Integration of Extra-Functional Properties in Component Models*

Séverine Sentilles, Petr Štěpán, Jan Carlson, and Ivica Crnković

Mälardalen Research and Technology Centre, Mälardalen University, Västerås, Sweden
{severine.sentilles, jan.carlson, ivica.crnkovic}@mdh.se
psn08003@student.mdh.se

Abstract. Management of extra-functional properties in component models is one of the main challenges in the component-based software engineering community. Still, the starting point in their management, namely their specification in a context of component models is not addressed in a systematic way. Extra-functional properties can be expressed as attributes (or combinations of them) of components, or of a system, but also as attributes of other elements, such as interfaces and connectors. Attributes can be defined as estimations, or can be measured, or modelled; this means that an attribute can be expressed through multiple values valid under different conditions. This paper addresses how this diversity in attribute specifications and their relations to component model can be expressed, by proposing a model for attribute specifications and their integrations in component models. A format for attribute specification is proposed, discussed and analyzed, and the approach is exemplified through its integration both in the ProCom component model and its integrated development environment.

1 Introduction

One of the core challenges still remaining in component-based software engineering (CBSE) is the management of extra-functional properties, often expressed in terms of attributes of components or of systems as a whole. In CBSE, one desired feature is the integration of components in an automatic and efficient way. The integration process is achieved by "wiring" components through their interfaces. The second aspect of the integration is the composition of extra-functional properties and this part is significantly more complex. The problem already appears in the specifications of attributes. While component models precisely define interfaces as a means of functional specification, specifications of attributes in relation to component specification is either not defined, or unclear. Is an attribute a property of a component or the result of interaction between components, or maybe the result of performing a function that is part of the component interface, or the result of combining a component and its environment? So far these questions have not been addressed in a systematic way.

This paper addresses the question of attribute specification in component models. The specification of attributes has several aspects that we discuss and demonstrate on a component model.

* This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS, and by the EU FP7 Project Q-ImPrESS.

First, we address the question of the form of attribute specifications. Our starting points are related to Shaw's specification which identifies the specification of attributes as a triple containing attribute name, value and credibility information [1]. We refine this definition in extension of values and credibility.

The second aspect of attribute specification that we address is related to the component and system lifecycle. During the lifecycle of a component an attribute changes with respect to how the value is obtained and the accuracy (credibility) of its value. In early phases of the component lifecycle a component is being modelled and then the attribute value can be an estimation or even a requirement. The accuracy of the estimation during the development process can be changed, as a result of an increasing amount of information or a change in the way the value is obtained. In the run-time phase (or even in the development phase in some cases), the attribute value can be measured.

The third aspect of the attribute specifications concerns the variations of the values — not only as a result of different ways of obtaining the value, but also different values depending on the external context. Some attributes are directly related to the system context — for example, the execution time of a component does not only depend on the component behaviour and input parameters, but also on the platform characteristics. For such cases it is obvious that we need to be able to specify these different values and the conditions under which the attribute value is valid.

There are also other aspects of integration of component models and their attributes. By nature the attributes are parts of (i.e., they characterize) components, but they also can be related to a particular element of a component or a system. For example, an attribute can be annotated to a component directly, or to a port in the interface of a component, or to a connector. In general, a component model that supports the management of attributes should have the possibility to relate attributes to different architectural elements of the component model.

The aim of this paper is to analyze the different aspects of attribute specifications to formalize their form and their integration with component models. A formal specification of an attribute format makes it easier to manage component and system properties. It also catalyzes the process of integrating extra-functional properties into component models.

Since attributes are very different, the concrete results can be shown on particular classes of attributes integrated with particular component models. To illustrate the attribute specifications in a component model, we use ProCom [2, 3], and annotations of attributes as an immanent part of the model. We also provide implementation examples.

The rest of the paper is organized as follows. Section 2 defines the attribute specifications. Section 3 discuses the attribute specifications of composite components in relation to the attributes of composable components. Since an attribute can include different values, i.e., different versions of an attribute can exist, in a system analysis or verification process it is important to select a particular version of an attribute. The selection principles and a possible support is discussed in Section 4. The principles of attribute specifications are exemplified in the ProCom component model, and a prototype tool that manages attributes is demonstrated in Section 5. Section 6 surveys related work, followed by a short discussion in Section 7, before the paper concludes with a summary and future work.

2 Annotation of Attributes in Component Models

The purpose of attributes is to provide additional information about the components, complementing the structural information that is provided by the component model.

This additional information is intended to give a better insight in the behaviour and capability of the component in terms of reliability, safety, security, maintainability, accuracy, compliance to a standard, resource consumption, and timing capabilities, among many others. In that sense, attributes bridge the gap between the knowledge of what a component does and its actually capabilities.

2.1 Attributes in a Component Model

As mentioned in [4], the additional information provided by attributes does not necessarily concern the component as a whole, but in fact often points more precisely to some parts of a component such as an interface or an operation of an interface. In our view, this relation should not be limited to components, interfaces and operations, but be extended so that attributes can be associated with other elements of a component model, including for example ports, connectors or more notably component instances. For instance, having an extra-functional property on connectors to capture communication latency, makes it possible to reason about the response time of complex operations that involve communication between components.

