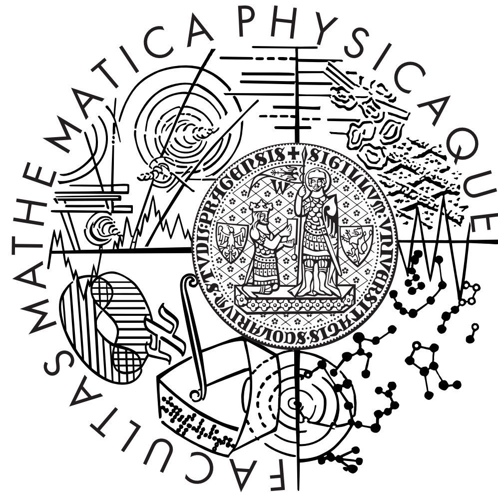
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Petr Štěpán

An extensible attribute framework for ProCom

Department of Software Engineering

Supervisor: RNDr. Tomáš Bureš, Ph.D.
Study Program: Computer Science, Software Systems

I would like to express my thanks to Séverine Sentilles and Jan Carlson for their guidance on my thesis — for fruitful discussions about my work and their numerous suggestions for its improvement as well as for their helpfulness in organizational matters.

I also wish to thank my supervisor Tomáš Bureš for opening up the opportunity for writing my thesis at Mälardalen University and helping me with the finalization of the thesis.

I hereby declare that I have elaborated my master thesis on my own and listed all used references. I agree with making the thesis publicly available.

Prague, March 12, 2009 Petr Štěpán

Contents

1 Introduction **1**
- 1.1 Goals of the thesis . . . 2
- 1.2 Structure of the thesis . . . 2

2 Background **3**
- 2.1 Theoretical background . . . 3
  - 2.1.1 Embedded systems . . . 3
  - 2.1.2 Component-based development . . . 4
  - 2.1.3 Progress . . . 5
  - 2.1.4 ProCom . . . 6
- 2.2 Technological background . . . 7
  - 2.2.1 Eclipse Platform . . . 7
  - 2.2.2 Progress IDE . . . 9
  - 2.2.3 Eclipse Modeling Framework . . . 9

3 Problem Analysis **11**
- 3.1 Attributes in the development process envisioned by Progress . . . 11
- 3.2 General attribute requirements . . . 12
- 3.3 Actors working with attributes . . . 13
- 3.4 Requirement specification . . . 14
  - 3.4.1 Common attribute requirements . . . 14
  - 3.4.2 Terminology clarification . . . 15
  - 3.4.3 IDE users requirements . . . 15
  - 3.4.4 Module developers requirements . . . 16
  - 3.4.5 Attribute contributors requirements . . . 17
- 3.5 Use cases . . . 18
  - 3.5.1 Use cases associated with an IDE user . . . 18
  - 3.5.2 Use cases associated with an IDE module developer . . . 19
  - 3.5.3 Use cases associated with an attribute contributor . . . 19

4 Solution Design **20**
- 4.1 Integration with the ProCom metamodel . . . 20
- 4.2 Attribute structure . . . 21
- 4.3 Attribute value structure . . . 23
  - 4.3.1 Conceptual structure . . . 23
  - 4.3.2 The implemented design . . . 24
  - 4.3.3 Rejected alternative designs . . . 26
  - 4.3.4 Possible alternative solution: Named extensible metadata . . . 30
- 4.4 Type extensibility . . . 31
  - 4.4.1 EMF-based methods . . . 31
  - 4.4.2 Externally serialized data . . . 32
- 4.5 GUI . . . 33
  - 4.5.1 Attribute View . . . 33
  - 4.5.2 Editors and viewers . . . 35
  - 4.5.3 Help . . . 38
- 4.6 Client API . . . 38
- 4.7 SPI . . . 40
  - 4.7.1 Attribute definition mechanism . . . 40

4.7.2 Attribute specification . . . 41

5 Prototype description **44**
5.1 Scope . . . 44
5.2 Implementation overview . . . 46
5.2.1 The framework core plugin . . . 46
5.2.2 The framework GUI plugin . . . 47
5.3 Demonstration . . . 48
5.3.1 Attribute contribution . . . 48
5.3.2 Prototype GUI . . . 52

6 Related work **56**

7 Conclusion **58**

8 Bibliography **60**

A Contents of the enclosed CD-ROM **62**

List of Figures

2 Background

1 ProSys subsystems communicating using a message channel [22] 6
2 ProSave components realizing a ProSys subsystem 7
3 Eclipse Platform architecture 8

3 Problem Analysis

4 Various information included in component's attributes 13
5 Actors in the attribute framework and their associated use cases 18

4 Solution Design

6 `Attributable` and its role in the attribute metamodel 21
7 Main entities forming an attribute 22
8 Attribute value structure 24
9 Attribute of the worst-case execution time (WCET) between two groups of a service 26
10 All-in-one attribute design exhibiting redundancy 27
11 Attribute design leading to type definition language 27
12 Combining data in an attribute value using multiple inheritance 28
13 Extensible metadata in attribute value combined using composition 29
14 Recursive attributes design 30
15 Named extensible metadata design 31
16 Displaying attribute values comprising of extensible metadata 34
17 Displaying attribute values having fixed metadata 35
18 Interfaces for attribute editors 36
19 Interfaces for attribute viewers 37
20 The client API implemented by the methods of `Attributable` 39
21 Attribute value proxies 39
22 Sample extension of the attribute contribution extension point 41

5 Prototype description

23 Simplified code of the validator 49
24 Simplified code of the reference provider 49
25 Contribution mark-up for the WCET between two port groups of a service . . . 50
26 Simplified implementation of the image data type 51
27 Serialization support for the `ImageData` class 51
28 Viewer displaying an image in an external application 52
29 Integration of the Attribute View and the model editor 52
30 Attribute View 53
31 Managed attribute viewer (the bitmap image viewer) 53
32 Various types of attribute editors 54
33 Assistance features overview 55

Název práce: An extensible attribute framework for ProCom
Autor: Petr Štěpán
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Tomáš Bureš, Ph.D.
E-mail vedoucího: bures@dsrg.mff.cuni.cz

Abstrakt:
Tato práce se zabývá konceptem atributů v ProComu, komponentovém modelu vyvíjeném v rámci výzkumu centra The PROGRESS Centre for Predictable Embedded Software Systems. Atributy jsou informace různých typů a úrovní abstrakce asociované v průběhu vývoje systému s entitami ProComu.
Na základě analýzy procesu vývoje, jak jej předvídá PROGRESS, jsou identifikovány požadavky na atributy entit ProComu a jsou analyzovány různé způsoby jejich realizace. Vybrané řešení přináší vysoce strukturované, vícehodnotové a rozšiřitelné atributy.
Práce dále obsahuje návrh a prototypovou implementaci atributového frameworku, který realizuje navržené koncepty a dokazuje tak jejich životaschopnost. Framework pokrývá potřeby všech, kteří v průběhu vývoje systému pracují s atributy: nabízí rozšiřitelné, modulární GUI pro zobrazování a úpravu potenciálně velmi komplexních informací uchovávaných v atributech, rozhraní pro přístup k atributům z jiných programových modulů a dobře definované mechanismy rozšiřování množiny atributů o nové atributy, jejich typy a prostředky pro manipulaci s jejich hodnotami. Framework je integrován do PROGRESS IDE, hlavního nástroje, který podporuje vývoj aplikací podle PROGRESSu.

Klíčová slova: atribut, komponentový model, mimofunkční vlastnost, spolehlivé systémy

Title: An extensible attribute framework for ProCom
Author: Petr Štěpán
Department: Department of Software Engineering
Supervisor: RNDr. Tomáš Bureš, Ph.D.
Supervisor's e-mail address: bures@dsrg.mff.cuni.cz

Abstract:
This thesis is focused on the attributes concept of ProCom, a component model developed within The PROGRESS Centre for Predictable Embedded Software Systems. Attributes are pieces of information of various types and levels of abstraction associated with the ProCom entities during the development of a system.
Based on the analysis of the development process envisioned by PROGRESS, the requirements for the attributes of ProCom entities are identified, and various alternatives of realizing attributes are analyzed. The chosen solution of highly structured, multi-valued, and extensible attributes is elaborated.
The thesis also consists of the design and the prototype implementation of an attribute framework realizing and proving the feasibility of the proposed concepts. The framework addresses the needs of all actors involved in working with attributes throughout the development of a system: It provides an extensible, modular GUI for viewing and editing possibly highly complex information contained in atributes, an interface for the programmatic access to attributes, and well-defined mechanisms for extending the attribute pool by new attributes, new attribute types, and means for their manipulation. The framework is integrated into the PROGRESS IDE, the main tool supporting the PROGRESS development.

Keywords: attribute, component model, dependable systems, non-functional property

Chapter 1

Introduction

In recent years the role of embedded systems[1] has grown. They surround us in our lives, often unnoticed in almost every electronic device we use. They can be found in consumer electronics, safety and control systems in vehicles, telecommunication, industrial automation, medical, robotics, military and many other systems [19].

In 1996 it was estimated that an average American came into contact with 60 microprocessors a day [19]. However, due to improving technologies and dramatical decreases in hardware prices, smaller and at the same time more capable devices are unceasingly coming to the market in greater numbers. This ubiquitous proliferation of embedded systems moves us to the *post-PC era*, where more and more information processing is being performed by embedded systems distributed in our surroundings rather than in our PCs. In the light of these facts, the statement of the journalist Mary Ryan does not seem so much exaggerated:

> ... embedded chips form the backbone of the electronic world in which we live ...

Despite the growing proliferation of embedded systems, their development is a challenging task since they are required to meet many strict resource limitations in terms of performance, consumed energy and space. Especially in the domain of control-intensive systems, where the cost of a system failure is significant or even critical due to their interaction with the physical world, embedded systems have to be *dependable* (correct, reliable, robust, high-performing, etc.), often working under tight real-time constraints. All of the aforementioned requirements are reflected in the development of these software systems, where low resource consumption and formally proved correctness are of a major importance, leading to monolithic, platform-oriented, not reusable and hardly maintainable software systems. Embedded system development is starting to become the bottleneck of their further growth in terms of increasing complexity of systems and its high cost.

Progress, a Swedish national research centre for applied research in development of predictable embedded software, has been established to devise theories and techniques dealing with the complexity of embedded systems and their development. The approach taken by Progress consists in employing the well-known development paradigm of component-based development (CBD) successfully applied in other areas of software. However, CBD process has to be adjusted to the specifics and hard requirements of embedded systems; new procedures, methods and appropriate tools have to be devised and implemented.

One of the cornerstones of the Progress approach is a component model, called ProCom (Progress Component Model), whose design reflects the needs of development process proposed by Progress and which provides the framework supporting the other key activities emphasized by Progress: analysis, verification, and deployment. All these activities demand some information to be associated with components, serving as their inputs or outputs. This feature of

[1] Embedded systems are information processing systems performing a limited set of specialized operations, usually being part of some bigger devices (ergo embedded).

the component model is furnished by *the attributes concept*. This thesis should elaborate the notion of an attribute in ProCom by designing its structure and refining its semantics.

However, the development process envisioned by PROGRESS and supported by ProCom is so complex that it is not feasible without proper and massive tool support. Therefore, one of the goals of PROGRESS lies in developing an integrated development environment, called PROGRESS IDE, providing this tool support. It is intended to be an integration platform, where various tools and methods coming from the PROGRESS research should be incorporated to become an ultimate tool backing up the whole development process.

Consequently, this thesis also proposes the design and implementation of an attribute framework integrated into the PROGRESS IDE. The framework aims at providing IDE users with a comfortable means for viewing and modifying component attributes of various types of information, accumulated throughout their development. Furthermore, the framework is extensible to support adding of new attributes and new attribute types since these are expected to arise frequently as another methods and tools will be developed within PROGRESS.

1.1 Goals of the thesis

Briefly, the goal of this thesis is to propose an attribute framework for the ProCom component model and prove the devised concepts in a prototype integrated into the PROGRESS IDE. Looking closer, there are two main sub-goals comprising the overall goal.

The first one lies in defining the structure of attributes in ProCom based on the analysis of the needs of different stakeholders involved in the development process envisioned by PROGRESS, extending ProCom to support these attributes and devising methods of adding new attributes and attribute types.

The second one relates to the prototype implementation of the attribute framework. The main actors working with attributes should be identified as well as their requirements imposed on the interface of the attribute framework. Their needs should be reflected in the design of interfaces exposed by the framework, which should then be implemented and integrated into the PROGRESS IDE.

1.2 Structure of the thesis

The structure of this document is the following. A brief introduction to the theories forming the context of this thesis together with the technologies used during its implementation part are presented in Chapter 2. The following two chapters represent the main contribution of the thesis. Chapter 3 gathers the main actors and key requirements for the properties of ProCom attributes and working with them. The designs realizing these requirements are elaborated and discussed compared to their alternatives in Chapter 4. Chapter 5 briefly describes the prototype implementation in terms of its scope and main features illustrating the main concepts on sample attribute additions. In Chapter 6 a summary of the related work to the topic of the thesis is given. The whole thesis is concluded in Chapter 7, where the contribution of the thesis is assessed and the possible future improvements are suggested.

Chapter 2

Background

In this chapter the theory and technologies forming the context of the thesis are briefly summarized. Embedded systems, component-based development, PROGRESS and ProCom are introduced in the theoretical background. The technological background consists of concise descriptions of the Eclipse Platform, the PROGRESS IDE and Eclipse Modeling Framework.

2.1 Theoretical background

2.1.1 Embedded systems

'Embedded systems are information processing systems that are embedded into a larger product.' [19] Nowadays, they can be found in a great variety of electronic devices ranging from consumer electronics (MP3 players, calculators, microwave ovens, etc.) to traffic lights controllers, vehicular control systems, and systems controlling nuclear power plants. Despite the diversity of this class of systems some common characteristics exist [19]:

- Typically, they interact with and control the physical environment they are embedded in.
- There is the strong need for *dependability* of these systems, which is caused mainly by the fact that they can physically influence the surrounding environment (e.g. an arm of the robot manipulating some product on the assembly line, or the autopilot controlling the course of the flight of an airplane). Dependability includes the properties of reliability, maintainability, availability, safety, and security.
- Embedded systems are supposed to be *efficient* in terms of energy, code-size (especially systems on one chip), run-time efficiency, weight, and cost.
- Embedded systems are usually special-purpose computers dedicated only to performing the intended functionality (as opposed to general-purpose computers).
- Often, embedded systems must satisfy real-time constraints, therefore being real-time systems.

Real-time systems are computing systems whose correctness depends on meeting timing constraints, i.e. the correct behavior depends not only on the result of computation but also on the time of producing the result [10]. There are two basic classes of these systems: soft and hard real-time systems. In soft real-time systems meeting the deadline is desired but not essential. In hard real-time systems the timing constraints must be guaranteed to be always met. Examples of real-time systems include vehicular control systems, certain medical devices or automated factories. Typically, real-time systems are embedded systems having to meet the above-mentioned requirements on embedded systems beyond the timing constraints.

The domain of real-time embedded systems brings new challenges beyond those known in general software development. Hard time and resource constraints of designed systems, which require thorough (often formal) verification and testing, tend to be reflected in the whole development process. The resulting software products are usually monolithic pieces of software heavily hardware-oriented and customized to a particular platform they run on. The preference for simple and thus easily verifiable systems is evident, as opposed to the criteria of versatility, reusability or design purity applied e.g. in desktop or enterprise software systems.

2.1.2 Component-based development

Component-based development (CBD) is an approach to the development of software systems putting emphasis on reusability. The approach is based on the notion of a component, a piece of software with well-defined functionality and clearly specified interface to the rest of the system. A component is a reusable unit of composition and deployment of a system [10].

Although reusability belongs to the long-known goals of software development, CBD has reestablished this notion by modifications introduced throughout the whole development process in all the phases of software life cycle in order to support and maximize the component reuse (see the following subsection describing the CBD process). A sub-discipline of software engineering studying and proposing customizations of the development processes to support CBD is called *component-based software engineering* (CBSE).

CBD has many benefits making it a successful development approach in many software areas (e.g. graphical desktop applications, enterprise systems): dealing with increasing complexity of software systems, shorter development times, greater productivity, and usability [10]. On the other hand, there are some issues that, if handled improperly, can pose a risk to the successful development of a system: too much effort spent on building too general and abstract components (trade-off between usability and reusability), difficult component requirements management (requirements coming from many systems, throughout — and regardless of — the component life cycle), component maintenance costs, and threatening a system reliability by component updates during the system operation [10].

CBD Process

The most evident modification of the development process is its splitting into the two interdependent processes [9] of

- building a system from components,
- development of reusable components.

From early phases of the development of a system (even in the requirement specification stage), the decisions made are strongly influenced by a set of components that can be (re)used in the project ('component pool'). There also arises the need for completely new processes of selecting, adapting, and testing components before they are integrated into the system. In comparison with other software development approaches, less time is spent on the actual implementation, which should ideally be restricted only to implementing the 'glue-code' connecting components and possible adaptation of components that do not fully satisfy the requirements. On the other hand, the processes of seeking for suitable components together with their verification and testing, which has to be performed for components in isolation as well as for assembled components, require more effort [9]. CBD even influences the support and maintenance phase of the software system life cycle, where processes similar to those taking place in the integration phase are performed during components updates.

Development of a new component is principally the same as in the classic approaches (arbitrary development process model can be used) with an emphasis on component reuse. However, developing a reusable component is much more demanding and thus requires more effort and

resources since it has impacts on its analysis and design striving for more general solutions. General design is beneficial for dealing with requirements both already imposed and yet unforeseen (possibly generated by systems designed in the future). Again, thorough verification and testing must be performed since the component is aimed to be reused in different systems deployed in various environments.

Although the two processes are not completely independent, they can be performed in parallel. And typically, their life cycles differ, which is always true for the components delivered by third parties.

2.1.3 Progress

Progress is a research centre aiming at facilitating the development of *predictable* embedded software systems with focus on vehicular, automation, and telecommunication domains. By predictability it is meant the ability to guarantee or at least reliably estimate certain properties of a system, namely the ones regarding functional requirements related to interfaces and behaviors, timing requirements, reliability, resource usage, and development life cycle [17].

Progress has chosen component-base development as a way how to tackle the growing complexity of embedded real-time software systems. Progress attempts to develop new theories, techniques, tools, and overall process improvements to CBD to better reflect the specific needs of the considered domain of software systems. The main areas considered crucial in tackling the challenges introduced by embedded systems and thus emphasized by Progress research are [6]:

- suitable component technology,
- deployment (and its significance in the development),
- analyses and verifications.

Progress proposes a very broad notion of a component as a primary, reusable, design-time element able to contain information of various levels of abstraction accumulated throughout its development. The component representation changes in the process of the development of a system to better reflect the requirements of its different phases. For the earliest phases of development with the most abstract and vague information about the system, some general modeling language (e.g. UML) is assumed to be sufficient. As the requirements become more concrete, another component model tailored specifically for the needs of the embedded systems should be used. This component model should support high-level design of a system as well as a more detailed design of smaller components comprising the system. Orientation to predictable embedded systems implies the support for associating extra-functional requirements with components and their composites (e.g. temporal constraints, reliability, robustness, safety, performance [10]). Currently, such component model is being developed within Progress, and it is called ProCom (Section 2.1.4). It should be noted that components are intended only as design-time entities, synthesized platform-specific artifacts will be executed at run-time.

The significance of a target platform belongs to the well-known specifics of embedded systems. This important role of the platform, or its abstraction, and the whole deployment is retained in the development process envisioned by Progress. Since early stages of development, it is possible to define requirements on the target platform. The platform can serve as a source of requirements influencing the design of the system but also conversely - requirements on the platform can be derived from the design of the system. Furthermore, specification of the deployment of the system to the platform is necessary for the assessment of predictability of the system, i.e. determining or estimating some properties of the system relating to the time constraints, reliability, performance, etc.

Various analyses and verifications are means for providing predictability of embedded software systems and its development [17]. They are intended to be performed throughout the whole development. Based on the level of detail of the system specification, they aim to provide rough

estimates (to guide the design of the system) or precise measurements of the system properties. The various analyses planned to be developed within Progress include reliability predictions, functional compliance analysis, timing, and resource usage analyses [6].

Apart from the three aforementioned main areas, Progress aims to focus on the overall amendments of the CBD process and on the support of legacy embedded software systems.

