researchers, for instance Strutzenberger et al [4] who cite one of papers of the author of this thesis. In the paper the problem is clearly expressed:

Various Software Development Kits exist which provide an extensive base for implementation, but nevertheless, for widespread use of OPC UA, the development effort for servers has to be reduced.

quasar– the focus of this thesis – addresses the problem by providing modelling capabilities which are (at the same time) close to the OPC UA information models and rich in different implementation hints that enable to significantly reduce development efforts.

Chapter 3

Distributed control system environment

The efficient creation of a large, software-based control system requires, at minimum, the fusion of the following technical domains: control systems theory and engineering, software engineering and distributed systems engineering.

3.1 Control systems theory and engineering

A control system is a *system of interconnected components to achieve a desired purpose* [37].

The common building blocks of control systems are:

- sensors, which determine state of object(s) under control and deliver information about it,
- controllers, which process the information and take decisions about actions to take,
- actuators, which transform decisions into physical actions on object(s) under control.

3.1.1 Sensors and actuators

Sensors are inputs to the control systems. Actuators are the outputs of the control system.

Physical quantities that both sensors and actuators relate to are copious: voltage, current, frequency, temperature, pressure, humidity, object position, velocity and acceleration, mechanical strain, color, light intensity, sound amplitude, presence of a chemical compound and many other. Multiple quantities are often required to determine state of given object. Often, multiple samples of quantities are needed; sometimes they undergo some aggregation procedure (e.g. average, median) before they are input to a controller; in other cases they are input as-is, i.e. as a sequence of samples. Therefore representing inputs and outputs of a control system (which is equivalent to describing their sensor and actuator information models) is not a trivial task.

Furthermore, additional complexity is introduced when interfacing of sensors and actuators is considered. The information which is passed between them and the control system might be analogue or discrete, discrete-time (periodic or irregular) or continuous-time, might be always tracking the physical quantity or perhaps it is triggered by an event or on controller request. All of those aspects must be taken into account by the control system design and will significantly impact the software creation.

3.1.2 Controllers

Controllers are responsible for taking decisions.

For recent decades of study on control systems (control theory domain), many different solutions came into existence and practical use: from relatively simple PID (proportional-integral-derivative) controllers processing one sensor and controlling one actuator, via fuzzy logic or neural networks up to systems made of multiple controllers of different types mutually exchanging information. While not much attention is paid to control algorithms in the thesis, the information exchange between controllers and controllers and sensors/actuators is given priority.

3.1.3 Software in the control systems

Modern control systems are often implemented using a digital computer [37].

In such systems, software plays the central role, because the controller is usually a software implementation of a chosen control algorithm.

The complexity of software scales with the complexity of the control system. Usually the scale is beyond-proportional, i.e. software gets complex faster than the control system overall, because software is charged with so many other responsibilities, like user interfacing, data validation, logging and tracing, archiving of information etc. Also, replacing software is usually simpler than replacing the control system overall (which would also include replacement of control system hardware). That makes creation and maintenance of software more hectic and more dynamic, and often error-prone.

Therefore there were many attempts to standardize design, implementation and maintenance of software-based control systems. All of that happened in the hope that a standardized solution would provide advantage in terms of lower cost, lower maintenance effort and higher robustness. Certain approaches - and often corresponding commercial software platforms - became common. For instance, MATLAB with its Simulink[38] extension became a widely known tool for design and simulation of controllers, and products exist which can turn those designs directly into control algorithm source code[39]. Software packages known as SCADA (Supervisory, Control and Data Acquisition) came into existence to ease running software-based control systems on different range of computers. Siemens WinCC Open Architecture (WinCC OA) [40] as well as National Instruments LabVIEW[41, 42] are good examples of such systems. However for control systems of characteristics laid out in 1, such software packages are not enough alone and further development must be done.

This thesis proposes an approach to build control systems software which can significantly extend aforementioned software packages. In addition, the approach can be used to simplify construction of stand-alone control systems software without usage of aforementioned commercial software packages.

3.2 Distributed systems

Typically, a control system characterized by conditions as in 1 falls beyond the possibility of being implemented as a simple computer program. Possible reasons include:

- size and complexity
  A control system running thousands or millions of controllers, each having multiple inputs (from sensors) and multiple outputs (to actuators) might require distribution into multiple computers because of needed processing power, memory and storage capacity and bandwidth and input/output capacities.

- redundancy and avoidance of a single point of failure
  A control system might require redundancy by duplication (or, often, triplication) of its core components, such that each of multiple computers is able to deliver the whole functionality running alone.

- physical distribution
  A control system might be built on a vast physical areas such that it is more practical to design it on a number of (interconnected) computers than on a single one.

- dynamic partitioning
  A control system might require than some of it parts might need to be put off-line for some periods of time (for example, for a planned intervention), while the remaining should stay on-line. In such a case distribution by such parts is an obvious strategy.

In all of those cases the distribution into multiple computers takes place.

Tanenbaum gives the following definition of a distributed system: "A distributed system is a collection of independent computers that appears to its users as a single coherent system" [19]. It is often practical to extend the definition by also including multiple independent and interconnected computer programs on the same computer, appearing as a single coherent system.

Distributed control systems are therefore software-based control systems implemented on independent computers and/or independent computer programs appearing as a coherent system.

3.3 Software engineering

The author uses the following definitions of software engineering, both from [43]:

1. systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software,
2. application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Both definitions were crucial for the author to serve as a point of reference on *how* creation of software in the *quasar ecosystem* should be approached.

The following key assumptions leading to *quasar* were made by the author:

- that topical knowledge, experience (both personal as well as gathered by querying experts) and prior art should be taken into account for design of the software automation,
- that formerly established methods, such as those known from Computer Aided Software Engineering (CASE), and related tools, should be re-evaluated for inspiration and possible re-use,
- that the software automation process should cover all aspects, from gathering of requirements, via design up to testing and deployment,
- that the process should be measurable to converge the software automation method to an optimal one.

3.4 The OPC UA standard

The OPC UA standard (OPC Unified Architecture) is a machine-to-machine information exchange technology proposed by the OPC Foundation [1]. It was standardized as the International Standard IEC 62541 [2].

The OPC UA standard has features particularly attractive to applications in the control systems domain:

- it is a complete standard for machine-to-machine information exchange, spanning from aspects such as data encoding on the wire through information modeling up to security and authentication of sessions.
- its model of attributes is extensible and generic.
- it supports any representation of information to be exchanged, as long as it can be modeled as an (possibly recursive) aggregation of primitive data types. Nowadays control systems process very varied information, from simple measurements (e.g. temperatures) up to snapshots of entire subsystems (which might be organized as graphs, trees, etc.).

- its information model (the description of information processed by given application) is based on graphs; graphs are more generic than many common data structures (e.g. any tree or any list can be represented as a graph). The information model is therefore very generic compared to many competing protocols.
- it features multiple communication patterns such as request-reply, polling and subscription-publication. Thus variety of communication patterns can be supported according to control system needs.
- its transport layer defines two types of data encoding: binary (which might be preferential for performance and low footprint applications) and XML-based. In addition one can define their own encoding.
- it is firewall-friendly by design.
- it has a built-in support for secure data exchange.

The communication in OPC UA follows the client-server model. The server opens endpoints and waits for client(s) to connect. Clients can send requests to the server. Among well defined requests there are invocations of services responsible for reading (information available on the server [1] is passed to the client), writing (information is passed from the client to the server), calling methods, accessing the history, etc.

OPC UA also permits to use other communication patterns, e.g. single publisher - multiple receiver model which does not enforce client-server architecture but still enjoys much of OPC UA features. These more complex information exchange arrangements are not related with topics discussed in the thesis so they are not further detailed (nevertheless they are compatible with ideas presented in the thesis).

The OPC UA standard focuses on how to exchange information between nodes of a distributed control system but it does not fully cover software engineering aspects of how such information exchanging software should be built. Nevertheless certain aspects were standardized to make the OPC UA software development easier:

- an interchange format (based on XML) called *OPC UA node-sets*[33] was created to represent the snapshot of OPC UA address-space as an XML document. The advantage it brings is that specialized tools (e.g. UaModeler[36]) can be used to model the address-space and store it as a file which can be then loaded by a chosen OPC UA Software Development Kit.
- also, some Software Development Kits (SDKs) propose a model of integration of custom code to serve as service calls handlers.

Despite of those easements, a lot of freedom is still left to software designers and developers.

The work done in this thesis fills this gap and proposes automated and efficient approach to OPC UA-based software creation, especially for large-scale control systems. It unifies different domains which typically need to be addressed together for efficient OPC UA software creation:

- OPC UA address-space management (including its creation),
- integration of custom code,
- managing configuration (e.g. how to describe instances of particular objects to be represented by OPC UA servers),
- co-generation of servers and clients,
- deployment and integration into distributed control systems.

All of the aforementioned aspects are part of the *quasar architecture* defined primarily by the author and detailed in this thesis.

[1] Precisely, the information does not need to be located on the server but the server knows how to obtain it.

Chapter 4

Implementation of the Quasar framework

4.1 Overview

A schematic of a simplified control system is shown in figure 4.1.

Despite simplification, the schematic demonstrates numerous challenges faced by the development of software for such a control system.

First of all, the system can be seen as composed of parts that are entirely hardware and those which are implemented in software. Sensors and actuators are the only elements which belong to both domains, i.e. they combine physical aspects (they measure physical quantities and clearly need some hardware implementation) but at the same time they appear as information inputs/outputs to/from the software-based system.

Sensors and actuators are often considered inputs and outputs. Often sensors also feature certain quantities that can be controlled (so are also outputs) and a symmetric relation holds for actuators (in figure 4.1 depicted as annotations "1" and "2"). A relevant example for a voltage sensor[1] is that it might have an adjustable range and/or rate of conversion; so it consumes information which is produced by the control system. Similarly any actuator might have embedded diagnostics which let the control system read-back (verify, cross-check) if the actuator really impacts the physical object as requested by the control system. The take-away message from this paragraph is that such aspects must be taken into account as one of the design decisions. Rarely such a simplified input-only and output-only model holds.

Another aspect relates to organization of the system. In the figure 4.1 one sees implied hierarchy of controllers, i.e. the master controller (a "parent" controller) versus controllers 1..3 which are implicitly slave controllers. In a design of a large control system such a hierarchy (which can be multi-level, i.e. tree-like) is common with certain exceptions. Sometimes controllers are free-running (i.e. they do not have a parent controller). Often the hierarchy must be configurable at run-time, e.g. a controller sometimes runs under a parent controller and at other times it runs freely. In any case, information exchange between controllers is usually common.

Thus one can see that for a given controller not only sensors are sources of information. Actually, sensors, actuators and other controllers can produce information in the control system. This further complicates the design and implementation process because information exchange can happen between any two elements of the system in any direction.

Often information arriving from sources (from sensors, actuators and other controllers) requires additional processing before it can be used by a controller. Such situation is seen in figure 4.1 by the note 3. The most common reasons for it are:

- the unprocessed information is too redundant or would require too much processing on the

[1] Analog-Digital Converter (abbreviated ADC) is a more common name of a voltage sensor.

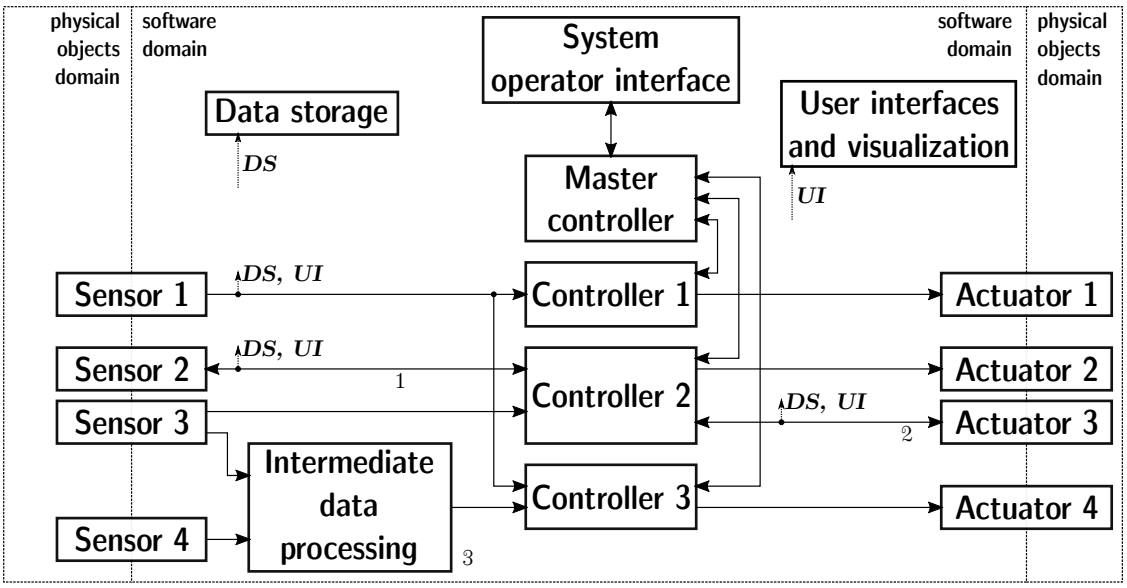


Figure 4.1: A simplified control system schematic. Connectors depict information exchange and arrows indicate direction of information flow. DS stands for Data Storage. UI stands for User Interface(s). Refer to text for notes in curly braces.

controller and it must be minimized in the sense of amount of information. This often implies usage of statistics or signal processing algorithms (e.g. averaging, taking a median or filtering-out high-frequency components of the signal which might be attributed to measurement noise).

- the information is at inconvenient, impractical or wrong (usually too low) level of abstraction and therefore some type of conversion is necessary.
- the format in which the information is represented does not fit the input specification of the controller thus it must be adapted or converted.

In those cases, an intermediate data processor can be used, serving one of aforementioned purposes. Such intermediate processors can of course be chained one after another to provide processing in a number of possibly independent steps. Also, such a processor can be used in the controller output information path, i.e. between a controller and an actuator. In this thesis, this concept is called "chaining" no matter on the location (before or after the controller) or cardinality (number of intermediate processors used).

In the figure 4.1 one can see that many information exchange links are "branched out" as "DS" (Data Storage) and "UI" (User Interface). In a big control system there is often a necessity to archive information exchanged by some (or all) links by making copy of transferred data to Data Storage. The applications of archiving are the following:

- monitoring of system evolution over potentially long time,
- input to control algorithms which might require information of past system state,
- storage of system state in case such requirements are imposed by regulatory requirements (law, authorities, certification process, etc.).

In addition to Data Storage, often other information consumers are needed, such as User Interfaces, which in big control systems visualize the state of certain parts of the system to their operators. The

aspect of "branching out" the information flow is identical to that of Data Storage. Design decisions should thus include architecture where information produced by a single source can be delivered to multiple consumers.

Controllers are often the most complex elements of the system. It was assumed that they can be connected with any number of information exchange links to any number of other elements. Controllers can be implemented either as independent computer programs (often called supervisory software) or they can be part of a SCADA software suite. Both cases are discussed separately in section 4.3.

4.2 Requirements analysis

The author performed analysis of requirements and constraints in three different viewpoints:

- of software product(s) which could be created by application of *quasar* - see section 4.2.1,
- of software automation method proposed by *quasar architecture* (i.e. of *quasar* itself) - see section 4.2.2,
- of foreseeable future events - see section 4.2.3.

4.2.1 Requirements applying to software product

The following requirements were identified:

- high robustness
  The software should be resilient to unwanted/unforeseen operational conditions and be designed such that the probability of failure is minimized. An example of such a behavior could be that the software resists malformed input and it reports an error in the format of input data rather than continuing and processing it and delivering unforeseen behavior [2]. Apart from the validation of inputs, some other examples include:
  - ensuring no memory leaks (either by extensive testing or by using mechanisms which delegate responsibility for memory management from a programmer to a compiler or execution environment - e.g. garbage collection),
  - being able to cope with run-time errors such as memory allocation issues or inability to write or read a file to/from a disk,
  - always checking for error flags (e.g. function return values) or promoting a mechanism which does it without programmer's explicit work (e.g. by usage of exceptions [45]),
  - ensuring that all possible execution paths deliver valid logic.
- high operational availability
  High operational availability of software is a result of high robustness but also of minimizing downtime due to predictable and planned maintenance activities, software updates etc. It can usually be expressed by a ratio of time in which the software must be operational, e.g. 99.99% ("four nines") annually. Therefore software characterized by lower maintenance requirements is preferred.
- ensuring that the software might be improved, developed, maintained and upgraded in the long term perspective
  Such a property is of particular importance if the software is expected to run and deliver value for relatively long periods of time between major upgrades (e.g. ten or twenty years). Even

[2] Such behavior is often called "garbage in – garbage out" in the computer science jargon[44].

though it seems exaggerated, one can support this statement by examples of systems still running software written in "legendary" languages (e.g. in COBOL) as of 2018 [3].

Control systems addressed by this thesis might be especially vulnerable because often their value is due to software-hardware relation rather than to software only (and hardware is far more expensive to replace than software, especially if no COTS solution exists). Long-term extensibility of software enables to add new features with reasonable effort long after the software was released.

- preferring proven, standard and validated technical choices
  Standardized and/or well-known technologies, products and software development approaches are more likely to be available for much longer as well as being of higher quality due to wider usage (so the ratio of uncovered software defects is smaller). Thus it helps to facilitate high robustness of software but also the long-term availability.

- clear decomposition into functional blocks; modularization; separation of concerns
  Clear decomposition is crucial for design, development and maintenance reasons. In the design phase, clear decomposition enables to focus on one component at a time rather than on a whole, so it boosts the chances of delivering higher level product. In the development phase, it makes it easier for multiple software developers to work concurrently and it also enables modularization, such that different components could be replaced with other components having the same inter-component API. In a similar fashion, different aspects such as business logic, information exchange, state and error reporting, configuration should be separated as well.

- being scalable to cover deployment on various range of systems
  As of 2018, with billions of embedded devices (including Internet-Of-Things (IoT)) in markets along very powerful server machines, it is important for software to scale for both low-end and high-end computers. It does not imply that the same software deployed on an embedded device could get the same performance as on a server machine but at least it should be possible to deploy it on virtually any target.

- portability
  Portability enables a software product to be compiled and run under/for various operating systems, compilers, API-compatible libraries etc.

- performance
  High performance ensures that most of the potential of the available processing power can be used wisely to deliver end-user processing. High performance is not necessarily the most important goal for a distributed control system, but if it can be obtained without sacrificing other aforementioned goals then it is considered an important advantage.

4.2.2 Requirements applying to software creation method

The following requirements were identified:

- minimizing the effort of a person implementing the software (typically, a software developer). Such effort can be quantified in:
  - the duration of implementing the project, usually expressed in an unit of time (e.g. months, years),
  - the effort of implementing the project, often expressed in man-months,
  - the cost of the implementation phase of the project, usually expressed in an amount of money,

[3] An article[46] written in 2018 states: "Companies involved in keeping COBOL-based systems working say that 95 percent of ATM transactions pass through COBOL programs, 80 percent of in-person transactions rely on them, and over 40 percent of banks still use COBOL as the foundation of their systems."

  - the number of lines of code to be written, or in another software metric capable to express software development efforts and software complexity,
  - the effort estimated by some formal metric known from the software engineering domain[47].

- optimizing time-to-market by shortening the development time.
  In some cases, time-to-market is a crucial property for guaranteeing economic feasibility of a project.

- minimizing probability of introducing software defects.
  Software defects negatively impact the value of the end-user product and make the project significantly more expensive due to post-deployment maintenance costs. In addition, software defects might have detrimental impact on control systems, making them dangerous to use or they might impact their reliability, availability and serviceability (RAS) requirements.

- minimizing the effort of a person designing the software.
  Software design establishes the architecture of a software product in a chosen language (notation), for example in the UML, SysML or a textual description of how the project is decomposed, what are the roles of its components and how do they interact together. Choosing a language(notation) which is clearer and simpler simplifies the software design phase, thus minimizes the effort spent in that phase and costs of software architects work.

- minimizing the effort of preparing software test plans and of executing them.
  A clearer and streamlined software design and development process will help to minimize software testing effort. In addition, significant savings can be gained if testing is automated (e.g. by automated generation of test cases) [48].

- minimizing the amount of required know-how and making the learning curve less steep.
  Such a feature lowers the threshold on skills of software developers capable of implementing the project. Therefore it makes the hiring process easier, it lowers the overall personnel cost and it facilities to match already hired personnel (if any) to join the project. In addition, mild learning curve enables to put less constraints on initial (preparatory) training, and it makes it less intense. Therefore the training costs are also lowered.

4.2.3 Projection of future requirements

The author tried to anticipate possibility of major changes and shifts of external conditions which could have serious impact on the viability of the *quasar ecosystem*. The motivation for such a query was to have supplementary constraints for subsequent design decisions which could better align the architecture for inevitable passage of time.

The following long-term events (e.g. over a time-span of 10 years) were identified:

- change of a chosen programming language, but keeping it in close conceptual relation.
  For instance, a change from C++ to the D programming language[49] was imagined.

- major change of a chosen programming language, involving paradigm shifts.
  For instance, a change from imperative programming into declarative programming was imagined.

- replacement of OPC UA with another middle-ware solution.
  An experiment was imagined in which another type of middle-ware (for example, Google Protocol Buffers[50] over ZeroMQ[25]) is used for information exchange while all other parts of *quasar*-based software components remain intact.

4.3 Products of the quasar ecosystem

All of previously described components of the control system under consideration (see section 4.1) can be considered potential products of software development automation:

- software components to interface sensors and actuators,
- controllers,
- intermediate data processors,
- interfaces to data storage,
- support for user interfaces.

4.3.1 Software components which interface sensors and actuators

These components can be realized either as OPC UA servers or OPC UA clients (nevertheless, in the author's experience, there is significant preference to have them realized as servers). The former choice is often preferred because a server contains its own OPC UA address-space so it can take full responsibility of maintaining coherency between the address-space model and presented data. The latter choice may be preferred in case the software component is required to have minimal possible resource usage (e.g. for deployment on an embedded system) and therefore maintaining full server functionality might not be possible with limited hardware resources.

In both cases it is expected that such a software component knows how to access the data of a sensor or actuator for input (monitoring, data acquisition) or output (controls). Typically such a task is assigned to a dedicated software layer (or a software component or a software library). When such a layer provides a programming interface which is dedicated to a particular hardware type then it is often called Hardware Abstraction Layer or Hardware Access Layer (abbreviated HAL in both cases). In other cases some form of indirect access might be provisioned, for instance via intermediate information exchange solution, like some dedicated networked protocol.

There are many examples of abstraction layers and intermediate information exchange solutions in control systems:

- *Linux Industrial I/O* [51] (abbreviated IIO)
  The Linux IIO subsystem provides support for devices which can be seen as Analog-to-Digital converters (ADCs) or Digital-to-Analog converters (DACs). The documentation[51] mentions the following examples of devices which are covered by the subsystem and consequently its software support: ADCs, accelerometers, gyroscopes, inertial measurement units, capacitance to digital converters, pressure sensors, color, light and proximity sensors, temperature sensors, magnetometers, Direct Digital Synthesis (DDS) units, Phase Locked Loops (PLL) and other. A Hardware Access Library called *libiio* is provided [52].

- Controller Area Networks (known as CAN)
  CAN [53] is a network standard spanning from the physical layer up to the network layer. It is considered a robust bus for message-based information exchange especially popular in the automotive applications and manufacturing industry. Transport-layer protocols exist on top of CAN such as *CANopen*[53]. CAN support is standardized in many operating systems; for example in Linux it is supported via *SocketCAN* subsystem [54] and a companion software library called *libsocketcan* [55].

- Modbus
  Modbus is a common communication protocol often used to interconnect various industrial systems using serial links or the Ethernet. The *libmodbus*[56] is a generic Hardware Abstraction Library capable of interfacing Modbus devices.

- Simple Network Management Protocol (SNMP)
  The Simple Network Management Protocol[57] is a common Internet protocol for monitoring and control of devices. There exist Hardware Access Libraries to interface SNMP-enabled networked devices. A common example of such a library is the *net-snmp*[58], an open-source project which aims at complete support of the SNMP protocol. The *net-snmp* is commonly used in different network management applications and is a library commonly seen especially in UNIX-based systems.

To construct an OPC UA compatible software component interfacing sensors and/or actuators, one must find how to exchange information between the sensor/actuator domain seen in software by its HAL or another API and the domain of OPC UA.

In a generic case, the following might be said about the sensor/actuator HAL/API:

- the way in which information (measurements, controls) is represented might vary significantly. There is no standardization as to how to represent different physical quantities. For instance, certain HAL/APIs might represent measured voltage in volts or in millivolts, while some other as ADC counts and provide scaling factors (gain and offset) to let the software perform conversion to volts on its own. In some cases HAL/APIs might be at a very low abstraction level, for instance they might provide measurements as a binary object requiring intricate knowledge on its internal representation. In other cases the API might be at reasonably high level such that even an inexperienced programmer could quickly understand it.

- the way in which communication with underlying data source/sink is organized.
  Many different communication patterns (information exchange patterns) are known, and among them request-reply and publish-subscribe are the most common. For request-reply communication, the API calls might be blocking (the caller waits until the called party finished) or non-blocking (the caller requests the communication and then the called party notifies by a call-back mechanism). For publish-subscribe, usually a call-back provided by the information consumer is called whenever there is new data available.

- the way in which a fault is communicated.
  A failure in sensor/actuator HAL/API might need to be considered a serious control system fault because it might mean that the control system loses track of controlled object state (fault in from-sensor direction) or it can not act on the controlled object (fault in to-actuator direction). Different HAL/APIs deal differently with this aspect. For blocking request-reply APIs it is common that the API functions either return success/fail values (from function/method calls) or throw exceptions (in languages which support this mechanism). However even for the blocking request-reply communication pattern certain APIs notify faults by asynchronous calls-backs, so a variety of approaches is commonly seen and therefore expected.

- performance aspects.
  For certain high-performance sensors or actuators (for instance based on high-throughput signal processing implemented on fast ADC/DACs) one often must take many more assumptions to profit from the API in full capacity (such as the impact of dynamic memory allocation, etc.).

On the other hand the following can be said about OPC UA aspects of sensor/actuator integration:

- OPC UA is very generic and can support different data types.
  In addition to its extensible information model one can equip OPC UA address-space with meta-information such as information about an unit in which given data is represented.

- OPC UA supports mechanisms analogous to request-reply as well as analogous to publish-subscribe to notify a client about data change.

- OPC UA supports very detailed list of possible failure codes which can be supplemented by application-defined diagnostic information.

Therefore to integrate sensor/actuator API into a OPC UA software component the following aspects must be covered:

- whether to integrate it as an OPC UA server or as an OPC UA client,
- how to match OPC UA information model into the specification of HAL/API,
- how to match communication patterns supported by OPC UA into what HAL/API provides,
- how to deal with software engineering aspects of given OPC UA protocol stack (e.g. to identify the division lines between what the protocol stack offers, what HAL/API offers and what should be the "glue code" between both domains).

Particular description of the solution is further discussed in section 4.4.

4.3.2 Controllers

Automation of OPC UA software creation for controllers is different from the one explained in 4.3.1. The reason for it is the existence of certain software products in market (often referred to as SCADA software) which are very well established and often used to build certain parts of control systems such as controllers. Examples of such products are Siemens WinCC OA [40] and National Instruments LabVIEW [41]. Therefore separate attention is given to controllers written as independent computer programs (and integrated to the control system using OPC UA) as well as controllers realized using one of such common products.

4.3.2.1 Controllers as independent computer programs

The primary task of a controller is to collect inputs from sensors (and possibly other controllers, or operator-given setpoints etc.), process them using some control algorithm (taking decisions) and output actuator controls.

One can therefore imagine a controller running in a mode of supervisory computer program which keeps getting up-to-date information on controlled object state, runs one iteration of control algorithm on it, communicates outputs and repeats the process. The repetition of this process can be scheduled from some periodic event source (e.g. operating system timers) or can be triggered by arrival of new data (e.g. notified by call-back invocation) or can be running free.

Therefore the problem to be solved here is finding a way to connect inputs/outputs to/from the controller, via OPC UA, to other OPC UA-based control system elements.

If the interface to sensors (so the controller inputs) are realized as OPC UA servers, then the task boils down to construction of OPC UA clients which could obtain sensors output via OPC UA. If, on contrary, inputs are realized as OPC UA clients then the task boils down to creation of OPC UA servers. Similar relation holds on the path from controller to actuators.

In any case there is a huge common part between such pairs of servers and clients. Major parts of their information models must be compatible with each other to enable information exchange. This observation is both a constraint (i.e. the system can only work well if there is the match) and also a very profitable opportunity (because a server and a client can be co-generated). Such co-generation, as observed by the author, reuses the same information model but creates multiple different products, therefore the yield is proportionally higher keeping the same cost. Thus the return on investment is higher which is a favorable property of any software creation method.

4.3.2.2 Controllers realized in Siemens WinCC OA

Siemens WinCC OA establishes its own software architecture to build control systems. At the core of the architecture [59] there is a concept of data points (abbreviated DP) which usually serve as process variables (i.e. they reflect current state of the control system). Data points are instances of Data Point Types (abbreviated DPT). Data Point Types are composed of Data Point Type Elements

(DPTE) which are declarations of individual variables of one of predefined data types (integers, strings, etc.).

A control system built using WinCC OA is composed of one or many (interconnected) WinCC OA projects. Each project has a collection of Data Points and it can run a number of so called managers which are independent computer programs performing different activities. The whole of communication between managers (which can, for instance, run different control algorithms) and between managers and the external world (input/output from/to sensors/actuators) is performed by writing and reading Data Point Elements.