Following this standpoint, we define as *attributable* an element of a component model (*component*, *interface*, *component instance*, *connector*, etc.) to which extra-functional properties (*attributes*) can be attached. By this means, all attributable entities are treated in similar way with regards to the definition and usage of attributes. Fig. 1 depicts these relations.

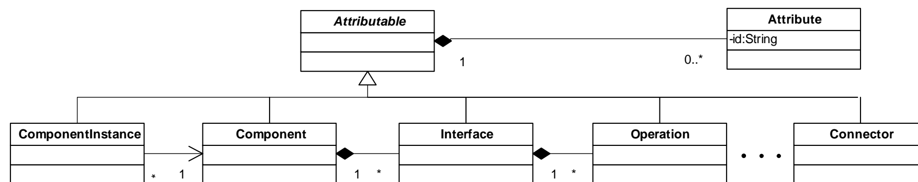


Fig. 1. The relation between attributes and the elements of a component model.

2.2 Attribute Definition

The exhaustive list of possible attributes to consider is endless and, as stated in [5], there is no a priori, logical or conceptual method to determine which properties exist in a system or in components. Furthermore, a single property can have a multitude of possible representations. This problem inheres in one of the fundamental characteristics of extra-functional properties and properties in general: they are issued by humans. Therefore, different users will consider different types of information important for the

development of the software system, and for the same property they might associate a different meaning and representation.

Consequently, the definition of a suitable format specification for extra-functional properties able to deal with the great variety of properties possibly of interest remains a challenge. This definition should be generic and flexible enough to handle the heterogeneity of properties while being extensible to support the emergence of new ones. This means that the specification format must be able to cope with different formats and different levels of formalism.

An informal way to specify these properties is to use annotations. However, it gives too much freedom concerning the definition and this brings problems to manage extra-functional properties at a large scale or in automated processes such as composition or analysis.

In order to move towards a precise formalisation of extra-functional properties, which allows an unambiguous understanding and a precise semantics both with respect to meaning and valid specification format of the value, we define the concept of *Attribute* as:

$$Attribute = \langle TypeIdentifier,\ Value^{+} \rangle$$
$$Value = \langle Data,\ Metadata,\ ValidityCondition^{*} \rangle$$

where:

- *TypeIdentifier* defines the extra-functional property (i.e. the identifier property in Fig. 1);
- *Data* contains the concrete value for the property;
- *Metadata* provides complementary information on data and allows to distinguish between them; and
- *ValidityConditions* describe the conditions under which the value is valid.

The remaining of this section details these concepts, based on diagrams issued from the meta-model of our attribute framework (the full meta-model is given in Appendix 8). However, an important aspect of this definition, which is worth noting already at this point, is the possibility for an attribute to have a several values. This is further explained in Section 2.5.

2.3 Attribute Type

Similarly to the concept of "class" in object oriented programming, an *attribute type* designates a class of attributes. In this respect, an attribute is then comparable to a class instance, and must comply with the specific structure imposed by the attribute type. An attribute type specifies thus an *identifier* which is a condensed significative name describing the principal characteristics of the attributes (e.g., "Worst Case Execution Time", "Static Memory Usage", etc.), a list of *attributable* elements to which the property can be attached, and a specification of the *data format* that the attribute instances must conform to. As illustrated in Fig. 1, the identifier of the attribute type is shared by all the attributes of the same attribute type, and an attribute belongs to a single attribute type only.

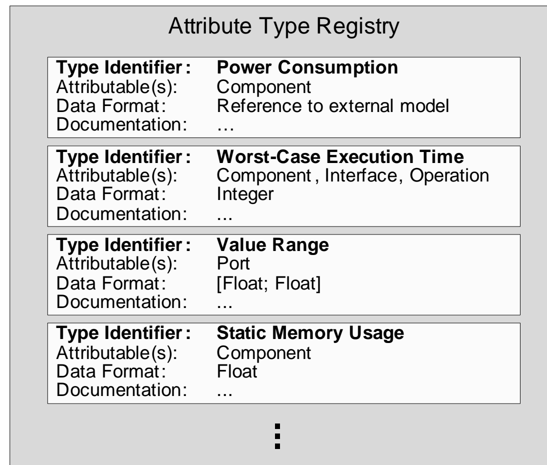


Fig. 2. Attribute type registry.

Consequently, the uniqueness of the attribute types must be ensured so that it is not possible to have two attributes with the same identifier but different value formats. This requires techniques outside the definition of the attribute concept itself. A simple technique is to keep a *registry* of attribute types, where all the declaration of attribute types are stored to ensure their uniqueness. Fig. 2 illustrates an attribute type registry containing several attribute types.

Although this way of specifying attributes types (or attributes, in a broader sense) provides the great advantages of being open and extensible so that it can fit the multitude of extra-functional properties which need to be defined, it still requires users to have an intuitive and common understanding of what the meaning and intended usage of the attributes were when they were created. Therefore it is important to provide proper attribute type *documentation*. This documentation is stored in the attribute type registry and consists of an informal text written in natural language. Nevertheless, it must supply enough information to primarily clarify the meaning of the attribute type as well as its intended usage.