2.1.4 ProCom

ProCom [5][6] (Progress Component Model) is a component model being developed within Progress to facilitate CBD in the considered class of embedded software systems. To achieve this goal, ProCom development is driven by the following main guidelines:

- To reflect different requirements imposed on the component model during different phases of development of a system.
- To associate information of different levels of abstractions, produced throughout the system development, with components to support the proposed feature of the Progress development process of non-linear transition between different development phases of a component as well as parallel performing of activities related to different development phases (e.g. specifying the target platform details, i.e. deployment specification, during the early design of a system).
- To sufficiently support the other two (beyond component modeling) important development process activities of analysis and deployment stressed by the Progress approach.

The first guideline has resulted in separation of ProCom into two interconnected layers: ProSys and ProSave, each of which aims to provide support for a particular phase of system modeling.

ProSys, the upper layer of ProCom, enables high-level modeling of a system in which the main parts of the system are identified and their basic relations are modeled through the specification of their communication. Components on this level are called *subsystems* representing large units with complex functionality typically deployed to different physical nodes. Subsystems are active units, able to have one or more threads of execution, communicating with other subsystems using asynchronous messages. A subsystem is specified by defining its *input and output message ports*, which are connected to the message ports of other subsystems by means of *message channels* (see Figure 1), and its representation, i.e. a specification of its internals. Depending on its representation, a subsystem can be *primitive* or *composite*. A primitive subsystem is not further refined by means of ProSys; instead, it is either a legacy subsystem adapted to have an interface of a ProSys subsystem or its internals are modeled using ProSave. A composite subsystem is composed of a set of communicating ProSys subsystems.

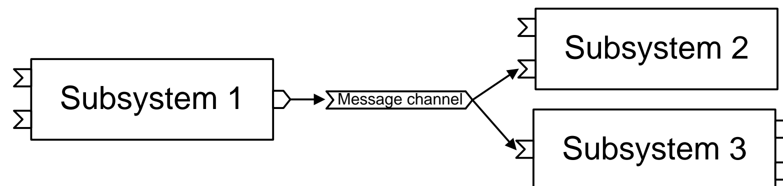


Figure 1: ProSys subsystems communicating using a message channel [22]

ProSave, the lower layer of ProCom, is a component model focused on modeling small-scaled components. It has been developed to facilitate the design of control structures found usually in embedded control-intensive systems. A component is a passive entity performing some action

only upon external activation, resembling in this respect the semantics of the function construct from procedural, imperative programming languages. A component's interface exposed to other components consists of a set of *services* provided by the component. A service is specified by its *input port group* and a set of *output port groups*, which allows for providing partial results of the service before its entire execution is finished. Port groups form an interface of a service, and they consist of a *trigger port* and a collection of typed *data ports*, thus separating control and data flows. The role of trigger ports is to notify the respective entity (the component's service in case of an input trigger port and a consumer of the service's results in case of an output trigger ports) about the fact that data in a particular port group are ready, and consequently invoke their processing by activating the respective entity. Again, a component can be primitive, realized by a set of C functions corresponding to the component's services, or composite, consisting of other ProSave components. Component communication is modeled by *connections*, directed edges connecting compatible ports. Both data and control connections can have their information flows modified by so-called *connectors* manipulating the flows using the domain-typical patterns (forks, muxers, demuxers, joins, selectors, etc.).

The two component models are smoothly integrated by allowing a ProSys subsystem to be realized by a ProSave component (typically composite), which is illustrated in Figure 2. Moreover, entities in both models share the ability to have various information associated with them (realizing the second guideline) by means of *attributes*, structured pieces of information containing component-related artifacts accumulated throughout their development. It is one of the majors goals of this thesis to elaborate the notion of an attribute in ProCom.

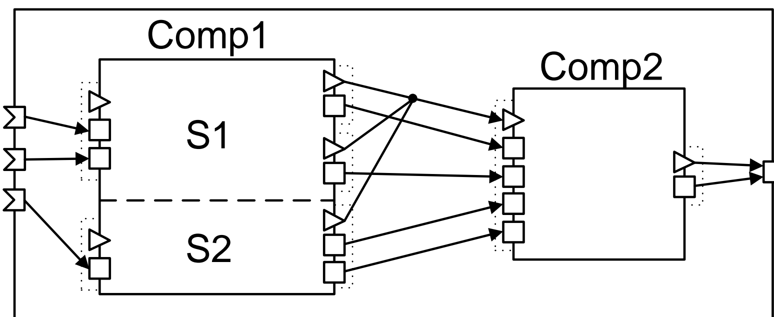


Figure 2: ProSave components realizing a ProSys subsystem. The subsystem is realized by a composite ProSave component which is composed of two sub-components: Comp1 and Comp2. Comp1 has two services (S1, S2), whereas the functionality of Comp2 is provided by an unnamed service. Dotted rectangles denote port groups. Squares and triangles designate data and trigger ports respectively.

2.2 Technological background

2.2.1 Eclipse Platform

The Eclipse Platform [1][11][8] is a Java, open-source platform for building applications. It provides developers with an extensible application framework and a set of components of various functionality able to build a rich client application. It has evolved from the Eclipse IDE, an extensible IDE supporting multiple programming languages, to a generic platform for development of both GUI and non-GUI applications of various types.

The cornerstone of the platform is an extensible plugin architecture allowing for integration of various functionality contributed by plugins. A plugin is a unit enriching the Eclipse Platform with some functionality or providing other resources (e.g. help, images). Apart from its actual contents (source code, help files, etc.), it comprises a *plugin manifest*, which is a specification

of plugin's run-time dependencies (plugins on whose functionality the plugin depends) and, in reverse, of classes exposed by the plugin to other plugins. In addition to run-time dependencies, the plugin manifest contains the definition of explicit points of extension of the plugin's functionality that can be extended by other plugins. The plugin manifest also comprises the declarations of extending extension points defined by some plugins in the application (termed as defining an *extension*). This mechanism, called *extension point mechanism*, serves as the main means for achieving extensibility of the Eclipse Platform.

Plugins are not only design-time but also run-time entities managed by the Platform Runtime. The Platform Runtime is a small kernel forming the basis of each application built on the Eclipse Platform which manages plugins and their life cycles. Its role is to discover all plugins comprising an application, read their manifests and build a *plugin registry*, a database of all available plugins, extension points and their extensions accessible via API to the plugins. The runtime also executes the first plugin forming the basis of the application, and consequently activates all the plugins required for the given plugin to be loaded. The plugin activation is lazy meaning that plugin is not active unless one of its classes is demanded by another plugin.

The whole Eclipse Platform is built using this mechanism where plugins provide some functionality and points of extension to other plugins which in turn extend the former plugins and provide other functionality for yet another plugins, etc. The brief summary of the parts comprising the Eclipse Platform is given in Figure 3. The Eclipse Platform provides the functionality to build a full-fledged IDE (which is far more than a typical application requires). A basic subset of plugins sufficient to create a rich client application consisting mainly from the Platform Runtime, generic UI toolkits, and several other features (see Figure 3) is called *Rich Client Platform* (RCP).

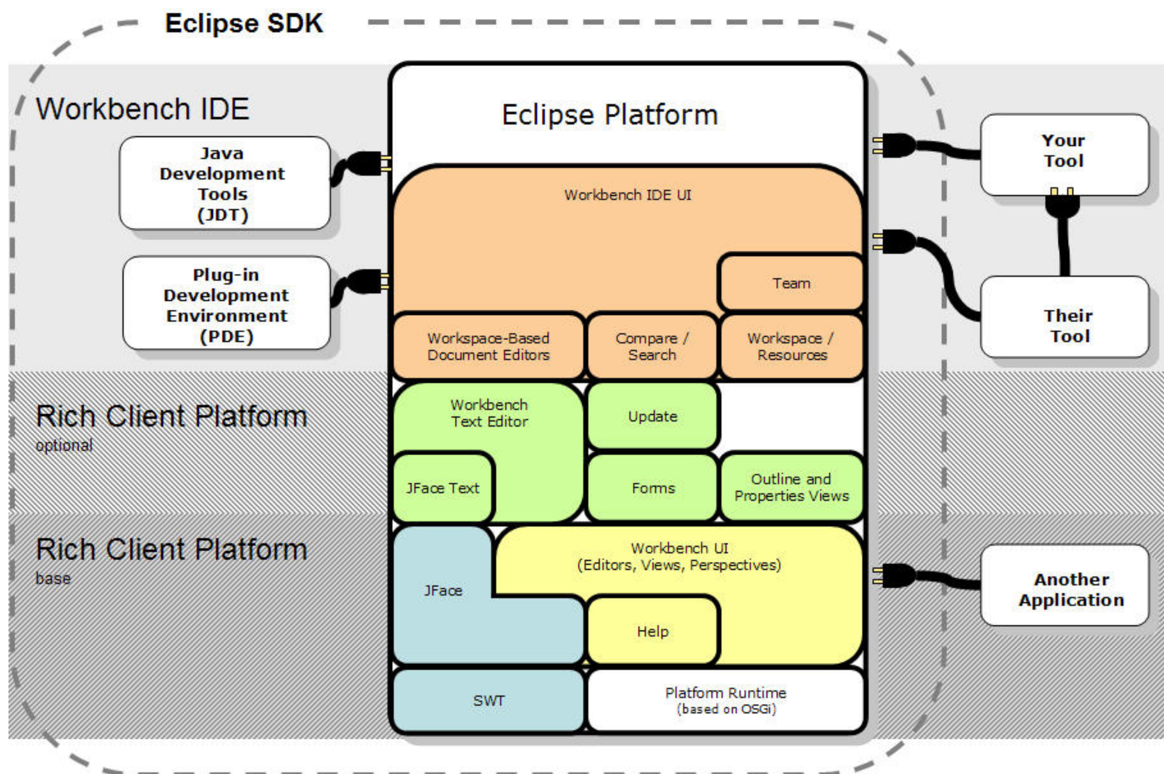


Figure 3: Eclipse Platform architecture [11]

Although the Platform provides a vast amount of functionality, there exist some parts that form a distinguishing characteristic of development in Eclipse. Since they were used heavily throughout the development of the attribute framework, their brief description is included. They are SWT, JFace, and Workbench UI. All of them are concerned with the GUI of an application

focusing on different levels of its design.

The Standard Widget Toolkit (SWT) is a low-level GUI toolkit providing implementation of common UI widgets. Its main characteristic is that the widgets are implemented using the native primitives offered by the underlying OS, as opposed to the Swing (standard Java GUI toolkit) approach which emulates all the widgets. *JFace* forms another level above SWT (not necessarily hiding the SWT layer to a client). It is an OS window system independent UI framework providing high-level UI elements like dialogs, wizards, actions and viewers, which are adapters for some sophisticated SWT widgets (table, list, tree) supporting MVC-based[1] design. *The Workbench UI* is a framework forming typical UI paradigm of Eclipse-based applications defining the behavior and layout of larger UI elements comprising an application: application main window (called workbench), editors, viewers, and perspectives. A main windows is composed of smaller UI parts of editors and viewers, whose layout configurations can be grouped and switched between by means of perspectives. Whereas editors serve as the main tool for modification of application domain objects, viewers provide detailed information about the currently selected element in a workbench.

2.2.2 Progress IDE

The PROGRESS IDE is an Eclipse RCP application developed within PROGRESS. It is intended to provide engineering support for the whole development process envisioned by PROGRESS. The output of the research conducted at PROGRESS in the form of various tools, techniques, and computations (e.g. analyses, verifications, synthesis) amending the process is to be integrated into the PROGRESS IDE. The attribute framework elaborated in this thesis should be integrated with the PROGRESS IDE as well.

2.2.3 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) [4][12] is a Java framework and a code-generation tool for creating parts of applications based on a model. By model it is meant some formal description of classes and their relations representing the entities from the application domain. EMF provides facilities for generating Java implementation of the model entities. The generated code can be further edited manually by a developer and it can also be regenerated when the model is changed having the ability to preserve manual code edits. At run-time, EMF offers several other utility features facilitating working with Java representation of the model entities: notifications about model entity changes, validation capabilities, persistence of the model entities (the XMI format by default). EMF also allows for an advanced reflective manipulation with model entities. Furthermore, one of the more advanced EMF features consists in defining models at run-time and being able to use the reflective manipulation for working with instances of those models (also called *dynamic EMF*).

Currently, input model representations can be in one the following formats: ECore XMI, annotated Java interfaces or an XML Schema. EMF converts these representations into its own modeling language called *ECore*. ECore is a simple modeling language able to represent classes and their features (primitive attributes or references to other classes). The above-mentioned reflective capability of working with model instances is realized by accessing object representation of ECore model instances.

Apart from the EMF core framework described above, EMF also consists of the *EMF.Edit* framework able to generate adapters to the model entities facilitating working with them in the Eclipse UI environment. Additionally, it is even capable of generating a simple GUI editor integrated with the Eclipse Platform for working with model entities.

[1] Model-view-controller, an architectural pattern distinguishing between model objects, their visual representation and controlling their behavior.

From a broader perspective, EMF can be seen as a simple application of the MDA (Model Driven Architecture) approach, which is based on defining a platform-independent model of an application, later on converted with the help of development tools to a platform-dependent model, and finally to the code. However, EMF lacks the ambition to define some complex models including the behavior of the whole applications; instead, it focuses on a simple structural description, a kind of formalization of class diagrams, aiming to do rather a small job but thoroughly so as to be usable in practice. Despite its simplicity, it still presents a new software development paradigm of modeling as another way of creating a software system by a (semi)formal specification used for generating the source code rather than programming manually.

In the text of the thesis, the terms of model and metamodel are used several times. Since they are closely connected with EMF, their explanation is included in this section. The model which is an input for the EMF code generation (the ECore model) is not a model of the real-world entities which are to be described. Instead, it is a specification of a description of these entities[2], i.e. what are the names of the kinds of entities, their structure and relations between each other. They are the instances of these descriptions which are actual models of the real-world entities. To terminologically distinguish between the two kinds of models, the former one is called a *metamodel* whereas as the latter one is denoted as a model.

[2] More formally, it is the definition of the grammar of the language whose words represent the real-world entities.

Chapter 3

Problem Analysis

In this chapter the key requirements for properties and features of the ProCom attribute framework are specified. The chapter begins with a closer description of a component-based development process in the domain of embedded real-time systems focusing on the role of attributes. This description serves as the main source of the requirement specification in Section 3.2 and Section 3.4. Finally, functional requirements are concretized in the form of a set of use cases in the last part of the chapter.

3.1 Attributes in the development process envisioned by Progress

Regardless of a particular development process model, several main phases in the development of a software product can be identified [9]. These include

- requirement analysis and specification,
- system and software design,
- implementation and unit testing,
- system integration,
- system verification and validation (in relation to the requirements),
- operation support and maintenance, and
- disposal.

Whereas in sequential process models (e.g. waterfall) these phases are carried out consecutively, in evolutionary models (e.g. iterative development, spiral model) they may be executed in parallel.

The CBD approach brings further amendments to the development process. There arises the need for completely new processes of selecting, adapting and testing components. Moreover, it introduces the separation of the whole development process into two interdependent, parallel subprocesses of assembling a system from components and a component development, as mentioned in Section 2.1.2.

In order to specify the requirements imposed on the attribute framework of the ProCom component model, it is necessary to specifically focus on the development process as envisioned by Progress. Since the development process describes all involved activities, including those handling with attributes of modeled entities, it forms the ideal ground for requirements elicitation.

The key characteristics of the development process according to the vision of Progress, whose context is detailed in Section 2.1.3 and Section 2.1.4, are as follows [6]:

- Pervasive usage of components (and subsystems on the system scale[1]) during the whole development cycle from the early project phases of requirement specification and analysis throughout design and implementation up to the deployment of the system.
- Ability to move non-linearly, backwards and forwards in the development process both on the level of an individual component and on the level of the whole development process. To illustrate the necessity of the former transition (backwards, on the component level), consider the process of refining analyses and designs in later stages when some implementation works have already begun. An example of the latter transition (forwards, on the system level) might be the specification of the characteristics of the deployment platform during the early phases of design.
- Different levels of abstraction of components in a system at the same time, the property inherent to the CBD, where the situation when a system consists of components that found themselves in different stages of development and thus of different levels of abstraction is the norm (e.g. a system consisting of a fully implemented component and a component whose development has not yet begun).
- Emphasis on analysis and verification during the whole process, dictated by the need of dependability and trustworthiness in the area of embedded real-time systems.

These characteristics have a major impact on the design of a component in ProCom. They require a component to be able to *contain information associated to it during the different phases of development and of different levels of abstraction.*

Having defined this key requirement for components in ProCom, the role of attributes in the whole process can be formulated. They are the attributes which are responsible for being able to contain all those different kinds of information associated to a component collected throughout the development process. The variety of included information is really immense (see Figure 4) as it ranges from requirement specifications, use cases and early sketches through results of various analyses (reliability predictions, functional compliance analysis, timing and resource usage analyses, etc.) up to detailed measurements of characteristics of the final version of a component.

Attributes are accessed by many actors taking part in the process, including humans (analysts, designers, developers, people in charge of deployment, etc.) and other programs (analyses, synthesis), for which attributes can play a role of both an input and an output.

A process of such complexity would not be possible without a massive tool support. In respect of PROGRESS, the PROGRESS IDE becomes an essential part of the development process and the central point for all actors involved in it. All information relating to components, i.e. components' attributes, should be accessed in a convenient way specific to the semantics and purpose of a particular piece of information by means of the IDE.

3.2 General attribute requirements

Taking into account the description of the PROGRESS development process given in the previous section, we will now attempt to formulate the general requirements that should drive the whole analysis and design of the ProCom attribute framework.

As mentioned earlier, the pieces of information attached to components throughout the development process come from many sources. They are of a great variety of types and levels of abstraction. The first and evident requirement therefore is the attribute's *ability to support the containment of a multitude of types of information.*

[1] For the sake of brevity, subsystems are not mentioned in this section, but the following statements about components can also be applied to subsystems.

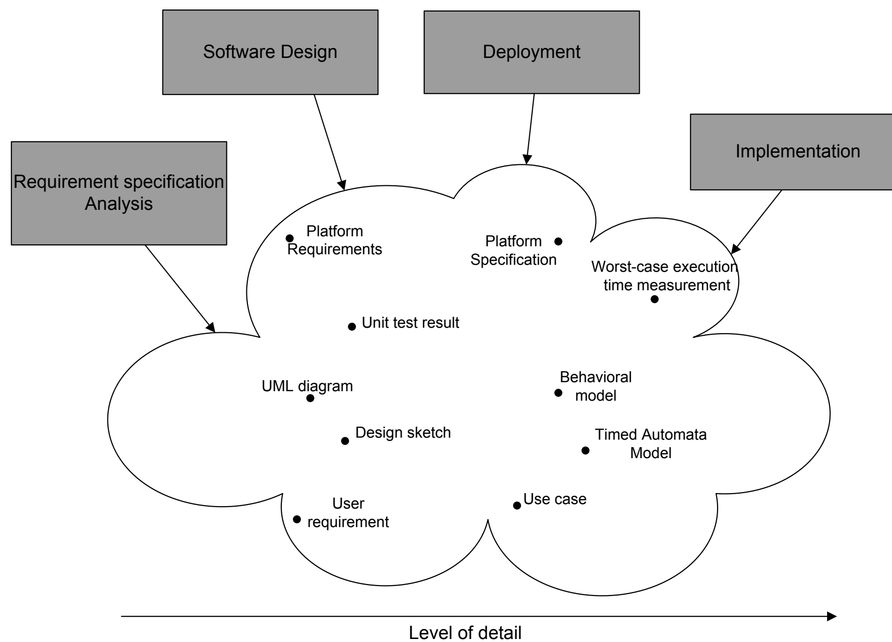


Figure 4: Various information included in component's attributes

In the current state of the PROGRESS project, it is not possible to obtain a definite list of all potential attributes that could be assigned to a component — the processes are not yet fully defined, many analyses and the synthesis are in a phase of sketches and early designs. But even if everything was already fully specified, there is always the possibility (approaching to certainty) of improving or adding some new step to the whole process. Hence, there arises the need for adding new information to a component, or in other words, attributes are required to be *extensible.* In fact, the importance of this requirement gave the name to the whole attribute framework.