Each Data Point Element (DPE) can be assigned a so called Peripheral Address (PA) which indicates that given DPE is intended for input/output rather than as to serve as a regular process variable. When such a Peripheral Address is assigned then writing to or reading from given DPE will be routed to a manager which can perform relevant communication via configured middle-ware (here: OPC UA). A manager charged with implementing input/output to an external entity (e.g. via OPC UA) is called a driver in the WinCC OA architecture.

Therefore by assignment of Peripheral Addresses to Data Point Elements one can establish information exchange between a SCADA-based control system built in the WinCC OA with another system using the OPC UA protocol. The author demonstrates how the act of efficient integration of *quasar*-based software with WinCC OA can be achieved, including peripheral address manipulation, in sections 4.4 and 4.9.3.

4.3.3 Intermediate data processors

Intermediate data processors, in the sense of their place in the OPC UA software components ecosystem, share a lot in common with controllers. The notable differences are that such processors usually do not run control algorithms and that their outputs are usually connected to inputs of controllers. However apart from these differences the same methods of software creation can be applied.

4.3.4 Additional products

In the author's experience, considering the following pieces of software in the overall collection of products further improves software creation efficiency:

- different test tools and test fixtures,
- test/development OPC UA clients,
- automation of documentation creation.

The particular products obtained are detailed in 4.9.1.12.

4.4 Strategy

So far the requirements and the scope of expected products were identified. In general, different methods can be used to implement such deliverables with different outcome.

Aforementioned observations should be summarized:

- there is often a common part in the exposed (servers) and expected (clients) OPC UA information models in the products explained above.
  For instance, when OPC UA information exchange between software component of given sensor and a controller using this sensor is made, both components by definition use common part of the information model. In other words there has to be an agreement between the information producer (sensor) and information consumer (controller) on addresses of variables, intended data types, information representation etc.

Therefore in such a case it does not make sense to construct those two components independently because it would not be efficient. Rather a common part should be identified and re-used.

In other cases the situation is similar: the same information model is shared between e.g. servers and SCADA-based systems.

- the OPC UA information model might not be sufficient for automated software construction. As pointed out in 4.3.1 many questions are left open about integration of HAL/API and OPC UA software. Most of those questions relate to matching of communication patterns and data representation.

- the OPC UA information model might not be the best choice when human aspects of software development are considered.
  As it was stated in section 1, the particular constraint of the problem is a possibly large number of contributors to the software development activities. Few of them are OPC UA software professionals. This indicates that OPC UA information model, which is a very rich and relatively complex language, might not be the preferred language that those contributors should be requested to speak. Lieberman[30] provides a comprehensive study on such human-related aspects.

Due to the reasons above, it is proposed that a notation of higher abstraction is introduced as the basic tool for developments of OPC UA software components for such control systems. The usage of common notation brings additional advantage of possibility to apply Model Driven Engineering techniques [60], and consequently to obtain deliverables (or their parts) by application of various model transformation techniques.

Such a notation should:

- be tailored to the problem domain, which combines aspects of control systems engineering, software engineering and distributed systems engineering,

- be as complex as necessary; no more or less [30],

- reuse elements of existing well-defined or standardized notation, for instance the UML[28] or its close relative, the SysML[29],

- support substantial degree of extensibility for future,

- benefit from the OPC UA information model where possible but without bringing too much of associated complexity and without bringing elements which are specific to OPC UA only.

The specification of the notation should be accompanied by a description of data interchange format:

- to support editing and storage of the notation,

- to support development tools process it, e.g. to apply techniques of Model Driven Engineering.

In addition, the author analyzed the prior art of software engineering for distributed control systems (e.g. the FESA project) where the aspect of matching of control engineering competencies with software engineering competencies was also visible. Finding a "common denominator" (e.g. a notation that both parties could understand) was the only possibility to make the large-scale control system software development possible.

4.5 The notation

The proposed notation is composed of classes, relations between classes, variables of different kinds, methods and configuration entries.

4.5.1 quasar classes

The proposed notation uses the concept of a class (similarly to classes known from the principles of Object Orientation[61]) as its central element [4]. Such a class, further referred to as *quasar class*, corresponds to a type of a physical object to be controlled in the system.

Examples of *quasar classes* could be: a pressure sensor class, a channel, a device, a property, a motor etc. By convention, the name of a *quasar class* is capitalized and uses camel-case notation, for instance `PressureSensor`.

Consequently the notation assumes that instances of *quasar classes* are called objects (or *OPC UA objects*) and they refer to a particular "thing" in the system to be controlled. A preference for usage of classes and objects comes not only from known advantages of thinking in terms of object orientation but also from the observation that in large control systems one often sees same types of control system elements (same type of sensor or actuator) in multiplicities.

4.5.2 hasObjects relation

Further on, a composition relation is used between classes; in *quasar* terminology it is called *hasObjects*. Such relation indicates that an object of one class shall be considered a part of an object of another class.

hasObjects relation is intended to indicate firm belonging in which objects are still discernible as individual parts but in normally operating system their presence is required. For example, one might imagine a thermostat; it might be said that a temperature sensor is an integral part of such a thermostat, so the relation between the thermostat and the temperature sensor is a good example of *hasObjects* relation.

To cover a wide range of real life situations, *hasObjects* relation is assigned cardinality of belonging objects. Such cardinality is chosen to be a numeric range specified by minimum and maximum number of belonging objects. Either of bounds might be absent; the defaults are zero for the lower bound and infinity for the upper bound.

hasObjects relation serves another very important goal overall: it can be used to model (describe) systems of any level of complexity. That is: it can nest multiple layers of classes one into another. Therefore one can build systems in tree-like fashion, from *quasar classes* on the very top (or, root of the tree, if tree is taken as the analogy) towards *quasar classes* at the very bottom (leaves), naturally the scope of the former would correspond to huge parts of the control system (or maybe even the whole control system) while the latter would correspond to many fine-grained elements such as individual sensors.

Moreover, *hasObjects* can be used to create cyclic relations between classes, in addition to the tree example mentioned in the previous paragraph. Therefore *quasar classes* are organized as a graph in which classes are the nodes and *hasObjects* relation builds edges. A tree, as an acyclic graph [62], is just a subset of the whole span of possibilities. The ability to organize *quasar classes* as a cyclic graph brings important features which are shown in section 5.3.

4.5.3 Variables

A *quasar class* may have any number of variables. Variables usually correspond to process variables, i.e. each variable corresponds to a particular aspect of state of given physical object, such as its position, temperature or another physical quantity.

It is important to point out that *quasar variables* take most of their properties after OPC UA variables which are more related to information exchange viewpoint than to programming viewpoint. Therefore *quasar variables* should not be confused with variables in any programming language.

The notation defines two types of *quasar variables*: *cache-variables* and *source-variables* [5]. *Cache-variables* are conceptually much simpler; they are the pristine example of storage of state. *Source-*

[4] In the OPC UA standard such term does not exist; one rather speaks about *object types*.

[5] The origin of names comes from technical aspects of how such variables are realized in software: cache-variables are usually stored in RAM while source-variables can not be stored in RAM and each read or write operation starts some computational or I/O activity to obtain or store its value.

variables, on contrary, do not relate to any persistent state. That difference has multiple consequences in software creation and many other aspects. *Cache-variables* and *source-variables* have the following similarities in the notation: they require attributes explaining type of data they handle as well as whether there are information consumers or producers or both.

Cache-variables, in addition, can be specified with an initializer which might be a value specified directly in the notation or obtained from configuration.

4.5.4 Methods

Apart from variables, a *quasar class* may also have methods. From a control system modeling perspective methods express invoking action(s).

Typical examples of method names would be: *reset*, *triggerAlarm*, *turnOff*, *programChip* etc.

Methods are allowed to take supplementary data for the invocation in the form of arguments and they might return supplementary return values. For instance, *reset* method defined in a certain class could take an argument specifying which type of reset to invoke (software restart or perhaps a full restart involving hardware reset) and return a value specifying if such a *reset* was successful.

4.5.5 Configuration data

Variables (both cache- and source-variables) and methods are involved in passing of "operational"[6] data and control. Often, there is other data which is not directly related to the operational state but which is still crucial - for instance, constants and configuration parameters.

Constants in such control systems typically relate to quantities which are a result of design decisions or come from physical limitations of hardware, for instance: maximum rotational velocity of a motor or maximum signal level that given analogue-digital converter can withstand.

Configuration parameters usually relate to supplementary information that has no direct use in the control algorithms but which is required for the system to operate properly, for example: information on how different physical quantities are mapped to different ADC channels or what is the network address of a host where the data source is. Configuration parameters are either constant or change very rarely, for instance only when the system (or its part) undergoes maintenance.

Configuration data in *quasar*-based projects is stored as *config entries* which can belong to a *quasar class*. Each configuration entry needs, at minimum, a name and the data type information.

Since configuration data comes from *outside*, the author proposed to support configuration constraints of different types (by pattern, by enumeration, by range etc.).

4.5.6 Device logic specification

As an important part of the *quasar architecture*, the author defined how custom code should be integrated with *quasar*-based OPC UA servers. For example, in some *quasar classes* there might be no necessity to use custom code at all. In the other cases, the custom code needs to be used but there is a variety of decisions to take, for example:

- which processing resources should be used to execute the code? (a thread-pool job or maybe the protocol stack thread?)
- how to synchronize concurrent invocations of the same device logic on the same OPC UA object?

Therefore the author proposed the usage of device logic integration hints, which can be defined as supplementary information not modeling the control system per se, but rather helping to describe the integration. Those topics are however very tightly related to software architecture and code generation, therefore they are detailed in sections 4.8 and 4.9.

[6] That is, related to current operation of the control system - such as current state of given physical objects.

4.6 The notation interchange

The notation, while being on a very high level of abstraction (i.e. it can be easily understood by a control system developer), needs a storage and interchange format.

The author identified the following requirements and desired properties for such a format:

- ability to encode the notation.
  The format should be able to store the information corresponding to the notation.
- extensibility.
  The format should be extensible according to future needs. For instance, new elements might need to be added to the notation. There should be support to either migrating between different versions of notation or by usage of innate extensibility of the chosen format.
- support of editing tools.
  The format is to be used by a control system engineer to write concepts in it. Therefore an editor application would be needed to compose a file in the chosen format. Being able to use existing, well-known editor applications would be a significant advantage, because a need to create such an editor from scratch would be eliminated and also because it would be easier for an engineer to use tools that he/she already knows.
- good fit for automated processing tools, including code generation tools.
  The information stored in the format would be used by computer programs, for instance to generate actual products. Therefore it is preferred that the format would permit the information to be easily manipulated by computer programs.

Taking into account the criteria, the author chose The Extensible Markup Language (XML) [23], constrained by a schema definition (in the XSD format [63, 64]). It supports all requirements given above: it can store the notation, it is extensible by nature, there is a wide selection of tools which can be used as editors.

In addition there is large selection of automated processing solutions for XML. XML is standardized, which is an advantage. In addition, XML is human-friendly (in case of need, a human being might read XML as-is) while being software-friendly at the same time.

4.6.1 Schema definition

The schema definition for the notation interchange was written in the XSD format. The visualization of the most relevant part is presented in figure 4.2.

The reasons for usage of schema definition are:

- guarantee of notation coherency and correctness. [7]
  The schema is complete with regards to notation constraints, for example:
  - different elements and attributes are characterized by usage constraints (i.e. *use* keyword in XSD parlance) such as *required* or *optional*,
  - naming conventions are imposed (e.g. a *quasar class* name must follow `CapitalCamelCase` convention),
  - different key constraints are introduced such that, for instance, variable names and config entry names are always mutually unique within a *quasar class*.
- possibility of usage of generic XML editor.
  Due to usage of a strict schema definition with a generic schema-aware XML editor it was possible to offer to final users an editor tool capable of editing the notation without the need to create such an editor from scratch.
- support for automated validation of a notation against the schema.

[7] A small part of *quasar design* validation is achieved using the XSLT due to pragmatic reasons. The relevant details are laid out in section 4.8.1.

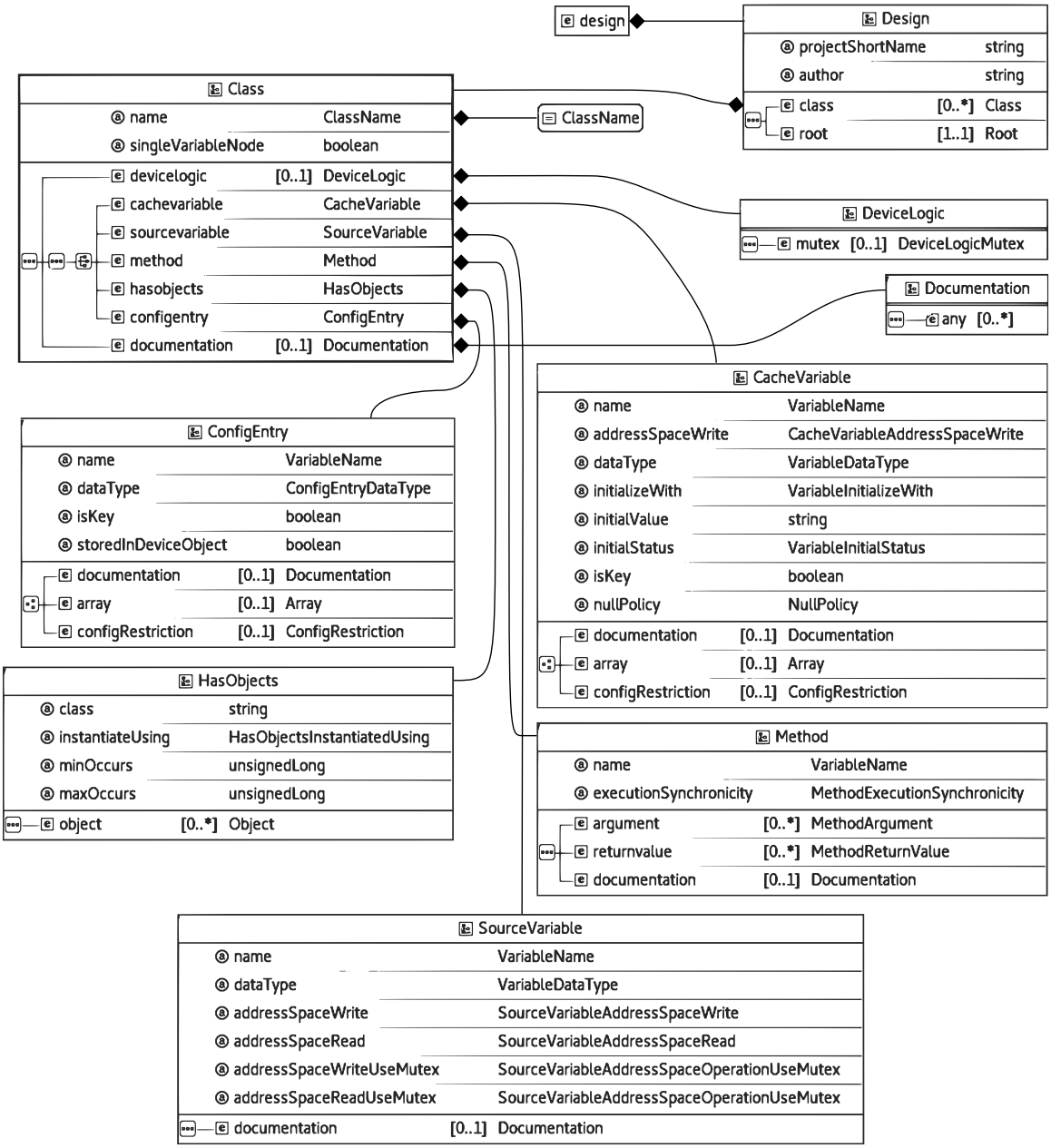


Figure 4.2: Schema definition visualization (partially produced by Eclipse XML Tools [65]). The visualization shows only certain relevant top-most elements like *quasar design*, *quasar class* etc.

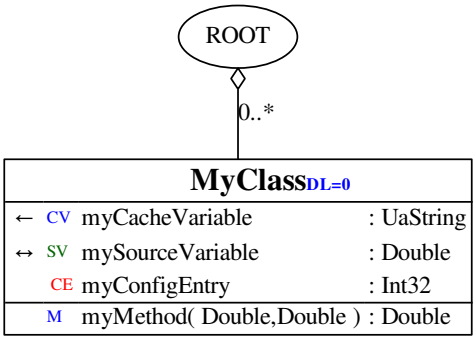


Figure 4.3: *quasar design diagram* of a very simple design comprising of one class (`MyClass`) with one cache-variable (`myCacheVariable`), one source-variable (`mySourceVariable`) and one method (`myMethod`).

4.6.2 Support for schema changes

Schema changes need to be taken into account in a software project of such a long duration. Over the 5 year period of work on the project the following approaches were used:

- backwards-compatible schema changes
  Backwards-compatible schema changes were used whenever new features were added to *quasar* without affecting existing features and their mode of operation. Thanks to extensibility (which is an innate feature of XML) adding new types of XML elements or adding XML attributes was easily accomplished [8].

- non-backwards-compatible schema changes
  For non-backwards-compatible schema changes a schema upgrade mechanism was made basing on XSLT[66]. In certain cases, when an automated decision could not be taken by the upgrade mechanism, expert manual fix was necessary.

4.7 The notation visualization

The notation stored in the format described in section 4.6 can be viewed directly by a developer, but such a way of working is not convenient and thus it would not be efficient. Therefore a graphical representation of visualization was proposed by the author. Such a visualization of the notation is termed *quasar design diagram* and *quasar* provides an accompanying tool to create it.

The graphical representation uses many conventions known from the UML[28]. The motivation for such reuse of UML is because many software professionals are familiar with UML, therefore the learning curve to read and understand *quasar design diagrams* is minimized.

A *quasar design diagram* of a very simple design is shown in figure 4.3. The design visualized in the figure was comprising of one class (`MyClass`) with one cache-variable (`myCacheVariable`), one source-variable (`mySourceVariable`) and one method (`myMethod`).

The ellipse labeled as "ROOT" is the *quasar design root* and it corresponds to the conceptual top-level of the design which is an imaginary element that "owns" top-most *quasar objects*.

A rectangle corresponds to a *quasar class*. The rectangle has three compartments:

[8] These attributes were either optional or they were required and their default value was provided in the XSD.

- the first one is labeled with the class name and supplemented with information if the class has device logic ("DL=")
- the second one is a list of cache-variables ("CV"), source-variable ("SV") and config-entries ("CE"). The arrow on the very left on each row with a variable indicates allowed information flow:
  - if the arrow points towards the outside boundary the variable can only be read (i.e. it is a source of information)
  - if the arrow points opposite the variable can only be written (i.e. it is a consumer of information)
  - if the arrow is bidirectional it can be both read and written

  The data type of the variable or the config entry is given after the semicolon.
- The third compartment is a list of methods ("M"). One finds a list of input arguments in the parenthesis. The return values(s) - if any - are given after the semicolon.

An arrow with the diamond shape at the end corresponds to *hasObjects* relation, similarly to UML. The side of the arrow with the shape corresponds the class which *owns* while the other one (without the shape) is the *owned* one. Cardinality is always shown, the default cardinality is shown as "0..*" which corresponds to any number of owned objects.

An example *quasar design diagram* of a design which was realized as a production control system is shown in figure 4.4.

The tool to create the visualizations was made as a transformation from *quasar design* to *Graphviz DOT* file, which is a file format describing graphs, which can be passed to *Graphviz*[67] software to visualize graphs. Such a transformation is described in section 4.8.

4.8 Transformation of design and code generation

The *quasar design* specified in aforementioned notation and stored as a XML file is used to generate a potentially large portions of source code as well as other artifacts.

Different techniques and approaches can be used to perform such a transformation. An overview is available in e.g. [31].

Out of many different choices, two were given attention: the first one based on XSLT, and the second one based on a generic template engine, namely *Jinja2*.

4.8.1 Design transformation with XSLT

The Extensible Stylesheet Language Transformations - XSLT[66] - is a language[9] defining transformations of XML documents into other text documents. The syntax of the language is based on XML. The overall usage mode is that two inputs are required: the factual input document and the transformation document (which is a XSLT transform), both of which are in XML format. A computer program called *XSLT processor* is then executed with both inputs and it generates output which can also be XML or just a generic text. Therefore it is possible to generate a wide selection of documents including source code in many programming languages, HTML documents, generic text (e.g. user instructions) and other.

XSLT is tightly bound to XML; it does bring many benefits, for instance it naturally employs the XPath language[68] which is used to efficiently traverse and query the input document.

XSLT is a declarative language [61], which means that it intentionally skips control flow and favors expressing the transformation rather than how to achieve it. Such a feature might be considered a double-edged sword: on one hand the language is more tailored to define transformations

[9] This thesis relates to XSLT in version 2.0 unless otherwise noted.

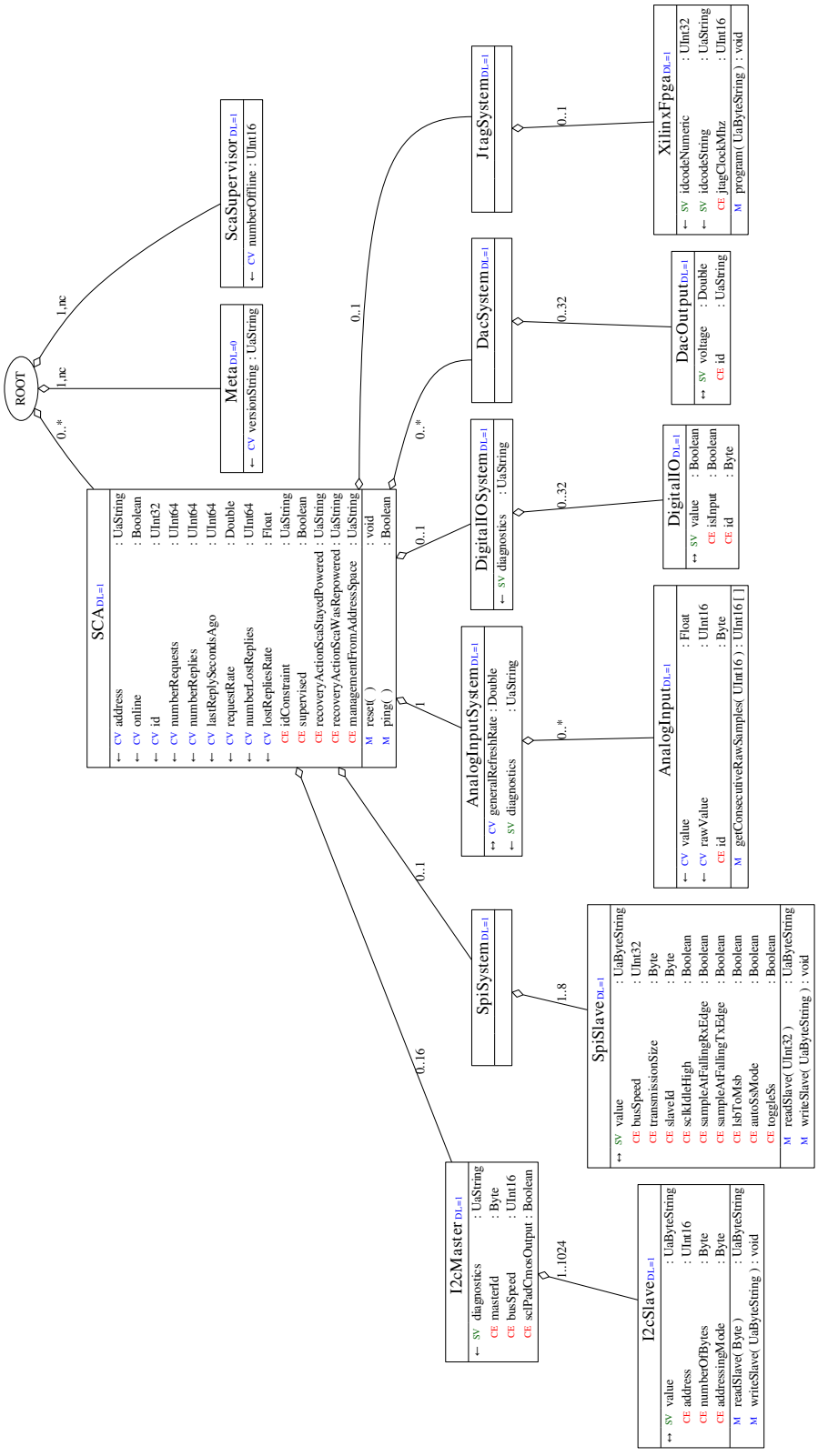


Figure 4.4: *quasar design diagram* of a more realistic example outlined further in section 5.4.

and less cluttered, but on the other hand implementing certain common programming constructs like loops is far more difficult and uncommon from imperative programming languages.

XSLT requires a computer program called *XSLT processor*. *Saxon2*[69] was used for all work performed in the context of the thesis.[10]

XSLT was used in *quasar* in multiple scenarios. In all cases the input document was the *quasar design*. The following output modes were used at the time of writing:

- C++ source code
  15 transformations were in use. From the XSLT point of view the transformations were simply configured as text output. A particularly inconvenient feature was observed that it was not possible to maintain desired indentation of the output source code, i.e. in the best possible case the indentation of the resulting C++ code followed the indentation of the XML transform, which can be considered a disadvantage. Therefore the author tried to employ automatic indentation and source-code formatting software as the post-processing step. Such a decision allowed to decouple formatting of the XSLT transform from desired formatting of the resulting C++ code. Common tools like *astyle*[71] or *indent* [11] or author's made indent tool were used successfully for the purpose. Consequently, they became supported by *quasar*.

- CMake scripting
  2 transformations were in use to transform from *quasar design* to files written in the CMake scripting language[72]. These files are used in the CMake-based build system employed by *quasar*. The motivation to generate CMake scripting came from the *quasar* design choice of storing source-code of different *quasar classes* in different files. However, the list of such files could only be obtained from the *quasar design*, which as a fairly complex XML file, could not be parsed by any built-in CMake mechanism.

- HTML documents
  2 transformations were in use. In both cases the purpose was to generate end-user documentation. Such an use mode is detailed in section 4.9.1.13.

- XML with *quasar design*
  Such a mode was used to perform migrations of XML files with *quasar designs* following schema changes, as remarked in 4.7.

- WinCC OA scripting
  2 transformations were in use to generate Siemens WinCC OA CTL scripting. The process is detailed in 4.9.3. Similarly to the C++ output mode, the additional post-processing step with automatic indentation was chosen.

- XML Schema Definition (XSD)
  1 transformation was in use. The generated XSD served as the XML schema for the configuration of *quasar servers*, as detailed in section 4.9.1.5.

- Graphviz dot language
  1 transformation was in use. The application was detailed in section 4.7.

- no output mode - but used as a validation engine
  Most of *quasar design* validation is achieved using XSD constraints detailed in section 4.6.1. However validation of certain aspects of such a design stored in XML using XSD constraints was sometimes found not pragmatic enough and the author realized that it would be much easier to achieve that goal with a declarative language (like XSLT) rather than with constraints language (such as XSD). Therefore it was attempted to distribute the *quasar design validation* into two steps, the first realized as XSD constraints and the second one as a XSLT transform

[10] There exists a widely used open-source XSLT processor called xsltproc[70] which supports XSLT in dialects older than 2.0 which is too old to implement features desired in *quasar*.

[11] *indent* can be easily found in numerous UNIX/Linux systems, under the typical package name of *indent*.

in which the output is not important but a possible validation fault is signalized using `< xsl:message terminate=''yes''/>` construct. Such a message will abort the transformation and the *quasar build system* could notify the user of the design fault. See *1* in figure 4.5 and in table 4.1.

Listing 4.1: Part of a XSLT transform used to generate *quasar DeviceRoot* (detailed in section 4.9.1.8). The example shows a difficult case in which XSLT code strongly dominates the C++ code making its analysis and maintenance difficult. In multiple lines (e.g. lines 2 and 5, inside `select` attributes) one sees embedded XPath queries which in the author's opinion is often a very favorable feature of XSLT. One also sees usage of XSLT 2.0 functions, for example of a function called `DClassName` in line 6, which is provided by *quasar* in a library of common functions for XSLT transformations.

```
1  /* find methods for children */
2  <xsl:for-each select="/d:design/d:root/d:hasobjects">
3    <xsl:variable name="class"><xsl:value-of select="@class"/></xsl:variable>
4    <xsl:for-each
5        select="/d:design/d:class[@name=$class]/d:cachevariable[@isKey='true']">
6      <xsl:value-of select="fnc:DClassName($class)"/> *
7        get<xsl:value-of select="$class"/>By<xsl:value-of
8            select="fnc:capFirst(@name)"/> (
9      <xsl:choose>
10       <xsl:when test="@dataType='UaString'">std::string </xsl:when>
11       <xsl:otherwise><xsl:value-of select="@dataType"/></xsl:otherwise>
12     </xsl:choose>
13     key) const;
14   </xsl:for-each>
15 </xsl:for-each>
```

An example of a design-code transformation implemented for the *quasar* project is shown in the listing 4.1.

4.8.2 Design transformation with Jinja2

Jinja2 is a template engine for Python[73]. It gained wide usage, mostly in combination with Web frameworks for Python such as *flask*[74] where in the *Model-View-Controller* architecture it serves the role of *View*. However as a generic template engine it is applicable to many different problems including those described in the thesis.