It is reasonable to assume that hundreds of attribute types or more will be introduced. Several classification schemes (e.g., [6] and [7]) have been proposed which can be used as basis to identify groups of attribute types such as “resource usage”, “reliability”, “timing”, etc. These categories could allow navigation across attributes more easily and possibly hide the whole set of attribute types that are uninteresting for a particular project. A remaining challenge is in this case to determine appropriate categories, as the proposed classifications are distinct and often non-orthogonal as mentioned in [5]. However, this is not within the scope of this paper.

2.4 Attribute Data

To elicit information on the element of the component model they are associated with, the part of attributes concerned with expressing data must be represented in an unam-

biguous and well-tailored format. This implies that in addition to supporting primitive types such as integers, floats, etc., and structured types such as arrays, complex types must also be covered. These complex types include representation of value distributions, various external models, images, etc.

For this, we define a generic data structure, called *data*, which is specialized into a number of simple data types and a reference to any complex object, as illustrated in Fig. 3. This structure can be extended to build more complex data structure such as records or tuples.

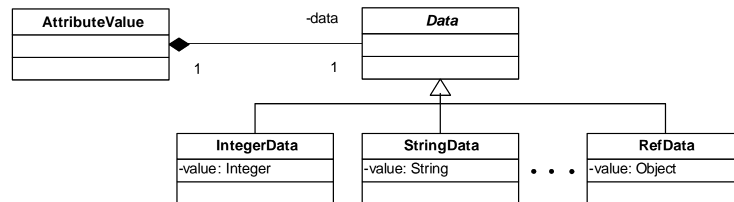


Fig. 3. Attribute data.

2.5 Multiple Attribute Values

Attributes emerge during the software development process as additional information needs to be easily available either to guide the development, to make decisions on the next step to follow, to provide appropriate (early) analysis and tests of the components, or to give feedbacks on the current status. This need for information starts already in early phases of the development, in which extra-functional properties are considered as constraints to be met and expected to be satisfied later on, thus becoming an intrinsic part of the component or system description.

This implies that through the development process, (i) the meaning of an attribute typically changes from a required property to a provided/exhibited property, and (ii) its value changes too as the knowledge and the amount of information about the system increases. Thus the actual data as well as the appropriate metadata needs to be successively refined to be replaced by the latest and most accurate value. For example, an attribute, estimated in a design phase, is replaced with a new value coming from a measurement after the implementation phase is completed, or with more information available the analysis become more efficient and reliable and therefore the confidence in the property, expressed by the accuracy metadata, increases.

However, the gradual refinement of an attribute towards its most accurate value is not always the expected way to deal with extra-functional properties. Often, values which are equally valid in the current development phase, need to exist simultaneously. In other words, this means that the latest value must not replace the previous one. This requires an ability for an attribute to have multiple values to cope with information coming from various context of utilization, to keep different values obtained through different methods, to keep the required value and a provided value for verifying the

conformity to the initial requirement, or to compare a range of possible values to make a decision. This ability of an attribute to have multiple values is depicted in Fig. 4.

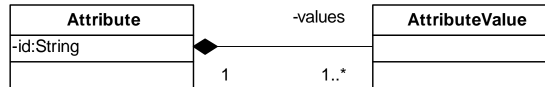


Fig. 4. Multiple attribute values.

2.6 Attribute Value Metadata

Introducing the possibility to have multiple values for attributes also requires the ability to distinguish between them. Furthermore, it is important to document the way an attribute value has been obtained to ensure that information about a component (or another element of a component model) is correct and up-to-date. These two functions are provided by the *attribute value metadata*, or simply *metadata*, which role is to capture the context in which the corresponding attribute value has been obtained: when, how and possibly by whom. However, the question of determining the complete list of elements that metadata should cover remains.

We define a partial list of metadata that we consider indispensable to provide a basic support for the concepts around the attribute definition (see Fig. 5). The list consists of the version of the current attribute value, the timestamp indicating when the attribute value was created or updated, the source of the value ("requirement", "estimation", "measurement", "formal analysis with the tool X", "simulation", "generated from model", "generated from implementation", etc.). Other metadata are optional; for example the accuracy of the value or some informal comments about the attribute value.

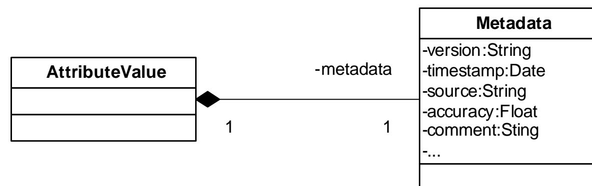


Fig. 5. Attribute value metadata.

2.7 Validity Conditions of Attribute Values

Reusability is a desired feature of component-based software engineering, which implies that a component is assumed to be (re-)useable in many different contexts. As an intrinsic part of components, revealing what the component is capable of, attributes are

intended to be reusable too. This means that the validity of their information must still be accurate in the new context in which the component is reused. Hence, to keep consistent all the information concerning the component, both its expected behaviour and capabilities, and the actual ones, it is necessary to specify in what type of contexts an attribute value is valid, i.e., fully or partially reusable.

We refer to these specifications of context restrictions as *validity conditions*. The validity conditions explicitly describe the particular contexts in which an attribute value can be trusted. Different types of contexts exist and, as with attribute types, an attempt to identify them all is bound to fail. They include, at least, constraints on the underlying platform, specification of usage profile, and dependencies towards other attributes, as illustrated in Fig. 6.