However, what has just been pointed out as the need for extensibility might be also realised, due to the vagueness of the requirement specification, by creating a fixed pool of attribute data types and letting the users of the framework add new attributes of some of the known types from the pool. Although this is also intended functionality, it obviously does not suffice to meet the needs of the development process. It is necessary to be able to introduce completely new kinds of information, instances of new data types. Therefore, another requirement imposed on the attribute framework refining the previous one is *type extensibility.*

3.3 Actors working with attributes

Before continuing further with specifying more concrete requirements, the main actors involved in working with the attribute framework are identified as it helps to structure the following discussion. Again, the development process description from Section 3.1 is the main basis for the following enumeration.

The first group of users of attributes are people who access components' attributes throughout the whole development cycle: analysts, designers, system architects, component developers, etc. Their common characteristic is that they use facilities offered by the PROGRESS IDE to work with the data associated with components using attributes, and these data are their primary concern. These users will be referred as *IDE users.*

As mentioned in the development process description section, attributes play often roles of

inputs and outputs of various computations over the model, e.g. various analyses and simulations. The second group of attribute framework users consists of developers programming those computations. They are using a completely different interface than IDE users to interact with the framework: they communicate through the framework's programmatic interface. Since the usage of this interface does not have to be necessarily constrained only to computations over model, but it might involve arbitrary invocation by any program module of the IDE, this group of users will be denoted as *IDE module developers*.

Although the previous actors interact with the attribute framework in completely different ways, both of these groups have something in common: they just use the existing attributes and attribute types and they do not extend them in any way. And since the key requirement for attributes is their (type) extensibility, another actor who is responsible for adding a new attribute or a new attribute type is needed. Typically, these actions will be carried out by a developer of the Progress IDE or an architect of the ProCom model, employing another part of the attribute framework which enables to plug in modules extending the capabilities of the framework. In the context of this thesis, such users will be called *attribute contributors*.

Not only do these three groups of users specify basic actors, but they also define three different areas of the attribute framework itself. This natural division is used many times in the following text.

3.4 Requirement specification

Having realized the main actors involved in working with the attribute framework, the discussion about the requirements imposed on the attribute framework can now be structured according to these actors. First, the requirements for attributes common to both IDE users and IDE module developers[2] are specified. Afterwards, we continue by enumerating requirements specific to the particular groups of users.

3.4.1 Common attribute requirements

So far, only components were considered as entities to which attributes can be attached. However, during the process it might be necessary to associate some information that is related to some entity inside a component (e.g. a port or a service) as well as to some object encapsulating a component (e.g. subsystem). Accordingly, attributes are required to be *attachable to many types of ProCom entities both in the ProSave and ProSys layers*.

Immediately, there follows another request refining the previous one. If attributes are allowed to be associated with many types of entities, it is almost automatically expected that there exists the *possibility of defining to which kinds of entities can a particular piece of information be attached and to which it is not applicable*.

Let us now focus more on the pieces of information associated with the ProCom entities. The first general requirement for the attribute framework was concerned with the support for a variety of kinds of information that can be attached to the model entities. Ranging from simple value types (e.g. integers, strings) to use cases and sophisticated behavioral models represented by e.g. timed automata, they truly represent a wide scale of data types. However, there might be the need for some parts of information to be common to several or all of the supported data types. It is not sufficient to have opaque data whose semantics is hidden inside the inner structure of the opaque pieces of information. There should be some data members common to all attribute data types (e.g. a version or a comment) known to the framework, helping the framework to classify associated information in a uniform way without actually peeking inside the main data chunk (e.g. a use case or a timed automaton). This requirement can be formulated as demand for *the structure of the pieces of information known to the framework*.

[2] Common attribute requirements describe the structure of attributes, which is of minor interest to attribute contributors.

One of the characteristics of the ProCom development process mentioned in Section 3.1 was the 'ability to move backwards to earlier phases in the development process of an individual component', which resulted in the design of a component where it contains information of different levels of abstraction associated to it during the different stages of the development.

Practically, this implies there can be several data values of the same characteristic on different levels of abstraction associated with a component. As an example, the attribute of the worst-case execution time of a service of a component can be taken: In the early phases of development there was made an estimate by an expert. Later, when behavioral models were elaborated, the same characteristic was computed in a simulation using the models. And finally, in the testing phase some precise worst-case execution time measurement was performed. Naturally, all these values need to be associated with the component (or rather with the service). This example illustrates one important implication of the above-mentioned ProCom feature. There is the need for the attribute framework to be able to *contain several pieces of information related to the same characteristic of a component.*

3.4.2 Terminology clarification

Before proceeding to the requirements specific to particular groups of users of the attribute framework, the usage of the terminology should be clarified and made consistent.

Up to now, the terms 'piece of information associated with a model entity' and 'attribute' were used intuitively and rather in an interchangeable way. However, the last requirement for the attribute framework, 'to contain several pieces of information related to the same characteristic of a component', forces us to distinguish two notions. The first is the notion of a characteristic of an entity, a property or a feature of an entity with precisely defined semantics, which will be referred as an *attribute* in the remaining text of this document. The second one is a particular piece of information, analogous to an instance of a data type (from the domain of programming languages), associated with an attribute, that represents or measures the quality of an attribute or, simply said, *attribute value.*

Using this terminology, the requirements stated above could be rephrased as follows:

- An attribute should be attachable to many types of ProCom entities both in the ProSave and ProSys layers.
- It should be possible to define to which types of entities a particular attribute can be attached and to which it is not applicable.
- An attribute value should be structured in a way known to the framework.
- An attribute should be able to contain several attribute values.

3.4.3 IDE users requirements

The following set of requirements reflects the needs of the Progress IDE users. The first one is *to integrate the attribute framework to the IDE* making the whole interaction possible. However, such specification needs to be refined by taking a closer look at the details of the development process.

Since actors start working with attributes as soon as they start using components, and the two activities are overlapping during the whole development process, there should be *smooth integration between the ProCom model editor*, which is responsible for creating and linking various ProCom entities, *and the attribute framework.* By the smooth integration it is meant that the framework should be aware of some important events (e.g. selection of a ProCom model entity) happening in the ProCom editor and it should then react to them accordingly.

Another important aspect of the framework's integration into the IDE should be its *intuitiveness.* Users using the IDE can be expected to have some experience with other applications

employing the well-known paradigm for working with properties of edited objects (e.g. text processors, vector graphics editors). Reusing some typical procedures might help to decrease learning time and increase user comfort.

Let us now focus more on the functional requirements imposed by IDE users. An IDE user, regardless of his role in the development, uses the attribute framework in order to access data associated with model entities, or more precisely, an IDE user accesses attribute values. They are the data stored using attributes that the user is primarily interested in as they become inputs or results of his or her work on the model. Consequently, the framework is required to *support basic operations with an attribute value*: adding a value to an entity, removing a value from an entity, viewing and editing a value. All of these operations are described closer in the section of use cases (Section 3.5). However, the last two represent more complex operations that yield another requirements.

The viewing of an attribute value is an action in which the framework provides an IDE user with a visual representation of the data stored in the attribute value. The *visual representation should be chosen in accordance with the data type and the semantics of the attribute* facilitating the user's work to a maximum extent. The editing of an attribute value is similar to the viewing in the respect that it should also be attribute-specific. But whereas the primary goal of viewing was to facilitate understanding of a value by proper visualisation, editing aims at *providing means for comfortable and effective modification of a value.* Thus, the above-mentioned intentions typically lead to different appearances of the GUI realizing them, which is the reason why viewing and editing are separated even though viewing can be generally perceived as a special case of editing with no modification performed.

As there exist plenty of possible kinds of information which could be eventually attached to model entities, it is likely that there will be a great number of attributes associated with entities. Therefore, several requirements whose motivation is to make it easier for a user to deal with many attributes are needed.

Attribute values *should be manipulated in a uniform way* independently on their attribute type. Although there will be some attribute-specific aspects of behavior (e.g. when viewing or editing values), the general manipulation procedures should be the same making it easier to work with types not yet known to the IDE user.

Another approach towards dealing with numerous attributes lies in *localizing and centralizing the control of attributes in a GUI element or a group of related GUI elements.* Attributes should be aggregated in one place, and a user should not be forced to search for a GUI element responsible for controlling a particular attribute value he or she wants to access.

Immediately, there follows an opposite requirement to the aggregation but facilitating working with many attributes as well. *Attributes should be divided into some semantically similar sets*, which would help a user in finding an attribute of his interest. For example, attributes might be divided into several categories (e.g. reliability, performance, deployment, resource consumption) according to what characteristic they measure.

Finally, in order to be well understood by users, *attributes should be supplied with documentation* accessible from the GUI.

3.4.4 Module developers requirements

IDE module developers use the attribute framework for working with attribute values similarly to IDE users. However, in contrast to IDE users, they access attribute values programmatically from the PROGRESS IDE modules they create. For this reason, they *require the framework to have a public API*[3] *allowing them to manipulate attribute values.* Because the PROGRESS IDE as well as its modules are intended to be implemented in Java, the API in this context means a public set of Java class(es) and/or interface(s).

[3] Application programming interface

In order to specify what features the framework's API must provide, the needs of IDE users can be taken as an inspiration. The API has to allow developers to *add, remove and access an attribute value of a particular model entity.* However similar these requirements are, it is worth noting the difference between the meaning of the notion of an attribute value for IDE users and module developers. Whereas the former group of users perceive an attribute value only indirectly by means of the facilities that the framework provides them to view and edit the value focusing then only on its visual representation, the latter group of users come in direct contact with the programmatic representation of an attribute value because it forms an output or input of the computations they perform.

The representation *should be flexible enough to be able to contain possibly highly structured or complex data* that might be stored in attributes, but at the same time it should also *allow the developers to access its structured contents in a comfortable way.*

3.4.5 Attribute contributors requirements

As mentioned before, attribute contributors are concerned with adding new attributes and attribute types to the framework and thus form the only group of users that is able to extend the framework capabilities. They do not use neither the GUI nor the API for module developers (*the client API*) to perform their job. Instead, *they require another interface to be provided by the framework.* It is also a specific kind of an API but with a different role. Whereas the client API was intended to provide its users with the attribute framework's functionality, this API, to the contrary, allows its clients to add new functionality to the framework. There exists a term for such a kind of API[4] - *Service Provider Interface* (SPI), which will denote the part of the framework used by attribute contributors.

In order to express which features the attribute contributors need the SPI to provide, it is necessary to realize what it means to add a new attribute or a new attribute type. A more general case of adding an attribute of a completely new type is considered. First, a *contributor has to specify an attribute value type*, a description of the data structure of an attribute value, which will later be associated with the attribute being added and whose realization will be exposed to module developers. Next, *they need to define the attribute itself.* Taking into account the requirements imposed earlier in the text (Section 3.4.1), an attribute definition consists of the following parts:

- a specification of an attribute value type, a type of values associated with the attribute being defined, realized by a reference to the type created in the first step of the contribution,
- providing the framework with the facilities for viewing and editing values of this attribute in the GUI of the PROGRESS IDE,
- a specification of type(s) of the ProCom model entity(ies) to which attribute can be attached,
- providing the framework with any other functionality that might be required to manipulate the values of a given attribute.

Compared to the requirements for the client API, the SPI might not be represented only by Java classes and interfaces. Here, we adhere to the broader semantics of an API (and consequently SPI) as a set of all means used in interaction between the attribute framework and its contributors.

[4] More information about SPI and its importance in designing an API can be found in [25].

3.5 Use cases

In this section, the functional requirements (see Figure 5) for the framework imposed by the actors defined in Section 3.3 are summarized in the form of use cases and provided with brief descriptions.

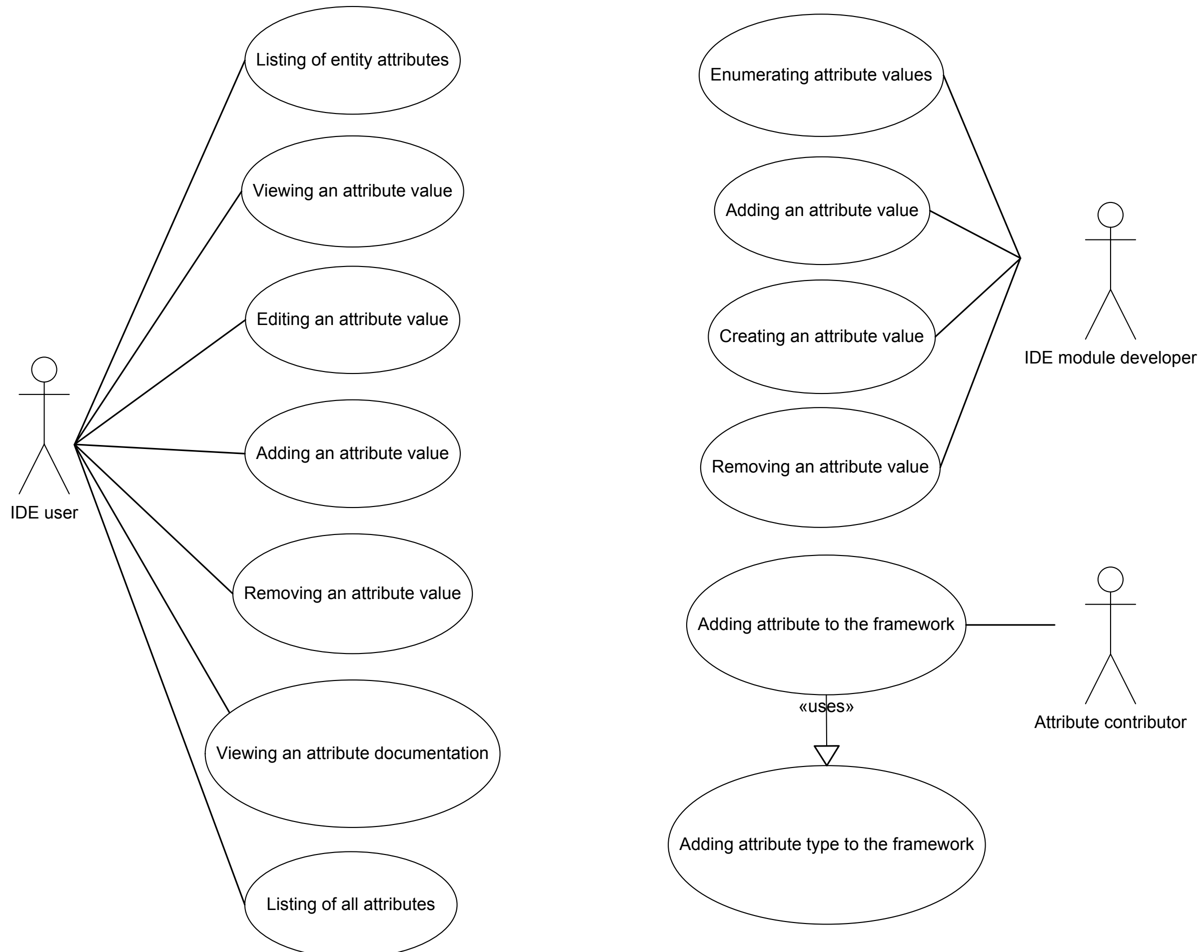


Figure 5: Actors in the attribute framework and their associated use cases

3.5.1 Use cases associated with an IDE user

Since in the following descriptions we need to refer to the GUI component of the Progress IDE responsible for working with attributes, we will denote it as Attribute View. However, it should be noted that it might not fully correspond to the final implementation, where the Attribute View may be represented differently, e.g. by several GUI elements.

Listing of entity attributes
An IDE user displays in the Attribute View all the attributes and their values attached to a ProCom entity selected in the ProCom model editor.

Viewing an attribute value
An IDE user views the value of an attribute selected in the Attribute View. Viewing of a value of some complex type might require opening a specialized viewer.

Editing an attribute value
An IDE user edits the value of an attribute selected in the Attribute View. Editing of a

value of some complex type might require opening a specialized editor.

Adding an attribute value
An IDE user adds a new attribute value to an attribute selected in the Attribute View.

Removing an attribute value
An IDE user removes an attribute value selected in the Attribute View.

Viewing an attribute documentation
An IDE user displays the documentation describing the usage of an attribute selected in the Attribute View.

Listing of all attributes
An IDE user displays the list of all the attributes contributed to the attribute framework, from which the attribute documentation will be accessible.

3.5.2 Use cases associated with an IDE module developer

All of the use cases contained in this section are performed programmatically using the framework's client API, and each corresponds to a Java interface or a class method call, which is the reason why the term 'caller' is used for an actor who carries out these use cases. Furthermore, it is assumed that a caller has programmatic access to entities of a ProCom model.

Enumerating attribute values
A caller retrieves a set of attribute values associated with an attribute of a model entity.

Creating an attribute value
A caller creates a new attribute value of an attribute.

Adding an attribute value
A caller adds an attribute value to the values of an attribute of a model entity.

Removing an attribute value
A caller removes an attribute value from the values of an attribute of a model entity.

3.5.3 Use cases associated with an attribute contributor

Currently, there are two use cases associated with the actor of an attribute contributor:

Adding attribute to the framework
An attribute contributor appropriately specifies all the items comprising the definition of an attribute (in the case of adding an attribute of a new type, the type is included, which is realized by the next use-case).

Adding attribute type to the framework
An attribute contributor appropriately specifies the definition of a new attribute type.

Contrary to the use cases mentioned previously, which were simple actions from the actor's point of view, the use cases associated with an attribute contributor are much more complex. They are refined in further chapters of the thesis. Requirements relating to them can be found in Section 3.4.5.

Here, let us explain why there are two separate use cases. It does not mean that these would have to be necessarily separate actions, but it should stress the fact that adding an attribute whose type is already known to the framework should be much more straightforward process than the case when even the attribute type specification is contributed.

Chapter 4

Solution Design

This chapter contains the important design decisions made during the development of the attribute framework. They are presented together with their explanation and evaluation of considered alternatives.

4.1 Integration with the ProCom metamodel

ProCom is formally defined in the form of a metamodel[1] describing entities, whose instances comprise the ProCom models of developed software systems, and relations between these entities. The metamodel is basically a UML class diagram where classes correspond to the ProCom entities and relations between them are modeled using the relationships of aggregation, composition, and association. The classical approach to modeling characteristics (or properties) of entities in UML employs class attributes[2]. Whereas early versions of the ProCom metamodel used this approach, it is insufficient in the context of this thesis.

The main reason lies in the anticipated high frequency of adding new attributes to ProCom during the initial phase of the PROGRESS evolution. As more and more parts of the development process (e.g. deployment to virtual and physical nodes, different kinds of analyses, synthesis) will get gradually specified in more detail, they will need to associate more kinds of information with components or other ProCom entities, i.e. they will need to add new attributes. If the ProCom attributes were modeled using the class attributes in the metamodel, every addition of a new ProCom attribute would mean extending the respective model entity class and consequently an extension of the whole ProCom metamodel. Changing the metamodel is an expensive operation because the metamodel does not only serve as a formalization of the ProCom but also as a source for the code generation process that produces code representation of the model (using EMF). There are even more code artifacts generated from the EMF model, including parts of the model editor. And all these would be influenced by the metamodel extension.

Another reason relates to the maintainability of the metamodel. It is expected that there will be tens or even hundreds of attributes, which would clutter the metamodel. Although attributes play an important role in ProCom, there exist more defining characteristics of the component model (e.g. components, services, ports, and their relationships). From this perspective, attributes are of a secondary importance and their presence in the metamodel would rather aggravate future ProCom evolution.

As a result, one of the tasks of the attribute framework consists in minimizing the effects of adding a new attribute on the ProCom metamodel. Still, attributes are the part of a component model and thus they should be reflected in its formalization, i.e. metamodel. The chosen solution detailed in this chapter is based on inclusion of the notion of an attribute and its refinement

[1] The difference between model and metamodel is explained in Section 2.2.3.

[2] As opposed to the rest of the text where 'attributes' refer to the ProCom attributes, here we have the semantics of the term as used in object-oriented programming in mind.

in the metamodel. However, particular characteristics of model entities, instances of attributes, are kept separately from the metamodel and are managed by the attribute framework.

After a closer examination of the role of attributes in the ProCom metamodel, it has been concluded that attributes, as a general means for associating various information with different ProCom entities, are not dependent on the remaining part of the metamodel.[3]

Conversely, certain parts of the ProCom metamodel must depend on attributes since some of its entities need to *have attributes*. To minimize the dependency between these two parts of the metamodel, an *attributable*, an entity capable of having attributes, has been introduced. The `Attributable` class is the only entity of the part of the metamodel related to attributes (*attribute metamodel*) which is directly used by the rest of the metamodel, and it therefore plays the role of an interface of the attribute metamodel. This design also makes it possible to completely hide the implementation details of the attribute metamodel and provide the facade in the form of methods of `Attributable`. ProCom entities having attributes are then modeled by subclassing `Attributable` (see Figure 6).