Jinja2 differs significantly from the XSLT transformations outlined in 4.8.1. While XSLT is tightly bound to the XML (so, a XML document is directly an input to the transformation), *Jinja2* can be considered a Python library and as such it uses data which are passed as arguments from a parent Python-based script. *Jinja2* is therefore entirely decoupled from the way in which the input data (here: *quasar design*) is stored physically (in a file, in a *Document-Object-Model* representation in the memory, on the Web, in a database etc.) as well as from the way in which it is represented as binary data (XML, JSON or yet another representation). A direct consequence of this fact is that a way to access *quasar design* must be implemented. A convenient way of doing this (for instance, similarly to *XPath* which is inherently built into XSLT) might be preferred to make *Jinja2*-based transformations not more cumbersome than XSLT-based ones.

An example transform taken from *quasar ecosystem* (the *Poverty* optional module[75]) is shown in the listing 4.2. In the example, the input data to the transform is taken from the *quasar design* using the *lxml* built-in Python module, then an *element tree* is constructed by parsing the XML file. Finally, a *XPath* query is done; its results are passed to the *Jinja2* input arguments.

4.8.3 Managing changes of generated files

In the set of design-code transformations provided by *quasar* (see section 4.8.4) there are two which produce source code files shared with a developer. Namely these are the transformations generating

Listing 4.2: A simple transform written in *Jinja2*. The transform takes an argument called `classes_cachevariables` which in this particular example is a Python dictionary. Note how *Jinja2* syntax (easily noticed by statements in curly braces and the percentage sign) is interleaved with C++ code: lines 1 and 12 show an iterative loop, and lines 2 and 5 show value substitution. In this particular example, if `classes_cachevariables` is a Python dictionary representable as `{'MyClass':['variable1']}` then one method with a signature `AddressSpace::ASMyClass* Poverty::getMyClass(const std::string& name)` will be generated in the C++ programming language.

```
1  {% for className in classes_cachevariables.keys() %}
2  AddressSpace::AS{{className}}* Poverty::get{{className}} (
3    const std::string& address)
4  {
5      auto result = AddressSpace::findByStringId<AddressSpace::AS{{className}}>(
6        s_quasarServer->getNodeManager(),
7        address);
8      if (!result)
9        throw std::runtime_error("Object not existing");
10     return result;
11 }
12 {% endfor %}
```

device-logic classes (see 4.9.1.6). The typical lifetime of those classes is the following:

1. once an initial version of *quasar design* is ready, a developer requests *quasar* to create (generate) *device-logic classes*. This step creates "empty" classes, that is: classes which are complete enough for the *quasar server* to be built but which have no useful implementation and they throw *NotImplemented* OPC UA status code.

2. then a developer adds source code, possibly using external libraries/frameworks such as Hardware Access Libraries.

3. it is expected that *quasar design* evolves over lifetime of a *quasar server*, for example: new features (including new classes or variables) are added. A *device-logic class* was already implemented and therefore there might be a need to "update" it to follow the *quasar design*.

The solution successfully attempted by the author was to use a high quality *diff and merge tool*. The *kdiff3* proved to deal efficiently with the task.

4.8.4 Overview of transformations in the quasar ecosystem

Most important transformations of *quasar design* into various artifacts are shown in figure 4.5. Detailed description is shown in the table 4.1.

4.9 Software architecture of produced software components

The software architecture of all types of products of *quasar* (servers, clients, SCADA integration, etc.) need to be defined. The particular aspects that need to be addressed are:

- decomposition into functional blocks and modules,
- paradigms to follow, for instance whether to apply event-driven architecture[61] or whether to follow certain design patterns[76],
- identification of which part of products must be generated, which might be generated and which should not be generated,
- the model of error handling (for example: whether to use exceptions and to which extent),

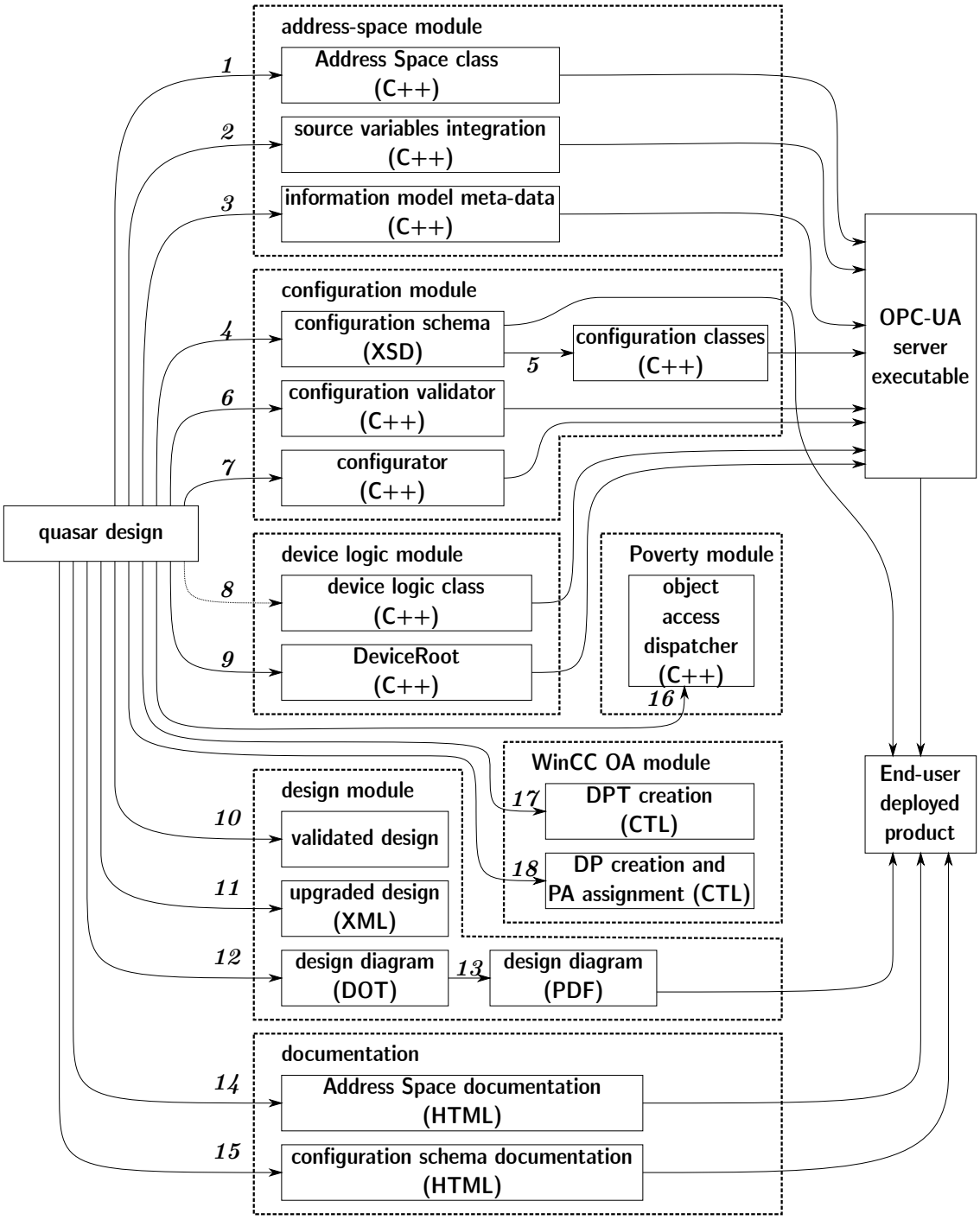


Figure 4.5: A diagram showing most important transformations of *quasar design* into various artifacts. Numbers in curly braces designate transformations types, see the table 4.1 for detailed description. Arrows in solid line designate transformations which are considered automatic, that is: a developer does not have to invoke them on request. Arrows in dashed line are invoked only on developer's request.

#	Name	Description	Section
1	ToClassHeader ToClassBody	Creates Address Space class header and body.	4.9.1.4.
2	ToSourceVariablesHeader ToSourceVariablesBody	Creates source-variables integration logic, e.g. distribution of source-variables work into independent thread-pool jobs.	4.9.1.4.
3	ToInformationModelHeader ToInformationModelBody	Creates code instantiating parts of the Address Space, such as Object Types.	4.9.1.1.
4	ToConfigurationXSD	Creates the XML Schema definition for target server configuration description.	4.9.1.5.
5	*XSD-C++ binding generator*	Creates C++ configuration classes.	4.9.1.5.
6	ToConfigValidator	Creates supplementary configuration data validation code.	4.9.1.5.
7	ToConfigurator	Creates source code which parses the configuration data and instantiates objects in the Address Space and Device Logic domain.	4.9.1.5.
8	ToDeviceHeader ToDeviceBody	Creates *device-logic classes*. Two versions are used, for base classes and the factual classes.	4.9.1.6.
9	ToRootHeader ToRootBody	Creates DeviceRoot.	4.9.1.8.
10	Validation	Passes through when 2^{nd} order validation is successful.	-
11	ToUpgradedDesign	Fixes the design due to backwards-incompatible design schema changes.	-
12	ToDot	Creates a representation of given *quasar design* in the graphviz notation.	4.7.
13	*Graphviz invocation*	Performs graph visualization using the *Graphviz* software.	4.7.
14	ToAddressSpaceDocHtml	Creates end-user documentation outlining address-space structure and behavior.	4.9.1.13.
15	ToConfigDocumentationHtml	Creates end-user documentation about configurations aspects.	4.9.1.13.
16	Poverty.jinja.hpp Poverty.jinja.cpp PovertyPythonModule.jinja.cpp	Creates object access methods for foreign programming languages.	5.5.3.
17	WinCC OA DPT creation	Creates *CTL* code capable of creating *data point types*.	4.9.3.
18	WinCC OA DP creation and PA assignment	Creates *CTL* code capable of creating *data points* and assign *peripheral addresses*.	4.9.3.

Table 4.1: Description of different *quasar design* transforms outlined in figure 4.5. Unless printed in italics noted, for the transforms of *quasar design* which are prevalent, the file name of given transform can be obtained by prefixing the literal *design* with the value from the column *Name*.

- the preferred programming constructs if there is multiple choice - for example, whether to prefer pointers or references in case C++ is used [45].

The software design and development aspects discussed in this thesis relate mostly to the C++ programming language. During the 5 years experience, often additional layers were built on top of it, for instance to use the deliverables in the Python programming language. Nevertheless the impact of designing for the C++ was important.

For instance, C++ is a compiled programming language with strong and static type system [12] [61] which is an attractive feature for high-reliability control systems because many errors can be found at compile time. In the author's opinion one should profit from such a compiler assistance as early as possible in the high reliability software development process. On the other hand, there are numerous features of C++ which are double-edged swords. For example pointers are a core element of C++ which often is applied without enough competence by programmers, jeopardizing computer programs stability and robustness. Careful architectural choices supplemented by firm programming decisions may alleviate such risks.

The software architecture presented in this section is an effect of author's work which initially was done by literally *writing* OPC UA software components. After short time of such a work style it became obvious that it is not a solution that could ever be effective and efficient taken into account requirements presented earlier in the thesis.

The architecture specified in the following subsections is given per type of software component (OPC UA server, OPC UA client, SCADA integration, etc...) rather than by its role in the control system (sensor interface, actuator interface, controller...) because the impact of software component type turned out to be by far greater.

4.9.1 OPC UA servers architecture

The primary differences between an OPC UA server and OPC UA client are:

1. the server serves requests generated by clients (i.e. the client always talks first),
2. the server instantiates and maintains the OPC UA address-space (the client does not have any address-space).

The relation between *quasar design* and the resulting OPC UA address-space is very important and therefore will be detailed.

4.9.1.1 Address Space model

There exist many possible mappings between a conceptual control system object (i.e. an instance of a *quasar class*) and its representation in the OPC UA information model.

The author attempted to choose a simplest possible mapping in which *quasar class* instances are represented by OPC UA objects. Those objects are further referenced to OPC UA variables and OPC UA methods related by, for instance, *HasComponent* OPC UA relation type. An example of such a mapping is shown in figure 4.6.

The following conventions were chosen as the basic mapping (which also is considered the default one unless another mapping is explicitly chosen):

- a *quasar class* is mapped to a node of ObjectType class,
- an instance of a *quasar class* maps to an Object,
- *quasar cache-variables* and *quasar source-variables* map to OPC UA variables (from the address-space point of view no difference between them is visible),
- a *quasar method* maps to OPC UA method; arguments and return values map to in- and out-arguments,

[12] There are certain mechanisms like *dynamic cast* and *Run-time type identification (RTTI)*.

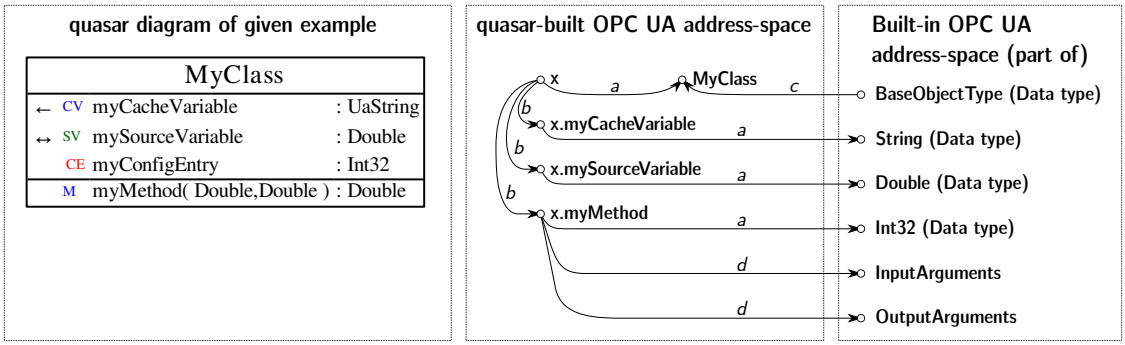


Figure 4.6: Example *quasar class* and its representation in the OPC UA address-space. *x* was chosen as the name of an object instantiated from the class *MyClass*. *a*, *b*, *c*, *d* stand for (built-in) reference types: *a* is *HasTypeDefinition*, *b* is *HasComponent*, *c* is *HasSubtype* (note reverse placement), *d* is *HasProperty*. For the chosen node classes: *x* is an Object, *MyClass* is an ObjectType, *x.myCacheVariable* and *x.mySourceVariable* are Variables and *x.myMethod* is a method.

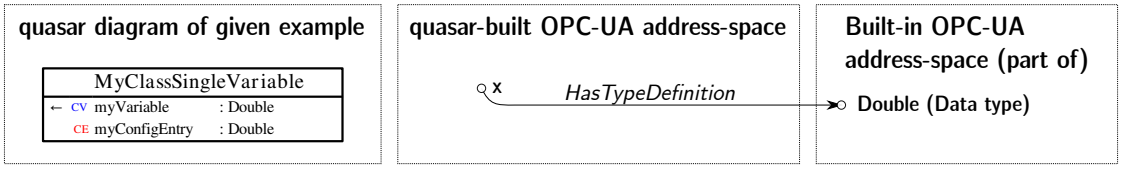


Figure 4.7: Example *quasar class* with singleVariableNode and its representation in the OPC UA address-space. *x* was chosen as the name of an object instantiated from the class *MyClassSingleVariable*. *x* is an OPC UA variable. Note that the variable name declared in the design as "myVariable" is not relevant in the address-space representation (i.e. it vanishes).

- *hasObjects relation* maps to OPC UA reference of a kind.

Such a basic mapping has an important advantage apart from simplicity: it is static in the run-time, and the obtained address-space can be predicted solely from the given *quasar design* and the objects instantiated from the configuration.

However in certain cases it is desired that the address-space behaved *as if* the variables instantiated per objects were entirely run-time dependent. For such a case the author proposes another mapping which is called *singleVariableNode*.

SingleVariableNode mapping might be chosen per *quasar class*. In the mapping the *quasar class* is allowed to have only one variable (either a *cache-variable* or a *source-variable*); it can not have any methods nor any references to dependent classes, therefore the usage of *hasObjects* is excluded. A *quasar object* gets represented by an OPC UA variable rather than by an OPC UA object. The usage of the mapping affects only the address-space representation, there is no impact either on Device Logic or on Configuration perspective.

An example of *singleVariableNode* is shown in figure 4.7.

The real-life application of the mapping is for all cases in which given *quasar object* needs to have OPC UA variables not defined by design but rather freely created in the run-time. Such an use case appears when given part of the control system can not be described a priori but rather is run-time dependent; for example the relevant description is obtained by a discovery process of some kind.

Another aspect of *quasar* address-space model is that it contains information which, from a control

system point of view, might be considered *meta-information* [13]. The meta-information is used to supplement additional knowledge about operational information. A particular example is that each *quasar class* has its type representation in the OPC UA address-space. Such a type representation can be seen in figure 4.6 with the reference labeled as *c*. The presence of meta-information brings many advantages, for example it might be used to create OPC UA clients in run-time as exemplified in section 4.9.2.2. The meta-information discussed in this chapter is inserted to the address-space by a design transformation called *ToInformationModel*, see section 4.8.4 for details.

4.9.1.2 Functional decomposition of quasar-based servers

In the course of *quasar* developments, the following functional decomposition of servers into modules[14] was assumed:

- AddressSpace
  The AddressSpace module builds, configures and maintains the OPC UA Address Space of a *quasar server*. It also serves as the glue code between OPC UA protocol stack and device-logic, for instance by adapting data representation from OPC UA domain to the programming language domain. The AddressSpace module provides functionality which is detailed in sections 4.9.1.1, 4.9.1.4 and 4.9.1.10.

- Device
  It is where device-logic code resides. The Device module provides functionality detailed in sections 4.9.1.6 and 4.9.1.8.

- Configuration
  Configuration module loads and processes configuration data and instantiates address-space and device-logic objects and builds links between them. the configuration module provides functionality detailed in section 4.9.1.5.

- server

  Server module is a top-level module which integrates (i.e. is a parent of) aforementioned modules and provides certain fundamental functions of OPC UA server such as the start-up of the OPC UA protocol stack.

4.9.1.3 relation of quasar design to address-space, device-logic and configuration classes

The *quasar design* was detailed in 4.5.
The following relation holds:

- every *quasar class* is followed by:
  - address-space C++ class (always),
  - configuration C++ class (always),
  - device-logic C++ class (only in case when given *quasar class* is specified to have device-logic).

- such a split into Address Space, Device and Configuration is motivated by clean separation of concerns relating to the same *quasar class*. Address Space relates to representation in the OPC UA domain, Device relates to the behavior while Configuration relates to configuration aspects of the class.

[13] The term should not be confused with *StandardMetaData* which is a part of *quasar* representing information such as logging levels of given server instance, etc.

[14] The name and functions of the module *Server* was inspired and taken after examples attached to the Unified Automation Software Development Kit[35]. The name of the module *Configuration* was taken after the CANopen OPC UA server product detailed in [17]. The names and functions of remaining modules listed in section were proposed by the author.

- The Address Space classes and Configuration classes are always generated from the *quasar design* and they are intentionally hidden from a developer. They are not supposed to be manually modified. On the other hand, the Device classes are partially generated: classes are generated with no factual implementation[15]. A detailed description of change management of Device classes is detailed in section 4.8.3.

4.9.1.4 Address Space class architecture

An address-space class serves as a proxy class[76] to the representation of objects of given *quasar class* in the OPC UA address-space.

Such an address space class is one of the main outcomes in the software automation discussed in this thesis because it fulfills a number of important requirements presented in the thesis.

For instance, it does serve as an interface between a software developer (who possibly is not an OPC UA expert) and the OPC UA software concepts (which require steep learning curve and significant expertise).

Functionally, the motivation of address-space class(es) is to provide software developers with:

- a way to use the address-space without expert knowledge of OPC UA, including:
  - instantiation, configuration and management of OPC UA nodes such as variables, objects, references, methods etc.
  - handling different ways in which variables can be used from the OPC UA perspective (e.g. read requests or monitored items)
  - handling call-backs coming from the OPC UA protocol stack which are to be finally served by manually written code running within given *quasar server*
  - encapsulation of software development kit or protocol stack API
- a native C++ type system instead of OPC UA derived one. This permits direct integration of existing source code as long as it uses standardized types (for example, the type system provided by the C++ standard library and by C++ STL, the standard template library) .
- safe data conversions with complete support of conversion faults (for example, safely handling casting between OPC UA variant type and C++ numeric data types).

An Address Space class inherits from a class representing OPC UA object in a chosen OPC UA protocol stack. In case of the chosen reference protocol stack - UASDK[35] - all address-space classes inherit from `OpcUa::BaseObjectType`.

In the *quasar architecture* defined by the author, an address-space class comprises of:

- a constructor, called when an instance of this class is about to be created. It requires the configuration class object describing the instance and an OPC UA address-space of a parent OPC UA object "under" which the new object will be inserted.
- a destructor.
- a number of methods which will be used to query the class about its OPC UA attributes [16], for example `typeDefinitionId()` which will be invoked whenever any OPC UA client queries this class for its type definition.
- a number of methods which are dependent on the *quasar design* for given class, for instance setters, getters and delegates detailed later on.
- a reference to all variables and methods belonging to the class.

[15] In the computer science jargon, it is called "stub"[44].

[16] The OPC UA standard defines a list[1] of common node attributes, like: node identifier, node class, browse name, description, etc.

Verb	Invoked by	Applies to	Effect
set	custom code	CV	sets the cached value and publishes the new value to all subscribed clients
get	custom code	CV	gets the cached value
write	protocol stack	SV, CV+W	passes the call to Device Logic
read	protocol stack	SV	passes the call to Device Logic
call	protocol stack	M	passes the call to Device Logic

Table 4.2: Address-space verbs chosen to name different operations on address-space class variables and methods. CV stands for cache-variables, CV+W stands for cache-variables with write delegates, SV stands for source-variables, M stands for methods.

Certain verbs were carefully chosen to express actions involving Address Space classes and listed in the table 4.2. The aforementioned verbs are used to create specific methods in the Address Space class for applicable cases. The methods are named by concatenating the verb with the relevant camel-cased name. Thus, those methods are called verb methods. For example, for a cache-variable called `measurementX`, the following C++ methods will be created: `setMeasurementX` and `getMeasurementX`.

The methods described in the previous paragraph are where in fact a lot of developer-perceived value is created. For example, a method which implements *write* verb guarantees that only conforming client requests will reach the custom code, other requests will be safely discarded.

Another aspect in which developer-perceived value is created is the support for both synchronous and asynchronous service calls. The aspect relates to *write*, *read* and *call* verbs. All of those verbs will create methods which are called by the OPC UA protocol-stack subsequently calling relevant Device Logic methods. In many cases of real-world control systems such methods might require significant time to execute, for instance, for a *write* call, it might take significant time for the hardware to actually apply new value which was sent along such an invocation. On the other hand, such a call in fact is originating from the protocol stack and therefore it might be required to be a non-blocking call. Thus the Address Space class methods, being the part linking protocol stack and the custom code, provide features to run the custom code in a separate thread [17], letting protocol stack continue its processing.

An address-space class serves yet another purpose: creation of its own OPC UA node address in the OPC UA address-space. The OPC UA standard provides different addressing modes. *quasar servers* use only *string addressing* by choice. By adopted convention (inspired by prior art), a node address of a child variable, method or object is created by concatenation of the parent string address, a dot and the child name.

4.9.1.5 The configuration module and configuration classes architecture

The configuration is understood as specification of objects of different *quasar classes* which should be instantiated on server start-up, including their configuration data. The configuration module is composed from two major parts:

- configuration classes - classes (one class per every *quasar class*) storing or interfacing configuration data for an instance of such a *quasar class*,
- configuration subsystem - composed of *configurator* and *config validator* which are responsible to load configuration data source using aforementioned *configuration classes* and instantiate *quasar classes* for a given instance of a *quasar server*.

[17] In fact, the custom code will be run from the context of a task spawned as a thread-pool job. The motivation of the usage of a thread-pool is to let possibly millions of such asynchronous operations to be concurrently run. Without thread-pool such scale would not be possible because operating systems often limit maximum number of threads to around few thousands and even if that was possible, memory constraints would possibly be too high (each thread requires its own stack at minimum). More detailed discussion on this topic is available in e.g. [77].

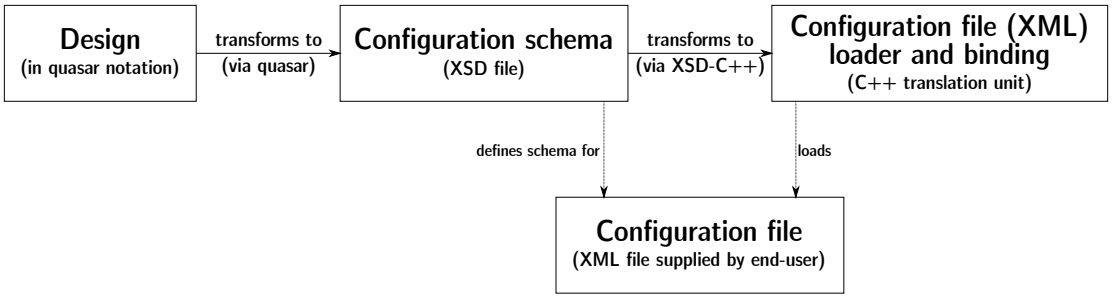


Figure 4.8: The work-flow (and also a dependency graph) of the Configuration module.

Configuration classes serve as means to access and store configuration data. The author chose the following criteria for the outlined architecture:

- safe handling of configuration data (i.e. with support of constraints, such that the whole of configuration data is guaranteed to be present and in the right format),
- support of configuration validation as a by-design feature (i.e. a *quasar server* should be able to validate configuration data without any external tools),
- possibility to use different types of sources of configuration data (for example, a XML file, a SQL database, a Web query etc.),
- no necessity to write any source code for the configuration module.

The only mandatory piece of (unconditionally present) data is called `name` and is of *string* type. Apart from being an identifier, it serves an important role of forming an OPC UA address. The procedure of address creation was detailed in section 4.9.1.4.

Each *quasar config-entry* and each *quasar cache-variable* with *configuration initializer* results in a new piece of data in the configuration class named after that config-entry or cache-variable. For example, if a *quasar class* has a config-entry called `identifier` declared as unsigned integer, it is expected that a Configuration class of that *quasar class* has a method called `int identifier()` returning the identifier from configuration data. Similarly, when *hasObjects* relation was used between a `Parent` and a `Child` class then it is expected that the configuration class of a `Parent` has a method called `Child()` returning a collection of children objects.

quasar does not restrict the source of the configuration data and it is expected that multiple providers might be implemented such as XML, data-base access, Web query etc. As of writing, only one source of configuration data was implemented: XML based configuration. A diagram showing how the source is integrated into *quasar* is shown in figure 4.8.

In such a configuration source, *quasar design* is transformed into *configuration schema* (which is a dependent XSD file) at first. At this stage, only configuration related aspects of a *quasar design* get carried over to the schema, and in addition they get transformed into *configuration schema* concepts. An overview of such a design to configuration schema mapping is shown in the table 4.3.

In the next step, the XSD-C++ binding generator is run on such a configuration schema. For this thesis, the CodeSynthesis XSD-C++ binding generator[78] was used [18]. This step generates a C++ translation unit (a C++ header and source file) where C++ mapping can be found, one C++ class per each XSD element type.

Apart from generating C++ interface to given XML configuration, the generator also provides the API to load the XML file into its memory representation (in a form of Document-Object Model, see e.g.[79]) and to perform the opposite operation.

[18]The generator has multiple ways of running and their discussion is not the scope of the thesis. It is however important to state that *tree mapping* mode was chosen.

quasar design element	XSD schema counter-part
quasar class	XML element type
scalar-type config-entry	XML attribute
array-type config-entry	XML element
scalar-type cache-variable with configuration initializer	XML attribute
array-type cache-variable with configuration initializer	XML element
cache-variables with different types of initializers	none
source-variables	none
methods	none
hasObjects relation	nesting of XML elements
config restrictions (for config-entry and cache-variable)	XSD *xs:restriction* mechanism
quasar data-type	XSD data-type

Table 4.3: A simplified view on the mapping of *quasar design* and the configuration schema for XML configuration loader. A direct mapping exists between most of *quasar* data-types, both scalar and arrays, and their XSD counter-parts. However certain types such as `ByteString` type require manually written parser because there is no obvious way on how to encode such data in XML documents.

Aforementioned steps explained the act of mapping a *quasar design* to configuration schema and to generation of C++ configuration classes. Nevertheless the most crucial task of the configuration module is to actually instantiate address-space and device logic objects basing on the processed configuration data. The author chose to place such functionality in a sub-module called *configurator* which is generated from the *quasar design*. The *configurator* loads an indicated configuration source using aforementioned *configuration classes* and then, recursively from the configuration top-level entry, creates address-space objects and inserts them into the address-space. If chosen by design, device logic objects are created and relations between them and corresponding address-space objects are created, as detailed in 4.9.1.7.

Some aspects of configuration data could not be validated using the generated XSD schema. To cover those cases, *configuration validator*, another sub-module of the configuration system is generated from the *quasar design*.

4.9.1.6 Device-logic classes architecture

Device-logic classes have two very important properties in the architecture proposed by the author:

- they take the primary responsibility for the behavior of a given *quasar server*.
- from the software architecture point of view, they are where much of developer's source code resides.

Each *quasar class* might have a corresponding device-logic class; the decision is made via *quasar design*. A class without device-logic is still an useful class from a developer point of view. It might be used solely for representation of additional data in the exposed OPC UA address-space but many OPC UA service calls would simply have no effect (for example, a source-variable declared in a class which does not have device-logic would persistently return *NotImplemented* OPC UA status code).