With the intentions of developing an automated process to select only valid values for the current context, the validity conditions must be defined in a strict manner and it is important that they are publicly exposed. However, strictly ensuring the respect of all the validity conditions is a too restrictive approach since in this case, only the attribute values for which the validity conditions are fully satisfied would be reusable. For instance, a component might be reused even though some of its attribute values are not trustworthy for the current design. This reuse might require a manual intervention to lower the confidence in the provided values. We envision that, as a conscious decision, some attribute values could be reused regardless of their validity conditions not being satisfied, but it would typically affect the values. For example, the value might be reused with a lower accuracy, or with the data modified to add some safety margins.

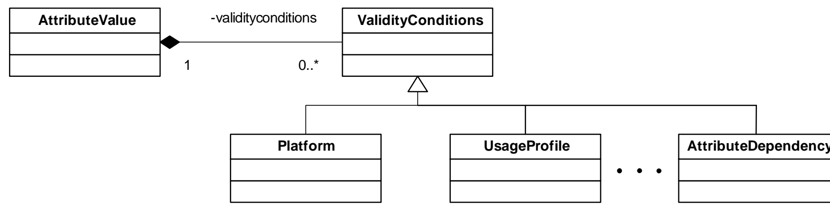


Fig. 6. Validity conditions of attribute values.

3 Attribute Composition

So far, the attributes has been in focus, and the attributable elements have simply been viewed as black-box units of design or implementation, to which attributes can be attached. However, the existence of hierarchical component models that also include composite components — components built out of other components — influences the ways in which the values of attributes can be established.

Ideally, all attributes of a composite component should be directly derivable from the attributes of its sub-components. While this is easily achievable for some attribute types, e.g., static memory usage, others depend on a combination of many attributes of the sub-components, or on software architecture details [5].

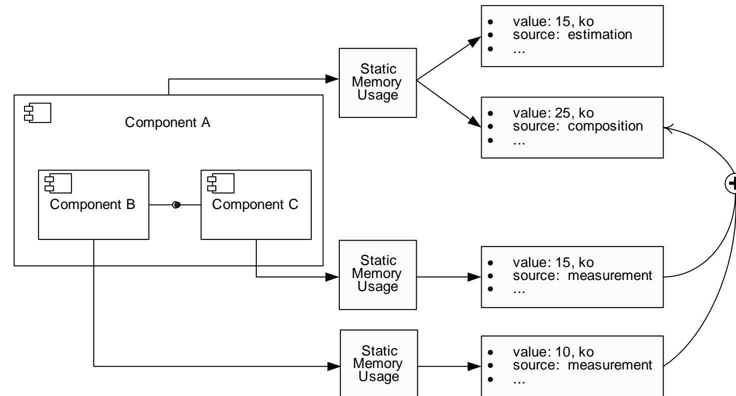


Fig. 7. A composite component with co-existing explicit and derived attribute values.

Even for composable attributes, we argue that it is beneficial to allow them to also be stated explicitly for the composite component as such. In particular, this allows analysis of the system also at an early stage of the development when the internals of a composite component under construction are not fully known, or not fully analyzed with respect to all attributes required to derive the attributes of the composite component.

The ability of the proposed attribute framework to store multiple values for a single attribute permits explicitly assigned information to co-exist with information generated by composition. To distinguish between them, the metadata field *source* can be given the value *composition* to indicate that the value was derived from the sub-components.

Specification of attributes of a composite is illustrated in Fig. 7. The composite component has been explicitly given an estimated value for the attribute representing static memory usage, and another value is provided by composition, which for this attribute simply means a summation over the sub-components.

Attribute composition can be viewed as the responsibility of the development process, i.e., it should specify when and how attribute values should be derived for composite components, possibly supported by automated functions in the development tools. An interesting alternative, in particular for easily composable attributes such as static memory usage, is to include the specification of a composition operator in the attribute type registry.

4 Attribute Configuration and Selection

From the previous sections we realize that an attribute can have many values. The question is which value of an attribute is of interest for a particular analysis, and what is the criteria to select it? The second question, related to the consistency of definition when using several attributes, reads: Which values of different attributes belong together?

This problem is addressed in version- and configuration management, and we apply the principles from Software Configuration Management (SCM). SCM distinguishes two types of versioning: (i) *versions* (also called revisions) that identify evolution of an

item in time. Usually the latest version of an item is selected by default, but also an old version can be selected, for example using a time stamp (select the latest version created before a specific time); and (ii) *variants* which allow existence of different versions of the same item at the same time. The versions and variants can be selected according to certain selection principles, such as: *state* (select the latest version with the specified state), *version name*, also called label or tag (select a version designed by a particular name). The latter is explicit since version names are unique, while states are not.

We adopt these principles in management of attributes. Since an attribute can have many values, each value is treated as an attribute version. A developer has two possibilities of managing attribute versions.

Attribute navigation The possibility to navigate through different versions of an attribute (i.e., through different values), and update the selected value (changing data, or metadata information, or modifying the validity conditions).

Configuration Values are selected, for one or several attributes, according to a given selection principle (e.g., based on version name or timestamp).