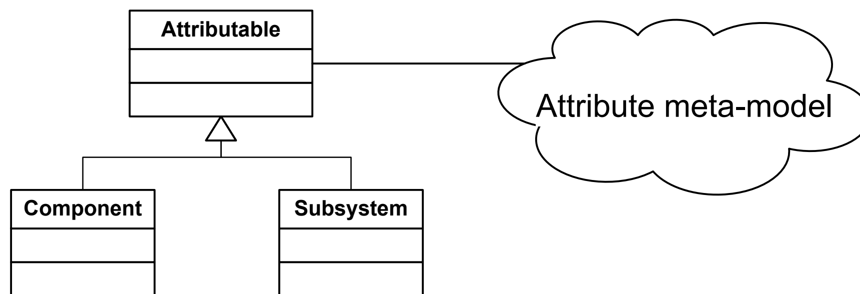


Figure 6: `Attributable` and its role in the attribute metamodel

An alternative to the realised solution is to use the relationship of composition. An entity with attributes could be modeled as a class that *is composed of* attributes, i.e. it has a composition relationship with a class representing an attribute. This design exhibits the following drawbacks:

- ProCom entities refer directly to the representation of an attribute, revealing the internals of the attribute metamodel.
- The attribute framework cannot determine whether an entity of the model has attributes or not from the class of the entity (as opposed to test whether an entity is a subclass of `Attributable`). This implies that there would have to be a part of the framework enumerating ProCom entities with attributes, which would make the framework dependent on ProCom and consequently less general.

4.2 Attribute structure

In this section the core classes of the attribute framework representing the notions of attribute and attribute value are elaborated. They can be viewed as more detailed and formalized versions of these notions, which were defined during the problem analysis, and they influence many aspects of the framework's design.

Three main entities forming an attribute and its associated values have been identified. They are common to all kinds of attributes regardless of the type of their values. These are an attribute, an attribute value and an attribute description. Each of these entities is directly represented by a class of the same name, as depicted in Figure 7.

[3] Here, only formal independence on the metamodel entities is meant. The overall design of the structure and semantics of attributes was heavily influenced by the context of ProCom.

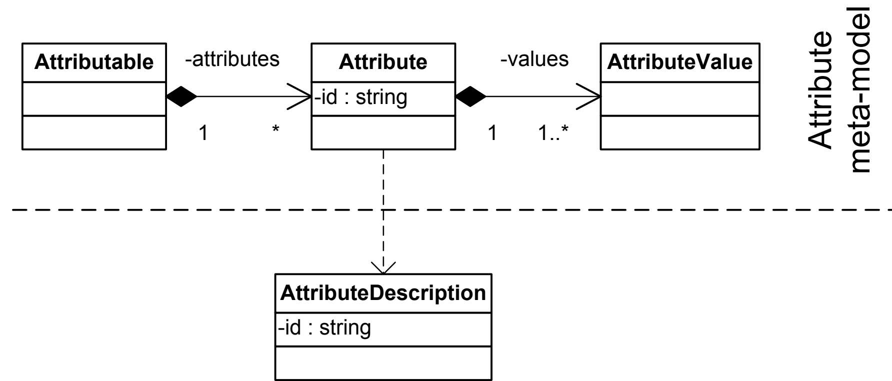


Figure 7: Main entities forming an attribute

Attribute

The `Attribute` class represents a characteristic of a ProCom model entity. As required in the analysis section, it can contain several attribute values. The semantics of an attribute is expressed by including the identification of an attribute description. As depicted in Figure 7, the relationships between `Attributable` and `Attribute` and between `Attribute` and `AttributeValue` are implemented using composition, which justifies another possible characterisation of `Attribute` as a container for attribute values.

Attribute value

A key class of the whole attribute framework. It represents a piece of information associated with a particular characteristic of a ProCom model entity. It is used indirectly by IDE users who view and edit instances of this class through the dedicated GUI of the attribute framework. It is used directly by IDE module developers who access it programmatically.

The design of this class also affects attribute contributors. Depending on the attribute type, information represented by an attribute value are of various types. This must be reflected in the internals of this class which must have attribute-specific parts. Type extensibility of the framework (see Section 4.4) is realized by providing new implementations of these attribute-specific internals of the `AttributeValue` class.

The refinement of the design of `AttributeValue` forms a substantial contribution of the whole thesis and is described in Section 4.3.

Attribute description

An attribute description is the only one from the triad of main entities that was not mentioned in the analysis section. As its name suggests, it contains information related to the attribute itself, e.g. the name of the attribute, its unique identification, the specification of the means of displaying and editing its values, etc. (see Section 4.7.2 for other attribute properties). Briefly, it aggregates all attribute-specific information that the framework needs for operating with the attribute values. Consequently, it indirectly influences attribute contributors since it models the information that should be supplied during the attribute contribution.

In this respect, `AttributeDescription` serves for internal needs of the framework and it is also the framework which is responsible for managing attribute descriptions. This is the reason why the `AttributeDescription` class is not a part of the attribute metamodel, which also implies that it is not included in the EMF serialized form of the ProCom models.

It can be argued that the division of these entities is optimal in the respect that joining any of these classes would result in some design troubles. Adding any extra-value information to `AttributeValue` would be partly redundant as there are possibly many attribute values attached

to an attribute and partly dangerous due to exposing information that should not be accessed by IDE modules working with the value. Joining attribute description and attribute would again be redundant since an attribute can be persisted in many ProCom models but attribute properties are shared between all instances of a particular attribute and they should therefore exist only in one copy.

4.3 Attribute value structure

This section contains the description of the design of the `AttributeValue` class together with the explanation of the major decisions made during its evolution. In the second part alternative solutions are discussed.

4.3.1 Conceptual structure

As mentioned before, an attribute value represents a particular piece of information attached to a ProCom model entity. It contains various kinds of information depending on the attribute type. However, apart from the actual information that an IDE user or an IDE module developer are interested in (e.g. a text description of a use case, a model of a timed automaton), another data describing the main information could be contained in an attribute value as well. These data include a version, a creation time, a source of a value, etc. All of them share some common characteristics and treating them separately from the 'main information' contained in an attribute value gains many benefits for the attribute framework. Accordingly, an attribute value has been conceptually separated into two parts called *data* and *metadata*.

Data generalize the actual piece of information representing a quality or a measure of an attribute (e.g. number of seconds measuring the worst-case execution time). Since the data part of an attribute value fully depends on an attribute type, its inner representation varies between different attributes. In the current design, it also holds that the data part is the only structural difference between values of two distinct attribute types. As a result, type extensibility of the attribute framework is implemented by having an extensible data part of an attribute value.

Metadata represent properties characterising the data part of an attribute value. Contrary to data, metadata do not depend on an attribute type and thus they can be shared among different attribute types. Metadata can be assigned different semantics; and accordingly, the possibilities of their usage are wide. One extreme approach considers metadata only as separated data with their own means for viewing and editing but otherwise identical with the same extensibility and flexibility as data. Another reasonable approach treats metadata in a completely different way. It requires a fixed (not extensible) set of metadata shared by all attribute types. Whereas data are used to represent an ever increasing variety of information, metadata are used to achieve the opposite tendency to unify, to categorize, to establish an order. The fixed set of shared metadata is used as a criterion for division to categories. The metadata part could be obligatory or optional depending on the required strictness of the categorisation.

The significance of metadata increases even more if multiple values of an attribute are taken into consideration. If we had an attribute of an integer type and we wanted to associate two values with it, an estimate made by an expert and a result of simulations, we would not be able to distinguish between the two values later. Inability to distinguish between the values of an attribute dramatically decreases their usability and practically makes them worthless. Metadata can play the role of a distinguishing characteristic refining the semantics of a particular attribute value.

On the other hand, metadata are only data, i.e. pieces of information. Consequently, it is possible for metadata to be included in the data part of an attribute value. Although it could be done, having a separate notion of metadata and distinguishing between data and metadata by the attribute framework brings several advantages to the users of the framework.

- various constraints on the values can be enforced by the framework (e.g. not allowing the values with the same metadata),
- contributors of new data types can focus only on the data structure they are interested in without having to deal with other data members like version, comment, etc.
- the framework can provide means for editing and viewing metadata, alleviating authoring of these modules.

4.3.2 The implemented design

The design of an attribute value that was implemented in the prototype implementation of the attribute framework (see Figure 8) respects the conceptual structure of an attribute value as described in the previous section. Since there are no other parts of an attribute value than data and metadata, its design description consists basically in refining the design of these components of a value.

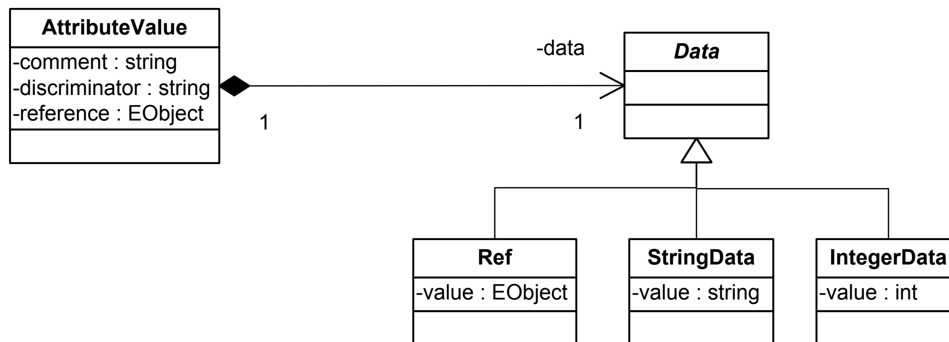


Figure 8: Attribute value structure

The data part of an attribute value is represented by a class inherited from the `Data` class. `Data` is an abstract EMF class having no own methods, which reflects the fact that there are no further behavior requirements imposed on the data part. The relationship between `AttributeValue` and its `Data` is implemented by a composition. This corresponds to the fact that value contains a data part. As depicted in Figure 8, there exist several descendants of the `Data` class representing the most common data types of attribute values. The `Data` class forms the point of extension of the attribute framework. The framework's type extensibility is realized by adding a new class inheriting from the `Data` class (see Section 4.4 for closer description of devised methods of extending the framework's attribute pool).

Implementing the metadata part of an attribute value (in Figure 8 corresponding to the attributes of the `AttributeValue` class) was not so straightforward process. The main decision shaping the resulting design was the choice whether to implement fixed or extensible metadata. By fixed metadata it is meant metadata that are hard-wired into the attribute metamodel giving no possibility for its extension other than changing the metamodel itself and are shared among all the attributes, as opposed to extensible metadata which could be extended in the same way as the data part of an attribute value. Although extensible metadata undoubtedly offer greater flexibility and are more general, for a number of reasons the fixed metadata design has been chosen.

Extensible metadata bring many implications which complicate the whole structure of the attribute framework. Similarly to extensible data, it would require the framework to offer an ability of defining new metadata types together with the means for their viewing and editing in GUI. Moreover, if certain constraints were to be enforced on the metadata, the framework would

require a language to describe them. And due to the extensibility of metadata, the language would have to be extensible enough to be able to express constraints on the new data types. Having potentially large and ever growing set of metadata also presents a serious challenge in designing an intuitive graphical user interface. In addition, the value added by this feature would be much less than the value of adding extensible data since changes in metadata are not supposed to be so frequent.

On the other hand, fixed metadata are significantly easier to implement since there is no need for general, extensible facilities because all the semantics and GUI editing support is hard-wired in the attribute framework. Not only are fixed metadata technically simpler, they also support the different approach to metadata (the latter approach from the previous section) where we want to minimize metadata for the sake of maximizing their capability to distinguish between attribute values independently on the attribute types. In other words, it is better to have a small number of well-defined metadata items widely used and understood by the users of the framework than having a lot of metadata with loose and unclear semantics.

In the prototype implementation, three metadata components of an attribute value have been included:

comment
: Comment is an optional part of an attribute value containing a string representation of a free-form user note related to the data part of the attribute value.

discriminator
: Earlier in the text, the possible usage of metadata as a distinguishing feature between the attribute values attached to the same attribute has been mentioned. Generally, the mere difference in metadata would be sufficient. However, in practice it might be convenient to simplify this general requirement. Especially, if metadata are used to categorize the attribute values, it is convenient to realize this by having a metadata element with the domain of all possible categories into which the values could be divided.[4] The only problem with this distinguishing metadata member is the precise definition of its semantics and consequently specification of its domain. It could be a source of a value: a human, an output of some computation, etc. Or, as proposed in [23], it could be *credibility*, another classification of a source of a value, consisting of asserted, verified, default and forced categories. Alternatively, it might express the trustworthiness of a value, etc.[5]

  The precise definition requires a deep expert understanding of the application domain, and although it is an interesting research question, it is out of the scope of this thesis. Nevertheless, a metadata member used for discriminating between the attribute values has been included in the prototype implementation as a proof of concept. Since there is currently no other semantics associated with it, it is represented as a string called discriminator. The framework does not strictly enforce its uniqueness among the values of the same attribute. Instead, if the uniqueness is broken, a warning is displayed and a user is informed about the fact having possibility to ignore the warning.

reference
: Introduction of this metadata element was motivated by an effort to support attributes relating to several model entities. For instance, we might want to measure the worst-case execution time of a computation between the two port groups of a particular service or, more precisely, the worst-case execution time between triggering a trigger port of an input port group and triggering a trigger port of one of the service's output port groups (since

[4] In this respect, the requirement for different metadata is a generalized case where a set of categories is a Cartesian product of the domains of all metadata elements.

[5] In all of the above mentioned proposals, the discriminator itself could not be the only distinguishing element between the values of the same attribute since there is a possibility of retrieving several attribute values from the same source (or using the same method) in different points in time.

there is only one trigger port per port group, it is really possible to speak about the execution time of a computation between two port groups). This situation is illustrated in Figure 9 where the worst-case execution time between the input port group (IPG) and the output port group (OPG1) within the service S1 is depicted. In this case, the attribute relates to more than one model entity and it is usable only if it is known to which entities it points. Owing to the lack of an entity able to group a set of other entities in the ProCom metamodel without assuming any semantics of this grouping, it is necessary to attach the information to one of the involved entities and then to specify other entities relevant for the attribute meaning (in our example we might want to attach the attribute to an input port group and then to specify a particular output port group[6]). In fact, the refinement of the semantics of the measured value to include the linked entities is needed. Accordingly, a metadata element specifying these links to other entities was chosen as the best way of implementing this feature.

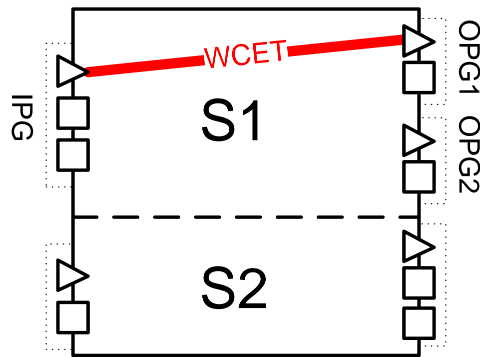


Figure 9: Attribute of the worst-case execution time (WCET) between two groups of a service

Nevertheless, having a reference able to point to an arbitrary model entity does not solve the whole problem. According to the semantics of the attribute type, the subset of model entities that can be possibly referred to by this reference should be specified (In our example, this set includes all output port groups of the same services as the input port group to which we attach the attribute). Since the conditions used to filter the set of reference targets can be possibly very complex, this filtering is delegated to an entity of *reference provider*. Based on the entity to which the attribute is being attached (and indirectly on the whole model which is accessible through this entity), the reference provider should return this set of entities. A reference provider can be specified as one of the items forming the definition of the attribute by attribute contributors (more information about reference providers can be found in Section 4.7.2).

Similarly to the discriminator, the reference serves rather as a proof of concept. The number of referred object has been intentionally restricted to at most one from two reasons. First, it would require implementing an identification of the references (e.g. by names) to distinguish between their roles in the relation. Second, we were not able to a devise a reasonable attribute from the application domain spanning more than two entities (thus requiring more than one reference). In the future, this needs to be reconsidered.

4.3.3 Rejected alternative designs

The evolution of the design realized in the prototype implementation was not a straightforward process. Many alternative solutions have been proposed and were later abandoned since they

[6] This example serves only for the illustration of the concept. Due to the fact that there is only one input port group in a service, the attribute can be attached to an output port group and the input port group can then be unambiguously determined without the need of having a reference to it stored in the attribute value.

exhibited some design flaws. In this section, the majority of these proposals is briefly mentioned, and their main benefits and drawbacks are summarized.

All-in-one

The initial set of designs originated before an attribute has been divided into the three conceptual parts of an attribute, an attribute description, and an attribute value. Their greatest disadvantage is the redundancy of extra-value data (`name` attribute in Figure 10) being repeated in every instance of attribute class (see Section 4.2 for closer explanation).

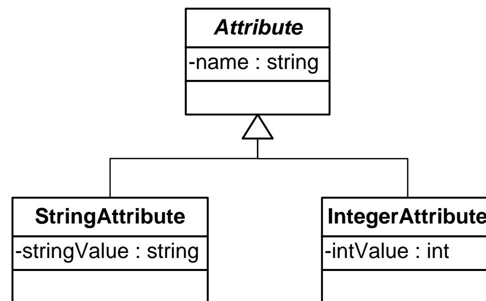


Figure 10: All-in-one attribute design exhibiting redundancy

Type definition language

This rather exotic alternative (depicted in Figure 11) includes the notion of a type into the ProCom metamodel. The fact that a value has a type can be modeled naturally by an association between the value and the type. By subclassing the `Type` class, it is possible to model a type system similar to type systems found in many programming languages. As a result, this proposal leads to the situation where type extensibility can be realized without extending the ProCom metamodel by simply modeling a new type from the constructs already present in the metamodel. It is worth noting that instances of descendants of the `Type` class are types, not instances of those types (e.g. an integer is *an instance of* `PrimitiveType`). These instances only describe some types, forming a language for definition of new types (hence the name of this design).

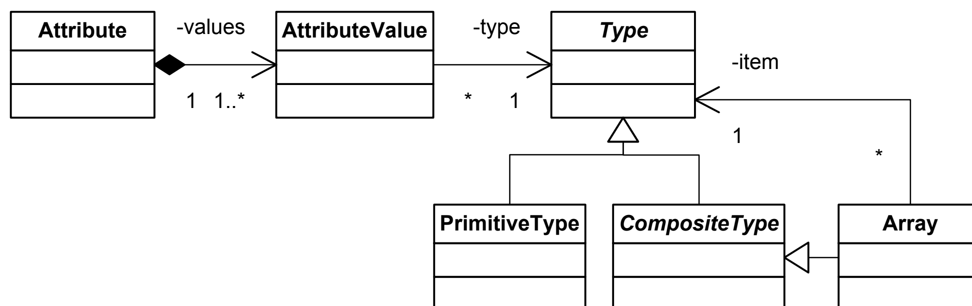


Figure 11: Attribute design leading to type definition language

The fact that at run-time there exist only type descriptions and not the classes implementing those types is the main disadvantage of this approach. The approach of type-less access to the data represented by the attribute value would have to be adopted to make this alternative feasible. Although dynamic EMF is able to dynamically create reflective representations of classes based on their structural description (our type definition in this case), instantiate them

and then provide reflective access to their instances, working with the contents of the values is much less convenient and more error-prone due to the lack of type safety (in contrast to the implemented solution). This resulted in the rejection of this design choice.

Inheritance-based approach to combining data

This design does not distinguish between data and metadata. However, it takes into consideration that a value might be composed of several parts, which is, in fact, a more general approach because no assumptions about the semantics of the value components are made, as in the case of metadata. The parts of the attribute value are either directly defined in the class representing the value, or they are inherited from another existing value class. In other words, inheritance is used as a method of combining existing value types. Due to the support for multiple inheritance in EMF, this mechanism was chosen to realize the actual combination (see Figure 12, where a value consisting of an integer and a reference is composed by inheriting from two classes containing respective components). The design was primarily created with the notion of a *composite editor* in mind. A composite editor would be a facility supplied by the framework for editing a composite value, created by composing several already existing editors. In the case of classes created purely using multiple inheritance (i.e. without adding any features that would not be inherited), composite editors could be automatically created from the editors of attribute types higher in the inheritance hierarchy (e.g. a composite editor for `IntegerRefValue` would be assembled from the editors for `IntegerValue` and `RefValue`).