The aforementioned term *behavior* comprises of the following aspects:

- actions which are supposed to happen after a solicited request from the OPC UA address-space This covers the case of an OPC UA client placing a request via either of the following mechanism:
  - a write invocation on a *quasar cache-variable* which has a *write delegate*,

  - a write or a read invocation on a *quasar source-variable* which is allowed to be (respectively) written or read,
  - a call invocation on a *quasar method*.

- actions invoked from inside of a *quasar server*, for instance from within of any thread constituting the server instance

- actions invoked as call-backs, typically triggered by an event in the control system and passed to the *quasar server* by an Hardware Access Layer or its equivalent.

For any of the possibilities mentioned above, it is expected from the server developer to supply source-code which implements the actions mentioned above. Such source code must fit the skeleton code which *quasar* prepares for the developer.

Device-logic classes is the only source code in the *quasar architecture* where generated code is interleaved with developer's source code (either hand-written or generated on his/her behalf).

A device-logic class is generally made of the following constituents:

- a constructor, usually called by the configuration module (see section 4.9.1.5) to instantiate given class, and a destructor
  The constructor, apart from a part which initializes constituents of the class which are intentionally hidden from the user, is also expected to be implemented by a developer with initialization of a respective part of the control system, for example by calling different functions/methods from the Hardware Access Library.

- user-written handlers which implement different verb methods (such as *set*, *get*, *call*)
  Apart from the constructor, the implementation of those methods is where much of developer's code is likely to reside.

- a reference to a parent object (generated by *quasar*).
  The parent object is defined as the object which holds the *hasObjects relation* to given child object. Such a reference is available only when precisely one *hasObjects* relation points to given object[19].

- a reference to address-space object representing given device logic object (see section 4.9.1.7).

- collections of references to children objects.

- coding easements.
  The author identified different supplementary additions to device-logic classes which, by means of automating software creation could significantly simplify human labor or make it unnecessary. A notable example is an attribute that can be specified per a *quasar config entry* called *isKey*. When the attribute is present, *quasar* generates methods which can query the hierarchy of objects to find a specific one (or a group) matching given key. Those methods can be used by a developer and therefore reduce tedious job of writing code traversing object collections. A practical example is given in section 4.9.1.9.

4.9.1.7 Relations between Address Space, Device Logic and Configuration classes

The architecture of Address Space, Device Logic and Configuration classes relating to a given *quasar class* was shown above. In this section, their relations are detailed. Such relations are relevant for many reasons:

- code from Device Logic class needs to publish (using *set* verb address-space class method) or retrieve (using *get* verb address-space class method) data, therefore a reference (or a pointer) to the corresponding address-space class must be known.

[19] *quasar design* permits multiple *hasObjects relations* to originate from different *quasar classes* to the same destination, in such a case it the type of parent reference could not be uniquely determined and is therefore not available.)

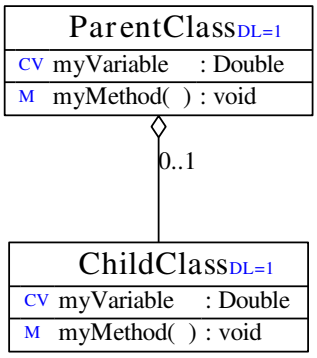


Figure 4.9: A sample *quasar design diagram* to illustrate relations between classes in the Address Space, Device Logic and Configuration domains. The UML class diagram illustrating classes resulting from this design diagram and their relations is shown in figure 4.10.

- generated code in the address-space class must pass a client request using a delegate to Device Logic class, therefore address-space class must know a reference (a pointer) to the corresponding Device Logic class.
- the constructor of the address-space class needs to know certain configuration data (e.g. the name of a class to establish its OPC UA address) so it needs a reference to the configuration class.
- the constructor of the device class class needs to know certain configuration data. Such a constructor in *quasar servers* is written by a developer. A reference to the configuration class might be needed in the constructor in order to initialize something (a part of the control system) using data supplied via configuration.

Let us assume an example *quasar design* as in figure 4.9. Then, from within device-logic of `ParentClass` (e.g. from the body of the verb method `callMyMethod`) one can:

- access the address-space to, for instance, publish a value:
  `getAddressSpaceLink()->setMyVariable( 3.1415, OpcUa_Good);`
- obtain collection of all `ChildClass` objects instantiated as children of given `ParentClass` object:
  `childclasss();` [20]

In addition, from the device-logic of `ChildClass`, one can obtain the `ParentClass` by the call to `getParent()` method. All of those methods are generated by *quasar* per-design.

4.9.1.8 Finding device-logic and address-space objects

The relations between classes explained in section 4.9.1.7 are of relative nature, that is: an address-space object of *quasar class* X can find its device-logic object and vice-verse, but it does not cover the case of *any* executed code inside a *quasar server* being able to find *any* requested object of chosen *quasar class*. Such a necessity is often required, for example when an event in the control system propagates to multiple address-space or device-logic objects which are not "close" to each other in the instantiated object hiearchy.

The problem in question can be independently solved in any *quasar server* from the perspective of device-logic (see section 4.9.1.9) as well as from the perspective of address-space (see section 4.9.1.10).

4.9.1.9 Finding by device-logic

The first solution relies on links between objects which are established between device logic objects, such as those available by `getParent()` or children reference access methods, as detailed in section

[20] The additional "s" is an effect of plural flexer which does not recognize that "es" would be more correct in the sense of English grammar.

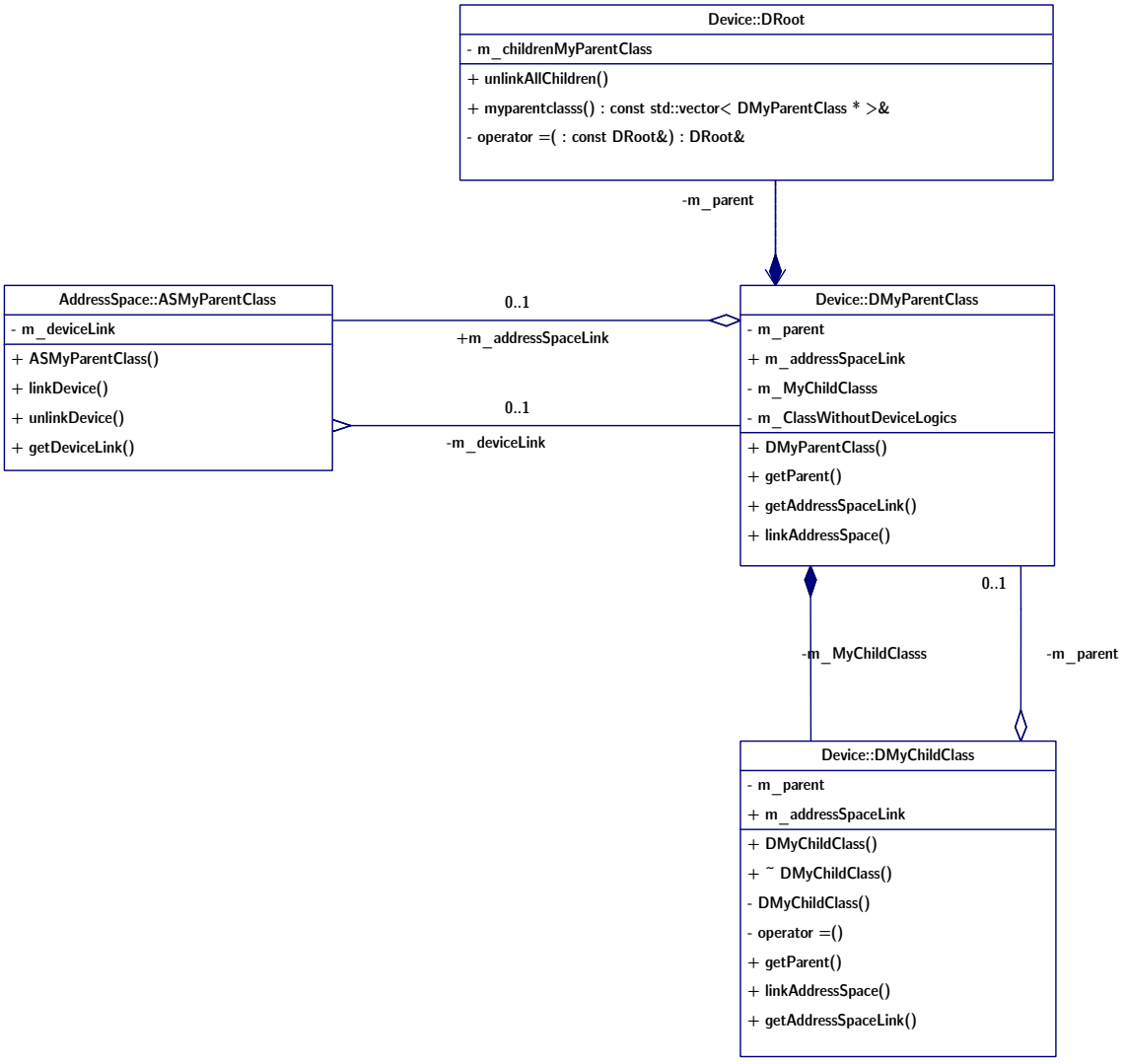


Figure 4.10: An UML class diagram of example address-space, device-logic and configuration classes described in section 4.9.1.7.

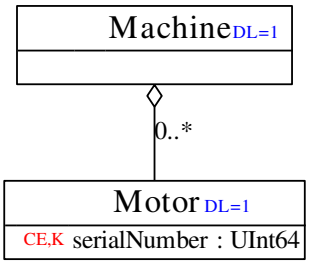


Figure 4.11: A sample *quasar design diagram* to illustrate simplification resulting from *isKey* feature. A tag *K* (seen after *CE*) signifies that the *quasar config entry* is declared as *key*.

4.9.1.7. In every *quasar server instance* there is a special singleton[21] class called *DeviceRoot*, which is a root of a tree formed by instantiated objects. *DeviceRoot* has the same interface to access its children like any other device logic object, e.g. if `Sensor` *quasar class* is referenced by *hasObjects relation* from *quasar design root* then the *DeviceRoot* class has a method called `sensors()` which returns all configured device-logic objects of `Sensor` *quasar class*. Therefore, a part of a program written by a developer can always start the traversal from the *DeviceRoot* and then (possibly recurrently) descend the tree towards its leaves to reach all device-logic objects of given *quasar server instance*.

The *isKey* feature mentioned in section 4.9.1.6 opens a way to traverse a path from *DeviceRoot* down constraining it only to objects where value matches in configuration fields chosen by the key. Such a feature proposed by the author can significantly simplify the traversal by reducing amount of *boilerplate* code that a developer would otherwise have to write.

There are two implications of declaring a *config entry* or a *cache variable* as *isKey*:

- the value (which is initialized from the configuration data) becomes constrained to be unique within all objects that belong to the same parent. When the default XML-based configuration engine is used, such a constraint is achieved by XSD keys.
- additional code is generated for the developer in the Device classes. The code lets efficiently perform queries based on key match.

An example is demonstrated by an excerpt *quasar design diagram* in figure 4.11.

Listing 4.3: Source code exemplifying simplification achieved by *isKey* feature, as per *design diagram* from the figure 4.11. The developer does not have to write any source code which performs searching for items by the value match.

```
1  void DMachine::start ()
2  {
3    getMotorBySerialNumber(123456)->start();
4  }
```

With a *quasar design* like in figure 4.11, the code shown in the listing 4.3 can be written in the Device Logic of `Machine` class. The function `getMotorBySerialNumber` was generated by *quasar* to reduce boiler-plate coding. It returns a device-logic object (here, an object of `Motor` class, as per design). A chain of consecutive calls can be done enabling convenient traversal from the Device Root down to the chosen leaf.

The key messages to take away is that accessing random objects via Device Logic hierarchy is possible and efficient for the developer as long as all classes on the traversal path have the declaration of Device Logic. For a selective traversal the *isKey* feature enables to efficiently code the query, even as an one-line function.

4.9.1.10 Unified address-space queries

In section 4.9.1.8 it was shown how to access an object via Device Logic domain. Another method bases on unified address-space queries. The method is significantly different from device-logic traversal explained in the previous paragraph. An advantage is that it can be used to query any

[21] A singleton class by definition has one instance.

address-space object[22] that was inserted into the address-space by source-code generated by *quasar* [23]. A disadvantage is that the query is based on an OPC UA address. While in many components of a *quasar server* it is very natural to use OPC UA addresses, in many other components - like Device Logic and all layers lower than it - the OPC UA address might simply not be available.

The address-space module of *quasar* delivers functions which can obtain given address-space object basing on a passed OPC UA address or a regular expression defined on an address. The functions are based on C++ templates parameterized by address-space class therefore they can be considered type-safe for a programmer.

An example of an unified address-space query is presented in the listing 4.4.

Listing 4.4: An example of an unified address-space query. The example comes from *StandardMeta-Data* module used in every *quasar*-based server. The source code obtains all (regular expression `.*` matches any string-based node address) address-space classes of `StandardMetaData` *quasar class* and stores them in the `objects` variable. `nm` is the address-space node manager.

```
1 std::vector< AddressSpace::ASStandardMetaData * > objects;
2 std::string pattern (".*");
3 AddressSpace::findAllByPattern<AddressSpace::ASStandardMetaData> (
4   nm,
5   nm->getNode(UaNodeId(OpcUaId_ObjectsFolder, 0)),
6   OpcUa_NodeClass_Object,
7   pattern,
8   objects);
```

4.9.1.11 Calculated variables

A *calculated variable* is an OPC UA variable residing in the OPC UA address-space whose value is calculated as a function of one or many OPC UA variables.

The feature of *calculated variables* is heavily inspired by the *CANopen OPC UA server project*[17] [24] however the design and implementation (and its generic application via *quasar*) is entirely the work of the author.

In the context of control systems software, *calculated variables* are used to solve the following problems:

- quantity type conversion
  For example, given OPC UA server delivers a measurement of period (of a periodic process) but it is more favorable for a final user to have it expressed as frequency.

- unit conversion
  For example, given OPC UA server delivers a valve pressure in millimeters of mercury (commonly abbreviated as *mm Hg*) but it is favorable to have it in more standard kilo-pascals (kPa).

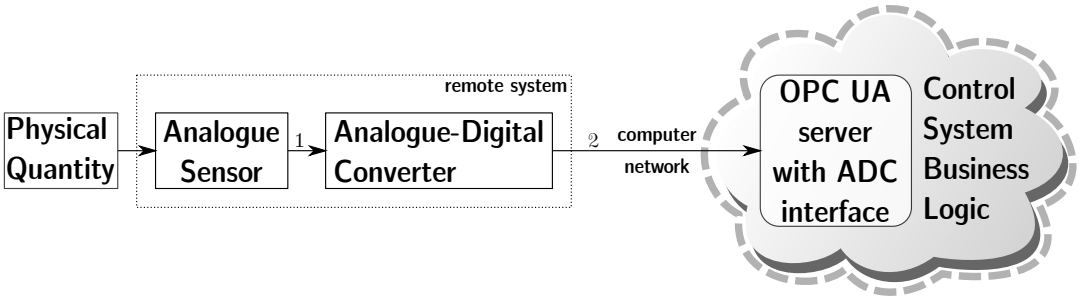
- calibrated value instead of raw value
  For example, a final user (or a control engineer, etc.) requires to apply a calibration fix to an uncalibrated value, for example by a simple linear gain and offset fix.

- measurement concretization
  Often, the data flow between a physical sensor and the representation of its measurement in the control system software is composed of multiple stages of processing. An example is given in the following picture:

[22] And, from there, using device-link, also access its device-logic part.

[23] In every OPC UA server instance there is a significant amount of information residing in the address-space which does not originate from *quasar*. Such contents usually is so called built-in OPC UA address-space as well as information inserted by given OPC UA toolkit/protocol-stack used for instance for debugging.

[24] The author thanks his colleague Viatcheslav Filimonov for inspiration.



In this situation, the data flow is composed of multiple stages:

- a physical quantity (temperature, pressure, etc.) of an element of the control system impacts the sensor,
- the sensor converts the physical quantity to electric signal, typically voltage (connection "1" in the figure),
- the analogue-digital converter converts from the electric signal into its digital representation (connection "2" in the figure) and sends over some type of network towards the control system
- an OPC UA server implements an interface to the digital representation and can therefore distribute the information as high-level piece of information over OPC UA

In the example above, often the Analogue-Digital Converter is a generic part, capable of interfacing a very wide range of different analogue signals. The transfer function of many (and certainly of most of practically used) analogue sensors can be given as an analytic expression. Therefore, for the reasons of economy and for practicality, it makes sense for the OPC UA server to be a generic ADC interface and use the *calculated variable* to describe the transfer function, thus provide an OPC UA variable representing concrete physical quantity (e.g. temperature, pressure...) rather than generic value (voltage).

The overall design of *calculated variables* feature is summarized as:

- each instantiated cache-variable is a subclass of `ChangeNotifyingVariable`, i.e. it can emit a change-notification.
- if such a variable is of numerical data-type (in the sense of its definition in the *quasar design*) then its reference is added to a list of potential sources of data for *calculated variables*. The sources are generally instances of `ParserVariable` class and the analytical expression parser knows how to find give `ParserVariable` by name when the formula is evaluated.
- when an instance of `CalculatedVariable` is approached while opening configuration, then a new analytical expression parser instance is created and new calculated variable is inserted in the OPC UA address-space. In addition, every created calculated variable is also added to the list of sources (e.g. it has a representation as `ParserVariable`) so it opens the potential of chaining multiple calculated variables.

The detailed description of implementation of calculated variables is available in [80].

The calculated variables feature was designed to be parser-agnostic. Current implementation uses *muParser*[81] as the analytical expression parser due to its favorable properties of high-quality implementation, performance and friendly build time.

In the evaluation, the implementation of calculated variables increased the overall CPU consumption related to information exchange by cache-variables by 1.4%. In the author conclusion it turned out to be a fair price to pay given the huge space of possibilities that calculated variables open.

4.9.1.12 Test and development easements

Several years of author's experience of application of *quasar* in designing and building of different parts of control systems middle-ware showed that there exists another domain of potential savings in the cost of software creation: the investment in "early" test and development tools. The term "early" refers to a phase in the software project lifetime *significantly* earlier and often just after the development activities started.

The justification for such a statement is that a successful software project in the domain of large control systems software often depends on achievement of different goals including:

- successful identification of requirements and constraints,
- successful creation of (a part of) control system model (requires the previous step),
- successful identification (and sometimes also design and implementation) of Hardware Access Libraries (or any other dependent pieces of software),
- successful combination of aforementioned model with the external libraries, which in *quasar* typically happens by implementation of Device Logic code,
- successful testing of obtained software with real control system hardware,
- successful integration of the software in the control system *as a whole*.

The particular challenge that accompanies the problem is that the goals listed above are not only interconnected but also on the critical path [25], for example: unavailability of the final specification of hardware might impact requirements identification or unavailability of the hardware might postpone the testing phase. Therefore, in author's point of view, any easement which could disentangle this problem and break it into smaller and independent pieces (i.e. divide-and-conquer approach) is worth consideration.

Therefore the *quasar ecosystem* offers (as a result of requirement analysis) different tools which implement aforementioned viewpoint. For instance, a tool [26] is provided which creates per-*quasar design* source code which is able to publish different data via the address-space (and according to its information model) *without* the presence of Device Logic and/or Hardware Access Libraries. Such a tool can be used in the following scenarios:

- to provide *fake* or *partially fake* data in the address-space before the real data can be published (for example, because the corresponding control system hardware was not manufactured yet) and therefore enable testing of whole system integration
- to test the throughput and performance of a *quasar-based* OPC UA server

Another tool is available [27] which can profit from *quasar design* to perform penetrative testing of various elements of OPC UA address-space, in the most extreme case the tool can invoke all supported OPC UA service calls on all elements of the address-space. Such a tool is able to force invocation of potentially whole code residing in given *quasar server instance* and therefore to identify software defects (e.g. resulting in software instability) well before they get exposed in the late testing or, even worse, in the production environment.

4.9.1.13 End-user experience

The author was examining end-user experience when different OPC UA software products created with *quasar* were being deployed in production systems. Two aspects were identified as the least user friendly for end-users, namely:

[25] A critical path in project scheduling is often defined as a path connecting tasks of a project that any delay in the tasks on the path will delay the completion of the project as a whole.

[26] For an interested reader, the tool can be generated by invoking *quasar* with the argument "generate honky_tonk".

[27] The tool is called OPC UA penetrator.

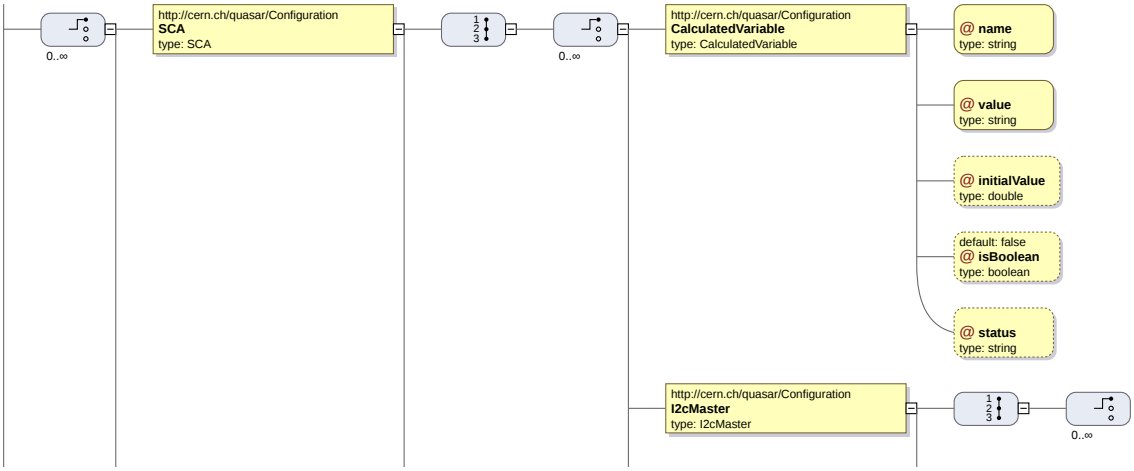


Figure 4.12: A part of an example configuration schema of one of software projects built using *quasar* - The SCA OPC UA server (see section 5.4) - visualized using the XsdVi tool [82]. In the author's opinion such a representation of the configuration schema, even though very mature and exact in technical terms, is not attractive for distribution to end-users because of its complexity.

- creating *quasar server* configuration data,
- performing OPC UA integration into SCADA systems.

Both aspects are on the critical path of software creation and integration process because typically the OPC UA integration follows the successful start-up of a server instance, for which configuration is obligatory. Therefore both aspects impact economical aspect of the software creation process described in this thesis because of amount of effort spent by end-users of product deployment as well as on end-user support costs.

Preparation of configuration data boils down to creation of an XML file[28] which describes instances of *quasar classes* of given *quasar server* along with mandatory or optional configuration data. The configuration file must be created in accordance with the schema which was detailed in section 4.9.1.5.

The author paid a lot of attention such that the configuration schema is augmented with information which makes end-user deployment as effort optimized as possible. In fact the chosen configuration schema format, XSD, enables to embed a lot of end-user hints such as XSD annotations (see [63], chapter 2.2.6). Along with different restrictions which XSD can store (and which are used by *quasar*) it presents a solution already.

In addition, the author provided means to supplement the end-user experience with partially generated documentation. Each *quasar*-based server can be equipped at no-cost with generated documentation of the address-space as well as the documentation of the configuration schema. Both documents profit not only from the information model stored in the *quasar design* but also from XML annotations that can be attached to any XML document. Such annotations are then transferred to the generated user documentation so the end-user enjoys simplified view of address-space and configuration without being able to read either *quasar designs* or any OPC UA information models.

4.9.2 OPC UA clients

In the writing so far, the usage of *quasar architecture* related mostly to creation of OPC UA servers. Nevertheless there are two features of the *quasar architecture* which stood out as particularly attractive

[28] Assuming that the XML configuration engine is used.

to encourage the author to gain additional economic advantage by creation of OPC UA clients in addition to the OPC UA servers:

- availability of a partial model of control system (in form of *quasar design*) which is sufficiently detailed and at right abstraction level,
- availability of the OPC UA information model (which, in terms of information, is a derivative of *quasar design*) via the OPC UA address-space of any *quasar server*.

The first of the features stands out during design and development time while the second one stands out during run-time. Therefore it occurred naturally to the author that the first feature could be attractive for creation of OPC UA clients for a compiled programming language, like C++ or Java. OPC UA clients generated in this approach are detailed in section 4.9.2.1.

On the other hand the second feature seemed most attractive for making OPC UA clients for a programming language in which a lot can be done dynamically at run-time, such as for Python. Such an approach is detailed in 4.9.2.2.

4.9.2.1 OPC UA clients in C++ using UaObjects for C++

Thanks to high-level modeling that *quasar* offers, the author observed that it should be possible to create (e.g. generate) a class in the C++[29] programming language which would closely follow given *quasar class*. Such a class would not show any relation to the OPC UA standard from the interface point of view. The class could represent a type of a control system object and expose its model (for example, methods) as a real programming language class interface. At the same time the class would hide that the object which it takes after is possibly deployed remotely, and there is OPC UA interfacing (and perhaps a complex network infrastructure) in between. It would be an example of *location transparency* which is a key feature of many widespread middle-ware solutions like CORBA[18] [30]. The author published a more detailed study in [3].

In software engineering (and particularly in the domain of *design patterns*), a class which provides or maintains access to another class (keeping the same interface) is often called a proxy class or a surrogate class [76].

The implementation of such a class (which normally is hidden from the eyes of its user) is where the actual middle-ware work is taking place. For instance, a straightforward implementation for OPC UA would imply that the act of using the class would:

- initialize the remote connection
- prepare invocation of the transaction, for example convert given input arguments to OPC UA data representation
- perform relevant transaction (e.g. an invocation of write, read, call service)
- if the transaction was successful, convert return values (*OPC UA output arguments*) to representation suitable to the programming language
- close the connection

A similar but more advanced solution was designed and implemented in the *quasar software ecosystem* by the author of this thesis under the name of UaObjects (abbreviated UaO) and open-sourced[83] in 2017.

In the UaObjects approach, a developer can request a C++ proxy class per each *quasar class* from a chosen *quasar design*. In addition tools exist which transform the whole *quasar design* (containing a potentially large number of *quasar classes*) into a set (e.g. a library) of proxy classes.

The relation of the most relevant elements of *quasar class* to the C++ proxy class are outlined in the table 4.4.

The basic architecture of a C++ class generated using the UaObjects approach is the following:

[29] Statements from this section apply equally well to the Java programming language and other languages sharing similar characteristics.

[30] The relation to CORBA is done only to exemplify *location transparency* feature.

quasar class design element	mapping to UaObjects proxy class
cache-variable	read verb method and write verb method (if given variable supports writing)
source-variable	read verb method and write verb method (if given variable supports writing)
method	call verb method
config-entry	not relevant for the proxy class
any device-logic aspects	not relevant for the proxy class
any *hasObjects* relation	not relevant for the proxy class

Table 4.4: The mapping of *quasar design* elements into elements of a class generated by UaObjects.

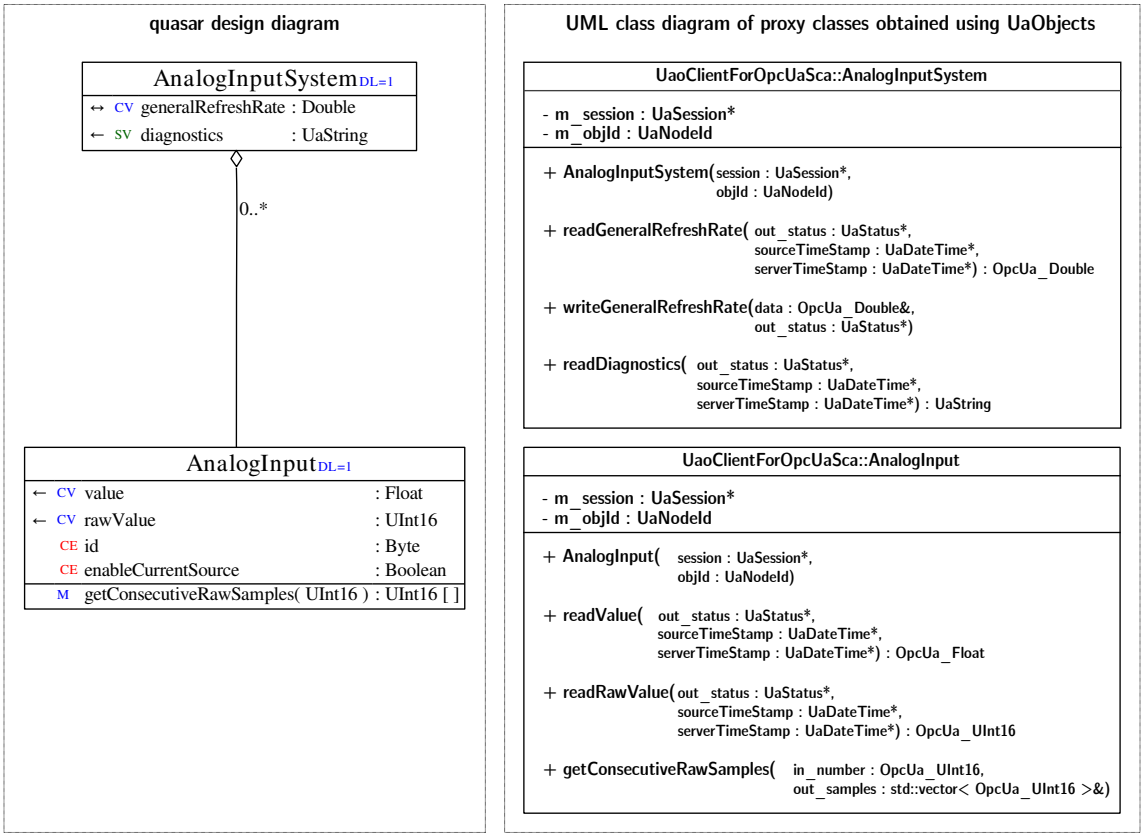


Figure 4.13: Side-by-side comparison of (a part of) *quasar diagram* and resulting proxy classes obtained using the UaObjects in the C++ programming language. The example comes from the SCA OPC UA server further detailed in section 5.4.