The *configuration filter* is important as it can be applied to the entire system, or to a set of components, and then all architectural elements expose particular versions of the attributes that match the filter. This is important when some system properties are analyzed using consistent versions of several attributes (for example in an analysis of a response time of a scenario performed on a particular platform).

The configuration filter is defined as a combination of attribute metadata and validity conditions, and the use of the following keywords:

Latest The latest version.
Timestamp The latest version created before the specified date.
Versionname A particular version designated by a name.

Metadata and validity conditions are equivalent from the selection point of view. In the selection process the filter defines constraints over metadata or validity conditions in the same way. The difference is however in understanding the filtering mechanism and in helping the developer in recording possible problems if the validity conditions that are filtered are contradictory (for example if the developer specifies to use attribute values valid for "platform X" and "platform Y").

The configuration filter is defined as a sequence of matching conditions combined with AND or OR operators. The conditions are tested in order, and if a condition is not fulfilled the next one is examined. The configuration filter is specified in the following format:

$Condition_1$ [AND $Condition_2 \ldots$] OR
$Condition_3$ [AND $Condition_4 \ldots$] OR
$\vdots$

The conditions within a line are combined by AND operator, while lines are combined with the OR operator. A concrete example of the configuration filter is the following:

(Platform: X) AND (Source: Measurement) OR
(Release 2.0) OR
Latest

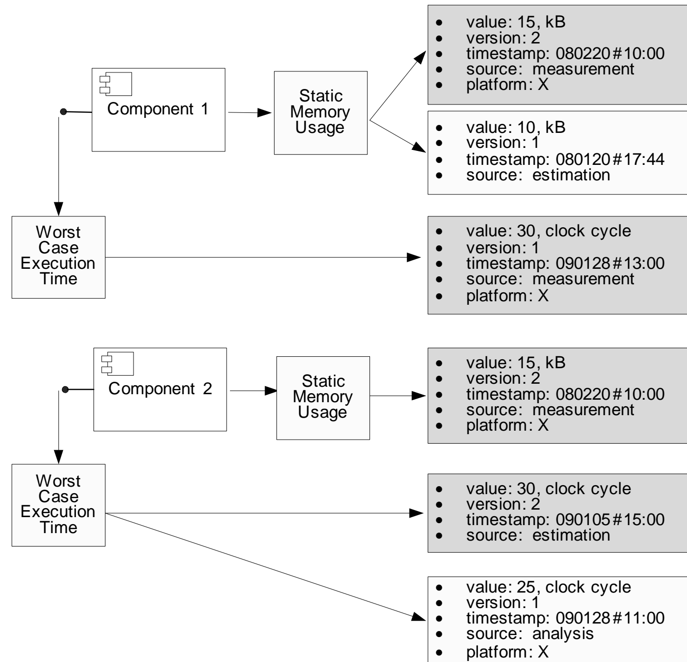


Fig. 8. Attribute value selection.

In this example the configuration filter will select first all values with validity conditions matching “Platform: X” and with “Source: Measurement” in the metadata. If such values exist, the latest one is selected; if not, the filter will select the latest version labeled with “Release 2.0”. If no such version was found, simply the latest version of the attribute will be selected. The selected attributes values are shown as gray boxes in Fig. 8.

5 A Prototype for ProCom and the PROGRESS IDE

This section concretizes and exemplifies the proposed attribute framework in the context of *ProCom*, a component model for distributed embedded systems [2, 3]. The characteristics of this domain make component-based development particularly challenging. For example, the tight coupling between hardware and platform, and high demands on resource efficiency, are to some extent conflicting with the notion of general-purpose reusable components.

ProCom applies the component-based approach also in early phases of development, when components are not necessarily fully implemented. Already at this point, however, it is beneficial that the components are treated as reusable entities to which properties, models and analysis results can be associated. Safety and real-time demands are addressed by a variety of analysis techniques, in early stages based on models and estimates, and later based on measurements, source code and structural information.

Table 1. Examples of attributes in ProCom.

Identifier	Attributable(s)	Data format	Documentation (short)
Static memory	Component, Subsystem	Int	The amount of memory (in kB) statically allocated by the component or subsystem.
WCET	Service	Int	The maximum number of clock cycles the service can consume before terminating.
Value range	Port	[Int;Int]	Upper and lower bounds on the values appearing on the port.
Resource model	Subsystem	External file	A REMES model specifying resource consumption.

Efficiency is achieved by a deployment process in which the component-based system design is transformed into executables that require only a lightweight component framework at runtime.

This extensive analysis support throughout the design and deployment process requires a large amount of information to be associated with various entities at different stages of the development. Information that is of interest to more than one type of analysis, or which should be reused together with the entity, is captured by attributes. Concretely, ProCom is based around two main structural entities — components and subsystems — both of which are *attributable* (as defined in Section 2.1). The attributable elements also include component services, message ports, and communication channels, among others.

The initial set of attribute types is influenced by the envisioned analysis of timing and resource consumption, and includes information about execution times, static and dynamic memory usage, and complex behavioral models handled by external model checking tools. Table 1 lists some of the attribute types used in ProCom.