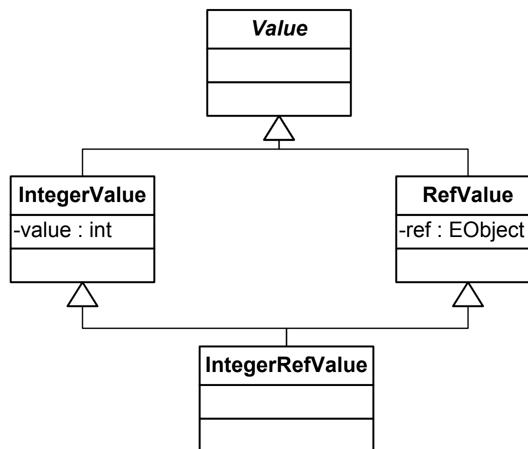


Figure 12: Combining data in an attribute value using multiple inheritance

A serious drawback of this design is that the number of composite classes in the metamodel grows very fast (exponentially in the worst case) since addition of a single class might require adding as many composite classes as there were classes before this addition, i.e. twofold increase of the number of classes in the metamodel. A slight modification of the design solving this disadvantage is to specify these composite classes outside the metamodel, during attribute contribution. But similarly to the solution described in the previous section, there would be no class implementing the composite class at run-time, which would again require dynamic EMF. Moreover, there are further drawbacks caused by the usage of inheritance. There is no order of inheritance meaning that composite classes represent only sets of data members inherited from their ancestors, not sequences as it might be required. Specially, it is not possible to inherit from some class multiple times to contain its data member more times (e.g. a pair of integers).

Extensible metadata

Treating metadata in the same way as the data results in the design in Figure 13. The advantage of being able to reuse the framework's extensibility mechanisms and the same facilities for editing and viewing metadata as the data of the same type are gained automatically. Again, the specification of which parts metadata should consist needs to be supplied externally during the attribute contribution. Contrary to the previous proposal, it is possible to include some data member several times since the order of elements in the metadata collection is preserved. Although it might seem that we can have extensible metadata at almost no cost, there are some serious disadvantages of this alternative.

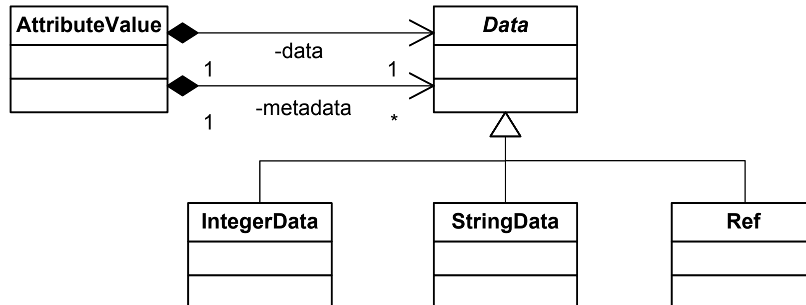


Figure 13: Extensible metadata in attribute value combined using composition

First, to fully use the power of metadata it is needed to have facilities for expressing various constraints on them (as discussed in Section 4.3.2). Second, as opposed to data, multiple metadata are expected to be associated with an attribute value. Although keeping them in an ordered sequence of elements suffices theoretically to distinguish their semantics based on their position, in practise it renders as an almost unusable solution. There arises the need of addressing metadata by name.

Recursive attributes

This design reacts to the need of addressing metadata by name. It investigates the approach employing recursion. The key idea motivating the whole design is the observation that named metadata basically do not differ much from the full-fledged attribute. By accepting this reasoning, the previous design of extensible metadata is modified in two ways. First, it is simplified by withdrawing the metadata relationship between `AttributeValue` and `Data`. Second, an attribute value is allowed to have attributes, representing conceptually metadata, by the standard way of subclassing the `Attributable` class.

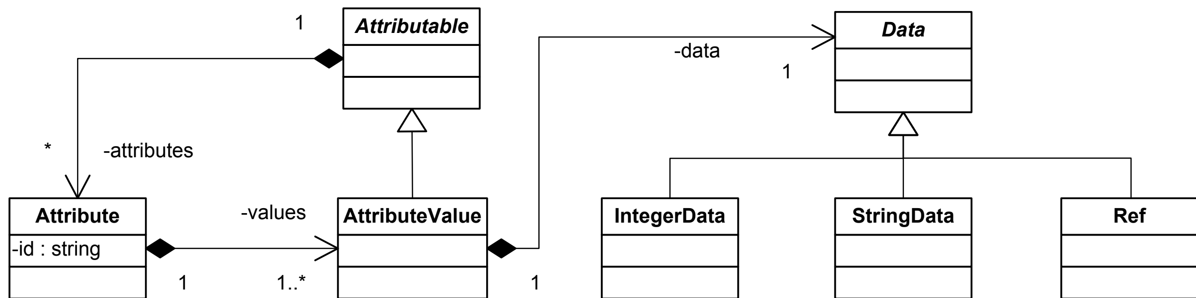


Figure 14: Recursive attributes design

The gains of this proposal include, similarly to the previous alternative, a complete reuse of the infrastructure provided by the framework for handling data (extensibility, editing and viewing GUI support). Moreover, by treating metadata as attributes they can be addressed by name as required, and the contribution mechanism for defining which metadata (i.e. attribute) should be applicable to which attribute can also be employed.

Using recursion brings an inherent problem of treating metadata in *exactly* the same way as attributes. Consequently, the rule that attributes can have attributes can be recursively iterated resulting in an unbounded nesting of attributes. Employing recursion also implies that metadata are multi-valued since attributes allow for multiple attribute values. These two facts were not intended to be incorporated in the design, and although the design offers unforeseen flexibility enabling to e.g. have several metadata distinguished by their version (employing both more than one meta-level and multi-valued metadata), it was dismissed since it is not possible to define data- or metadata-specific behavior.

4.3.4 Possible alternative solution: Named extensible metadata

As opposed to the alternatives presented in the previous section, the design described in this section was not implemented because it would have some major design flaws but because it realizes the approach of extensible metadata, which was not adopted in the current version of the prototype application. Nevertheless, it could be used in the future to increase the capabilities of the attribute framework.

This design proposal represents the improved version of the ‘Extensible metadata’ design replacing the unnamed collection of metadata by the collection of named `MetaData` entities. Contrary to the previous, recursive design it is possible to define different behavior for metadata and attributes, which can be seen in Figure 15, where metadata can contain only one data element.

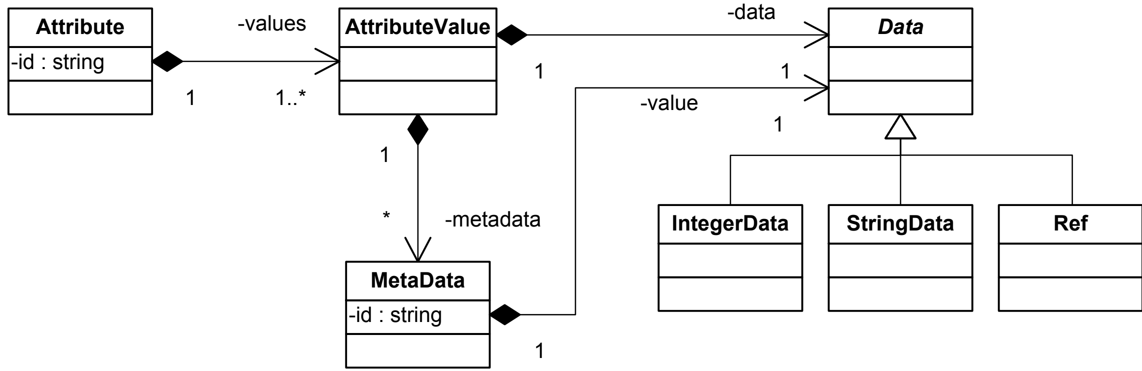


Figure 15: Named extensible metadata design

Due to the similarity of the designs, all advantages of 'Extensible metadata' are retained, adding the feature of named metadata. Providing metadata with identification does not imply only more convenient access to them, the identification can also be used for defining additional semantics.

To summarize, this design exhibits the best properties among all of the designs supporting extensible metadata, and although there are still some issues to be solved (e.g. how precisely the semantics of the metadata should be defined), it is a preferred way of the future evolution of the attribute value structure (in case that the extensible metadata would be required).

4.4 Type extensibility

The whole ProCom (including the attributes) is modeled as an EMF metamodel. One of the advantages of using EMF is the EMF-provided serialization support for models assembled from EMF-generated classes. Since ProCom models are expected to employ this serialization mechanism, attributes, as a part of the ProCom metamodel, should support it as well. However, this requirement complicates the process of extending data type pool of the framework. As mentioned in Section 4.3.2, the type extensibility of the attribute framework relies on adding new classes extending the `Data` class. Unfortunately, due to the properties of the EMF serialization mechanism, providing a class that was not modeled using EMF and is only implementing the `Data` interface[7] is not sufficient. Although instances of such a class can be normally used during working with the model, they are not serialized properly. Accordingly, the design of the framework's type extensibility must take this issue into consideration.

In the first part of this section, the EMF serialization is described to understand why the above-mentioned approach does not work, and an overview of methods of extending the framework's type pool using the EMF facilities is given. Next, another method avoiding working with EMF metamodels is proposed.

4.4.1 EMF-based methods

The EMF serialization mechanism relies on the structural description of features (attributes and references) of an EMF object represented by the `EClass` class. During the code generation process when EMF model elements are transformed into a Java code, not only Java classes directly representing the model entities are created. Information about structural features of these entities present in their EMF model is also transformed to the form of static instances of `EClass` in a corresponding model package (descendant of `EPackage`). When serializing a particular EMF object, EMF obtains its structural description (in the form of an `EClass` instance), which

[7] EMF classes are converted to Java interfaces during the code generation, see Section 2.2.3 for the description of EMF.

drives the whole serialization process — attributes of primitive types are serialized directly, and referred entities are serialized by calling the same mechanism recursively.

When a descendant of `Data` (precisely, a descendant of the `DataImpl` class) not modeled using EMF is being serialized, the framework attempts to obtain the `EClass` instance describing the object features. The element it receives is the description of the contents of the `Data` class, which does not contain any structural features. Consequently, an empty instance of `Data` is serialized. Obviously, the structural description corresponding to the `Data` descendant class is needed.

The first group of solutions to this problem relies on EMF facilities to generate these type descriptions. The solution in which the attribute metamodel is directly modified (i.e. a new class inheriting from `Data` is added) and then costly re-generated violates the aim of minimizing the impact of adding a new type on the metamodel (stated in Section 4.1). The reason is that code re-generation affects all the model classes located in the same EMF package where the modification was performed. Consequently, this alternative has been rejected.

The following two methods avoid the costly code re-generation and are therefore implemented in the prototype. They use different approaches and are appropriate for different cases.

The first one employs the EMF capability of creating a new model extending some existing EMF model. This means that the entities from the existing model may be referred from an extending model. Since the extended model is not modified in any way, its code is not re-generated during the code generation of the extending package. An attribute contributor models a new data type, a descendant of `Data`, in a separate EMF model extending the attribute metamodel, then he or she generates its Java code representation and specifies the type during attribute contribution.

In the second method, an EMF model can be referred from an attribute value even without the model being descendant of the `Data` class. This method uses one of the predefined classes inheriting from `Data`, the `Ref` class, which contains a reference to an arbitrary EMF model entity (`EObject`). Originally, it was intended to refer to the entities in the same ProCom model. But due to the EMF behavior in Eclipse platform based applications where all EMF models are 'published' so that they can be used without any special initialization, it can refer to an entity of another EMF model contributed to the running instance of the Eclipse platform. This approach does not require any modifications of the referred model, which makes it especially appropriate for already existing EMF models not designed with the attribute framework in mind. On the other hand, programmatic access to the model is complicated by being referred indirectly from an attribute value via the `Ref` class instance (resulting in one more dereference than the previous solution).

4.4.2 Externally serialized data

Both of the approaches suggested in the previous section share one disadvantage: they rely on EMF and therefore require attribute contributors to have an advanced knowledge of this technology in order to extend the framework's type pool. Considering that people who may need to add a new attribute type to the framework come from various backgrounds, the prior knowledge required to perform the action should be minimized. Motivated by this observation, we have devised a method employing only standard Java mechanisms.

At the same time, EMF was intended to be retained as a final means for serialization since it allows for having all the attributes serialized in one EMF-managed resource together with the rest of the ProCom model. This is achieved by delegating the serialization of the non-EMF contributed types to the attribute framework itself. However, creating a complete automatic serialization framework within the attribute framework is not desired. Instead, a user-supplied code performing the actual serialization of an object into a string is required to be contributed together with the non-EMF attribute type. The framework acts during the serialization as a translator. Upon receiving a request for an instance of a non-EMF type to be serialized, the

framework instantiates the class providing serialization support contributed in the definition of that type. The framework uses this instance for converting the object into a string of characters. This string is in turn wrapped by an instance of the `ExternallySerializedData`, a descendant of an EMF class containing just the string. Since it is a proper EMF class, it is serialized in a standard EMF way (the contained string is only copied to the persisted resource). When the attribute value consisting of `ExternallySerializedData` is accessed, the same procedure in reverse order must be performed (the persisted string is converted by the serialization support to an instance of the externally serialized type).

The key assumption making the whole method feasible is its *transparency* to the clients of the framework API. For them `ExternallySerializedData`, string-to-object and object-to-string transformations should be completely invisible. The default access to the data part of an attribute value (both reading and editing a value) has to be modified to include the transformation code. Details about implications of this method on the design of the client API can be found in Section 4.6.

For attribute contributors, extending the type pool using this method consists of implementing a class representing a new data type (inheriting from a special non-EMF descendant of `Data` to be able to be referred from an attribute value) and a class performing both transformations (to a string, from a string). To further facilitate adding new types through this method, the serialization support for classes ready to be serialized using the standard Java serialization mechanism has been implemented. This relieves contributors of such classes of their duty to implement the class performing serialization and deserialization.

4.5 GUI

In this section the design of the parts of the attribute framework related to GUI is covered.

4.5.1 Attribute View

In the part of analysis summarizing the requirements of IDE users (Section 3.4.3), we have concluded that there is a need for a centralized GUI control which displays attribute values of various types in a convenient way allowing for a uniform control, regardless of the attribute type. Taking the attribute value design into consideration, the complexity of displayed information further grows since it is not caused only by a variety of attribute types but also by the structuring a value into data and several metadata elements. In designing the GUI, the major task is to tackle the complexity of displayed information and, on the contrary, to provide a user with as simple GUI as possible. In accordance with Section 3.5.1, the central attribute GUI control is called Attribute View.

The Attribute View listens for the changes of selection in the Eclipse workbench (employing the standard Eclipse mechanism of selection service and lacking any direct dependence on the ProCom model editor). On the detection of an attributable ProCom entity being selected, it displays attributes and their values belonging to the selected entity. Thus, there arises the need for designing the way to display sets (categories) of multi-valued attributes of various types whose value is further decomposed into a data part, a discriminator, a reference, and a comment.

To comply with the requirement of intuitiveness of control, the design should reuse the widely used GUI paradigms well-known from other applications to a large extent. The closest paradigm is editing object properties as known e.g. from Eclipse. Here, a user is presented with a list of property-value pairs. Regardless of the type of the value, there is the same space allocated for each value limited typically by the height of a single row of text.

The alternative approach that was considered was inspired by the much greater type variety of ProCom attributes in comparison with e.g. Eclipse or other typical application using the properties paradigm. It lies in allocating different amount of space to attribute values according to their type allowing for much richer type-specific rendering of values, e.g. rendering a preview

of a timed automata or a graphical representation of probability distribution. However, this approach was dismissed due to anticipated high numbers of attributes (tens to hundreds). Having potentially tens of completely different visual representations of values, where each of them would be of a different size, possibly of different colors, would result in a confusing user experience caused by revealing too much of underlying information complexity.

Instead, in order to provide a user with a simple and thus more effective GUI, a plain text value representation, equally spaced regardless of the value type, has been chosen. Consequently, the definition of transformation to a string (realized by supplying the class implementing the `StringRenderer` interface) forms a part of an attribute type specification. The default plain text rendering of a value is supplemented by the rich, type-specific visualisation (described in Section 4.5.2) invokable by a user, needed to fully view complex values for which text representation is not sufficient.

In further decreasing the complexity of displayed information, the concept of categories (semantically related sets of attributes), similar to the one present in Eclipse, has been adopted. It relies on attribute contributors to classify the attribute being contributed according to the semantics of the attribute into one category. Contributors are allowed to extend the category pool if they conclude that there is not a proper one for their attribute. Additionally, when the category is not specified, the attribute is automatically moved into the General category with no semantics assumed. In the GUI, users are given the option of hiding the whole categories, which can significantly decrease the amount of information they have to deal with. Due to the anticipated maximum number of attributes within a range of hundreds, hierarchical categorisation of attributes has been dismissed, i.e. categories cannot be nested. Based on the assumption that attributes will be divided into categories uniformly, tens of categories including tens of attributes suffice to contain hundreds of attributes in total. Every increase of the depth of category nesting by one, adds an order of magnitude to the total number of attributes possible to be displayed (assuming again tens of attributes per category), but it also makes navigation through categories more complicated, which is undesirable.

Contrary to the typical concept of properties with simple values, the attribute framework introduces highly structured multi-valued attributes, which has to be reflected in the GUI. Multiple values of an attribute are rendered on separate lines in the graphical control and are visually nested in the attribute entry to indicate a user to which attribute they belong. The separation of a value into a data and metadata parts require further splitting of a graphical representation of an attribute value into several rendering cells. If the extensible metadata design was adopted, the number of value parts would be unbounded, posing an interesting problem of how to display them efficiently. One of the solutions can be found in Figure 16, where metadata parts are nested in the value entry.

Attributes

ExecutionTime	
Value	10
Ref	PortA
Ref	PortB
Comment	Rough estimate
Value	10.00808
Ref	PortA
Ref	PortB
Comment	Result of simulation

Figure 16: Displaying attribute values comprising of extensible metadata

Due to the adoption of fixed metadata, the design can rely on the constant upper limit of number of metadata members. This allows us to lay out the information in a more space efficient way taking into account also the semantics of the metadata. Thus, the area of the Attribute View is divided into several columns: one describing the data and the other columns corresponding to the particular data or metadata part of an attribute value. The discriminator metadata member, whose role is to distinguish between the attribute values, is integrated into the first column, as depicted in Figure 17.

Property	Value	Reference	Comment
◢ Category A			
◢ Execution Time			
Result of Simulation	12		Simulated using XY method.
Estimate	20		John Moose is responsible.
Measurement	14		This is an arbitrary memo...

Figure 17: Displaying attribute values having fixed metadata

The Attribute View is also a central point for controlling attributes. All of the use cases performed by an IDE user are realized through the means of the Attribute View except for 'Listing of attributes', which is invoked by selecting a model entity outside of the Attribute View.

4.5.2 Editors and viewers

Editors and viewers represent the framework's means for editing and viewing attribute values, realizing the requirements from Section 3.4.3. Since these facilities depend on the attribute value type, they have to be specified during the attribute contribution. From the perspective of an IDE user, these modules form the most important part of the attribute framework because they allow the user to access the attribute value, which is the main attribute-related activity performed by this group of users. Due to this fact, the design of editors and viewers has been created carefully putting emphasis on a close integration with the Attribute View and not limiting the authors of these modules by too restrictive requirements.

Both of these modules conceptually interact with an attribute value. However, working with data and metadata parts of an attribute value has been separated. Whereas editors and viewers interact directly only with the data part of an attribute value, accessing metadata is, due to the fixed metadata design, the responsibility of the framework. In the case of adopting the extensible metadata approach, metadata editors and viewers would be contributed together with each metadata element. Separating data and metadata editing and viewing, enables the authors of the modules to work with a less complex data structure (the data part versus the whole attribute value structure) and removes the redundancy of implementing the means for manipulating metadata in every attribute editor and viewer.

Editors

Editors are invoked by an IDE user from the Attribute View to change the data part of an attribute value. In order to deal with the opposing requirements of (i) seamless integration into the GUI and of (ii) the least possible restrictions on the editor classes, there exist three variants of data editors in the attribute framework. They differ in the extent to which they realize the latter or the former requirement, and it depends upon the editor implementor to decide what type of editor is the most proper for a particular data type. The editor types include:

- attribute editor

The most general editor interface contains only the methods necessary for editing the data (the methods will be described later in this section). Since it is not dependent on the Eclipse framework or SWT, it provides editor implementors with the option of implementing the editor using non-Eclipse technologies or, which is considered as more important, reusing already existing editor (e.g. an editor implemented in Java Swing or even an external application written in an arbitrary programming language).