- an object of the class holds internal data, namely:
  - a reference to a session[31] which will be used to perform given operation,
  - an address of a node in the OPC UA address-space to which this operations applies.
- a constructor, which serves only to gather aforementioned internal data. The act of instantiating the proxy class does not trigger any OPC UA activity.
- an empty destructor.
- verb methods, zero, one or two per every variable (depending on allowed information flow direction) and one per every method. Refer to the table 4.4 for detailed mapping.

A client generated using *UaObjects for C++* will most likely become a part of another application, for example as depicted in section 5.5. Therefore an important problem to solve is interfacing between different representations of data, e.g. those in OPC UA and those preferred in C++. The author chose a convention in which the proxy class interface exposed by UaObjects uses as much of standard C++ type system as possible. The motivation for such a decision came from the need to minimize the learning curve for software developers as well as for other benefits of adhering to standards. Only in certain cases the OPC UA data typing was preserved, for example *UaVariant* data type is preserved in case it is used in the *quasar design*.

Another important aspect is handling of failures. The following failure modes were identified:

- failures in the service calls - for example, as a result of a timeout related to network failure. In such a failure mode it is assumed that the OPC UA service call failed as a whole therefore no attempt is done to even attempt partial interpretation of results.
- bad returned status-code (despite successful service call)
  Such a situation would occur when server consciously finished given service call with a status code different from good. It could for instance signify that handling of given request by the server was successfully attempted but, for instance, a call to hardware access layer (HAL) failed.
- data conversion faults
  Such a situation could take place if the data sent in either direction was not convertible to a format expected by the receiver, for example a method call would expect an argument of integer data type, but a non-integer number was transmitted on-the-wire (a non-integer can not be converted to an integer without information loss). Pragmatically, such a situation is excluded in *quasar* as long as *quasar design* coherency is maintained. It could happen though that over long lifespan a change in *quasar design* is implemented, therefore both server and client should be regenerated and consequently deployed in the production environment, but by human mistake one of them is forgotten. Then such a failure scenario is expected and should be protected against.

The author chose the following convention regarding *the way* in which failures are propagated to the software using (i.e. calling) UaObjects:

- *read* verb methods feature a reference argument [32] (called `out_status`) which by default is *null* signalizing that a potential failure should be signalized by *exceptions*. Only when a programmer explicitly supplies a referenced `out_status`, then instead of *exceptions* the status code will be returned. In such a case the responsibility to check the return value falls onto the developer.
- for *write* verb method similarly `out_status` reference is available; if empty then a fault is signalized by *exceptions*.

[31] A session corresponds to one connection between an OPC UA client and an OPC UA server.

[32] References as arguments are often used to obtain *output argument semantics*, that is: another way - different from regular return values - for a function to return results.

- for *call* verb method the faults from invoking methods are always returned as exceptions.

The name-space of the generated client code can be parameterized externally according to the choice of an user. This might help to let use multiple different *UaObjects*-based clients as well as ease maintain clarity and organization of the project.

4.9.2.2 UaObjects for Python

On contrary to *UaObjects for C++* detailed in section 4.9.2.1 the requirements, constraints and wishes driving *UaObjects for Python* are substantially different. The reason for such differences come predominantly from fundamental differences between the C++ programming language and the Python programming language:

- C++ is a strongly typed programming language[61] which means that each variable needs to be assigned a type it adheres to.
- on contrary, Python is dynamically-typed and it evaluates object access at run-time.

Therefore, not only the approach of mapping of *quasar class* to *Python class* is entirely different but also the space of possible features is enlarged.

First of all, many attractive features of object mappings in Python come from the following special methods which are defined by the core of object orientation of Python[84]:

- `__getattr__(self, name)`
  A built-in method of every Python object to get value from a field of an object.
- `__setattr__(self, name, value)`
  A built-in method of every Python object to set value to a field of an object.
- `__call__(self, *args)`
  A built-in method of every Python object to call a method of an object.

For instance, if a class `MyClass` defines the `__getattr__` method, then one can provide an implementation which will be called each time the following code would be invoked: `objectOfMyClass.fieldName` and in such case the `__getattr__` will be called with the value of `name` as `'fieldName'`.

One can imagine creating classes which profit from this feature for elegant mapping of OPC UA address-space into Python.

The author proposed [3] a library for Python called *UaObjects for Python*[85]. The library is based around the following classes:

- `Session` - represents an open connection to OPC UA server.
- `Node` - represents a node in the OPC UA address-space, stores OPC UA node address.
- `Object` - a specialization of Node class representing an object in the address-space.
- `Variable` - a specialization of Node class representing a variable in the address-space.
- `Method` - a specialization of Node class representing a method.

The library uses PyUaf[86] as an OPC UA client for Python.

A practical example is shown in the listing 4.5.

Listing 4.5: Example of UaObjects for Python. `'anObject'` is the string identifier of a *quasar*-built object residing in the address-space. `ns_index` is the OPC UA name-space index (typically 2 in *quasar*-built servers).

```
1  session = uao.Session(client, server_uri)
2  obj = session.get_object('anObject', ns_index)
```

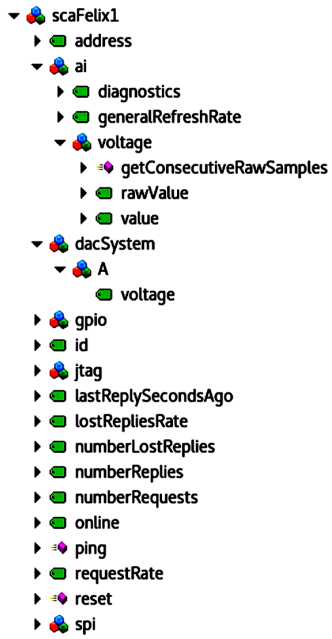


Figure 4.14: A part of a screen-shot from the UaExpert[88] OPC UA client partially showing OPC UA address-space including an object `scaFelix1` and a number of its children. An example UaObjects code interacting with the address-space is shown in the listing 4.6.

The programmer's entry point is by instantiation of a `Session`, supplying PyUaf client handle and the Uniform Resource Identifier (URI) of the server, as shown in line 1. Then the object of `Session` class can be requested to perform the object mapping from OPC UA to Python (line 2). The mapping is performed recursively by walking the address-space graph using the *Depth-First-Search* graph algorithm[87]. As the effect, the whole hierarchy of descendant objects becomes accessible to the user's code in Python. Thanks to this, the hierarchy of objects can be walked like it were nested classes with no further calls to UaObjects functions.

A more concrete example, using the GBT-SCA OPC UA server (see section 5.4) is shown in the listing 4.6. An example OPC UA address-space is shown in figure 4.14.

Listing 4.6: Example of UaObjects for Python in interaction with a real OPC UA server whose address-space is illustrated in figure 4.14.

```
1  session = uao.Session(dcs_opc_ua.client, dcs_opc_ua.server_uri)
2  scaFelix1 = session.get_object('scaFelix1',2)
3  pong = scaFelix1.ping()
4  scaFelix1.ai.voltage.value.value
5  scaFelix1.dacSystem.A.voltage = Float(0.75)
```

In the listing 4.6:

- line 2 demonstrates how a proxy object (made by UaObject) is obtained. The object is present in the OPC UA address-space in the name-space 2 (2^{nd} argument) using the string address `scaFelix1` (1^{st} argument).
- line 3 demonstrates objectified method call.
- line 4 demonstrates reading a variable, i.e. the value of the expression corresponds to a result from OPC UA read service call. A potential fault (e.g. network failure) is thrown as an exception.
- line 5 demonstrates writing a variable, i.e. the value 0.75 represented as OPC UA `Float` data-type will be written.

quasar design element	mapping to WinCC OA
class	Data Point Type
cache-variable	Data Point Type Element
source-variable	Data Point Type Element
method	*no support*

Table 4.5: The mapping of *quasar design* elements into concepts of the WinCC OA architecture.

4.9.3 SCADA integration

SCADA integration of a *quasar server* is an act of interconnecting the server with a control system implemented using given SCADA platform, using the OPC UA protocol, such that both parts form a combined and unified control system. All concepts and examples studied in this section refer to the Siemens WinCC OA [40] as an example of a professional-grade SCADA system. An introduction to (parts of) control systems implemented in the platform is given in section 4.3.2.2.

The following aspects of integration work were researched in this thesis:

- creation of WinCC OA Data Point Types (and their Elements) from a *quasar design*,
- instantiation of WinCC OA Data Points following the configuration file of a *quasar server instance* (possibly supplemented by a corresponding *quasar design* if necessary),
- assignment of WinCC OA peripheral addresses which bind given Data Point Elements to particular sources/sinks of information in an instance of *quasar*-based OPC UA server.

A Data Point Type in the WinCC OA architecture has similarity to classes in programming languages:

> The user may configure an appropriate data point type for each real device type (drive, valve, stirring unit, controller, intrusion sensor, ...). A data point for each real device is derived from this data point type (kind of template). In the object-oriented software engineering you would call the data point type a "class" and the representation of an individual device (the data point) an instance. [59]

A Data Point Type also functionally resembles a *quasar class*[33] Therefore the author attempted to benefit from possible analogies, and the proposed mapping is listed in the table 4.5.

In order to obtain automated Data Point Type (DPT) creation (including children DPTEs), a transform was created to translate *quasar design* into a control script [34] which once run from a WinCC OA control manager [35] is capable of creating respective Data Point Types and Data Point Type Elements. The author considered an alternative way to create Data Point Types by means of creating a transform able to generate so called ASCII representation [36] of Data Point Types which could then be loaded by the ASCII manager of WinCC OA software package. However the former approach, by means of CTL script, was considered superior because of achieved portability (the ASCII format in case of WinCC OA is not very well documented and it is considered “internal” representation).

The two other remaining deliverables - creation of Data Points themselves and assignment of Peripheral Addresses - require both *quasar design* and a configuration file of given *quasar server instance*. Due to limited support of XML in WinCC OA the author implemented generation of a schema-aware XML parser in the CTL programming language from respective *quasar design*. The

[33] The author acknowledges his colleague, Benjamin Farnham for pointing out this analogy.

[34] Control scripting (often abbreviated CTL) is a built-in programming language of the WinCC OA platform [59]. It resembles the C programming language and it features certain extensions that are natural to the WinCC OA architecture, like built-in support for information exchange via Data Points.

[35] A WinCC OA control manager is a computer program which is capable of executing CTL scripts.

[36] The ASCII manager is an integrated part of the WinCC OA software which can store and restore Data Point Types and Data Points (including Peripheral Addresses) as an ASCII text files, thus the name ASCII representation. The ASCII representation is analogous to SQL dumps which are commonly used to in relational databases practice.

generated parser is able to traverse given *quasar server instance* configuration file and once an instance of an object is approached it is able to:

- create Data Points, according to previously created Data Point Types, naming them accordingly to their position in hierarchy,
- assign OPC UA addresses, predicting them from their position in hierarchy.

The complete framework for the WinCC OA platform was released by the author in 2018 under the name of *fwQuasar*. Its application on a real use-case is visualized in figure 4.15.

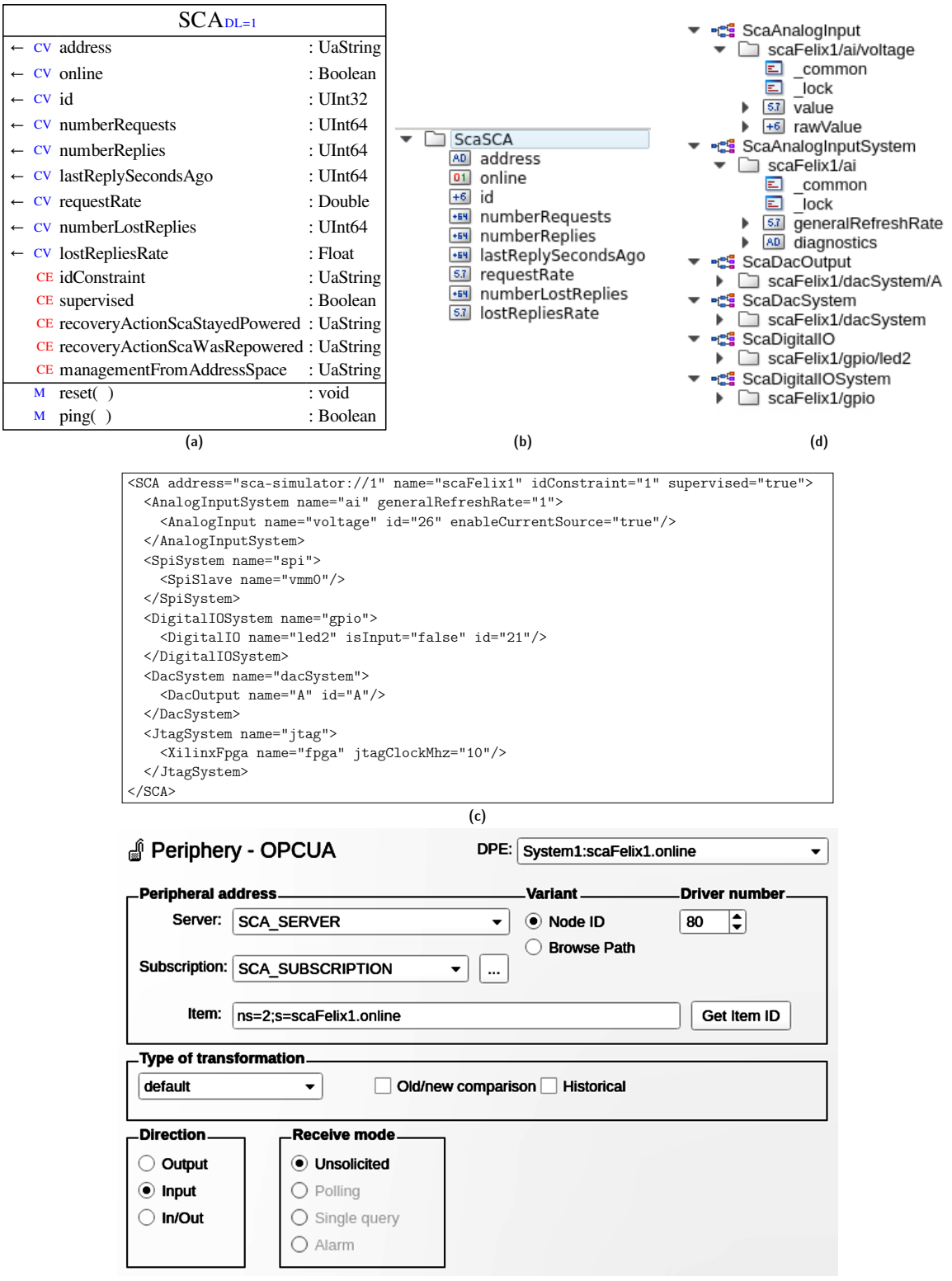


Figure 4.15: Different phases of SCADA integration of a specimen class *SCA* from the project detailed in section 5.4. (a) shows a part of *quasar design* focused on the class. (b) shows respective Data Point Type in the WinCC OA system derived using the discussed solution. The Data Point Type was expanded and therefore it also shows the Data Point Type Elements. The icons represent the data type. (c) shows an excerpt of a sample config file used to produce (d) and (e). (d) shows DPs instantiated using (c). (e) shows a Peripheral Address derived using (c).

Chapter 5

Applications and evaluation

quasar was conceived as a software ecosystem capable of supporting efficient creation of OPC UA software. Among its key requirements stand its flexibility to fit into variety of applications typical to heterogeneous control systems of large scale.

To demonstrate the flexibility, in this chapter, a number of real-life applications of *quasar* are presented. The applications are given with software measurements which can be used to estimate the effort saving of software creation which *quasar* brings. The order of presented examples of OPC UA servers (sections 5.2, 5.3, 5.4) is determined by their perceived conceptual complexity.

In addition to the examples of OPC UA servers, to further outline the support of diverse systems, certain less typical applications are detailed, for example *quasar*-based OPC UA servers integrated into software projects made in mixed programming languages.

5.1 Description of evaluation methodology

There exists a scientific domain called Software Measurement (often called also Software Metrics) usually classified as a part of Software Engineering. An overview of the Software Measurement domain is available in e.g. [89, 47]. Software measurement delivers methods to measure various aspects of software creation:

- process (understood as the act of software development)
  Software development process can be characterized (measured) by effort (e.g. man-months), duration (days, months, ...), cost etc. These measures are to some extent subjective, e.g. man-months is highly dependent on skills and experience of developers.

- products (the output of the software development process)
  Products include source code, executable programs, libraries, deployment recipes, associated documentation etc. Products can be characterized by complexity (size of source code, e.g. in lines of code – LoCs, or size of object code), quality (number of defects found in given time after end-user deployment), requirements (amount of processing power or memory).

- resources (the input of the software development process)
  The inputs are usually requirements definitions and constraints. Resources seem to be highly dependent on chosen software development approach thus they are the most vague term..

Therefore different methods might be used to measure the automation of the software development process.

In the environment in which the author implemented his idea, it was possible to measure the product only. Therefore careful application of measurement by ELoCs (executable lines of code) was chosen to evaluate the method and products discussed in the thesis.

The Lines-of-code (LoC) metric, while not ideal, is still a tool of major usability to assess size of source code in *quasar*-based projects:

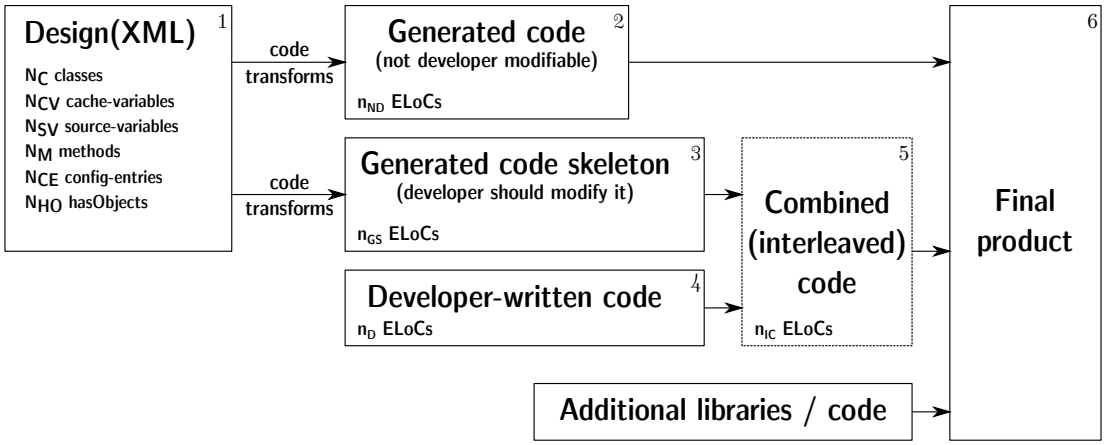


Figure 5.1: Typical work-flow of *quasar* with points of measure. The metric for Design is based on numbers of different design elements (result set: N_C, N_{CV}, N_{SV}, N_M, N_{CE}, N_{HO}). The metric for transformed source code is based on number of design-dependent executable lines of code (ELoC) (result set: non-developer code n_{ND}, generated skeletons n_{GS}, developer code n_D and combined (interleaved) code n_{IC}).

- to assess overall size, when used in absolute form,
- to compute automation gain, when used in relative form: for example comparing the source code size before and after applying the automation.

Nevertheless such a metric must be applied carefully, with context knowledge about the measured source code and observations.

For its application in *quasar-based* projects, the following key decisions were made by the author:

- only design-dependent source code was taken into account, comprising of:
  - address-space classes[1] – headers and body – of the AddressSpace module,
  - device-logic classes[2] – headers and body – of the Device Logic module,
  - configuration schema – the XSD document – of the Configuration module.
- for device-logic classes which are made of partially generated code, which is then made complete by a developer, both parts were measured separately.

The author assumed that the size of design-dependent source code (as listed above, compared of results sets of multiple variables) is the key measurement deliverable. From the measurements and the information obtained by analyzing *quasar design* (such as number of classes), various automation statistics can be obtained:

- the ratio of automation (comparing the total size of design-dependent source code to the non-generated design-dependent source code)
- the ratio of code-generation (comparing how many lines of design-dependent code is generated per different elements of design)

The figure 5.1 shows how different stages of *quasar* work-flow correspond to evaluation of software automation:

[1] See section 4.9.1.4 for reference.
[2] See section 4.9.1.6 for reference.

- in the step 1 a *quasar design* is created. The content of the *quasar design*, and particularly the number of different elements like *quasar classes*, *quasar cache-variables* etc. is likely to impact the measured size because of their expected dependency on the *quasar design*. The counting of such elements in the *quasar design* is a straightforward task.

- in the step 2, parts of design-dependent source-code are generated, excluding user-modifiable code. The step involves address-space classes, base classes of device-logic and the configuration schema. The size measurement is not difficult: a count of non-empty non-commented generated lines of code need to be derived.

- in the step 3, the skeleton code is generated (the device-logic classes). However, such skeleton code is usually created once in project lifetime per each *quasar class* and then it undergoes manual edits in which a developer adds/changes/removes the implementation. Such state is marked as 5 in the picture. The state is actually a manual combination of generated code (3) and developer code (4).

It is straightforward to obtain the measure from step 5 by counting ELoCs, however in order to relate it to the amount of work done by the programmer, it is deserved to also know the number of ELoCs relating to 4 in the figure.

If the following assumptions hold:

- developers do not change or remove lines of code generated by *quasar* (which is an official recommendation of the *quasar architecture*),

- developers add their implementation to the generated code,

- the device-logic is kept maintained with regard to the used version of *quasar*[3],

then:

- one can generate the code skeleton (i.e. step 3) à nouveau, as a temporary output file, just to measure its ELoCs. The obtained ELoCs would correspond to n_{GS} in the figure.

- then one can subtract from the ELoCs of inter-leaved (combined) code - n_{IC} the size of generated stubs n_{GS} to calculate the size of design-dependent developer code - n_D.

Therefore, the evaluation algorithm can be summarized as:

- measure non-developer design-dependent code and store it as n_{ND},

- measure inter-leaved (combined) design-dependent code and store it as n_{IC},

- generate stubs as temporary output, measure it and store as n_{GS},

- obtain developer dependent code as $n_D = n_{IC} - n_{GS}$,

- to obtain the automation ratio, compare the ELoCs *in case quasar was not used* - which is $n_{ND} + n_{IC}$, to ELoCs when *quasar is used*: n_D.

Therefore the final automation ratio formula is given as:

$$A.R. = \frac{n_{ND} + n_{IC}}{n_D} = \frac{n_{ND} + n_{IC}}{n_{IC} - n_{GS}}$$

[3]Code generation templates might change among different versions, therefore the coherency should be maintained to obtain valid results.

5.1.1 The tool to evaluate a server

The author made a tool (a computer program) to measure given individual *quasar-based* OPC UA server and obtain statistics from it. The tool was named *QuasarMetrics*. It was open-sourced and is available as [90].

Overall, the tool:

- can process the *quasar design* of a *quasar server* in which it was deployed,
- is capable of deriving all figures listed in section 5.1 such as n_{ND}, n_{IC}, n_{GS}, n_D as well as the final A.R. (automation ratio),
- can count lines-of-code as ELoCs (the default) or LoCs (on request),
- can output a *Python pickle* with results, therefore it can be easily integrated into another application willing to use the data.

The processing of source files in the C++ programming language to obtain the ELoC figure turned out not to be an easy task. For example, taking into account the complexity of the C++ syntax, intelligent removal of comments, including all variants in which they might appear, was difficult. Therefore, in the end, the preprocessor of the *gcc* compiler was used in the following way:

```
gcc -fpreprocessed -dD -E -P input_file > output_file
```

5.1.2 The tool to evaluate a set of servers

Consequently, a set of 13 *quasar*-based servers in advanced state of development, each of them authored or co-authored by 10 different developers, was chosen for statistical evaluation. The *QuasarMetrics* tool detailed in section 5.1.1 was used as the per-server evaluation engine and the results were further aggregated. The tool was used to produce table 5.1 in section 5.7 with abbreviated results as well as the table A.1 with full results in the appendix A.

5.2 Case study: ATLAS Wiener OPC UA server

5.2.1 Introduction and context

The ATLAS Wiener OPC UA server was conceived to integrate monitoring and controls of VME crates (that is: equipment which can host expansion cards in the VME standard [91]) into a large-scale distributed control system. The particular solution addressed crates from a particular supplier (W-IE-NE-R Power Electronics[11]) who designed a custom communication protocol[92] [4] running on top of CAN bus[53]·

The crates have numerous operational variables:

- relating to powering of hosted equipment: voltages, currents and status of different power lines
- relating to cooling of hosted equipment: temperatures, fan speeds, etc.
- relating to VMEbus: error flags, etc.

The communication protocol is straightforward: aforementioned operational variables of such VME crates are bound to particular types of CAN messages, usually one type of CAN message is used to deliver information about multiple variables. The communication is generally oriented on requests and replies, only in exceptional cases the equipment sends notifications on its own initiative. As per CAN standard, there can be multiple crates (in theory, up to 127) on each CAN bus.

Apart from monitoring the crates can be controlled: switched on and off, the setpoints (e.g. current trip setpoints) can be altered, VME reset signal can be invoked, etc.

[4] In fact the same protocol is reused by many more types of equipment from that particular supplier, such as industrial power supplies, and the author's product was also used to their monitoring and control needs.

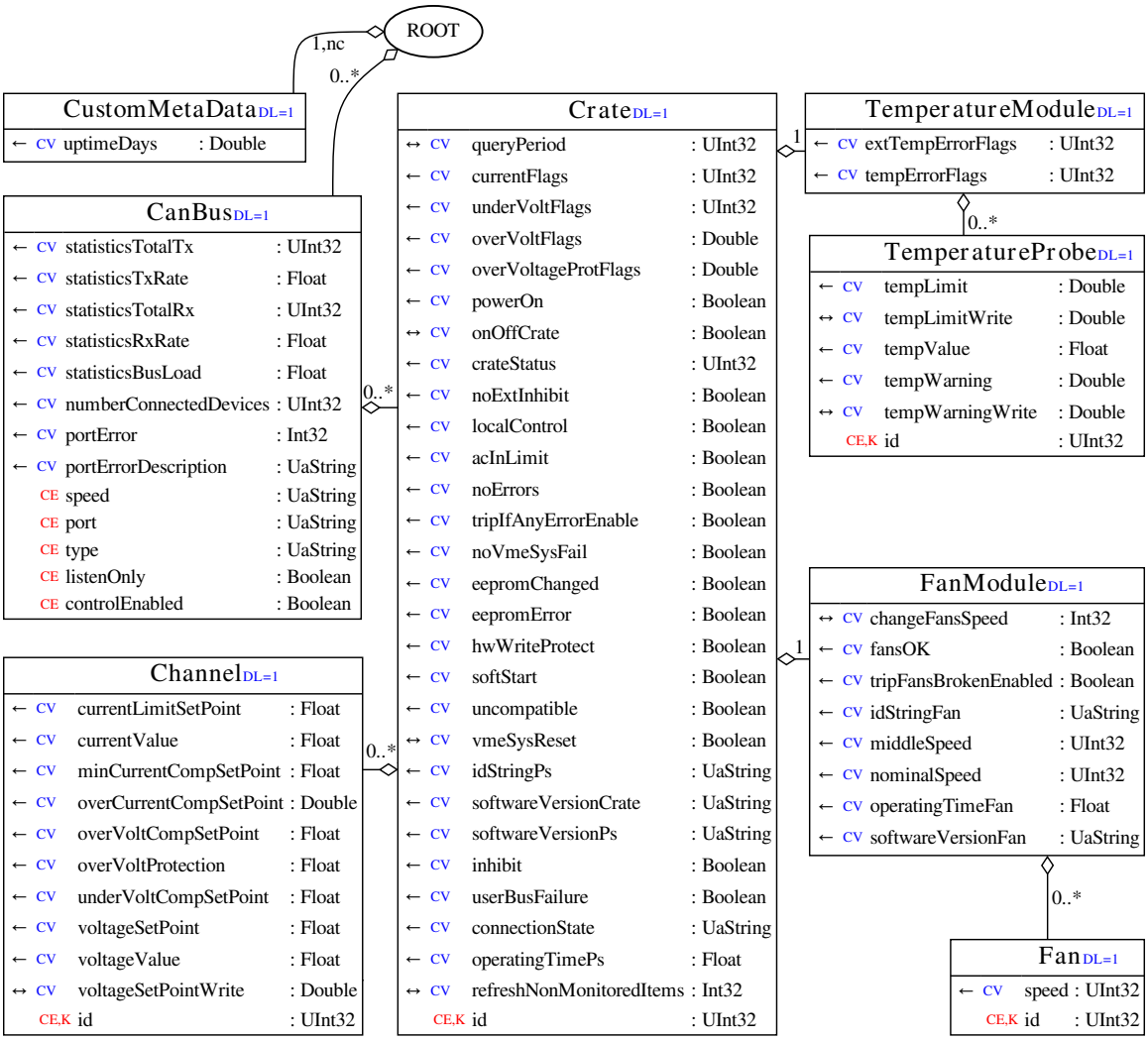


Figure 5.2: A *quasar design diagram* of the ATLAS Wiener OPC UA server detailed in section 5.2.