To ease the development in ProCom, an integrated development environment called PROGRESS IDE is being developed. It is a stand-alone application built on top of the Eclipse Rich Client Platform, and includes a component repository, architectural editors to independently design components and systems, a C development environment, and editors to specify behaviour and resource utilization.

A variant of the proposed attribute framework is included in the PROGRESS IDE, in the form of two plugins: one for the core concepts that are required e.g., by analysis tools interested in, or producing, attribute values; and one for the graphical user interface through which the developer can view and edit attributes. In its current version, the prototype does not support validity conditions, nor is the selection mechanism fully implemented. For a detailed presentation of the attribute framework prototype, see [8].

The graphical part of the framework consists of an additional tab in the property view, where the attributes of the currently selected entity are presented. In Fig. 9, a component is selected in the top editor, and its attributes (*Resource model* and *WCET*) are shown in the property view below. In the depicted scenario, each attribute has two values, distinguished by the metadata timestamp.

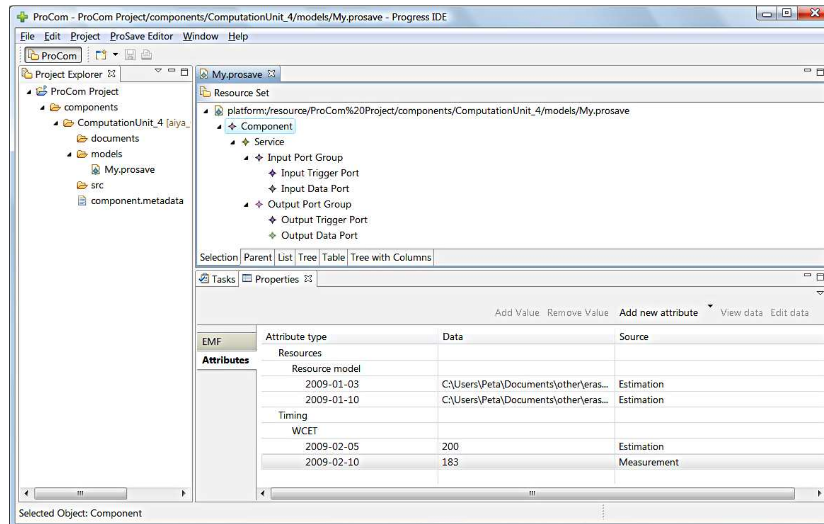


Fig. 9. The Attribute framework integrated in the PROGRESS IDE.

The attribute type registry is realized by an extension point that allows other plugins to contribute new attribute types. In addition to the information specified in Section 2.3 (e.g., data format and documentation), the extension can also define how the new attribute type is handled by the graphical interface, by defining classes for viewing, editing and validating its data.

6 Related Work

Although a lot of work has been done studying extra-functional properties in general, few component models actually integrate support for specifying and managing extra-functional properties. When this support exists, it concerns specific types of extra-functional properties such as temporal properties or resource-related properties and is intended for reasoning and predictability purposes.

The relation between extra-functional properties and functional specifications of component models was first explicitly addressed in the Prediction-Enabled Component Technology project (PECT) [9]. In PECT, extra-functional properties are handled through "analytical interfaces" conjointly with analytical models to both describes what are the properties that a component must have and the theory that should support the property analysis.

In Robocop [10] the management of extra-functionality is done through the creation of models: a resource model describes the resource consumption of components in terms of mathematical cost functions and a behavioural model specifies the sequence in which their operations must be invoked. Additional models can be created.

The support for extra-functional property proposed by Koala [11] handles only static memory usage of components. The information about this property is provided through

an additional analytic interface which must be created and filled for every components existing in the design. It is not possible to add information about this property to already existing components. Moreover, through diversity spreadsheets, Koala proposes a mechanism outside the analytical interface to deal with dependencies between attributes.

Contrary to our approach, which allows various elements of a component model to have attributes, these components models manage extra-functional properties on component- or system-scale only.

The closest approaches to our concept of attributes are those which define extra-functional properties as a series of name-value pairs; for example Palladio [12] and SaveCCM [13]. Palladio uses annotations and contracts to specify extra-functional properties concerned with performance prediction of the system under design. SaveCCM follows the concept of credentials proposed by Shaw [1], where extra-functional properties are represented as triples $\langle Attribute,\ Value,\ Credibility\rangle$ where Attribute describes the component property, Value the corresponding data, and Credibility specifies the source of the value. Similarly to our registry of attribute types, these credentials should be used conjointly with techniques to manage the creation of new credentials.

Other approaches not related to a particular component model have also been proposed. Zschaler [14] proposes a formal specification for extra-functional properties with the aim to investigate architectural elements and low-level mechanisms such as tasks and scheduling policies that influence particular extra-functional properties. In this specification, extra-functional properties are split between intrinsic properties which are inherited from the implementation and are fixed, and extrinsic properties which are properties which depend on the context. In [15], a specification language for specifying the quality of service of component-based systems is proposed. The language supports specification of derived attributes for composites, and links between attribute specification and measurement.