- managed attribute editor

  This editor features the balance between a smooth GUI integration and not too restrictive interface for its developers. Managed editors are shown in windows managed by the attribute framework, but they contribute their own contents of the window. Having no prior limitations on the size or the contents of the editor window allows for editing of very complex types. However, the tighter integration with the framework introduces dependency on SWT. This type of editor is preferred over the former one when implementing an editor from scratch.

- cell attribute editor

  This kind of editor is integrated directly into the Attribute View. Upon its activation, it is placed into a cell of the table inside the Attribute View containing the data part of the attribute value. The seamless integration with the GUI implies strict restrictions on the editor implementations. The editor area is limited by the fixed size of a table cell, which makes it usable only for not so complex data types. Despite its restrictiveness, the cell editor is supported since it is assumed that the substantial number of attributes will be of primitive data types (e.g. strings, integers or decimal numbers).

Java interfaces corresponding to the particular editor types are shown in Figure 18. It should be noticed how the more restricted editors extend the more general editors. The methods of `AttributeEditor` are concerned only with actions required for editing a data whereas the other two editors have an extra method for creating their GUI in SWT. `AttributeCellEditor` is due to its different semantics treated separately even though it implements the same set of methods as `ManagedAttributeEditor`.

```java
public interface AttributeEditor {
  void edit(Data oldValue);
  void setContext(EditorContext context);
  Data getValue();
}

public interface ManagedAttributeEditor extends AttributeEditor {
  Control createContents(Composite parent);
}

public interface AttributeCellEditor extends ManagedAttributeEditor {
}
```

Figure 18: Interfaces for attribute editors

Although it has been noted that the methods of `AttributeEditor` are concerned only with data editing, their design might not appear so straightforward. Editing of a value can be conceptually represented as a single function taking the old value as an argument and returning the new edited value as the function value. This can be directly translated into Java as a method, leading to a simple editor interface. However, this approach proves to be too limiting.

The whole value editing must happen in a single method call, i.e. execution of the method calling the editing function is blocked during the editing since the methods of creating of the

editor and editing of the value are executed on the same call-stack *above* the calling method. Consequently, during editing of a value, the whole IDE would be blocked. Although possible, such behavior is not desired. Users might want to display the values of other attributes or to perform other actions in the IDE during editing the attribute value, and the editor design should not limit them in doing so. Moreover, this approach is suitable neither for cell editors nor for editors invoking an external application.

The blocking nature of a single editing function can be avoided by splitting it into two separate methods, from which one receives a value for editing (and returns immediately) and the other is used later for retrieving the edited result. However, this solution brings a problem of when should the framework retrieve the edited value. Obviously, after the editing is finished. But the framework itself is not able to determine this fact since the process of editing depends entirely on the editor and it might be quite complex (e.g. consisting of several steps only after which the new value is set). Instead, it is necessary to provide the editor with the means for informing the framework when the value editing is finished. This is achieved by providing the editor with the `EditorContext` object, which contains methods for reporting the editing status. Thus, `EditorContext` represents a communication point between the editor and the framework, which can be used for sending other information to the editor, e.g. currently, it is used for providing editor with validators (see Section 4.7.2). The implemented design is more general than the single editing function since the blocking behavior can be easily simulated by calling the `EditorContext`'s call-back on the end of the `edit` method.

Viewers

Viewers are invoked from the Attribute View to visualize the data part of an attribute value. Contrary to editors, they are not necessary as for the simple data types the default plain text representation can be sufficient. In this respect, string renderers can be viewed as default attribute viewers.

Similarly to editors, the effort to provide developers with comfortable viewer interfaces resulted in designing more types of attribute viewers:

- attribute viewer

  The general attribute viewer interface analogous to the attribute editor. Focusing solely on providing desired functionality independent on any technology, it is intended to be used for reusing existing means for viewing data.

- managed attribute viewer

  The managed attribute editor's counterpart dependent on SWT and supplied with the framework's managed window. It is required to provide the contents of the viewer window.

The design does not contain any attribute cell editor because as argued in Section 4.5.1, in order to keep the GUI simple, only the plain text representation is used in the Attribute View, which is the responsibility of string renderers.

```
public interface AttributeViewer {
  void displayValue(Data value);
}

public interface ManagedAttributeViewer extends AttributeViewer {
  Control createContents(Composite parent);
}
```

Figure 19: Interfaces for attribute viewers

As indicated in Figure 19, viewer interfaces were designed to be as simple as possible. Viewers are perceived as light-weight and short-lived means for visualizing the value, similar to e.g. the concept of a tooltip. This tooltip-like behavior is implemented by managed viewer windows, and it is also reflected in the fact that viewers are, due to keeping the maximum simplicity, not equipped with any mechanism notifying them about possible changes of displayed values.

4.5.3 Help

Another goal of the framework GUI lies in providing IDE users with documentation describing particular attributes, their semantics and usage. Although the documentation could be maintained separately from the IDE, the more tight its integration into the Progress IDE is, the more efficiently it can be used by IDE users. The following forms of help are proposed:

- help integrated into the Attribute View

  Temporary hints in the form of a tooltip invoked from the Attribute View by a user, displaying only short descriptions of selected attributes.

- standard context help

  The standard Eclipse context help, which displays the help to the currently selected GUI element in a separate view, is employed to display a short description of selected attributes, similarly to the previous kind of help. Additionally, it also provides a link to the more detailed attribute description (see the next item).

- detailed attribute description

  This documentation provides users with the complete information available to the framework about a particular attribute. It consists of two parts. The first is generated by the framework based on the information supplied during the attribute contribution. The second is a rich text document (HTML document) specified by an attribute contributor, containing the detailed description of the attribute, examples of usage, etc. This kind of help is accessible via links from the standard context help or using Eclipse help browsing facilities.

Due to the mechanism of attribute contribution (see Section 4.7), where attribute specifications are not necessarily located physically in one place but they can be distributed in many locations, the framework's ability to automatically generate an attribute documentation increases in importance. The generated summary of attributes becomes the only available and always up-to-date attribute reference manual accessible to IDE users.

4.6 Client API

As stated in Section 3.4.5, the client API represents the attribute framework's programmatic interface for module developers. The functional requirements imposed on the interface are expressed by the use cases listed in Section 3.5.2 and can be summarized as 'working with attribute values'.

The interface for working with attribute values is defined implicitly by the attribute metamodel (see Figure 7 and Figure 8) since EMF generates Java interfaces corresponding to the metamodel entities. However, from a module developer point of view, this interface is unnecessarily complex and confusing. First, the `Attribute` class is of no interest to the module developer, and it only presents an extra dereference when accessing attribute values. Additionally, navigating manually through a collection of attributes searching for a particular attribute is a tedious task resulting in blocks of boilerplate code, which is also undesirable. A possible solution, which has been chosen in the current version of the attribute framework, consists in modifying the implicit EMF-generated API to better suit the needs of module developers.

It has been designed to be simple and effective. All the functionality is provided through the methods of the **Attributable** class that were added to the attribute metamodel. The methods correspond to the particular use-cases for IDE module developers[8] (compare Figure 5 with Figure 20), and they use only objects that are significant for this group of users: attribute identifiers and attribute values. Additionally, the composition relationship between **Attributable** and **Attribute** is hidden in the generated **Attributable** interface. Unfortunately, this modification has to be performed manually in the EMF-generated Java code since there is no support for the concept of visibility of relations and entities in EMF.

```
public interface Attributable extends EObject {
  EList<AttributeValue> getAttributeValues(AttributeId id);
  AttributeValue createAttributeValue(AttributeId id);
  void removeAttributeValue(AttributeId id, AttributeValue value);
}
```

Figure 20: The client API implemented by the methods of **Attributable**

The methods of **Attributable** carry out the above-mentioned tedious tasks instead of module developers (e.g. navigating through the collection of attributes). Moreover, the attribute framework provides certain services that has to be performed transparently during attribute values manipulation in the background without the client's explicit demand, e.g. serializing or deserializing externally serialized data (Section 4.4.2) and validating operations performed on the data part of an attribute value (Section 4.7.2). These additional functionalities are also invoked from the methods of **Attributable**. As a result, there exists the tight coupling between the attribute metamodel and the framework since **Attributable** needs to access the framework's services and, on the contrary, the framework operates with the attribute metamodel entities. Due to this coupling, the attribute metamodel and the attribute framework had to be merged into one plugin to avoid circular inter-plugin dependencies. This has led to the separation of the ProCom metamodel and the attribute metamodel, making the attribute framework independent on the ProCom metamodel and thus reusable in other contexts.

The greatest drawback of this design is the necessity of manual modification of the generated code (event though it is only hiding one method in a generated interface). On the other hand, it brings an intuitive and easy to use interface for module developers.

A possible alternative, which was implemented before adopting the current design, is creating a completely new interface forming another layer above the EMF-generated interface (e.g. a singleton [14] called **ValueProvider**). The transparent validation and serialization of attribute values could be realized by **AttributeValue** proxies [14] wrapping the original **AttributeValue** instance and providing the respective functionality (Figure 21).

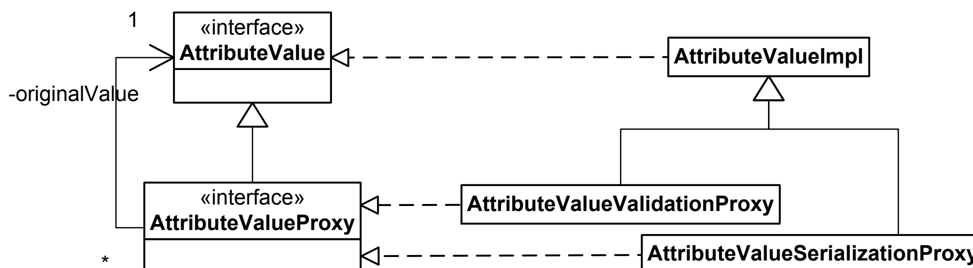


Figure 21: Attribute value proxies

[8] The **createAttributeValue()** method combines the use-cases of 'Adding an attribute value' and 'Creating an attribute value' by adding the newly created attribute value to other values of the attribute.

Although the design nicely separates the framework code from the EMF-generated code, its inherent disadvantage, which led to its rejection, is impossibility to hide the EMF-generated interface. Bypassing this high-level interface by using the EMF-generated one would lead to inconsistencies due to skipping the functionality provided by serialization and validation proxies.

4.7 SPI

The framework's interface for extending its attribute pool has been termed SPI in Section 3.4.5. Contrary to the client API, which is implemented by methods of a single class, the SPI is formed by more constructs. Basically, the SPI is comprised from the two following logical components:

- *an attribute definition*, the specification of various items forming together support for working with the attribute and its values using the attribute framework,
- *a realization of the items specified in the attribute definition* which either influence some aspect of the semantics or some support for handling the attribute values in the PROGRESS IDE, e.g. Java classes implementing some required interfaces, declarative behavior specifications of some parts of the framework, HTML documents.

These logical parts of the SPI are described in the following subsections. In Section 4.7.1 the chosen mechanism for attribute definition is discussed. The detailed list of all items of which the attribute definition consists is presented in Section 4.7.2.

4.7.1 Attribute definition mechanism

As explained in Section 4.1, due to the difficult adding of a new attribute and cluttering the metamodel, attributes cannot be defined directly in the ProCom metamodel. There arises the need for another mechanism for attribute definition. Attribute contributors should define new attributes by means of this mechanism and it should serve as a source for initialization of the framework's attribute pool as well, thus being a persisted form of the attribute pool. One of the possible candidates is an XML file edited by attribute contributors playing the role of the framework's configuration file since, if properly designed, it can be easily comprehensible for humans as well as being readily parsed programmatically due to the strong tool support. Furthermore, the tree structure of an XML file is suitable for the high numbers of items of various types specified during the attribute definition. However, owing to the fact that the PROGRESS IDE is an Eclipse RCP application, there exist more variants to choose from.

In a certain sense, each addition of an attribute can be viewed as extending the framework's capabilities. In Eclipse, one of the fundamental mechanisms aiming to support contribution of functionality between plugins is called an *extension point mechanism*.

An extension point is a precisely defined point of extension of plugin capabilities. It consists of an XML schema definition and a set of Java interfaces or classes, which essentially define the API or, more accurately, the SPI for extending the plugin's functionality. A plugin contributing some functionality into a particular extension point defines so-called *extension*. The extension is formed by an XML mark-up in the plugin's manifest file (complying with the extension point's XML schema) which specifies classes from the contributing plugin implementing the required interfaces or inheriting from the required classes realizing the contributed functionality. The Eclipse platform run-time aggregates all extensions of an extension point and makes them available via its API.

The extension point mechanism was preferred to the variant of XML files and was chosen to be the attribute definition mechanism of the attribute framework. The extension point mechanism retains the above-mentioned advantages of using an XML file because extensions are just portions of an XML code. Additionally, the framework can use the tools provided by the Eclipse platform for convenient accessing the extensions, and so can do the attribute

contributors. Based on the supplied extension point schema, the Eclipse IDE generates GUI editors allowing for defining extensions in a comfortable way without the need of working directly with their XML representation.

The attribute framework defines the extension point which enables expanding the framework's attribute pool, and attribute contributors can define extensions of this extension point in their plugins (see Figure 22) contributing new attribute types and other classes necessary for the framework to work with new attributes.

```
<extension point="se.mdh.progresside.attributes.registration">
      <attribute
            id="se.mdh.progesside.attribute.memo"
            name="memo"
            targetType="Component"
            targetPackageURI="http:///ProComMetamodel/ProSave.ecore"
            categoryId="my_category"
            dataType="StringData"
            dataPackageURI="http:///BasicAttributeTypes.ecore"
            editorClass="se.mdh.progresside.attributes. ↩
               BasicAttributeTypes.editors.StringDataCellEditor"
            stringRendererClass="se.mdh.progresside.attributes. ↩
               BasicAttributeTypes.viewers.StringDataRenderer"
            briefDescription="Arbitrary memo"
            documentation="html/memo.html"/>
</extension>
```

Figure 22: Sample extension of the attribute contribution extension point

Possible drawback of extension points might be their inherently distributed nature. Since every extension is defined in the manifest file of a contributing plugin, there is no central location from which the set of attributes that the framework's attribute pool actually contains could be easily determined. However, this can be alleviated by giving this option at run-time, using the facilities provided by the Eclipse platform to aggregate all the extensions. Indeed, this approach was adopted and the option of listing all the attributes contributed to the framework was incorporated in the help (see Section 4.5.3).

4.7.2 Attribute specification

To define a new attribute, an attribute contributor has to specify all characteristics of the attribute as well as facilities used by the framework for working with values of the attribute. Throughout this chapter, most of them has already been discussed. Here follows the complete list of items[9] that can be specified during attribute contribution:

identification
A unique identifier of the attribute used for programmatic access to the attribute.

name (label)
A user-friendly label of the attribute to be shown in GUI.

target type
A specification of the class of model entities to which the attribute can be assigned (e.g. component, subsystem).

target package URI
A URI of an EMF package containing the target type class.

[9] Items with asterisk (*) are optional, i.e. they do not have to be specified during the attribute contribution.

category*
: An identifier of the category to which the attribute belongs. Unless specified, the attribute belongs to the general category (Section 4.5.1).

value data type
: A specification of a class representing the attribute value. It can be either an EMF class or a Java class, which will then be externally serialized (Section 4.4.2) by the framework.

data package URI*
: A URI of an EMF package containing the value data type. The item has to be specified if and only if the value data type is an EMF class.

serializer
: A utility (a Java class) used for converting an externally serialized value (Section 4.4.2) to a string and vice versa. This item has to be specified if and only if the attribute value is an externally serialized Java class.

viewer*
: A utility (a Java class implementing one of the viewer interfaces) responsible for viewing the attribute values (Section 4.5.2).

editor
: A utility (a Java class implementing one of the editor interfaces) responsible for editing the attribute values (Section 4.5.2).

string renderer
: A utility responsible for rendering the attribute value as a string in the Attribute View (Section 4.5.1).

reference provider*
: A utility (a Java class) providing a set of model entities to which the attribute can refer. The set is offered to IDE user to choose from when they specify the reference metadata member of the attribute value (Section 4.3.2).

This item is specified in the case that the attribute relates to another model entity than the one it is attached to. The type of the referred object depends on the semantics of the attribute (e.g. execution time between two ports, attached to an input port, requires an output port as the type of referred entity). However, the condition constraining entities that may be potentially referred to can be quite complex, involving even other model entities and their attributes (e.g. only output ports in the *same* service should be considered). Instead of developing a declarative constraint language for this purpose, the task of determining the set of potential reference targets is delegated to a reference provider. The set of model entities returned by the reference provider is based on the model entity to which the attribute is attached (and through this entity indirectly on the whole model).

If specified, the reference provider item causes the framework to require IDE users to fill the reference metadata element. Otherwise, the reference cannot be filled by the means of the GUI.

validator*
: A utility (a Java class) evaluating validity of the data part of an attribute value and determining the cause of invalidity.

Attribute value data part validation is another service provided by the attribute framework enabling restriction of the data domain beyond the constraints defined by the data type itself. It was developed to support data types which are minor modifications of existing data types, having a subset of the original data type domain as their domain (e.g. even

integers as a subset of the integer data type domain). Although every constraint can be realized by introducing a new data type enforcing it, validators give authors the possibility to reuse existing facilities for handling the original type (e.g. viewers and editors), which would not be possible if implemented as a new data type. The framework warns IDE users, or it even aborts any modifying operation leading to invalid data (see the next item for further explanation). Furthermore, editors can access validators through the editor context (Section 4.5.2) and consequently integrate validation into the process of editing of the data within the editor in order to provide an IDE user with more comfortable user experience.

invalid data action*

Specification of the framework's behavior in reaction to an invalid attribute value data (classified as invalid by a validator). There exist two possible options. The first one has a strict semantics when the framework aborts any operation leading to an invalid attribute value (even the operation made programmatically by an IDE module developer). The second one allows every operation on an attribute value. The framework only displays warnings for the invalid attribute values.

brief description*

A very brief (at most a couple of sentences) summary of the meaning of the attribute shown as a tooltip in the Attribute View and in the context help (Section 4.5.3).

documentation*

A path to the file containing the text (in the HTML format) thoroughly explaining the semantics of the attribute. This documentation is merged with an automatically generated attribute documentation into the 'detailed attribute description' (Section 4.5.3).

Chapter 5

Prototype description

In this chapter the prototype implementation of the attribute framework for ProCom is briefly described. First, the main parts of the prototype are outlined so as to specify its scope (Section 5.1). Second, the internals of the framework implementation are shortly described. Finally, the usage of the prototype from the perspective of an attribute contributor and an IDE user is demonstrated.[1] (Section 5.3).

5.1 Scope

Since the attribute framework is integrated to the PROGRESS IDE, an Eclipse RCP application, it is split into several plugins (see Section 2.2.1 for the definition of a plugin and its role in the Eclipse Platform). The following list enumerates the plugins which were created during the works on the attribute framework:

- EMF-generated component model editor

  Due to the absence of a component editor in the current early stage of the IDE development, a simple component editor has also been developed to provide a platform into which the attribute framework could be integrated. It is based on the ProCom metamodel extended with the attribute metamodel which was implemented as a separate EMF model.

  The editor has been generated by EMF and then customized to cooperate with the attribute framework. First, the attribute metamodel editing is not performed by the editor; instead, it is delegated to the attribute framework. Second, the default editor behavior of displaying properties in the standard Eclipse Properties view has been changed to use the attribute framework GUI.

  The component editor comprises three plugins:

  – `se.mdh.progresside.procom`
    The plugin contains the ProCom metamodel and its EMF-generated Java representation (classes and interfaces).

  – `se.mdh.progresside.procom.edit`
    EMF.Edit framework-generated adapters of metamodel entities enabling easy integration with the Eclipse UI (providing labels, pictures and properties) can be found in this plugin. The editing support for attribute metamodel entities has been removed from the plugin as it is ensured by the attribute framework.

  – `se.mdh.progresside.procom.editor`

[1] The last group of users not mentioned, module developers, is supposed to use only the API described in Section 4.6.

This plugin contains simple ProSys and ProSave component model editors generated by EMF.Edit. Models are displayed in a tree from which all the modification actions are invoked. The editors have been adjusted to use the attribute framework GUI for displaying and modification of entity attributes.