5.2.2 Product requirements

The following requirements were given or identified:

- complete support of the provided communication protocol enabling full profit of crates controls and monitoring
- system size per a running OPC UA server instance: support of unconstrained number of CAN buses with up to theoretical maximum of crates on every CAN bus
- fault tolerance and fault recovery (e.g. following CAN bus issues)
- sophisticated error reporting and propagation via OPC UA towards control system operators

5.2.3 Propagation of requirements to quasar design

Following the requirements (section 5.2.2) and the knowledge of the protocol[92] a *quasar design* was composed. Its visualization is presented in figure 5.2.

Thanks to the fact that the set of all operational variables (to be monitored or controlled) is known way in advance (including their data types and mapping into protocol data units), a static model was chosen (that is: all variables are declared a priori in the *quasar design*). Apart from advantages of being explicit, such a model is better for creation of other deliverables such as OPC UA clients and SCADA integration, because whole knowledge about the system stays in the design and is not dependent on configuration files and/or run-time information.

5.2.4 Implementation and evaluation

Following the design stage (including creation of *quasar design*, as per section 5.2.3) the Device Logic code was written. The Device Logic is oriented on event-driven architecture:

- the library for CAN communication is requested to install a callback which gets invoked at reception of every CAN frame on a bus
- there is a list of handler functions per each declared *quasar class*, for instance one sees a method `handleVoltageCurrentUpdate` in Channel Device Logic class. All those methods know the data operational data is stored in given type of CAN frame so at each invocation they process the receive frame and update the OPC UA address-space.
- the aforementioned callback dispatches all received messaged to the handler functions.
- since all received frames must be first requested, there is a module called `CrateCommunicationController` which sends out such requests.

In addition to the `CrateCommunicationController`, which serves monitoring of operational variables, a class called `NonMonitoredItemsController` was designed to support either one-off or on-request data, such as identifiers, versions, serial numbers etc.

The server was made in 2014 and deployed in multiple instances in production control system in 2015. Each server instance usually runs for few months (typically 6 months) non-stop between system maintenance periods. No crash or another abnormal event was noticed between 2015 (deployment date) and 2019 (this writing).

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
8	63	0	0	9	4998	414	1397	983	6.5

5.3 Case study: Generic OPC UA–SNMP gateway

In comparison with the simple example detailed in section 5.2, which covered a well defined type of control system objects (so, was specific rather than generic), the Generic OPC UA–SNMP gateway was conceived to support any control system objects communicating by the Simple Network Management Protocol (SNMP).

The Simple Network Management Protocol (SNMP) is a standardized and widely used protocol to exchange information between devices and hosts monitoring/controlling them[57]. In the parlance of SNMP, a party initiating communication (i.e. a master) is called manager, and a party that reports data to the manager (or is controlled by the manager) is called an agent. The protocol is relatively simple and thanks to that, SNMP agents can usually be easily implemented even in simple computing nodes, such as low-end embedded systems. Despite simplicity, the protocol does not exclude well-structured information models, so different individual variables representing state of monitored/controlled agents can be recognized, handled and exchanged.

Most often agents are implemented in the computer network equipment (e.g. manageable switches), high-end telecommunication crates (e.g. μTCA, ATCA[93]) but also industrial equipment (e.g. industrial power supplies).

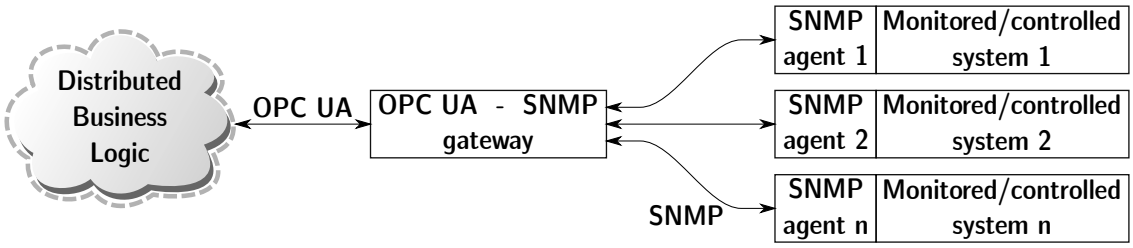


Figure 5.3: A simplified architecture of OPC UA–SNMP information exchange.

A concept of a generic gateway enabling information exchange between SNMP agents and a distributed system in the OPC UA standard was studied. The study was motivated by testing whether *quasar* is suitable to generate generic information-exchange applications, without much assumptions taken at compile time, and rather configured by end-users in the run-time. This stands in contrast to the assumptions from the previous example in section 5.2. The reason that SNMP was chosen for such a test was because:

- the SNMP is not overly complicated while still generic enough
- it has good software support (e.g. support libraries such as *net-snmp* [58])
- there were many possible applications for such a software product in the control systems that were in the author's reach so the applications were motivated by real-life cases.

The author proposed a simplified concept of a distributed system built using the gateway discussed in this section is detailed in figure 5.3.

5.3.1 Requirements analysis

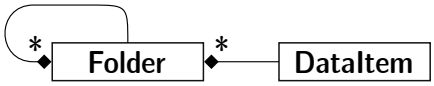
The following functional requirements were identified for the gateway:

- support for one-off writes and reads of SNMP data objects
  The gateway should be able to handle a write and a read request from an OPC UA client, pass it to the SNMP domain and translate the results back to the OPC UA domain. Failure conditions (e.g. no acknowledgment from the SNMP agent) need to be gracefully handled (e.g. as an appropriate OPC UA status code).

- support for sampling of data objects
  The gateway should be able to sample SNMP data-objects when such a mode is configured by the end user. The SNMP, as a request-reply protocol, does not support any communication pattern similar to "publish-subscribe", therefore developing such a feature brings multiple features to the end-users of the gateway:

  - reduction of network traffic: using the gateway, the network traffic resulting from polling the SNMP agent for data might be limited to the network segment between the gateway and the agent, independently of the number of consumers in the control system business logic (because the distribution of data is done on the OPC UA level, which is a far more efficient and advanced protocol than SNMP for data distribution in big systems).
  - enabling use of limited resources devices in big control systems: using the gateway, even relatively low-end devices can supply data via SNMP because the act of interacting via possibly big number of consumers is shifted to the gateway.

Figure 5.4: Recursive structure which permits generation of an arbitrary-depth tree representing the OPC UA address-space graph.



  - simplification of control system algorithms: using the gateway, the algorithms which continuously process the data (for example, supervision software) might be rewritten such that their processing is activated only on data change (which is technically easy and robust when OPC UA *Monitored Items Service* is used).
  - performance - in case of slowly changing data (the typical use of SNMP is for monitoring data, which from the author's experience often do not change for relatively long time) the computing resources might be reallocated much better (e.g. to faster application) because the gateway would only emit on-change notifications.

- support for access rights
  The end-user should be able to configure the system such that data bound to SNMP and exposed via OPC UA can be given access rights such as *read-only*, *write-only*, *read-write*.

- flexibility of information mapping (including support for any mapping between OPC UA address-space and the data objects layout imposed by particular SNMP agent)
  SNMP Agent impacts data objects layout by its own preferences (which come from SNMP agent author/vendor preferences). Such layout can be easily seen when respective Managed Information Base (MIB) file is studied. However, such layout might not be preferred by control system developers and the usage of gateway should permit to "remap" the structure arbitrarily according to the configuration defined by control system developers or end-users.

5.3.2 Software design

5.3.2.1 Decomposition into quasar classes

The author proposed the following *quasar classes*:

- Agent
  It corresponds to a SNMP agent (which typically is an instance of software, running in a *managed* device, which is capable of handling SNMP network requests and replying to them). A typical attribute of a SNMP agent is its network address and so-called *SNMP communities* which server a purpose similar to user names.

- DataItem
  It corresponds to an individual SNMP data object and the act of accessing it is maintained by its parent SNMP agent. A SNMP data object is identified by its Object-ID (in the scope of given Agent) and is characterized by its data type at minimum.

- Folder

  Folders serve only as means to better organize the OPC UA address-space. Folders can contain other folders as well as DataItems, as depicted in figure 5.4. Thanks to that, an arbitrary-depth tree can be configured by an end-user.

The *quasar design diagram* illustrating the chosen decomposition into *quasar classes* is shown in figure 5.5.

5.3.2.2 Organization of data flow

The author identified three possible paths of data flow (to exchange information between SNMP and OPC UA domains via proposed gateway):

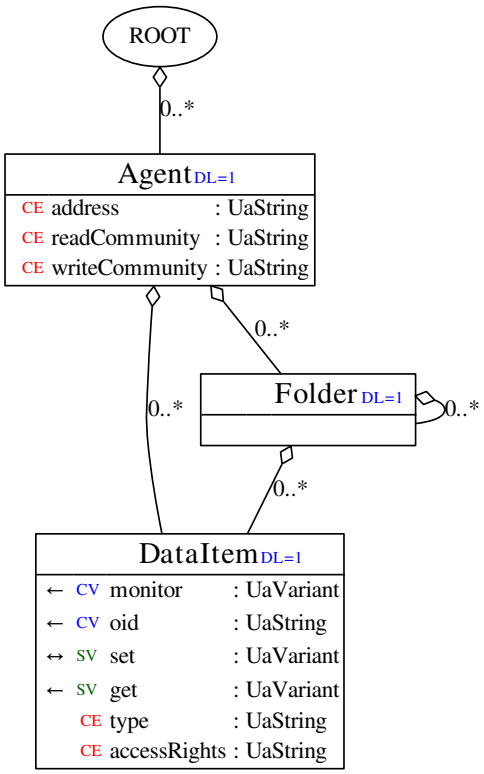


Figure 5.5: *quasar design diagram* of the OPC UA–SNMP gateway from the section 5.3. In the Agent class, `address` is the network address of the Agent; `readCommunity` and `writeCommunity` are the names of so called *SNMP communities* which help to group users of SNMP Agent according to rights they have. In the DataItem class, `monitor` is the *quasar cache variable* which stores the last known states of monitored SNMP data object (it is the output of the sampling performed within OPC UA server); `oid` is SNMP Object Identifier; `set` and `get` are individual variables to perform write and read operations; `type` is the data type that should be assumed for given SNMP data object and `accessRights` is one of *r* (read-only), *w* (write-only) or *rw* (read-write). Note that all variables participating in data flow between SNMP and OPC UA domains (`monitor`, `get` and `set`) are of *variant* OPC UA data type (which signifies *any* type).

- monitoring of SNMP data objects
  Monitoring (resulting in publish-subscribe communication pattern in the OPC UA domain) is attractive for multiple applications. However, SNMP only supports reading by request-reply communication pattern, so the implementation of publish-subscribe via means of request-reply polling must be implemented in the gateway with configurable polling interval(s). The author proposed that the gateway would keep a (configurable) number of polling threads which would perform SNMP *get* requests. The monitoring mechanism is available via `monitor` *cache variable* of DataItem class.

- reading
  It is expected, that certain control system applications, in which continuous monitoring or performance are not required, would prefer reading on demand. Therefore the author proposed support for such one-off requests via `get` *source variable* of DataItem class.

- writing
  Support to write data (via SNMP *set* requests) is needed to perform controls. Such support is provided via `set` *source variable* of DataItem class.

5.3.3 Implementation and usage examples

- Wiener RCM
  The project was successfully evaluated to be used as controls and monitoring of the Wiener Remote Control Module of Maraton power supplies.

- ATLAS Fast Tracker cooling and ventilation monitoring
  The project is contracted to be used to integrate custom-built monitoring solution (based on a custom-build SNMP agent deployed on an Arduino board) for the ATLAS Fast Tracker detector control system.

5.3.4 Evaluation of software automation

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
3	2	2	0	5	1008	175	373	198	7.0

5.4 GBT-SCA OPC UA project

The author presents the GBT-SCA OPC UA project here because, to his knowledge, it is the most complex set of OPC UA software components realized using *quasar* in terms of diversity, size and span of used features of *quasar*.

The project comprises of an OPC UA server, OPC UA clients (developed using UaObjects detailed in the section 4.9.2.1) and SCADA integration layer (as per section 4.9.3). In additions, on top of it another OPC UA layer was built applying the chained-servers concept (see for instance the OPC UA server detailed in section 5.6.2).

5.4.1 GBT-SCA overview

The GBT (Gigabit Transceiver) project [94] is a new ecosystem of electronic and computing technologies enabling wide bandwidth data acquisition (a typical example is a system able to process data from 24 optical fibers each at 4.8Gbps data rate on a single server machine). The special feature of the technology is that all data sources (i.e. all electronic devices which transmit data over fiber optics) are qualified to work in very harsh physical conditions (especially regarding radiation levels and the magnetic field).

One of the core parts of the GBT ecosystem is the GBT-SCA [5] integrated circuit [95]. The GBT-SCA chip is an universal analog/digital input/output device, which was used (and, as of writing, is planned to be used) as the basic building block of a vast range of control systems applications.

The following properties of the chip are relevant in the context of its integration in an OPC UA based control system:

- the chip can be interfaced using a custom protocol [96], encapsulated in the HDLC (High-level Data Link Control) protocol [97], transferred on top of the e-link link technology [98]
- the chip is composed of 21 processing blocks which can run concurrently. Among the blocks one can find the Serial Peripheral Interface block [99], a general purpose digital I/O block, 16 I2C blocks [99], a JTAG master block [100], an analog-digital converter and a digital-analog converted.
- all blocks mentioned above can be talked to (concurrently) using a request-reply communication pattern and each of them can process at most one command concurrently (there is no buffer which could store requests). In addition, all blocks are communicated via shared HDLC link, which enforces usage of common sequence number. Therefore the aspects of synchronization of communication can not be ignored.

5.4.2 Constraints and requirements

- system size (number of GBT-SCA per system)
  Many of contracted applications (that is: systems in which the software under discussion is to be used to interface the GBT-SCA chips) are relatively large: they need to integrate hundreds or thousands of GBT-SCA chips per system. For example, the New Small Wheel project will use 6944 SCA chips [101]. Another application, the Liquid Argon (LAr) Trigger Digitizer Board

[5] SCA stands for "Slow Controls Adapter". A bit pejorative adjective "slow" refers to the rate of signals which are used in controls applications, which are significantly slower than signals carrying data.

will use 620 SCA chips. Both applications will use the software described in this section to establish information exchange between GBT-SCA chip and the distributed control system.

- diverse interfaces of GBT-SCA connectivity
  The GBT-SCA is a device interfaced to the control system via e-link. There are many possible architectures permitting ELink interfacing to the GBT-SCA, for example using APIs coming from the FELIX software[6] libraries.

- layer of abstraction
  The product is expected to provide an abstraction layer to hide away the low-level details of communication with the GBT-SCA chip (such as the GBT-SCA protocol, the HDLC protocol, the e-link connectivity etc). The end-user should see (via the OPC UA address-space) the input/output features of the GBT-SCA, such as ADC conversion values, digital I/O states etc.

- reliability
  The product must be very robust because it is expected to deliver 24/7 service in uninterrupted duration of few months (typically 6 months). The service delivered by the product will be used to monitor, control and configure a large scale industrial machine, so a failure to deliver the service is likely to produce down-time to the whole machine.

- generic support
  The GBT-SCA is an universal analog/digital input/output chip. As such, it exposes a number of universal interfaces like: general-purpose digital input/output(GPIO), analogue input, Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I^2C). It is planned that control system engineers would connect devices performing specialized functions to those universal interfaces. When that happens, the GBT-SCA OPC UA server would be used as a transport layer between a specific device and the distributed control system. In such a case it is expected that specific OPC UA servers would be created to fulfill specialized functions (for an example, see section 5.6.2). Therefore the server must be designed such that serving the functions of a transport layer is feasible, robust and practical.

5.4.3 Decomposition into SCA Software (the abstraction layer) and the OPC UA server

An effort to decompose the problem into parts was made. As the effect, the author decided to organize the product as two distinct software components:

- an universal library for GBT-SCA support, called *SCA Software* and abbreviated SCA-SW
- a *quasar-based* OPC UA server built on top of that library

The SCA Software should:

- provide a high-level Application Programming Interface to support features of different components of the GBT-SCA (e.g. ADC, DAC, ...)
- abstract out the details of the communication interface
- hide away the particular type of e-link implementation for GBT-SCA connectivity
- support book-keeping of transactions per chip such that synchronization aspects would be hidden and traceability of lost/failed transactions was assured

The SCA Software was implemented [103] [7]. The package delivers a high-level C++ API which can be exemplified by the following demonstrator program:

[6] FELIX (Front-End Link Exchange)[102] is an ecosystem of electronics and software which offers an interface between GBT systems and software deployed on common servers. FELIX-based architecture of GBT-SCA interfacing is shown in figure 5.6.

[7] The author would like to acknowledge the following contributors: Paraschos Moschovakos, Henk Boterenbrood, Aimilianos Koulouris, Stefan Schlenker.

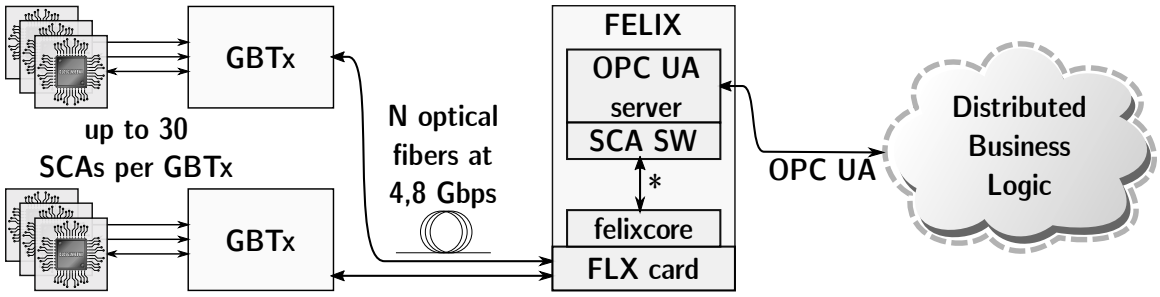


Figure 5.6: GBT-SCA integration into a control system using OPC UA in the *FELIX* variant (which is one of many studied architectures). The asterisk symbol (*) relates to the fact that varied ways of connecting SCA-SW to the felixcore are possible, including the netio project [104], Unix Domain Sockets [105] and even direct function calls in case felixcore is integrated with the SCA-SW and the OPC UA server into one executable. The architecture shown in the picture was been successfully implemented, and as of most recent update (May 2019) it was undergoing integration and systems test.

Listing 5.1: SCA Software code example.

```
1  Sca::Sca sca ( ''sca-simulator://1'' );
2  std::cout << sca.adc().convertChannel(5);
```

The SCA Software package supports multiple connectivity options out of which the most popular are:

- the *netio* mode (which can be used to connect to a GBT-SCA using the FELIX architecture - see figure 5.6)
- the simulator mode (which is a purely software based simulation engine, transparent to the end-user).

5.4.4 Architecture of the server

A *quasar design* was created, as visualized in 4.4.

All conceptual features of *quasar* were used for the SCA OPC UA server, as can be judged from the design diagram (figure 5.7):

- *quasar cache variables* are used when server is supposed to poll periodically (e.g. analog-to-digital conversion results) or when an one-off value (i.e. constant during server life time) is enough (e.g. for GBT-SCA serial number)
- *quasar source variables* are used whenever each OPC UA read or write must precisely correspond to one GBT-SCA operation - e.g. getting or setting value of a GPIO line
- *quasar methods* are used when it is impossible to solve information exchange problem by either of variable types. For example, the `AnalogInput` class has a method `getConsecutiveRawSamples` with an argument of number of samples: that is: a potential OPC UA client might request parameterized size of such data (raw samples).
- *quasar config entries* represent configuration data which are fixed in the lifetime of server instance, e.g. a direction of given GPIO line (such data is considered fixed because the direction is imposed by hardware, which is unlikely to change).

In addition, it is expected that the *Calculated Variables* feature (see section 4.9.1.11) will be massively used, especially to provide calculation of physical quantities from the values obtained by GBT-SCA Analog Inputs.

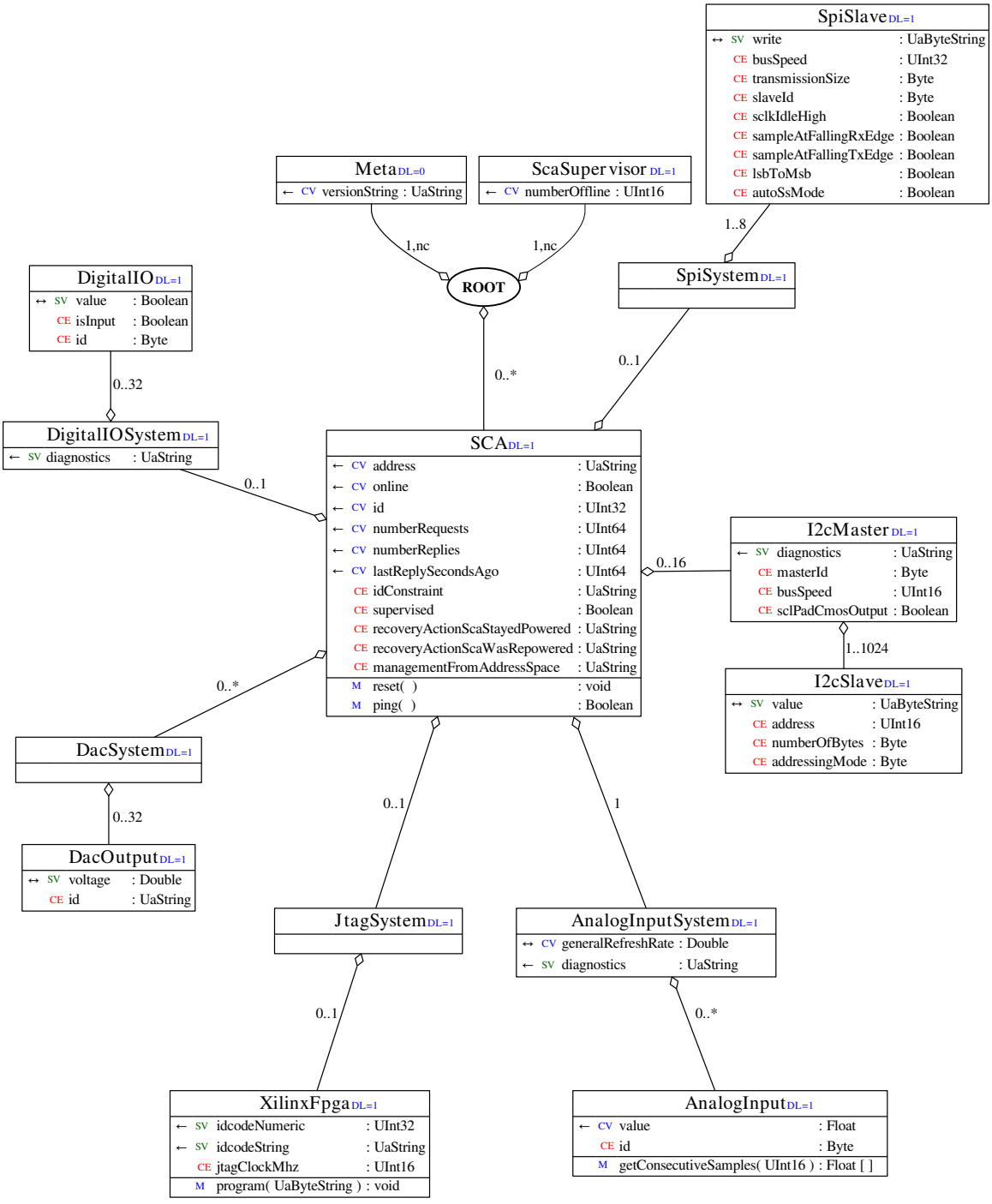


Figure 5.7: *quasar design diagram* of the GBT-SCA OPC UA server.

5.4.5 GBT-SCA OPC UA server and the related software ecosystem

The GBT-SCA OPC UA server project, apart from the factual OPC UA server for the GBT-SCA chip, is a primary building block of an ecosystem of related software solutions. A simplified illustration of the ecosystem was presented in figure 5.8.

Overall the following paths (as in figure 5.8) can be identified which span from a potential control system user towards the control system hardware:

- the simplest monitoring path shown in the figure from the hardware via path *a*, *b* up to SCADA-based user interface
  Such a path can be used for multiple applications where the GBT-SCA OPC UA server can deliver an intelligible measurement without any peripheral server. An example would be analogue value measurements by the GBT-SCA embedded Analog-to-Digital converter (with or without usage of Calculated Variables[8]).
- another path is exemplified from the hardware via *a*, *c* and *d*
  Such a path is applicable to all cases where the GBT-SCA OPC UA is used as a transport layer for another protocol running on top of it, interfacing a specialized device over one of common (typically digital) interfaces (e.g. SPI, I^2C). An example could be an Analog-to-Digital converter connected via SPI (for to-SCADA direction) or an intelligent motor driver with I^2C protocol (for from-SCADA direction).
- another path is via *a*, *e* and *a*, *c*, *f*
  Such a path is provided for non-SCADA cases, that is: when the end-user is not related to the control system operations. The typical examples are interfacing of OPC UA servers with Data Acquisition Systems (which is an alternative path to control system). An example of this path is the calibration and configuration of the New Small Wheel hardware. The calibration and configuration is both technically and in terms of responsibility organized as a part of the New Small Wheel DAQ group. Thus clean separation of responsibilities is required. However, due to the fact that GBT-SCA in the NSW is used concurrently by controls and DAQ systems, the DAQ systems adopted usage of UaoCpp (see section 4.9.2.1) to integrate OPC UA connectivity to the GBT-SCA in their software. Such a case is exemplified by path *a*, *e* in the figure.
- another one is *a*, *h* and *a*, *c*, *g*
  This connectivity path is normally opened only temporarily for diagnostics of issues.

Overall, the following components of the *quasar* eco-system were used in GBT-SCA related activities:

- *quasar* itself, for creation of both the main OPC UA server as well as the peripheral servers in the chained architecture
- UaoCpp (see section 4.9.2.1) for creation of the C++ clients
- fwQuasar for performing SCADA integration of both the main server as well as peripheral servers (see section 4.9.3)

5.4.6 Evaluation of software automation

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
17	15	9	8	27	5889	968	1653	685	11.0

In the case of the GBT-SCA OPC UA server, additional deliverables were obtained using *quasar*:

[8] See section 4.9.1.11.

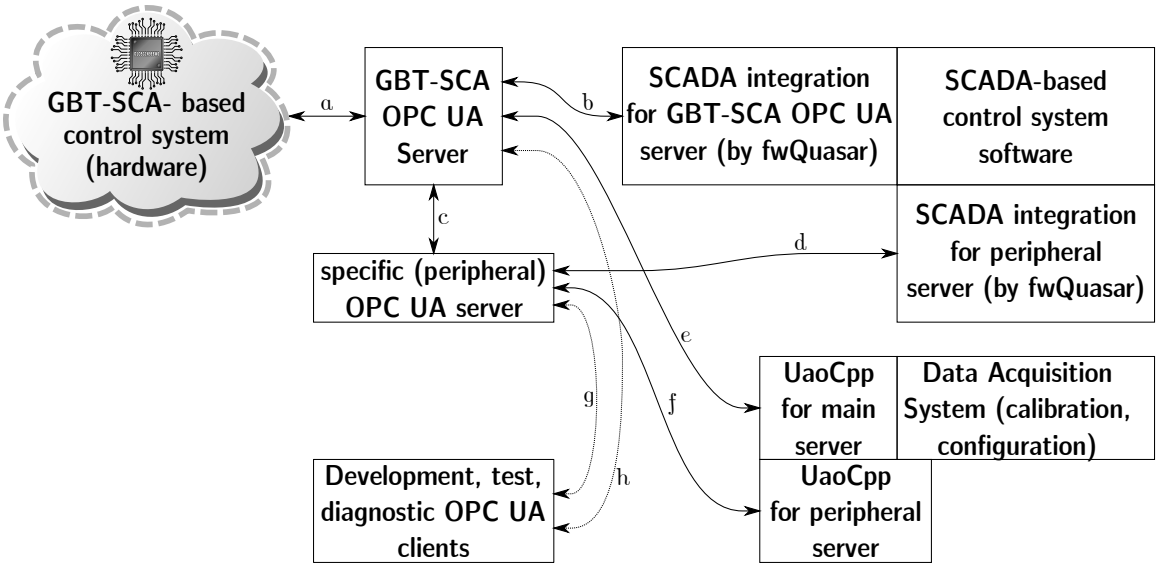


Figure 5.8: A simplified illustration of the software ecosystem based on the GBT-SCA OPC UA server. Path *a* relates to information exchange between the GBT-SCA and the OPC UA server, it might be implemented using different solutions (e.g. via FELIX project). Paths *b, c, d, e, f, g, h* are all OPC UA. Dotted lines correspond to OPC UA connections which are usually not permanently present in the control system.