Comparing with what exists for UML, our approach relates to the MARTE subprofile for non-functional properties [16] which extends UML with various constructs to annotate selected UML elements. Similarly, extra-functional properties are defined in a "library" as types with qualifiers and used in the models. Attribute values can be specified through a Value Specification Language, which also defines value dependencies between attributes through symbolic variables and complex expressions. Dependencies involving more than one element are expressed through constraints. MARTE also acknowledges the need for co-exisisting values from different sources, but the associated information is not as rich as our metadata concept, and the selection mechanism is not elaborated. However, MARTE does not support component-based development and design space exploration, nor provide means to manage refinement of non-functional properties. Our work could gain in integrating the generic data type system and also in integrating the value specification language for supporting the specification of the attribute values, which are now left to the creator of the attributes.

Our approach also relates to work on service level agreements (SLA) in service-oriented systems [17], although our motivation for capturing non-functional properties comes mainly from the need to perform analysis, rather than as the basis for negotiation of quality of service between a service provider and consumer. In the context of SLA, non-functional properties are used in the formal specification of services, defining, e.g.,

the availability of a service or the maximum response time, while we associate non-functional properties with architectural entities to facilitate predictable reuse.

In summary, our approach differs from previous in focusing on reuse of attribute values, proposing an attribute concept allowing to have multiple values and a mechanism to select among them, and encompassing context dependencies that must be satisfied for a value to be valid in a new context.

7 Discussion

Our purpose with this attribute model is to provide a structure for managing extra-functional properties closely interconnected to the component model elements with the long-term vision of supporting a seamless integration and assessment of extra-functional properties in an automatized and efficient way. This structure is intended to be used throughout a component-based development process from early modelling to deployment steps (for an overview of this development process, see [18]). In particular, it should be possible for reused components with extensive, detailed information to co-exist with components in an early stage of development, and for analysis to treat the two transparently.

With regards to other models, our proposition is characterized by the support for multiple attribute values. Although for some simple attributes such as *number of lines of code*, one and only one value is correct at a given point in time, for other attributes the value vary according to the methods or techniques used to obtain it, and it is not always possible to say that one value is more correct than another. An example of such attributes is the *worst-case execution time* for which different analysis techniques give different values, all of which can be considered equally true in the characterization of the attribute. For instance, a "safe" static analysis technique gives a higher number than a probabilistic method but the confidence in the fact that the value cannot be exceeded is higher. For components in an early stage of development, even a simple attribute such as *lines of code* could be estimated by several approaches, and thus have multiple values that are equally correct at the time.

One possible way to manage multiple property sources would be to create a separate attribute type for each variant of the property, treating e.g., *estimated worst case execution time* and *measured worst case execution time* as two separate attribute types. However, viewing them as a single attribute with multiple values facilitates analysis that use attributes as input. For example, analysis that derives the response time of an operation can be based on the execution time attribute without having to deal with the different possible sources of this information. Thus, the same response time analysis can be performed based on early execution time estimates, safe values from static code analysis, or measurements. Multiple values also significantly reduces the amount of properties types which can be defined (in the case in which the methods provide results for the same property) while preserving the source of information through the metadata and the usage context through the ValidityConditions.

Another noticeable characteristic of our model is the specification of validity conditions for individual attribute values. Many attributes depend on factors external to the entity, such as underlying middleware or hardware. When a component is reused in the same, or similar, context, the attribute value can also be reused without restrictions. If,

on the other hand, the component is reused in a context that does not match the validity condition, the value will not be used (e.g., in analysis) unless proceeded by a conscious decision by the developer. For example, the value can be used with lower confidence as an early estimate, or fully reused if the developer believe that it still applies in the new context.

The approach presented in the papers aims for increasing analysability and predictability of component-based systems. It however introduces a complexity in the design process. By having many attribute types and different versions of attributes, there is a need for a selection of a "proper" attribute version. There is also a need for ensuring consistency between attributes of different types. We propose that this is handled outside the attributable entity, by a configuration management-like mechanism in the development environment. This allows the developer to specify which attribute version, from a number of currently "correct" ones, that should be used in the analysis performed at this point. The attribute version can be determined by different parameters, such as specification of the context (identified by ValidityConditions).

The defined infrastructure for attributes facilitates a complete analysis that includes analysis of different properties and relations between them, including a trade-off analysis. For example, by simple changes of the configuration filters, the process of the analysis and presentation of the results for all attributes is simpler, and consistent.

8 Conclusion

Providing a systematic way of attribute specifications and their integration into a component model is important for an efficient development process; it enables building tools for attribute management, such as specification, analysis, verification, and first of all efficient management of different attributes, or the same attributes attached to different components. It also facilitates integration of different analysis tools. This paper proposes a model for attribute specification which is expandable in the sense of allowing specification of new attribute types or new formats of attribute presentations. The model distinguishes attribute types (defined by a name and a data type), attribute values which include metadata and specification of the conditions under which the attribute value is valid. The main challenge in the attribute specification formalization is to provide a flexible mechanism to cover a large variety of attribute types and their values, and keeping them manageable. This is the reason why the model is extensible.