- Attribute framework core (`se.mdh.progresside.attributes.framework`)

  The plugin provides the whole functionality of the attribute framework with the exception of GUI. It includes:

  - the attribute metamodel and its EMF-generated Java representation,
  - the client API (modified `Attributable`),
  - the definition of extension points used for contributing new attributes and new categories of attributes,
  - interfaces of the classes that should be included in the attribute specification during its contribution (editors, viewers, string renderers, serializers, validators, reference providers),
  - support for external serialization and a sample serializer using the Java Serialization API,
  - validation support,
  - an attribute registry, an internal structure of the framework keeping records of contributed attributes and categories and providing them to the rest of the framework,
  - an instance provider, an internal entity responsible for managing instances of contributed classes.

- Attribute framework GUI (`se.mdh.progresside.attributes.framework.ui`)

  All the GUI-related functionality of the attribute framework integrated with the Eclipse UI is located in this plugin. It is comprised mainly from the implementation extending the standard Properties view[2] to work with the ProCom attributes. The more detailed list of included items follows:

  - an own property sheet page, a GUI component forming the contents of the Eclipse Properties view,
  - a set of classes forming the model for the tree viewer displaying attributes and categories in the page mentioned in the previous item,
  - commands of adding a value, removing a value, viewing a value, editing a value and adding a new attribute to an entity, contributed to the Properties view,
  - a tabbed property sheet page aggregating the own property sheet page (the first item of this list) and the standard EMF property sheet to enable editing of both attributes defined in the ProCom metamodel and those contributed through the means of the attribute framework,
  - support for managed editors and viewers (managed windows for these modules and their manager),
  - facilities integrating with the Eclipse help system (components providing the help context for the selected attribute GUI items, generators of attributes documentation, and the table of contents of the help section containing the attribute reference manual).

[2] The Attribute View mentioned in the analysis and design is in the prototype implementation realized by the means of this standard view.

- Sample attributes (`se.mdh.progresside.attributes.framework.basicTypes`)

  The plugin contains several attributes of various types contributed to the attribute framework to illustrate the methods of extending the attribute pool. Apart from the type definitions, it also includes editors, viewers and other classes extending the framework's capabilities to allow for working with the values of the contributed attributes.

5.2 Implementation overview

Although many parts of the attribute framework have already been described in detail, there are still areas that have not been covered in the preceding text. Mostly they are the internal parts of the framework infrastructure not exposed through the client API or SPI. However, to summarize functioning of the framework prototype, it is helpful to give a brief overview of these parts of the framework.

5.2.1 The framework core plugin

All the main functionality of the core framework plugin (`se.mdh.progresside.attributes.framework`) is accessed through the facade of the `Attributes` class. It provides the basic entities supporting the framework[3]: attribute and category registries, and an instance provider.

The attribute and category registries are singletons created and initialized during the initial invocation of `Attributes`. The Eclipse Platform runtime is queried to obtain the list of all the registered extensions of the extension points defined by the framework for adding new attributes and new categories (carried out by the classes in the `contribution` subpackage). Based on this information, inner data structures of the registries are initialized and the registries are ready to be queried. The registries keep information about attributes and categories in the form of instances of `AttributeDescription` and `CategoryDescription` respectively. They basically encapsulate attribute or category definition supplied during contribution and can be accessed conveniently based on a given information (all attributes of a given entity possibly filtered by a category, the attribute or category of a given identification).

The instance provider is another singleton accessible by means of the `Attributes` facade. It is responsible for managing instances of the classes supplied during the attribute contribution. When the framework needs an instance of one of these classes (string renderers, editors, viewers, validators, serializers, reference providers), the instance provider is delegated to carry out the task. The instance provider, which is again only a facade to the factories corresponding to the above-mentioned classes, in turn obtains the required instance from the respective factory and returns it back to the framework. One factory is always responsible for the classes implementing one interface or inheriting from one base class. Apart from instantiating the classes, a factory is able to reuse the class instances. This is realized by adding the `ReuseInstance` Java annotation to the definition of a class contributed to the framework. When a factory is about to instantiate a class with this annotation, it does not create a new instance; instead, it reuses the single instance of this class. Thread-safe classes can use this feature to make the framework more efficient by avoiding instantiation each time the instance is requested.

There exists an interesting implementation detail shared by all of the above-mentioned classes and, in fact, by all classes and interfaces in the attribute framework that take an attribute identifier as a parameter of their methods. Although an attribute is logically identified by its name, a string supplied during the attribute contribution, in the framework the attribute is primarily identified by an instance of the `AttributeId` class. If it was implemented as a string, clients would be able to easily create strings that are not valid identifiers of the attributes contributed to the framework (regardless of whether intentionally or mistakenly). This would have to be reflected in the design of interfaces of the classes taking attribute identifiers as parameters by

[3] The client API, realized by `Attributable`, has been described in detail in Section 4.6.

including mechanisms indicating an error when the incorrect identifier had been entered (e.g. by throwing an exception). As a result, the class interfaces would be more complicated and, even worse, their usage would be inconvenient due to the necessity of checking whether the error occurred during their invocation. Inspired by the advice of avoiding strings in an API whenever possible [2], `AttributeId` has been created to alleviate this problem. Contrary to strings, when using instances of `AttributeId` as attribute identifiers, it is possible to ensure that every existing instance of this class corresponds to a contributed attribute. Therefore, there is no need for checking the validity of identifier and no means for indication of an invalid identifier are required, the mere existence of an `AttributeId` instance is a sufficient guarantee of validity. This invariant is ensured by restricting instantiation of the `AttributeId` class. There exist no public constructors; instead, a static factory method is supplied that takes a string attribute identifier as its only parameter. Upon invocation, the factory method checks whether the identifier is a valid attribute identifier by means of the attribute registry, and if so, it creates the corresponding instance. Otherwise, it throws an exception indicating the failure. In other words, the necessity of error checking in every method in the framework using the attribute identifier has been delegated to a single method of the `AttributeId` class.

5.2.2 The framework GUI plugin

Whereas the framework core plugin mainly provides functionality by exposing various APIs and own extension points, the framework GUI plugin primarily extends the Platform[4], frequently using the API exposed by the core plugin. As mentioned in Section 5.1, the major part of the GUI plugin consists of the implementation of an own property sheet page, associated UI components, and their models.

Although the Eclipse Platform provides a standard property sheet page[5], which supports viewing and modification of the properties of an entity selected in the workbench, it is not extensible enough to meet the requirements of the designs of the framework GUI. First, it is based on the property-value paradigm enforcing two-column realization of a user interface, which restricts the set of possible designs (in fact, only the design depicted in Figure 16 would be feasible). Second, the lack of support for the standard Eclipse extension mechanisms of publishing the entity selected inside a view through the selection service and an extensible context menu implies impossibility of creating user actions adding some new functionality. As a result, the own implementation of a property sheet page has been created. It uses the standard JFace tree viewer to display categories, attributes, and their values. Contrary to the standard property sheet page, it contains multiple columns corresponding to the attribute value metadata elements, and it supports the above-mentioned extensibility mechanisms of extensible context menu and publishing a selection.

Commands in the context menu and the upper toolbar can be therefore contributed by the standard Eclipse means — through extensions extending a particular view or a context menu. The handlers of these commands obtain a selection from the tree viewer and consequently perform the requested action using the API exposed by the core framework plugin (the client API, attribute and category registries, or the instance provider).

However, by using the MVC-based control of the tree viewer, a model for this control has to be implemented. Since the entities of category, attribute and attribute value should not be of a concern to a UI component, a different set of classes has been developed to become the model of the tree viewer. They are the classes implementing the `AttributePageEntry` interface, which represents a tree node having references to children and a parent and providing the labels used by the tree viewer. In addition, they contain a link to the attribute metamodel entity they represent, which is required by the commands handling the selection in the tree

[4] Currently, it contains 10 extensions of different extension points.

[5] Property sheet denotes the Property view, and a property sheet page represents the UI component rendering the contents of this view.

viewer manipulating these metamodel entities. Furthermore, attribute page entries listen to the changes of the metamodel entities they encapsulate and react to them accordingly by modifying their labels, sets of their children, etc. ensuring an up-to-date state of the Property view.

Another important part of the GUI plugin is formed by the implementation of user assistance features of the attribute framework. They include classes extending various Eclipse extension points: they dynamically provide the context help for the UI items selected in the Properties view, generate the table of contents of the help section listing the attributes contributed to the framework, and generate the documentation to these attributes.

5.3 Demonstration

Some of the main features implemented in the prototype application are demonstrated in this section. The first part of the demonstration focuses on attribute contribution whereas the second part is dedicated to describing the attribute framework GUI. As its name suggests, this section is not intended to be a complete user documentation; instead, its goal is to illustrate the usability and versatility of the mechanisms proposed during the design of the attribute framework.

5.3.1 Attribute contribution

Attribute contribution is exemplified by contributing two attributes: the worst-case execution time and a bitmap image. The attributes employ different mechanisms provided by the framework illustrating its various features.

The worst-case execution time

The worst-case execution time[6] (WCET) between two port groups of a service (see Section 4.3.2 and Figure 9) or, more precisely, between an input trigger port of an input port group and an output trigger port of one of the output port groups within the same service is measured as an integer number of milliseconds, always greater than zero. More specifically, it is an attribute of the input trigger port, and the output trigger port to which the WCET is measured is referred by the attribute value.

The integer type is one of the ProCom sample data types and thus it can be reused. The additional constraint requiring positive integers can be enforced by a validator (see the code in Figure 23). Additionally, because it is unacceptable for a value of this attribute to contain non-positive integer, the framework should be configured to abort any operation that would violate the constraint (including programmatic modifications of the value). Preferring a validator over implementing a new data type with the constraint built-in brings the advantage of possible reuse of viewers and editors already implemented for the integer type. Since an integer is a primitive type, the default string renderer is sufficient, and it is not necessary to contribute any viewer. From the same reason, a cell editor (Figure 32a) represents the optimal choice of an editor for this type.

[6] The worst-case execution time of a computational task is the maximum length of time the task could take to execute on a specific hardware platform.

```
public class PositiveIntegerValidator implements DataValidator {
  public boolean isDataValid(Data data) {
      IntegerData integer = (IntegerData) data;
      return integer.getValue() > 0;
  }
  public ValidationReport getValidationReport(Data data) {
    boolean valid = isDataValid(data);
    return new ValidationReport(valid ? "" : "Integer is not positive ↩
       ", valid);
  }
}
```

Figure 23: Simplified code of the validator (error checking omitted)

When specifying the reference to an output trigger port in the IDE, an IDE user has to specify only output trigger ports within the same service as the input trigger port to which the attribute is attached. To ensure correctness of the user input as well as user comfort, the mechanism of reference providers has been devised. Contributing the reference provider has two implications. First, it indicates that the attribute must refer to another entity to be valid. Second, it provides users with a set of correct possible reference targets to choose from. In this case, the reference provider returns a set of appropriate output trigger ports based on an input trigger port to which the attribute is attached (see Figure 24).

```
public class OutputTriggerPortProvider implements ReferenceProvider {
  public List<EObject> getObjects(EObject object) {
    List<EObject> outputTriggerPorts = new ArrayList<EObject>();
    InputTriggerPort port = (InputTriggerPort) object;
    Service s = (Service) port.eContainer().eContainer();
    for(OutputPortGroup outGroup : s.getOutputPortGroup()) {
      outputTriggerPorts.add(outGroup.getTrigger());
    }
    return outputTriggerPorts;
  }
}
```

Figure 24: Simplified code of the reference provider (error checking omitted)

The contribution of every attribute must also contain a unique attribute identification used for programmatic access to the attribute, and, optionally, an attribute category, a brief attribute description, and attribute documentation in the form of an HTML document.

To complete the attribute contribution, it is necessary to create an extension of the extension point defined by the framework for contributing new attributes. This can be done either by means of a GUI editor of extensions points or directly by entering the extension mark-up into a plugin manifest, as depicted in Figure 25.

In summary, to contribute the considered attribute it was sufficient to implement two simple Java classes (validator and reference provider), three other classes were reused (data type, editor and renderer), an HTML documentation was created, and finally the attribute was contributed through an extension point mark-up.

```
<attribute
      id="se.mdh.progesside.attribute.portsWorstExecTime"
      name="Worst-case Execution Time"
      targetType="InputTriggerPort"
      targetPackageURI="http:///ProComMetamodel/ProSave.ecore"
      dataType="IntegerData"
      dataPackageURI="http:///BasicAttributeTypes.ecore"
      editorClass="...IntegerCellEditor"
      stringRendererClass="...IntegerDataRenderer"
      validatorClass="...PositiveIntegerValidator"
      invalidDataAction="OPERATION_ABORT"
      refProvider="...OutputTriggerPortProvider"
      briefDescription="Worst-case execution time between an input ↩
         and an output trigger ports within the same service"
      documentation="html/wcet.html"
      categoryId="your_category" />
```

Figure 25: Contribution mark-up for the WCET between two port groups of a service (class names shortened)

Bitmap image

The next sample attribute contribution involves adding of an attribute of a bitmap image to a component. Whereas the previous contribution example employed rather advanced facilities of the framework without extending the attribute type pool, this attribute represents a fairly complex data type with no existing data representation and with no means for viewing or editing to be reused.

There exist two options when adding a new attribute type: to create an EMF metamodel or to use externally serialized data. In this illustrative contribution, the latter option will be selected[7]. As described in Section 4.4.2, this option consists in creating a Java class inherited from the `AbstractData` class (a special descendant of `Data`) and specifying the way how the class instances should be serialized (so-called serialization support) to strings that should encode the values represented by the instances (and vice versa for deserialization). However, in the case of very complex types whose instances can consume much more memory than the rest of the model (as in this case), the direct string encoding might not be appropriate. Instead, an indirect encoding, in which the serialized string refers to a resource containing the actual value data, is more suitable. In case of a bitmap image, such an indirect encoding can be a file-system path to the image file.

Thus, the Java representation is a class which contains a path to an image file and provides access to the data contained in the file, as shown in Figure 26. Consequently, serialization support restricts to saving the image path during serialization and building the class instance initialized by the path when deserializing (see Figure 27).

[7] In the case of taking the EMF metamodel approach, the metamodel extending the attribute metamodel would have to be created. After generating its Java code representation, viewers and editors would be implemented in the same way as in the case of the externally serialized data.

```
public class ImageData extends AbstractData {
  private String file;
  public String getImageFile() {
    return file;
  }
  public void setImageFile(String file) {
    this.file = file;
  }
  public Image getImage(Display display) {
    return new Image(display, file);
  }
}
```

Figure 26: Simplified implementation of the image data type

```
public class ImageDataSerialization implements SerializationSupport {
  public Object deserialize(String str) throws SerializationException ↩
      {
    ImageData data = new ImageData();
    data.setImageFile(str);
    return data;
  }
  public String serialize(Object o) throws SerializationException {
    ImageData data = (ImageData) o;
    return data == null ? null : data.getImageFile();
  }
}
```

Figure 27: Serialization support for the `ImageData` class

Another question that needs to be solved when contributing a completely new type lies in deciding what types of viewers and editors should be chosen. Generally, the more complex types, the more general editors (i.e. managed or general editors rather than cell editors) and viewers, and consequently worse integration with the IDE. Often, there are more ways in which editors or viewers could be designed, and it then depends on the semantics of the attribute which to choose. For instance, editing of the bitmap image attribute might be realized either by selecting another image from a file-system or by changing the image data (pixels) or by combination of the both ways. In this example, the former style of editing will be adopted. The editor can be implemented as a general attribute editor using the system dialog for selecting another image file (Figure 32c). The image viewer can be realized as a managed viewer (Figure 31), which implies that images will be displayed in managed windows directly in the IDE. Alternatively, a general viewer could also be used. A possible application might involve executing an external program capable of a more comfortable viewing (and even editing). Figure 28 shows how easily such an external viewer can be implemented (it launches the system default image viewer).

```
public class ExternalImageViewer implements AttributeViewer {
  public void displayValue(Data value) {
    if (value instanceof ImageData) {
      ImageData data = (ImageData) value;
      Program.launch(data.getImageFile());
    }
  }
}
```

Figure 28: Viewer displaying an image in an external application

Additionally, before contributing a new data type, a string renderer, a component responsible for the default rendering of a value in the Attribute View has to be implemented. The contribution is completed by creating an extension mark-up specifying the serialization support, the editor, the viewer, the string renderer, and the remaining obligatory items (listed in Section 4.7.2).

5.3.2 Prototype GUI

The cornerstone of the attribute framework GUI is the Attribute View integrated into the standard Eclipse Properties[8]. From here, an IDE user performs the majority of attribute-related operations.

As proposed in the design (Section 4.5.1), the Attribute View is listening for a selection change in the model editor. Once the change is registered, it displays the attributes of the selected model entity and their values (illustrated in Figure 29).

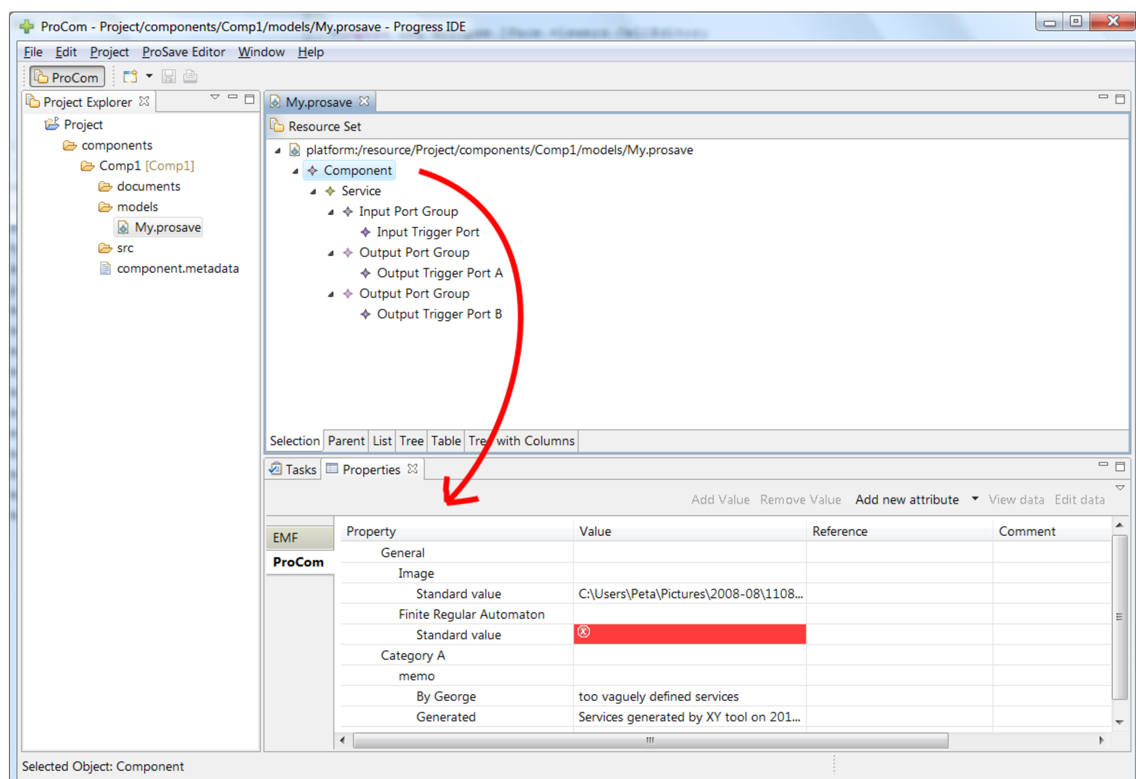


Figure 29: Integration of the Attribute View and the model editor

The Attribute View consists of two main parts (Figure 30): (1) the upper toolbar with actions

[8] Still, in the following text it will be regarded as the Attribute View.

that can be performed if an appropriate GUI item is selected, and (2) the viewer which displays categories, attributes, and their values in a UI control behaving both like a tree (collapsible items in the first column) and a table (several columns corresponding to the parts of the attribute value structure).