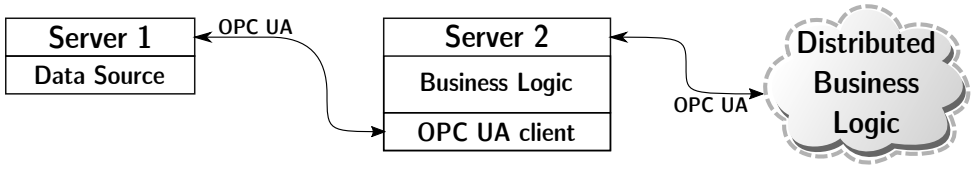


Figure 5.9: One of many examples of simplest chained OPC UA systems. The lines and arrows illustrate the information flow.

- UaObjects-based C++ client
  The design-dependent generated client code was measured as 2103 ELoCs.
- SCADA integration layer
  The design-dependent generated SCADA integration layer (in the CTL language) was measured as 1715 ELoCs.

Comparing the numbers to the n_{ND} figure given above, about 35% additional automation ratio gain is seen for the UaObjects C++ generation and about 30% further is seen for SCADA integration layer.

5.5 OPC UA to OPC UA distributed components

In the thesis, a "chained" OPC UA distributed information exchange system is defined as a system in which multiple OPC UA information exchange links exist and information passing over one of them is in some way propagated to another one.

The figure 5.9 shows one of simplest "chained" systems. The aforementioned propagation of information between links can be shown on the following example:

- on Server 1, the data source generates information; the information is published for example through a *cache variable* on Server 1

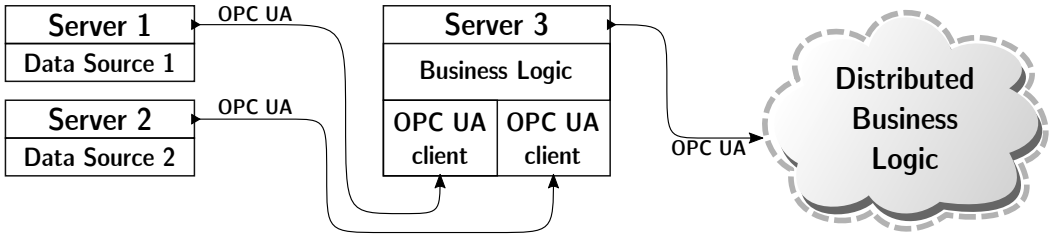


Figure 5.10: An example of a chained OPC UA system with multiple clients and servers. The lines and arrows illustrate the information flow.

- the OPC UA client which is incorporated in the Server 2 receives the information, perhaps processes it in some way and then publishes it through a *cache variable* on Server 2

It could therefore be said that Server 1 and Server 2 are chained.

Such chaining can be imagined in multiple ways - between two clients, between two servers; by usage of variables or methods etc.

The reasons to apply such chaining might be multiple:

- to provide additional data processing layer (or, "adapt" one OPC UA interface to another OPC UA interface)
- to distribute computing load
- to process combined data from multiple sources (applies to cases when more than two interfaces are considered)

5.5.1 A simple example

An interconnected system is shown in figure 5.10.

quasar permits efficient creation of such chained software. In this section it is shown, that if the servers (especially Server 1 and Server 2 in the figure) are made using *quasar*, then one can also efficiently obtain corresponding client code for integration into Server 3 (in figure 5.10).

Therefore, the act of construction of Server 3 boils down solely to provision of the "Business Logic" code.

For simplicity, both Server 1 and Server 2 are instances of the same *quasar server*. The data sources of Server 1 and Server 2 are just samples of sinusoidal function over time, with configurable frequency and phase. A *quasar design diagram* for both servers is shown in figure 5.11.

The key take away message of the section is that *quasar* can efficiently generate the client code to be embedded in the Server 3 using *UaObjects* approach, as per section 4.9.2.1.

The *quasar server* shown in the picture 5.11 as (a) was put to the UaoCpp (UaObjects for C++) tool. The listing 5.2 presents the declaration of the resulting class.

Listing 5.2: Declaration of the SineGenerator class obtained from the *quasar diagram* shown in figure5.11 as (a), using UaoCpp.

```
1  class SineGenerator
2  {
3
4  public:
5
6    SineGenerator(
7      UaClientSdk::UaSession* session,
8      UaNodeId objId
9      );
10
11   OpcUa_Double getOutput ();
12   OpcUa_Double getFrequency ();
```

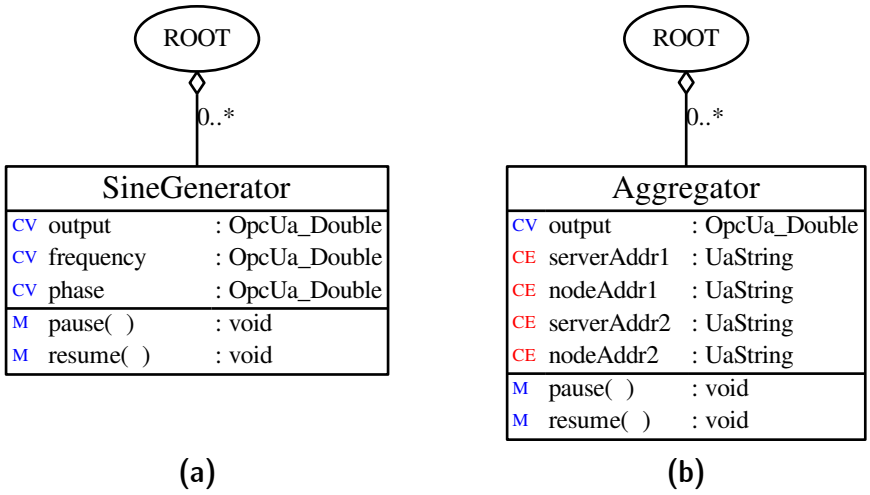


Figure 5.11: quasar design diagrams of cascaded middle-ware example from section 5.5.1. (a) shows *quasar design* of Server 1 and Server 2. (b) shows *quasar design* of Server 3.

```
13    OpcUa_Double getPhase ();
14
15    void pause();
16
17    void resume();
18
19  private:
20
21    UaClientSdk::UaSession  * m_session;
22    UaNodeId      m_objId;
23  };
```

Looking at the constructor (lines 6-9) one sees that creation of such a proxy object requires an OPC UA session object (which corresponds to OPC UA connection) and an object identifier. The identifier in this case is the OPC UA address of such object in a running instance of Server 1 or Server 2 accordingly. Such identifiers are typically constant per deployment [9] so they can be stored as *config entries* of the Server 3.

The most important asset in the generated `SineGenerator` class are the following methods: `getOutput` , `getFrequency` and `getPhase`. The methods hide (abstract out) the implementation of *read*, *write* and *call* service calls through OPC UA. In addition their inputs and outputs have common data types and at the same time they guarantee type safety and error detection and propagation, so a developer is freed from such a responsibility.

An "aggregating" OPC UA server (corresponding to Server 3 in figure 5.10) can then be easily derived. A developer should prepare a design, for example similarly to the design (b) in figure 5.11. The content of the design must obviously correspond to the requirements and expected features of such a chained software component.

In the example, the aggregating server (Server 3) reads the samples from Server 1 and Server 2 and performs a rather trivial data processing: an output value, y is transformed accordingly to the formula $y = a^2 + b^2$. The reader is reminded that both Server 1 and Server 2 sample a sinusoidal function, so if the same frequency, amplitude and phase reference regarding chosen time is ensured, then the published output on Server 3 should always be equal to 1, as per the trigonometric identity $sin^2x + cos^2x = 1$

In addition, both methods `pause` and `resume` of Server 3 propagate to respective method invocations to Server 1 and Server 2. Therefore, by demonstrating the information flow in the opposite direction, it is shown that a bi-directional chaining was achieved between three computer programs forming

[9] The author would like to point out that the proposed solution does not impose them to be constant.

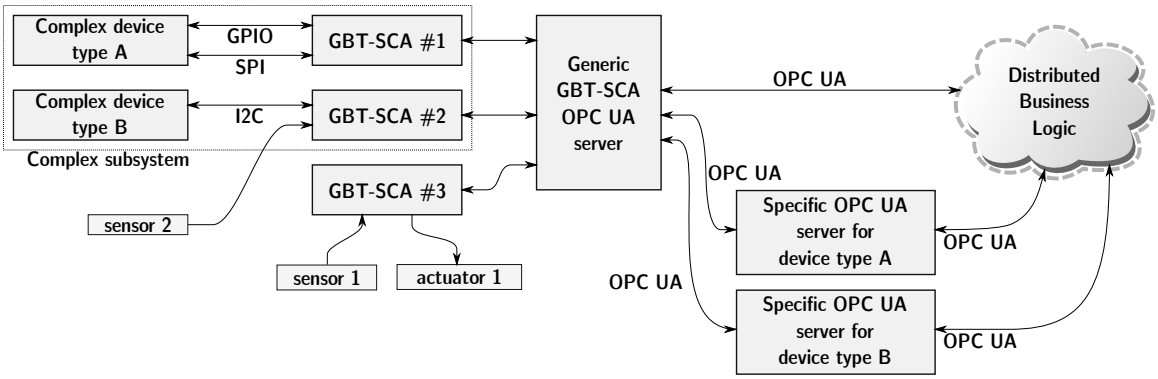


Figure 5.12: Distributing GBT-SCA OPC UA system into generic and specific servers.

a simple model of a control system, and based on 5 software components (3 OPC UA servers and 2 OPC UA clients), all of them made using *quasar*.

An excerpt of the implementation of Server 3 business logic is presented in the listing 5.3.

Listing 5.3: An excerpt of business logic of Server 3 (as per the figure 5.10).

```
1 float val1 = sg1.getOutput();
2 float val2 = sg2.getOutput();
3
4 float agg = val1*val1 + val2*val2;
5 getAddressSpaceLink()->setOutput(agg, OpcUa_Good);
```

The source code of OPC UA clients generated by the UaoCpp tool can be put (included as source code, or linked against if distributed as a compiled library) to the Device Logic of the Server 3.

5.5.2 Distributing logic of the SCA OPC UA integration

A more complex example of chaining can be presented basing on a factual problem from a real-world control system deployment. It was mentioned in section 5.4 that the GBT-SCA supports some interface types - like SPI or I2C - which themselves are very generic and can provide connectivity to plethora of different devices.

Therefore, the OPC UA based system should support such connectivity and let the distributed logic communicate to such devices. However at the same time the server should stay generic and avoid inclusion of logic relevant only to some specific device types. How could such a conflicting requirement be handled?

One of the possibilities is distribution of the logic of the system into a generic GBT-SCA solution (the same software as outlined in 5.4) and a number of specific, separate OPC UA servers that would handle those specific devices. The distributed business logic would therefore use the services exposed by the generic GBT-SCA server for concrete device types and a number of separate services for the specific device types.

An example of such distribution is shown in figure 5.12.

5.5.3 Integration of quasar-made OPC UA servers into different programming languages

Even though *quasar* was conceived to support creation of software for the C++ programming language, often it happens that such software must be further included in another software project which is based on another programming language. In the author's experience such a case arose for instance for the Python programming language. Namely, it was required that an existing piece

of software (of production-grade maturity) written in Python would be extended with OPC UA interfacing by usage of *quasar* ecosystem [10]. Therefore the Python application would be considered a *master* and an *quasar server* would be a *slave*, and both would preferably be integrated without visible seams.

Three approaches have been attempted to perform *quasar server* integration into a program in Python, using:

- C Foreign Function Interface [106]
  CFFI is described in the documentation as "Interact with almost any C code from Python, based on C-like declarations that you can often copy-paste from header files or documentation."
- Simplified Wrapper and Interface Generator (commonly known as SWIG) [107]
  According to [107]: "SWIG is an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby, and Tcl."
- Boost.Python [108]
  Boost.Python is "a C++ library which enables seamless interoperability between C++ and the Python programming language"[108].

The goal to accomplish was the same for all three approaches:

- provide "a control path" that could start and stop the server - i.e. run the server with a chosen configuration, such that it would become indiscernible part of the hosting process
- access objects of the server, so the hosting process could invoke e.g. Address Space objects methods to publish data via OPC UA
- provide elegant handling of data representation of OPC UA from Python

In all approaches the author attempted to respect the key advantages of *quasar* such as adherence to the model, type safety, developer's efficiency etc.

5.5.3.1 Embedding using CFFI

The CFFI approach was studied as the first one. The approach was from the beginning controversial, due to the fact that CFFI is based on the C programming language, therefore to respect object orientation (and especially object mapping in *quasar*) would require extra work to be achieved. Such work would be needed both on the server side (to map from the object orientation to plain C programming) and then again on the higher level programming language (here: in Python). The author obtained a working partial solution but it was considered too distant from the advantages of *quasar* and therefore discontinued.

5.5.3.2 Embedding using SWIG

The SWIG approach was evaluated afterwords. From the specification study, SWIG made an impression as a most automated solution of all three of them, that is, the least amount of "manual" coding work seemed to be required to map different classes and free functions to be used from another programming language. At the same time, SWIG seemed to be available to generate bindings to many different programming languages, therefore the investment could be very attractive and the return on investment could be potentially significant. After performing different experiments it turned out that the parser of SWIG is unable to deal with certain constructs of the C++ preprocessor which were used in headers of libraries which needed to be used to compile the servers. The inspection of code revealed that the problematic source code was conforming to standards and certainly did not use any particularly exotic or recent features of the C++ programming language.

[10] There exist Python-based OPC UA server solutions but there are significant advantages to maintain a solution based on *quasar*, for instance because of possibility to efficiently create corresponding OPC UA clients (see section 4.9.2) as well as to efficiently provide SCADA interfacing (see section 4.9.3).

Manual modification of the headers made the SWIG wrapper accept them, but the overall experience was considered not robust enough for acceptance into efficient creation of robust software. The SWIG approach was therefore put on hold and Boost.Python was evaluated.

5.5.3.3 Embedding using Boost.Python

The Boost.Python approach was significantly different from both previous experiences. Compared to the CFFI experience, Boost.Python (as a part of Boost C++ libraries) is entirely object-oriented and C++ based, so no additional efforts were required for make up. Compared to the SWIG experience, Boost.Python did not use any external interface parser and every step of mapping was resolved by the chosen C++ compiler, therefore as long as the C++ compiler was able to parse the used libraries header files, it was equally able to provide the mapping. On the other hand, due to no support of reflection in C++, it was necessary to *specify* the list of methods to be mapped, including supplementary information such as:

- value passing semantics - for instance, how to treat pointer or reference based arguments and return values and who owns an object pointed/referred to
- attributes such as whether a method is *static*
- how to map data types which do not have any direct equivalent in the C++ standard library, such as OPC UA basic string type, often referred to as `UaString`

The effort was overall successful and it deserved to be generalized. The author therefore decided to spawn it as an optional module of *quasar* and made it public[75].

5.6 Summary of known applications

This section aims to outline the use cases of *quasar* framework known to the author up to May 2019.

5.6.1 ATCA Shelf Manager OPC UA server

Advanced TCA (Advanced Telecommunications Computing Architecture, abbreviated ATCA) is an open standard of high-end, industry-grade computing equipment, especially for telecommunication markets [93].

ATCA Shelf Manager is a part of an ATCA shelf which monitors and controls blades and other Field Replaceable Units (FRUs) [93]. The Shelf Manager communicates with the external world using some common protocols including SNMP. The OPC UA server is specifically made for Shelf Manager with Pigeon Point IPM Sentry, and it monitors and controls the activities within the shelf via the Intelligent Platform Management Bus (IPMB). In addition, the server features support for custom sensors realized on the IPMC.

The author would like to thank his colleague and the author of the ATCA Shelf Manager OPC UA server, Paris Moschovakos, for delivering the information about the project and granting permission to publish it in this thesis.

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
16	178	12	0	15	11817	1003	1313	310	42.4

5.6.2 ATLAS Liquid Argon (LAr) LTDB OPC UA server

The LAr Trigger Digitizer Board (LTDB) is a specialized device for signal processing for the LAr sub-detector of ATLAS which is planned to be deployed from 2021 on[109]. Apart from complex signal processing electronics, the device uses the GBT-SCA chip [95] to perform different monitoring,

configuration and controls activities. Due to the usage of GBT-SCA, the GBT-SCA OPC UA server (see section 5.4) is planned to be used as the base interface to the chip. On top of it, the LTDB OPC UA server is planned to be used, accordingly to the "chained servers" architecture presented in section 5.5.2.

As of May 2019 the server was in the testing phase.

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
18	89	0	59	14	13111	1394	2085	691	22.0

5.6.3 ATLAS Liquid Argon (LAr) Purity OPC UA server

The server delivers data about quality of argonium gases in the LAr sub-detector of the ATLAS detector. The data arrives from a custom FPGA-based system and is then processed (curve fitting, etc.) and published through OPC UA as high-level estimation of gases quality.

The server was deployed into production use in 2016.

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
3	25	0	0	0	2018	145	1033	888	3.4

5.6.4 Wiener VME crates OPC UA server (ATLAS version)

See section 5.2 for details.

5.6.5 FTK SBC OPC UA server

FTK SBC OPC UA server is a relatively small project providing monitoring of parts of ATLAS FTK sub-detector deployed on single board computers (SBCs). The device-logic mostly comprises of functions communication via shared memory of a VME-based system to perform SBCs monitoring.

The project was deployed to production in 2016.

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
2	45	0	0	0	2435	92	571	479	6.3

5.6.6 HVSys OPC UA server

The server is an OPC UA middle-ware application which controls and monitors multi-channel high density High Voltage Systems[110] from the HVSys company customized to conform to ATLAS TRT detector requirements [111].

The high-voltage system uses dedicated serial communication protocol in the RS-232 standard. Hardware Access Library for HVSys OPC UA server was extracted and upgraded from previous OPC (DA and UA) servers developed for ATLAS TRT DCS. The deployed server controls up to 840 HV Cells independently managed by micro-controllers.

The project was deployed in production systems in April 2018.

The author would like to thank his colleague and the author of the HVSys OPC UA server, Jolanta Olszowska, for delivering the information about the project and granting permission to publish it in this thesis.

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
6	41	6	0	14	3789	437	951	514	9.2

5.6.7 CAEN power supplies OPC UA server

The server delivers controls and monitoring for a range of high-voltage industry-grade power supplies from the CAEN[10] company.

The server uses the the Hardware Access Library from the CAEN company implementing a custom protocol on top of TCP/IP.

The project was developed in collaboration[12] between CERN (primary author: Benjamin Farnham) and the CAEN company. It entered production environment in 2018 and it is contracted to be used in the control systems of all LHC experiments at CERN.

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
9	12	0	0	11	3151	403	559	156	23.8

5.6.8 ISEG crates OPC UA server

The server controls and monitors commercial industry-grade power supplies from the ISEG company [9]. The device-logic uses a software component implementing a custom protocol on top of the CAN network [53].

The project was developed in collaboration[12] between CERN (primary author: Benjamin Farnham) and the ISEG company[9].

The server is used in production environment since 2017.

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
10	19	0	0	12	3319	363	532	169	22.8

5.6.9 Wiener OPC UA server (JCOP version)

The project delivers controls and monitoring for industry-grade power supplies as well as VME crates developed by the Wiener company [11]. It is akin to the ATLAS Wiener OPC UA server (see section 5.2) on the level of functionality but on contrary, on the level of contributions, it is developed collaboratively with the hardware developer [12]. As of the writing (May 2019) it is contracted to be used in the existing control systems and the development is ongoing.

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
8	4	0	0	11	2351	385	596	211	14.0

5.6.10 Generic OPC UA–IPbus gateway

The project is a generic gateway for information exchange between a distributed system in the OPC UA standard and devices compatible with the IPbus[112] protocol.

The IPbus protocol is a state-less communication protocol built on top of the UDP/IP which is particularly attractive for implementation in FPGA-based systems. Many of such systems run without a factual CPU and the usage of a lean protocol like IPbus is favorable due to minimized amount of used resources of programmable logic. The CACTUS[113] libraries were used as a Hardware Access Library for the IPbus communication.

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
4	22	0	0	0	2230	207	621	414	6.9

Project name	R.S.	N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
ATCA ShelfManager	5.6.1	16	178	12	0	15	11817	1003	1313	310	42.4
ATLAS FTK SBC	5.6.5	2	45	0	0	0	2435	92	571	479	6.3
ATLAS LAr LTDB	5.6.2	18	89	0	59	14	13111	1394	2085	691	22.0
ATLAS LAr Purity	5.6.3	3	25	0	0	0	2018	145	1033	888	3.4
ATLAS Tile HvMicro	5.6.13	5	105	0	0	0	6716	374	1688	1314	6.4
ATLAS Wiener	5.6.4	8	63	0	0	9	4998	414	1397	983	6.5
HVSys	5.6.6	6	41	6	0	14	3789	437	951	514	9.2
JCOP CAEN	5.6.8	9	12	0	0	11	3151	403	559	156	23.8
JCOP ISEG	5.6.7	10	19	0	0	12	3319	363	532	169	22.8
JCOP Wiener	5.6.9	8	4	0	0	11	2351	385	596	211	14.0
Generic IPbus	5.6.10	4	22	0	0	0	2230	207	621	414	6.9
Generic SNMP	5.3	3	2	2	0	5	1008	175	373	198	7.0
SCA	5.4	17	15	9	8	27	5889	968	1653	685	11.0
total	–	–	–	–	–	–	62832	6360	13372	7012	–
average	–	–	–	–	–	–	–	–	–	–	10.9

Table 5.1: Summary table of few OPC UA servers made with *quasar*. Legend: R.S. - reference section in this thesis, N_C - number of classes, N_{CV} - number of cache-variables, N_{SV} - number of source-variables, N_M - number of methods, N_{CE} - number of config-entries, n_{ND} - non-developer code (i.e. totally generated code not supposed to be modified), n_{GS} - generated stubs, n_{IC} - interleaved code, n_D - developer code (without generated part), A.R. - automation ratio. n_{ND}, n_{GS}, n_{IC}, n_D are expressed in ELoCs (executable lines of code). Automation ratio is specified unit-less. Detailed description of measurement technique is presented in section 5.1.

5.6.11 Generic OPC UA–SNMP server

See section 5.3 for details.

5.6.12 SCA OPC UA server

See section 5.4 for details.

5.6.13 Tile HV Micro OPC UA server

The server features controls and monitoring of a custom-built high voltage delivery system for the Tile calorimeter of the ATLAS Detector. The project was delivered to production in ATLAS in April 2015.

The software metrics as per section 5.1 are presented below.

N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
5	105	0	0	0	6716	374	1688	1314	6.4

5.7 Code metrics summary

The summary table of many of projects known to the author is shown in 5.1.

The lines-of-code reduction factor is within range of 3.4 to 42.4 times, considering only the OPC UA servers without generation of related clients or related layers. The average reduction factor from 13 evaluated *quasar-based* projects is 10.9x.

Chapter 6

Summary and conclusions

The author started his research in 2014 asking the following question: how difficult is it to *efficiently* build OPC UA based distributed control systems such that they would be applicable to very heterogeneous systems planned for a decade or more of 24/7 operation?

The initial studies showed that the OPC UA standard, despite its numerous attractive properties, might not be sufficiently rigid, complete and detailed to enable out-of-the-box process of creation of compliant software, especially when efficiency of development and conceptual stability of solution is expected. These early observations, published by the author in 2015, were confirmed by independent researchers in 2019, who cited the author's paper crediting him (and his team) with researching the problem (and proposing one of the first solutions) a few years earlier. General review of such papers as well as of the state of the art is presented in chapter 2.

A method addressing the problem was proposed and was under development since 2014. A new notation, supplementary to OPC UA, was proposed (see section 4.5), as well as a related software architecture (see section 4.9.1). The concept was then enhanced with a software engineering framework and tools collectively named *quasar* (abbreviation of *Quick OPC UA server generation framework*).

Thus the *quasar* framework (and an engineering method which it builds upon) presented in this dissertation was conceived to addresses different fundamental requirements of design and implementation of large, complex and heterogeneous control systems made in the OPC UA standard. These requirements were scrutinized in section 4.2 and are not re-discussed here due to being numerous and detailed.

These requirements (enhanced with the requirements of the proposed thesis of this dissertation), were achieved by providing OPC UA standard-based software development automation methods, concepts and tools.

Development and maintenance of control system software spanning across millions of channels and with scale of the order of tens of years has to be built on a solid software framework which is on one hand stable and on the other hand enables extensions introduced by constant and fast development of computers and electronics. The proposed method, architecture and framework is claimed to support the statement.

The prime property of the *quasar* framework detailed in chapter 4 is the built-in and intrinsic automation of software development and maintenance. It features:

- a method of automation of OPC UA based software development for distributed control systems, which is characterized by:
  - holistic approach to the software creation problem, involving all the aspects from requirements through design, implementation, testing and finally the deployment and post-deployment maintenance,
  - high efficiency of development: the most tedious and laborious part of the software

making process, including writing of „the boilerplate code"[1] gets automated,
  - mild learning curve for developers: the developers do not need to learn complex APIs of communication stacks but instead focus on engineering problems from their own problem domain,
  - high quality of solutions which are created: the framework establishes a number of guarantees to the problem domain, protecting against some classes of programming issues such as data format issues (establishing safe conversions), assuring all execution paths to be well handled (e.g. no uncaught exceptions), etc.
- a complete and integrated framework and software ecosystem:
  - enabling rapid creation of OPC UA servers, clients, SCADA integration tools and other products,
  - based on architecture which favors high-reliability, modularization, ease of maintenance,
  - supporting C++ and Python,
  - supporting variety of deployment modes typical to heterogeneous control systems.

A key part of this thesis is the application of the presented method and the framework for implementation of numerous software components for the control systems of experiments built around the LHC particle accelerator at CERN in Geneva, Switzerland. The system serves a real example of a large scale distributed control system which is a very heterogeneous distributed software-based control system planned for many decades of operation and serving critical tasks.

The research detailed in the thesis was done in the period 2014-2019, with few updates done in 2022.

The direct contributions of the author were:

- study of the technology (the OPC UA standard) as well as related technologies and the prior art,
- initial research on applicability of OPC UA for efficient software creation and preparation of first research software prototypes,
- definition of software architecture, including architecture of OPC UA servers, OPC UA clients, SCADA integration layer and other,
- definition of notation of *quasar design* and its interchange format,
- implementing transformation and code generation engines,
- server-client co-generation facilities and consequently chaining facilities,
- implementation of a build system with support for portability to native builds as well as cross-compilers,
- interfaces to SQL and No-SQL data storage for data archiving and historic data retrieval,
- design and implementation of integration with different programming languages which permitted embedding of OPC UA software components in software packages written in different programming languages, for instance in Python,
- design and implementation of integration with Siemens WinCC OA industry-grade SCADA system,
- design and implementation of various related software components like *open62541-compat*[14] and other.

[1] Source code which is often repetitive and of little intellectual value to the project but still crucial, such as error handling, data type conversions etc.

The evaluation of the method and the framework involved a query to software architects and developers who used the proposed method and framework to design and implement software products. Thus the evaluation was based on real-world software projects.

The evaluation (detailed in chapter 5) shows that two types of software development automation were achieved:

- quantitative

  As demonstrated in section 5.7, the overall automation rate in terms of lines-of-code reduction for OPC UA servers is 10.9x on average, with 3.4x minimum and 42.4x maximum. That is, a developer would have to write 10.9x more executable lines of code (on average) himself/herself to obtain the same functionality[2]. The measure is subjective to the code style of given project. As demonstrated in section 5.3.4, obtaining other products such as OPC UA clients or SCADA integration easements, can easily boost the figure by a significant amount (i.e. +35% and +30% respectively for given example).

- qualitative

  As demonstrated in the chapters 4 and 5, the application of the *quasar architecture* defined by the author:

  - frees the developers of the complexities of OPC UA, letting them speak in terms of higher-level control system constructs which are familiar to them,
  - standardizes creation of multi-component OPC UA-based distributed control systems,
  - raises chances of creation of high-reliability software by shifting aspects such as type safety, fault handling, synchronization and job distribution away from the programmer and handle them in an uniform way.

The quantitative results obtained in the evaluation let assume, that the research thesis, initially stated in chapter 1, was proven.

All of the software products of the thesis were open-sourced and are open to the general public: the *quasar* itself[5] as well as a number of associated pieces of software [14, 83, 75].

The widespread usage of *quasar*-based software engineering in the LHC accelerator experiments control systems which started in 2014 and is well present in 2022 let assume that the proposed method and framework well serve distributed software-based control systems of big industrial/scientific installations that are planned for many decades of operation. In the ATLAS experiment (one of the LHC experiments) in 2022 all but one officially assigned OPC UA server projects are realized using *quasar*-based technologies. The only remaining non-*quasar* based OPC UA server is being re-engineered to be *quasar*-based and planned to enter production environments in mid-2022.

[2] Non-executable lines of code are not included in the measurements

Appendices

Appendix A

Detailed information on projects used for effort evaluation

This appendix extends section 5.7 with additional details such as versions of projects used for evaluation.