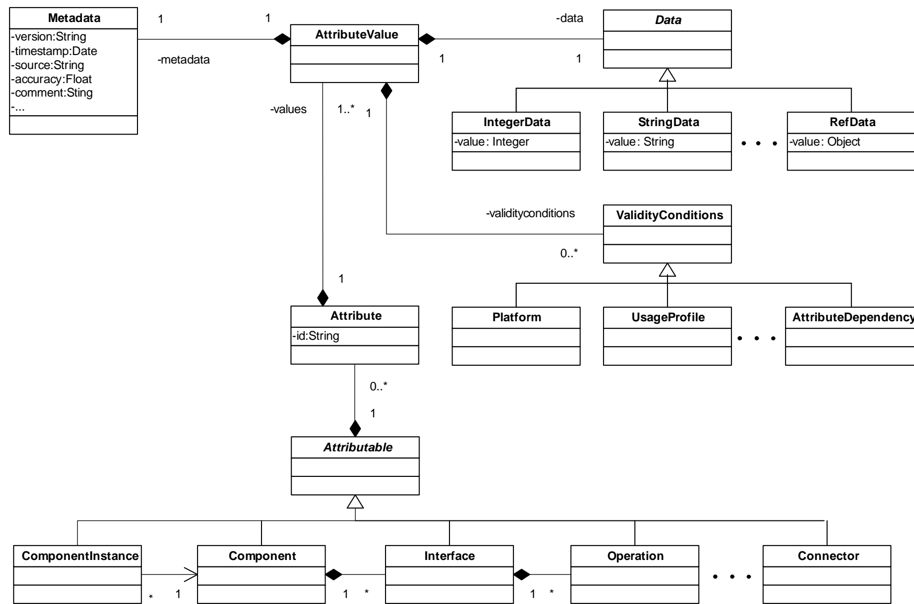
The proposed model has been integrated into ProCom, a component model aimed for development of component-based embedded systems for which the modeling, estimation and prediction of extra-functional properties are of crucial importance. The prototype, developed and integrated in the PROGRESS IDE, covers both introduction of new attribute types and specification of attributes for components and other modeling entities, with data formats ranging from primitive types to complex models handled by external tools.

Our plan is to further develop the model and the tool. The validity conditions can be further formalized to enable automatic selection of attribute values depending on the context in the development process. The same is true for the filter selection mechanism that should enable the developers an easy selection process. Further, we plan to

develop an attribute navigation tool that will be able to show differences between different attribute values and validity conditions. Finally, a set of predefined attributes will be specified for the ProCom component model, which will improve the efficiency and simplicity of attribute management.

Appendix A: Attribute Framework Meta-model

Below, the full attribute framework meta-model is presented.



References

1. Shaw, M.: Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does. International Workshop on Software Specification and Design (1996) 181
2. Sentilles, S., Vulgarakis, A., Bureš, T., Carlson, J., Crnković, I.: A component model for control-intensive distributed embedded systems. In Chaudron, M.R., Szyperski, C., eds.: Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008), Springer Berlin (October 2008) 310–317
3. Bureš, T., Carlson, J., Crnković, I., Sentilles, S., Vulgarakis, A.: ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University (June 2008)
4. Crnković, I., Larsson, M.: Building Reliable Component-Based Software Systems. Artech House, Inc., Norwood, MA, USA (2002)
5. Crnkovic, I., Larsson, M., Preiss, O.: Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes. In: Architecting Dependable Systems III. Volume 3549 of LNCS. Springer Berlin (2005) 257–278

6. ISO/IEC: Information Technology - Software product quality - Part 1: Quality model. Report: ISO/IEC FDIS 9126-1:2000 (2000)
7. Bertoa, M.F., Vallecillo, A.: Quality attributes for COTS components. In: 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2002. (2002)
8. Štěpán, P.: An extensible attribute framework for ProCom. Master's thesis, Mälardalen University, Sweden (2009)
9. Hissam, S., Moreno, G., Stafford, J., Wallnau, K.: Packaging predictable assembly with prediction-enabled component technology. Technical Report: CMU/SEI-2001-TR-024 (2001)
10. Maaskant, H. In: A Robust Component Model for Consumer Electronic Products. Volume 3 of Philips Research. Springer (2005) 167–192
11. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. Computer **33**(3) (2000) 78–85
12. Koziolek, H.: Parameter dependencies for reusable performance specifications of software components. PhD thesis, Oldenburg, University (2008)
13. Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., Tivoli, M.: The SAVE approach to component-based development of vehicular systems. Journal of Systems and Software **80**(5) (May 2007) 655–667
14. Zschaler, S.: Formal specification of non-functional properties of component-based software. In: In: Proc. Workshop on Models for Non-functional Aspects of Component-Based Systems. (2004)
15. Aagedal, J.Ø.: Quality of Service Support in Development of Distributed Systems. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo (2001)
16. Espinoza, H., Dubois, H., Gérard, S., Pasaje, J.L.M., Petriu, D.C., Woodside, C.M.: Annotating UML models with non-functional properties for quantitative analysis. In Bruel, J.M., ed.: MoDELS Satellite Events. Volume 3844 of LNCS., Springer (2005) 79–90
17. Bianco, P., Lewis, G.A., Merson, P.: Service level agreements in service-oriented architecture environments. Technical Report CMU/SEI-2008-TN-021, Carnegie Mellon (2008)
18. Land, R., Carlson, J., Larsson, S., Crnković, I.: Towards guidelines for a development process for component-based embedded systems. In: Workshop on Software Engineering Processes and Applications (SEPA) in conjunction with the International Conference on Computational Science and Applications (ICCSA), Springer (June 2009)