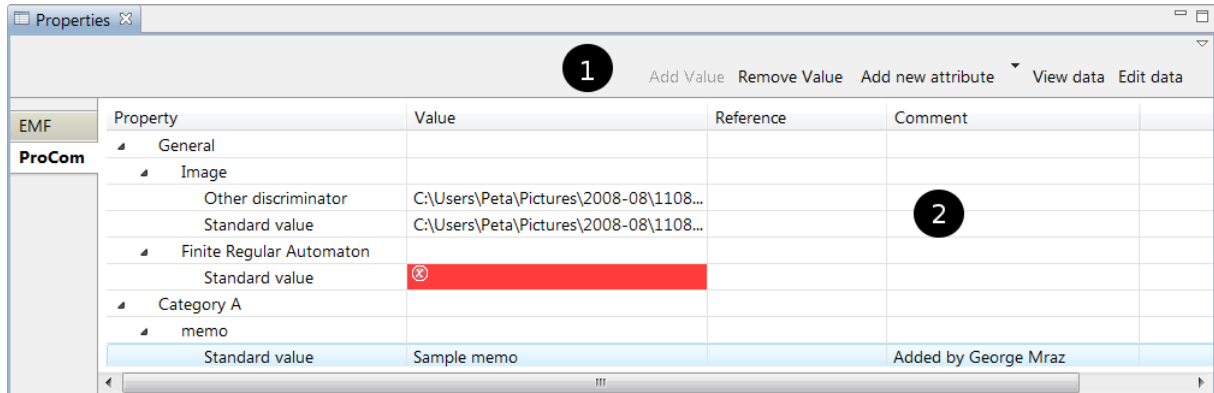


Figure 30: The Attribute View comprised from (1) the upper toolbar and (2) the viewer part.

Apart from actions manipulating items in the viewer (adding, removing an attribute value, and adding yet unassigned attribute to an entity), the other two actions invoking further GUI facilities of the framework are viewing and editing an attribute value. As explained in Section 4.5.2, there exist two types of viewers besides the default string renderer. Both of them are invoked in the same way — by the upper toolbar or by means of the viewer's context menu, and it depends on the attribute definition which one is eventually invoked. A managed viewer (Figure 31) is displayed in a window inside the IDE managed by the framework whereas a general viewer has no restrictions on how it should be rendered. It may launch an external application, use a system dialog or create an own window adjusted to the needs of viewing a particular attribute type.

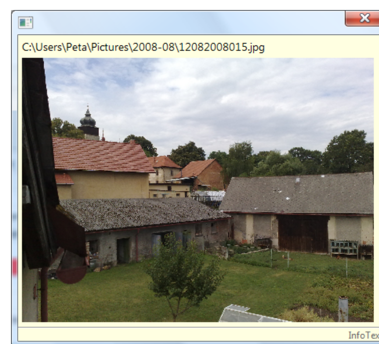
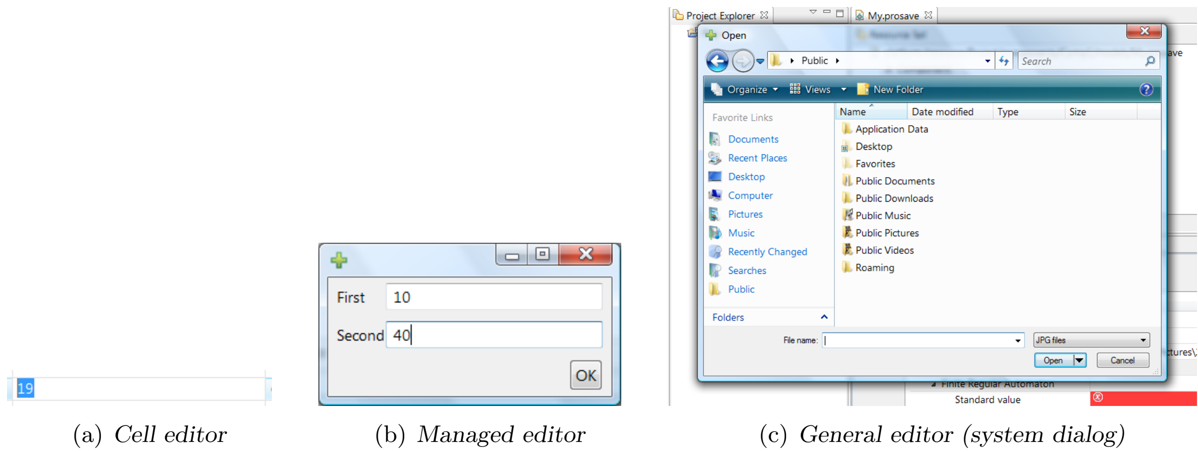


Figure 31: Managed attribute viewer (the bitmap image viewer)

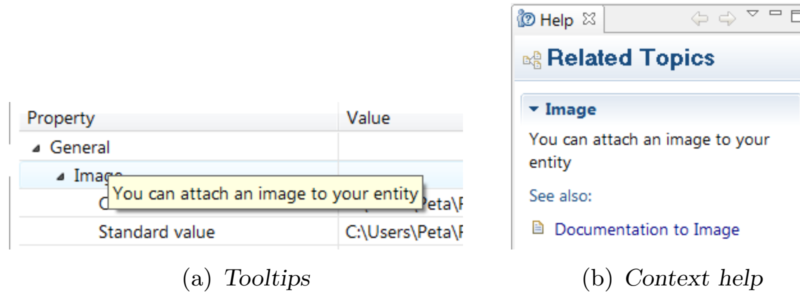
Editors are invoked either by clicking a respective cell in the viewer or, similarly to viewers, through the upper toolbar. Editors can be of three types (Section 4.5.2). Apart from general (Figure 32c) and managed editors (Figure 32b) with the behavior analogous to the corresponding viewers (except for the fact they are able to modify the attribute value), a cell editor (Figure 32a) is additionally supported. It can be considered as a counterpart of a string renderer since it is also fully integrated into the attribute viewer. However, due to its space limitations it is appropriate only for less complex types.



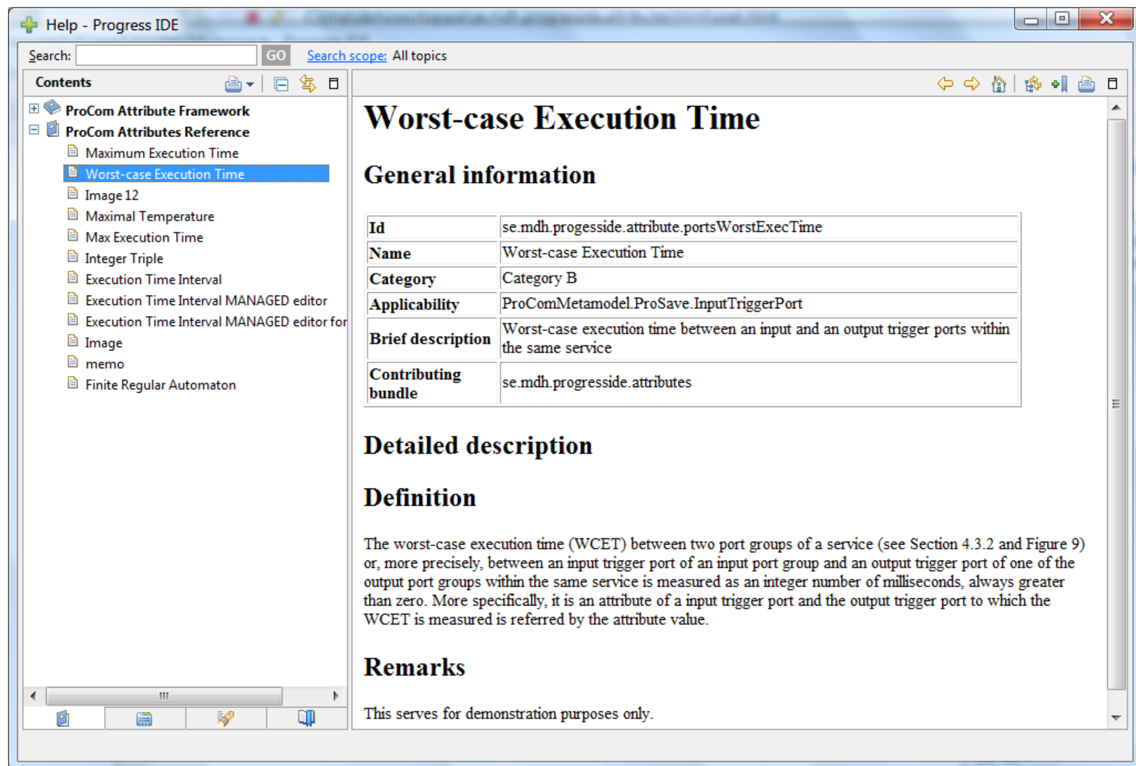
(a) *Cell editor* (b) *Managed editor* (c) *General editor (system dialog)*

Figure 32: Various types of attribute editors

Another important part of the framework's GUI is formed by user assistance features. They are implemented by several means: in the form of tooltips showing a brief description of an attribute in the Attribute View (Figure 33a), as an Eclipse context help (Figure 33b), and as a part of the standard Eclipse help system displaying the full attribute documentation (Figure 33c).



(a) *Tooltips* (b) *Context help*



(c) *Eclipse help system*

Figure 33: Assistance features overview

Chapter 6

Related work

This chapter contains a brief summary of other works dealing with attributes in component models.

Many component technologies which support *run-time components* perceive attributes as pieces of information that can be accessed by component instances at run-time, serving as configuration parameters. CORBA Component Model [15] uses attributes defined in component interfaces for this purpose: values of attributes control components behavior and can be set during deployment, component initialization or at run-time. Similarly, SOFA [7] uses *properties* that can be associated with frames (component interfaces) and architectures and are set during deployment. Fractal [3] provides attribute support for components that implement an extension of the `AttributeController` interface where attributes are defined by means of getters and setters (like JavaBean properties). However, regardless of the method of attributes definition (whether using special syntax, declaratively, or using normal interfaces), all of the afore-mentioned attributes realizations treat attributes as information read by component instances at run-time.

To the contrary, ProCom attributes are primarily design-time entities since the whole ProCom is a design-time component model, and their main purpose is to capture non-functional properties of model entities. The general-purpose component models usually have such properties hard-wired in the component model. For example, in SOFA an interface has an attribute of communication style which determines what communication paradigm should be used (method invocation, streaming, etc.). Furthermore, there exists a concept of communication features associated with an interface being even closer to the semantics of ProCom attributes. Each communication feature is a name-value pair specifying an extra-functional property serving as an input for connector generator. Compared to ProCom attributes, feature values are not structured since they are represented as strings, and they can be associated only with interfaces.

Naturally, component models specifically designed for domains where extra-functional requirements are of great concern exhibit better support for extra-functional properties. The Rubus component model [16] for resource constrained real-time systems features run-time profiles describing execution time and memory consumption of a component on different platforms. PECT (Prediction-Enabled Component Technology) [26] puts emphasis on predicting properties of assemblies of components based on the properties of their subcomponents. In [26], a property is vaguely defined as an n-tuple consisting of a name, a value, and 'arbitrary, perhaps property-specific information' (e.g. a confidence interval of a property value). Similarly to ProCom, properties can be associated with many elements, not only components as in the majority of other component models, by means of annotations. Unlike ProCom, PECT employs reasoning frameworks, strong and complex tools for reasoning and predicting properties of assemblies.

The closest approach to attributes is adopted by SaveCCM [18]. A SaveCCM attribute is a triple (`Name`, `Type, Value`, `Credibility`), a typed named value with possibly associated confidence measure, which can be only associated with components. Compared to ProCom attributes, SaveCCM attributes' type information and value representation are free-form strings.

Credibility of attribute values is a special case of value metadata. Another distinguishing feature of ProCom attributes consists in allowing multi-valued attributes.

Whereas there exist many component models capturing extra-functional properties of model elements in their specifications and proposals, only a few of them offer some usable tools for working with these properties. SaveIDE [21][20] is such a tool for development of component-based embedded systems by means of the SaveCCM component model. It provides an architectural editor for designing the architecture of a system and a behavioral editor for creating timed automata which model component behavior and are used by various analyses. However, support for working with attributes seems rather limited in the current version of SaveIDE [20]. The GUI of SaveIDE, realized by EMF and GMF[1]-based editors and property sheets, directly follows the structure of SaveCCM attributes without attempting to hide the underlying simple representation of types and values as strings. Consequently, values can be only rendered and edited in a simple text field and there is no choice of predefined types of values; instead, a user is required to fill in the type name. Thus, editing attributes is error-prone and utterly inconvenient for more complex types.

Similarly, graphical tools for working with SOFA (SOFA 2.0 IDE [24]) and Fractal (F4E, Fractal For Eclipse [13]) rather focus on modeling the architecture of a system by means of editing its architectural descriptions expressed in respective ADL[2]. Again, they use EMF and GMF facilities to manipulate entities from the underlying metamodels. As a result, working with the above-mentioned constructs closest to ProCom attributes is restricted to editing and viewing free-form strings, sharing all drawbacks of editing attributes in SaveIDE. On the other hand, the GUI support is in accordance with the secondary importance of these constructs in SOFA and Fractal.

What distinguishes the ProCom attribute framework is the concept of structured, multi-valued, typed, extensible attributes and notably complete coverage of all phases of working with attributes throughout the development of a system by providing respective interfaces (SPI, GUI, and API). Additionally, despite still being a prototype, the implementation of the framework is mature in comparison with existing tools and enables comfortable work with attributes in the Progress IDE.

[1] Graphical Modeling Framework provided by the Eclipse platform.

[2] Architecture description language.

Chapter 7

Conclusion

In this thesis an attribute framework for the ProCom component model has been proposed and realized as a prototype implementation integrated into the Progress IDE.

Based on the analysis of the development process envisioned by Progress, the requirements for the attributes of the ProCom entities have been identified. In the analysis driven by these requirements, many alternative proposals have been devised and their benefits and drawbacks have been evaluated. The notion of an attribute has been conceptually divided into more precisely defined entities of an attribute, an attribute description, and an attribute value. The inner structure of the attribute value has been refined to contain the data and metadata parts. Whereas the data part represents the attribute-specific information, a measure of the quality represented by the attribute, the metadata part consists of information describing the data and specifying closely its semantics. Several possible usages of metadata have been proposed, and finally the fixed set of metadata elements with well-defined semantics have been preferred over the other alternatives. The solution chosen in the analysis part specifies highly structured, multi-valued attributes designed so as to be extensible both by adding new attributes and adding new attribute types.

The investigation of metadata usage has raised questions related to the role of metadata as a characteristic distinguishing multiple values of the same attribute: Is there a single characteristic that could have this role? If so, what is its precise semantics, and its domain? What should be other metadata elements to effectively categorise attribute values? These questions remain to be answered by further research which has to carefully examine the development process and the role of attributes in it.

Furthermore, the thesis has identified the main roles of users working with the ProCom attributes throughout the development process and their requirements on the interfaces used for this interaction. These have been reflected during the design of the attribute framework, a set of features provided by the IDE, APIs, procedures and techniques enabling users to efficiently and comfortably work with the ProCom attributes. The main focus of the design of the attribute framework has been on an extensible, modular GUI, supporting reuse of its components, able to display and modify complex information possibly contained in the ProCom attributes. The design of the other two interfaces, the one for the programmatic access to attributes and the interface used for adding new attributes and attributes types have been elaborated to facilitate the work of the respective user roles.

The prototype application which implements the proposed designs has shown the flexibility and versatility of the devised mechanisms. It has been successfully integrated to the Progress IDE and tested on a set of attributes of various information complexity.

To summarize, the contribution of the thesis consists in analysis of various possibilities of realizing attributes in ProCom, elaborating the chosen design, and designing and implementing a usable attribute framework for working with the ProCom attributes in the Progress IDE.

The future work consists in answering the open questions, further enhancing the attribute metamodel and the framework facilities, and notably evaluating the viability of the proposed mechanisms by testing them in real-world settings.

From a short-term perspective, the open questions raised during the analysis of metadata usage (mentioned in the previous section) should be investigated and eventually answered. This could either mean that the proposed notion of a semantic-less discriminator in the current design would be reestablished and it would be given a precise semantics or it would be removed and replaced by a different concept. Also attributes related to multiple model entities should be supported properly, replacing the reference metadata element, which is just a proof of concept.

Additionally, it might be interesting to deal with other ideas that have not been explored in the attribute framework yet. For instance, there might be attribute values which depend on the context of the element they are attached to (e.g. a memory consumption of a component that is deployed to a particular platform). Consequently, there should be a possibility of filtering attribute values that are valid for the current context of a component. There arises the need for some specification of the valid context of the attribute, (possibly automated) management of attribute values depending on their context, etc. These possible features of the framework heavily depend on the precise usage of attributes during the development of a system.

For the next successful evolution of the attribute framework it is vital that new attributes and new attribute types together with the modules responsible for their visualization and modification are constantly added and used by the users of the PROGRESS IDE. This will help in proving suitability of the mechanisms currently implemented but, more importantly, it will help in determining the direction of further development of the ProCom attributes. Hopefully, with the next advancements in the PROGRESS research this condition will be met.

Chapter 8

Bibliography

[1] Wayne Beaton and Jim des Rivieres. Eclipse Platform Technical Overview. Technical report, The Eclipse Foundation, 2006.

[2] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, 2 edition, May 2008.

[3] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean B. Stefani. The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[4] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework: A Developer's Guide*. Addison Wesley Professional, 2003.

[5] Tomas Bures, Jan Carlson, Ivica Crnkovic, Séverine Sentilles, and Aneta Vulgarakis. ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report, Mälardalen University, June 2008.

[6] Tomas Bures, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. A Component Model Family for Vehicular Embedded Systems. In *The Third International Conference on Software Engineering Advances*. IEEE, October 2008.

[7] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. *Software Engineering Research, Management and Applications, ACIS International Conference on*, 0:40–48, 2006.

[8] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins (2nd Edition)*. Addison-Wesley Professional, 2006.

[9] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. Component-based Development Process and Component Lifecycle. *Journal of Computing and Information Technology*, 13(4):321–327, November 2005.

[10] Ivica Crnkovic and Magnus Larsson, editors. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[11] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer's Guide to Eclipse (2nd Edition)*. Addison-Wesley Professional, 2004.

[12] Eclipse Foundation. The Eclipse Modeling Framework (EMF) Overview, June 2005. Eclipse SDK Help.

[13] F4E (Fractal For Eclipse), version 1.2.4.200902091217. `http://fractal.objectweb.org/f4e/`.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[15] Object Management Group. CORBA Component Model 4.0 Specification. Specification Version 4.0, Object Management Group, April 2006.

[16] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. The Rubus Component Model for Resource Constrained Real-Time Systems. In *3rd IEEE International Symposium on Industrial Embedded Systems*, June 2008.

[17] Hans Hansson, Ivica Crnkovic, and Thomas Nolte. The world according to PROGRESS. Technical report, Mälardalen University, 2007.

[18] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The SAVE approach to component-based development of vehicular systems. *J. Syst. Softw.*, 80(5):655–667, 2007.

[19] P. Marwedel. *Embedded System Design.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[20] SaveIDE, snapshot 20081114.
`http://forge.ow2.org/project/showfiles.php?group_id=54`.

[21] Séverine Sentilles, Anders Pettersson, Dag Nyström, Thomas Nolte, Paul Pettersson, and Ivica Crnkovic. Save-IDE - A Tool for Design, Analysis and Implementation of Component-Based Embedded Systems. In *Proceedings of the Research Demo Track of the 31st International Conference on Software Engineering (ICSE)*, May 2009.

[22] Séverine Sentilles, Aneta Vulgarakis, Tomas Bures, Jan Carlson, and Ivica Crnkovic. A Component Model for Control-Intensive Distributed Embedded Systems. In Michel R.V. Chaudron and Clemens Szyperski, editors, *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*, pages 310–317. Springer Berlin, October 2008.

[23] Mary Shaw. Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does. In *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996.

[24] SOFA 2 IDE, version 0.5.2.
`http://forge.ow2.org/project/showfiles.php?group_id=54`.

[25] Jaroslav Tulach. *Practical API Design: Confessions of a Java Framework Architect.* Apress, Berkely, CA, USA, 2008.

[26] Kurt C. Wallnau. Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC). Technical Report, Carnegie Mellon University, 2003.

Appendix A

Contents of the enclosed CD-ROM

The enclosed CD-ROM is organized as follows:

`readme.txt`
: A description of the contents of the CD-ROM and instructions for using it. Please read this file first before using the CD-ROM.

`thesis.pdf`
: This thesis in the PDF format.

`progresside/`
: PROGRESS IDE application.

`dev/`
: The complete development environment of the PROGRESS IDE.

`doc/`
: Programming documentation in the HTML format generated by JavaDoc from the source files.

`metamodel/`
: Attribute and ProCom metamodels designed in IBM Rational® Software Architect™.