Project name	quasar v.	git commit	developers	R.S.	N_C	N_{CV}	N_{SV}	N_M	N_{CE}	n_{ND}	n_{GS}	n_{IC}	n_D	A.R.
ATCA ShelfManager	1.3.3	b1535e5	P.M.	5.6.1	16	178	12	0	15	11817	1003	1313	310	42.4
ATLAS FTK SBC	1.3.4	4525391	I.M.	5.6.5	2	45	0	0	0	2435	92	571	479	6.3
ATLAS LAr LTDB	1.3.0-rc1	239f9b8	E.M.F.	5.6.2	18	89	0	59	14	13111	1394	2085	691	22.0
ATLAS LAr Purity	1.3.4	ed6e202	M.B.	5.6.3	3	25	0	0	0	2018	145	1033	888	3.4
ATLAS Tile HvMicro	1.3.4	f005261	P.G.	5.6.13	5	105	0	0	0	6716	374	1688	1314	6.4
ATLAS Wiener	1.3.3	2.2.0-0	P.N.	5.6.4	8	63	0	0	9	4998	414	1397	983	6.5
HVSys	1.3.3	42c974b	J.O.	5.6.6	6	41	6	0	14	3789	437	951	514	9.2
JCOP CAEN	1.3.3	4e054f7	B.F.	5.6.8	9	12	0	0	11	3151	403	559	156	23.8
JCOP ISEG	1.3.4	76be54a	B.F.	5.6.7	10	19	0	0	12	3319	363	532	169	22.8
JCOP Wiener	1.3.3	41b8963	B.F.	5.6.9	8	4	0	0	11	2351	385	596	211	14.0
Generic IPbus	1.3.4	88a71ba	C.V.S.	5.6.10	4	22	0	0	0	2230	207	621	414	6.9
Generic SNMP	1.3.4	0921431	P.N.	5.3	3	2	2	0	5	1008	175	373	198	7.0
SCA	1.3.4-rc0	1.3.3-rc1	P.N.,P.M.,A.K.	5.4	17	15	9	8	27	5889	968	1653	685	11.0
total	–	–	–	–	–	–	–	–	–	62832	6360	13372	7012	–
average	–	–	–	–	–	–	–	–	–	–	–	–	–	10.9

Table A.1: Legend: quasar v. - used version of *quasar*, git commit - identifier (git commit or tag) of the version used for evaluation, R.S. - reference section in this thesis, N_C - number of classes, N_{CV} - number of cache-variables, N_{SV} - number of source-variables, N_M - number of methods, N_{CE} - number of config-entries, n_{ND} - non-developer code (i.e. totally generated code not supposed to be modified), n_{GS} - generated stubs, n_{IC} - interleaved code, n_D - developer code (without generated part), A.R. - automation ratio, developers - list of initials of developers assigned to given project. n_{ND}, n_{GS}, n_{IC}, n_D are expressed in ELoCs (executable lines of code).

Appendix B

External contributions

This thesis, unless explicitly noted, shows only the author's conceptual and implementation work. However in certain places in this thesis it was needed to include work done together with other contributors to keep the logical flow understandable and complete. Such contributions are listed below. The author acknowledges the listed collaborators and expresses gratitude for their contributions.

- support of array data types - the author acknowledges his colleagues Benjamin Farnham and Michael Ludwig for collaboration on this topic. Benjamin Farnham wrote a formal specification of arrays in *quasar*, Michael Ludwig performed significant implementation and validation work on arrays, and the author of this thesis performed conceptual and implementation work of integrating the arrays in the *quasar* project.
- calculated variables - the author acknowledges his colleague Viatcheslav Filimonov for the inspiration of idea of "Calculated Items" from his project named *CANopen OPC UA server*[17]. Nevertheless, the design and implementation of *calculated variables* in the *quasar* project was done by the author of this thesis and it is detailed in the section 4.9.1.11.
- use cases for *singleVariableNode* mapping were proposed by Benjamin Farnham for projects: CAEN OPC UA server (see section 5.6.7), ISEG OPC UA server (see section 5.6.8), JCOP Wiener OPC UA server (see section 5.6.9). The mapping was implemented by the author of this thesis.
- the idea for modeling device-logic synchronization directly in the *quasar design* (i.e. *mutex* element of *quasar design*) was proposed by Stefan Schlenker and further elaborated by the author of this thesis.
- the idea to split device-logic classes into base device-logic classes and the factual device-logic classes was proposed by Benjamin Farnham and implemented by the author of this thesis.
- the proposal of *storedInDeviceObject* feature was proposed by Benjamin Farnham and further elaborated by the author of this thesis.

Bibliography

[1] W. Mahnke, S.-H. Leitner, and M. Damm. *OPC Unified Architecture*. Berlin: Springer, 2009.

[2] International Electrotechnical Commission. *OPC Unified Architecture - Part 1: Overview and concepts*. Technical report IEC TR 62541-1:2016. International Electrotechnical Commission, Oct. 2016. URL: https://webstore.iec.ch/publication/25997.

[3] P. P. Nikiel and K. Korcyl. "Object Mapping in the OPC-UA Protocol for Statically and Dynamically Typed Programming Languages". In: *Computing and Informatics* 37.4 (2018). ISSN: 2585-8807. URL: http://www.cai.sk/ojs/index.php/cai/article/view/2018_4_946/915.

[4] D. Strutzenberger, T. Frühwirth, T. Trautner, R. Hinterbichler, and F. Pauker. "Communication interface specification in OPC UA". In: *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019, pp. 1329–1332.

[5] *quasar: Quick OPC-UA Server Generation Framework*. the quasar team. 2019. URL: https://github.com/quasar-team/quasar (visited on 03/31/2019).

[6] P. P. Nikiel, B. Farnham, V. Filimonov, and S. Schlenker. "Generic OPC UA Server Framework". In: *J. Phys.: Conf. Ser.* 664.8 (2015), 082039. 8 p.

[7] P. P. Nikiel, B. Farnham, S. Schlenker, C.-V. Soare, V. Filimonov, and D. Abalo Miron. "quasar - A Generic Framework for Rapid Development of OPC UA Servers". In: *Proceedings of ICALEPCS2015, Melbourne, Australia*. 2015, WEB3O02. 4 p. DOI: 10.18429/JACoW-ICALEPCS2015-WEB3O02.

[8] Knowledge Transfer Group. *CERN Knowledge Transfer Report 2016*. Annual report. CERN, 2016. URL: https://kt.cern/sites/knowledgetransfer.web.cern.ch/files/file-uploads/annual-report/knowledge-transfer-report-2016.pdf.

[9] *High Voltage Power Supply(HVPS) - iseg Germany*. iseg Spezialelektronik GmbH. 2019. URL: https://iseg-hv.com (visited on 02/16/2019).

[10] CAEN S.p.A. *Power Supply - CAEN - Tools for Discovery*. 2019. URL: https://www.caen.it (visited on 02/16/2019).

[11] W-IE-NE-R Power Electronics. *Welcome to WIENER | Electronics for Research and Industry*. 2019. URL: http://www.wiener-d.com/ (visited on 02/16/2019).

[12] B. Farnham, F. Varela, and N. Ziogas. "A homogenous approach to CERN/vendor collaboration projects for building OPC-UA servers". In: *Proceedings, 16th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2017): Barcelona, Spain, October 8-13, 2017*. 2018, THPHA009. DOI: 10.18429/JACoW-ICALEPCS2017-THPHA009.

[13] Linux Foundation. *Yocto Project – It's not an embedded Linux distribution – it creates a custom one for you*. 2019. URL: https://www.yoctoproject.org/ (visited on 05/25/2019).

[14] P. P. Nikiel et al. *open62541-compat: adapts open62541 API to Unified Automation C++ OPC-UA Toolkit API*. the quasar team. 2019. URL: https://github.com/quasar-team/open62541-compat (visited on 05/14/2019).

[15] M. Arruat, L. Fernandez, S. Jackson, F. Locci, J.-L. Nougaret, M. Peryt, A. Radeva, M. Sobczak, and M. Vanden Eynden. "Front-End Software Architecture". In: *11th International Conference on Accelerator and Large Experimental Physics Control Systems, Knoxville, TN, USA*. 2008, 3 p.

[16] Katarina Sigerud. *FESA – Front-End Software Architecture*. CERN. 2018. URL: https://be-dep-co.web.cern.ch/content/fesa (visited on 04/14/2019).

[17] P. P. Nikiel, B. Farnham, S. Franz, S. Schlenker, H. Boterenbrood, and V. Filimonov. "OPC Unified Architecture within the Control System of the ATLAS Experiment". In: *Proceedings of ICALEPCS2013, San Francisco, CA, USA*. Mar. 2014, MOPPC032.

[18] M. Henning and S. Vinoski. *Advanced CORBA programming with C++*. Addison-Wesley Professional Computing Series. Boston, MA: Addison-Wesley, 1999. ISBN: 9780201379273.

[19] A. S. Tanenbaum and M. Van Steen. *Distributed Systems: Principles and Paradigms; 2nd ed.* Upper Saddle River, NJ: Prentice-Hall, 2007. ISBN: 9780132392273.

[20] A. S. Tanenbaum and D. Wetherall. *Computer networks; 5th ed.* Boston: Prentice Hall, 2011. URL: https://cds.cern.ch/record/1598499.

[21] L. H. Etzkorn. *Introduction to Middleware: web services, object components, and cloud computing*. Milton: CRC Press, 2017.

[22] International Telecommunication Union. *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), Recommendation ITU-T X.690, also known as ISO 8825-1*. Tech. rep. 2015.

[23] World Wide Web Consortium. *Extensible Markup Language (XML) 1.1 (Second Edition)*. 2006. URL: https://www.w3.org/TR/2006/REC-xml11-20060816/ (visited on 03/16/2019).

[24] *The JavaScript Object Notation (JSON) Data Interchange Format*. Tech. rep. Internet Engineering Task Force (IETF), 2017. URL: https://tools.ietf.org/pdf/std90.pdf.

[25] P. Hintjens. *ZeroMQ: messaging for many applications*. Sebastopol, CA: O'Reilly, 2013. ISBN: 9781449334062.

[26] N. Brown and C. Kindel. *Distributed Component Object Model Protocol – DCOM/1.0*. Tech. rep. Microsoft Corporation, 1998. URL: https://tools.ietf.org/html/draft-brown-dcom-v1-spec-03.

[27] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. Tech. rep. Internet Engineering Task Force (IETF), 1999. URL: https://tools.ietf.org/pdf/rfc2616.pdf.

[28] R. Miles and K. Hamilton. *Learning UML 2.0. A programatic introduction to UML*. Sebastopol, CA: O'Reilly, 2006. ISBN: 0596009828.

[29] L. Delligatti. *SysML distilled: a brief guide to the systems modeling language*. Upper Saddle River, NJ: Addison-Wesley, 2014. ISBN: 9780133430356.

[30] B. A. Lieberman. *The art of software modeling*. Boca Raton, FL: Auerbach, 2007. ISBN: 978-1-420-04462-1.

[31] J. Herrington. *Code generation in action*. Greenwich, CT: Manning Publ., 2003. ISBN: 978-1-930-11097-7.

[32] T. Hoffmann, M. Schwickert, and G. Janša. "FESA at FAIR - The Front-End Software Architecture". In: *Particle accelerator. Proceedings, 23rd Conference, PAC'09, Vancouver, Canada, May 4-8, 2009*. 2010, FR5REP009.

[33] OPC Foundation. *UA Nodeset*. 2019. URL: https://github.com/OPCFoundation/UA-Nodeset (visited on 05/25/2019).

[34] *open62541: an open source implementation of OPC UA*. 2018. URL: https://open62541.org/ (visited on 05/14/2019).

[35] Unified Automation GmbH. *C++ Based OPC UA Client and Server SDK (Bundle)*. 2019. URL: https://www.unified-automation.com/products/server-sdk/c-ua-server-sdk.html (visited on 05/28/2019).

[36] Unified Automation GmbH. *UaModeler*. 2017. URL: https://www.unified-automation.com/products/development-tools/uamodeler.html (visited on 05/31/2019).

[37] R. C. Dorf and R. H. Bishop. *Modern control systems; 12th ed.* Upper Saddle River, NJ: Pearson, 2011. ISBN: 9780131383104.

[38] P. H. Petkov, T. N. Slavov, and J. K. Kralev. *Design of embedded robust control systems using Matlab/Simulink*. Institution of Engineering & Technology, 2018. ISBN: 9781785613302.

[39] The MathWorks, Inc. *MATLAB Coder - MATLAB*. 2019. URL: `https://ch.mathworks.com/products/matlab-coder.html` (visited on 04/04/2019).

[40] ETM professional control GmbH. *Modern, efficient and flexible SIMATIC WinCC Open Architecture V3.15*. Tech. rep. 6ZB5370-1EG02-0BA0-V2. ETM professional control GmbH, a Siemens Company, Jan. 2017. URL: `https://siemens.com/wincc-open-architecture`.

[41] National Instruments. *What Is LabVIEW?* 2019. URL: `http://www.ni.com/en-us/shop/labview.html` (visited on 04/04/2019).

[42] J. Essick. *Hands-on introduction to LabVIEW for scientists and engineers; 4th ed.* Oxford: Oxford University Press, 2018. ISBN: 9780190853068.

[43] IEEE. "Systems and software engineering – Vocabulary". In: *ISO/IEC/IEEE 24765:2017(E)* (Sept. 2017), pp. 1–418. DOI: `10.1109/IEEESTD.2010.5733835`.

[44] A. Butterfield, G. Ngondi, and A. Kerr. *A Dictionary of Computer Science*. Oxford Paperback Reference. Oxford University Press, 2016. ISBN: 9780199688975. URL: `https://books.google.fr/books?id=GDgICwAAQBAJ`.

[45] M. Gregoire. *Professional C++; 4th ed.* Newark, NJ: John Wiley and Sons, 2018. ISBN: 978-1-119-42122-1.

[46] G. Fleishman. *On the past, present, and future of COBOL – Increment: Programming Languages*. 2018. URL: `https://increment.com/programming-languages/cobol-all-the-way-down/` (visited on 05/26/2019).

[47] M. Shepperd. *Fundamentals of software measurement*. Hemel Hempstead: Prentice-Hall, 1995. ISBN: 9780133361995.

[48] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. San Francisco, CA: Morgan and Kaufmann, 2007. ISBN: 9780123725011.

[49] A. Alexandrescu. *The D programming language*. Upper Saddle River, NJ: Addison-Wesley Professional, 2010. ISBN: 9780321635365.

[50] Google Inc. *Google: Protocol Buffers*. 2019. URL: `https://developers.google.com/protocol-buffers` (visited on 05/31/2019).

[51] The kernel development community. *Industrial I/O - The Linux Kernel documentation*. 2017. URL: `https://www.kernel.org/doc/html/v4.12/driver-api/iio/index.html` (visited on 02/23/2019).

[52] Analog Devices, Inc. *A cross platform library for interfacing with local and remote Linux IIO devices*. 2019. URL: `https://github.com/analogdevicesinc/libiio` (visited on 02/23/2019).

[53] CAN in Automation (CiA). *CAN lower- and higher-layer protocols*. 2019. URL: `https://www.can-cia.org/can-knowledge/` (visited on 05/31/2019).

[54] Linux kernel developers. *Controller Area Network Protocol Family (aka SocketCAN)*. 2017. URL: `https://www.kernel.org/doc/Documentation/networking/can.txt` (visited on 02/23/2019).

[55] libsocketcan developers. *libsocketcan*. 2018. URL: `https://lalten.github.io/libsocketcan/` (visited on 02/23/2019).

[56] libmodbus.org. *A Modbus library for Linux, Mac OS X, FreeBSD, QNX and Win32*. 2018. URL: `https://libmodbus.org` (visited on 04/04/2019).

[57] D. R. Mauro and K. J. Schmidt. *Essential SNMP; 2nd ed.* Sebastopol, Calif.: O'Reilly, 2005. ISBN: 0596008406.

[58] The Net-SNMP project. *Net-SNMP*. 2013. URL: http://www.net-snmp.org/ (visited on 04/04/2019).

[59] ETM professional control GmbH. *WinCC OA Online help, v.3.15*. 2018.

[60] D. Gasevic, D. Djuric, V. Devedzic, J. Bézivin, and B. Selic. *Model driven engineering and ontology development; 2nd ed.* Berlin: Springer, 2009. ISBN: 978-3-642002816.

[61] R. W. Sebesta. *Concepts of programming languages; 7th ed.* Boston, MA: Pearson, 2006. ISBN: 9780321312518.

[62] J. Clark and D. A. Holton. *A first look at graph theory*. Singapore: World Scientific, 1991. ISBN: 9789810204907.

[63] World Wide Web Consortium. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. 2012. URL: https://www.w3.org/TR/xmlschema11-1/ (visited on 03/16/2019).

[64] World Wide Web Consortium. *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. 2012. URL: https://www.w3.org/TR/xmlschema11-2/ (visited on 03/16/2019).

[65] Eclipse Foundation, Inc. *Eclipse XML Editors and Tools*. 2016. URL: https://marketplace.eclipse.org/content/eclipse-xml-editors-and-tools-0 (visited on 03/16/2019).

[66] M. Kay. *XSLT 2.0 programmer's reference; 3rd ed.* Indianapolis, IN: Wiley, 2004. ISBN: 978-0-764-56909-8.

[67] Graphviz. *Graphviz - Graph Visualization Software*. May 22, 2019. URL: https://www.graphviz.org/ (visited on 06/01/2019).

[68] World Wide Web Consortium. *XML Path Language (XPath) 2.0 (Second Edition)*. 2010. URL: https://www.w3.org/TR/xpath20/ (visited on 03/19/2019).

[69] Saxonica. *Saxonica: XSLT and XQuery processing*. 2019. URL: http://saxonica.com/welcome/welcome.xml (visited on 05/31/2019).

[70] D. Veillard. *The XSLT C library for GNOME - libxslt*. 2017. URL: http://xmlsoft.org/XSLT/ (visited on 03/18/2019).

[71] *Artistic Style 3.1: A Free, Fast, and Small Automatic Formatter for C, C++, C++/CLI, Objective-C, C#, and Java Source Code*. 2017. URL: http://astyle.sourceforge.net/ (visited on 04/05/2019).

[72] Kitware, Inc. *CMake*. 2019. URL: https://cmake.org/ (visited on 04/05/2019).

[73] A. Ronacher. *Welcome | Jinja2*. 2014. URL: http://jinja.pocoo.org/ (visited on 03/24/2019).

[74] M. Grinberg. *Flask web development: developing web applications with Python; 2nd ed.* Sebastopol, CA: O'Reilly Media, 2018. ISBN: 9781491991732.

[75] P. P. Nikiel. *Poverty, a quasar module for trivial integration of quasar servers into Python*. 2019. URL: https://github.com/quasar-team/Poverty (visited on 05/14/2019).

[76] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series. Boston, MA: Addison-Wesley, 1995. ISBN: 9780201633610.

[77] A. Williams. *C++ concurrency in action: practical multithreading*. Shelter Island, NY: Manning Publ., 2012. ISBN: 9781933988771.

[78] Code Synthesis Tools CC. *C++/Tree Mapping Getting Started Guide*. 2018. URL: http://www.codesynthesis.com/projects/xsd/documentation/cxx/tree/guide/ (visited on 03/25/2019).

[79] E. R. Harold. *Processing XML with Java: a guide to SAX, DOM, JDOM, JAXP, and TrAX*. Boston: Addison-Wesley, 2003. ISBN: 0201771861.

[80] P. P. Nikiel. *Calculated Variables*. Available from quasar repository: CalculatedVariables/doc/-CalculatedVariables.html. Nov. 2018.

[81] I. Berg et al. *beltoforion/muparser: official repository of the muparser fast math parser library*. 2018. URL: https://github.com/beltoforion/muparser (visited on 05/27/2019).

[82] Václav Slavětínský. *Vizualizace XML schémat*. 2008. URL: http://xsdvi.sourceforge.net/ (visited on 03/30/2019).

[83] Nikiel, Piotr P. *UaObjects client generation facility for C++, for Quasar*. 2019. URL: https://github.com/quasar-team/UaoForQuasar (visited on 04/03/2019).

[84] M. Lutz. *Python: pocket reference; 5th ed.* Sebastopol, CA: O'Reilly, 2014. URL: https://cds.cern.ch/record/1670689.

[85] P. P. Nikiel. *UaObjects for Python*. 2017. URL: https://gitlab.cern.ch/pnikiel/UaObjects (visited on 05/15/2019).

[86] W. Pessemier, G. Deconinck, G. Raskin, P. Saey, and H. Van Winckel. "UAF: a generic OPC unified architecture framework". In: *Proc. SPIE* 8451 (2012), 84510P.

[87] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms, second edition*. MIT electrical engineering and computer science series. MIT Press, 2001. ISBN: 9780262531962.

[88] Unified Automation GmbH. *UaExpert – A Full-Featured OPC UA Client*. 2019. URL: https://www.unified-automation.com/products/development-tools/uaexpert.html (visited on 05/31/2019).

[89] N. Fenton and J. Bieman. *Software metrics: a rigorous and practical approach; 3rd ed.* Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. Hoboken, NJ: Taylor and Francis, 2014. ISBN: 9781439838235.

[90] P. P. Nikiel. *QuasarMetrics*. 2019. URL: https://gitlab.cern.ch/pnikiel/QuasarMetrics.

[91] J. Black. *The system engineer's handbook: a guide to building VMEbus and VXIbus systems*. San Diego, CA: Academic Press, 1992. ISBN: 9780121028206.

[92] A. Ruben and A. Köster. *CAN-BUS Interface for W-IE-NE-R Crate Remote Control*. W-IE-NE-R Plein & Baus GmbH, Jan. 25, 1996.

[93] PICMG. *AdvancedTCA® | PICMG*. 2019. URL: https://www.picmg.org/openstandards/advancedtca/ (visited on 05/13/2019).

[94] K. Wyllie, S. Baron, S. Bonacini, Ö. Çobanoğlu, F. Faccio, S. Feger, R. Francisco, P. Gui, J. Li, A. Marchioro, P. Moreira, C. Paillard, and D. Porret. "A Gigabit Transceiver for Data Transmission in Future High Energy Physics Experiments". In: *Phys. Procedia* 37 (2012), 1561–1568. 8 p. DOI: 10.1016/j.phpro.2012.02.487.

[95] A. Caratelli, S. Bonacini, K. Kloukinas, A. Marchioro, P. Moreira, R. De Oliveira, and C. Paillard. "The GBT-SCA, a radiation tolerant ASIC for detector control and monitoring applications in HEP experiments". In: *JINST* 10.03 (2015), p. C03034.

[96] S. Bonacini, A. Caratelli, R. Francisco, K. Kloukinas, A. Marchioro, P. Moreira, and C. Paillard. *GBT-SCA: The Slow Control Adapter ASIC for the GBT system, user manual*. Version 8.2. CERN. Apr. 2017.

[97] W. Buchanan. *Applied data communications and networks*. Boston, MA: Springer, 1996. ISBN: 9781461312079.

[98] S. Bonacini, K. Kloukinas, and P. Moreira. "E-link : A Radiation-Hard Low-Power Electrical Link for Chip-to-Chip Communication". In: *Proceedings, Topical Workshop on Electronics for Particle Physics (TWEPP09)*. CERN. CERN, 2009. DOI: 10.5170/CERN-2009-006.422.

[99] J. A. Dell. *Digital interface design and application*. Chichester: Wiley, 2015. ISBN: 9781118974353.

[100] Texas Instruments Incorporated. *Application note SSYA002C: IEEE Std 1149.1 (JTAG) Testability Primer*. SSYA002C. 1997.

[101] P. Moschovakos. *Software support for the SCA ASIC*. Oct. 14, 2016. URL: https://indico.cern.ch/event/576215/contributions/2335368/attachments/1355482/2048404/SCA_NSW_Software.pdf.

[102] J. Narevicius, J. T. Anderson, A. Borga, H. Boterenbrood, H. Chen, K. Chen, G. Drake, M. Donszelmann, D. Francis, B. Gorini, D. Guest, F. Lanni, G. Lehmann Miotto, L. Levinson, A. Roich, F. P. Schreuder, J. Schumacher, W. Vandelli, and J. Zhang. *FELIX: The New Approach for Interfacing to Front-end Electronics for the ATLAS Experiment*. Tech. rep. ATL-DAQ-PROC-2016-009. Geneva: CERN, May 2016. DOI: 10.1109/RTC.2016.7543142.

[103] P. P. Nikiel, P. Moschovakos, S. Schlenker, H. Boterenbrood, and A. Koulouris. *ScaSoftware Gitlab repository*. 2019. URL: https://gitlab.cern.ch/atlas-dcs-common-software/ScaSoftware (visited on 05/12/2019).

[104] J. Schumacher, C. Plessl, and W. Vandelli. "High-Throughput and Low-Latency Network Communication with NetIO". In: *Journal of Physics: Conference Series* 898 (Oct. 2017), p. 082003. DOI: 10.1088/1742-6596/898/8/082003.

[105] W. R. Stevens, B. Fenner, and A. M. Rudoff. *Unix network programming. v.1; 3rd ed.* Boston, MA: Addison-Wesley, 2004. ISBN: 9780131411555.

[106] A. Rigo and M. Fijalkowski. *C Foreign Function Interface for Python, CFFI documentation, CFFI 1.12.3*. 2019. URL: https://cffi.readthedocs.io/ (visited on 04/27/2019).

[107] The SWIG team. *Simplified Wrapper and Interface Generator*. 2019. URL: http://www.swig.org (visited on 04/27/2019).

[108] D. Abrahams and S. Seefeld. *Boost.Python*. 2019. URL: https://www.boost.org/doc/libs/1_66_0/libs/python (visited on 04/27/2019).

[109] K. Chen, H. Chen, M. Citterio, H. Deschamps, A. Grabas, S. Latorre, M. Lazzaroni, H. Liu, P. Schwemling, S. Simion, H. Xu, and H. Zhu. "Design and Evaluation of LAr Trigger Digitizer Board in the ATLAS Phase-I Upgrade". In: *21st IEEE Real Time Conference (RT2018) Williamsburg, Virginia, June 11-15, 2018*. June 2018. arXiv: 1806.08046.

[110] HVSys company. *Multi-channel High Voltage System Controller SC508. User Manual (v.100406)*. HVSys, Dubna, Russia, 2006.

[111] HVSys company. *High Voltage Cell CA2K. User Manual - option for TRT, Atlas Experiment (v.190106)*. HVSys, Dubna, Russia, 2006.

[112] C. G. Larrea, K. Harder, D. Newbold, D. Sankey, A. Rose, A. Thea, and T. Williams. "IPbus: a flexible Ethernet-based control system for xTCA hardware". In: *Journal of Instrumentation* 10.02 (Feb. 2015), pp. C02019–C02019. DOI: 10.1088/1748-0221/10/02/c02019.

[113] *IPbus: Software that implements a reliable high-performance control link for particle physics electronics, based on the IPbus protocol*. 2019. URL: https://ipbus.web.cern.ch/ipbus/ (visited on 05/27/2019).

[114] OPC Foundation. *AnsiC version for OPC UA*. 2018. URL: https://github.com/OPCFoundation/UA-AnsiC-Legacy (visited on 05/14/2019).

[115] M. F. Lungu. "Reverse engineering software ecosystems". PhD thesis. Università della Svizzera italiana, 2009.

Chapter 7

Dictionary of terms

ADC analogue-digital converter.

address-space class a class (e.g. in the C++ programming language) which represents the OPC UA perspective of a *quasar class*. See section 4.9.1.4.

COTS common-of-the-shelf, opposite to: custom.

CTL programming language the default programming language of Siemens WinCC OA used to express behavior and actions of control systems realized in WinCC OA.

device-logic class a class (e.g. in the C++ programming language) which represents the behavior perspective of a *quasar class*. See section 4.9.1.6.

DeviceRoot a singleton class (thus also an object) representing top-level element of device-logic objects in a *quasar server instance*.

ELoC, executable line-of-code an unit of source code measurement based on lines of code, in which lines not translating into computer program (e.g. comments, blanks) are not counted.

HAL, hardware access/abstraction library a software component providing an application programming interface to exchange data with a device.

LoC, line-of-code an unit of source code measurement based on lines of code. Lines-of-code metric typically involves all lines of code including comments and blank lines. See ELoC for improved measure.

OPC UA protocol stack a software-based implementation of the OPC UA protocol, typically distributed as a software library. Commonly used OPC UA protocol stacks are, for instance, open62541[34] and the OPC Foundation ANSI C protocol stack[114].

OPC UA server a computer program able to exchange information with *OPC UA clients* using the *OPC UA protocol*[1]. The OPC UA server has and exposes an OPC UA address-space.

quasar architecture software architecture (described in this thesis) applicable to OPC UA servers and clients built using *quasar*.

quasar class a type of control system object modeled using *quasar*. See section 4.5.1.

quasar config-entry a piece of configuration data belonging to a *quasar class*. See section 4.5.5.

quasar design a model of a control system (or its part) which was made using *quasar notation* and is stored in a format standardized by *quasar*. The *quasar design* is usually composed of *quasar classes* and relations between them.

quasar design root an element of *quasar design* which symbolizes top-level element of a server. From the address-space perspective it corresponds to *OPC UA Objects Folder*[1, 2]. From the XML configuration perspective it corresponds to the top-level element of the XML document.

quasar server a software product delivering functionality of an OPC UA server which was created using *quasar* framework.

quasar server instance a *quasar server* being executed on a given computer and serving OPC UA server functionality (e.g. maintaining connections to OPC UA clients).

SCADA software Supervisory Control and Data Acquisition software. A type of software often used in different industries, facilities (power plants, manufacturing plants), research labs etc. for monitoring, controls and data acquisition.

SDK, software development kit a software library which provides higher layer software support for given problem. In the case of OPC UA it is the API to OPC UA built on top of the protocol stack (for example, it encapsulates protocol-defined data types into convenience classes, etc.). Common examples of OPC UA software development kits are the UASDK developed by Unified Automation [35] as well as open62541-compat[14] which is an open-source project started by the author of this thesis, which builds such a SDK on top of open62541[34] protocol stack.

software ecosystem a collection of software projects which are developed and which co-evolve together in the same environment [115]

RAS, reliability, availability and serviceability aspects of designing, implementing and operating computer systems related to their reliability and operational up-time.

verb method a method concretizing one of verbs chosen for *quasar* (e.g. set, get, read, write, call, etc.) for given element of *quasar design*.