Refining Extra-Functional Property Values in Hierarchical Component Models

Thomas Lévêque
Mälardalen Real-Time Research Center
Mälardalen University
Västerås, Sweden
thomas.leveque@mdh.se

Séverine Sentilles
Mälardalen Real-Time Research Center
Mälardalen University
Västerås, Sweden
severine.sentilles@mdh.se

ABSTRACT

It is nowadays widely accepted that extra-functional properties (EFPs) are as important as functional properties for system correctness, especially when considering systems such as safety-critical embedded systems. The criticality and resource-constrained nature of these systems necessitate to be able to predict tight and accurate extra-functional property values all along the development, from early estimations to measurements. By using a hierarchical component model that allows implementing components as an assembly of subcomponent instances, the same component can be instantiated in several assemblies, i.e. in different usage contexts. Many EFP values are sensitive to the usage context and knowing information about the enclosing assembly enables refining the values of the properties on the subcomponents. Such refinement is usually not supported and the consistency between refined values and the original ones not ensured. This paper presents the concepts and mechanisms to support EFP refinement in hierarchical component models with explicit property inheritance and refinement policies which formally define consistency constraints between refined value and the original one. These policies are interpreted and ensured for all actors and in all workspaces. The paper also describes the related experiments performed on the ProCom component model.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics;
D.2.13 [**Reusable Software**]: Domain engineering;

General Terms

Measurement, Performance

Keywords

extra-functional properties, multiple values, inheritance policy, component type, component instance, virtual workspace

CBSE'11, June 20–24, 2011, Boulder, Colorado, USA.

1. INTRODUCTION

The development of embedded systems is nowadays challenged by a rapid and important increase in the complexity of the systems to build. This complexity results not only from a will to provide more advanced software functionalities, but additionally from intricate interplays between several factors including distribution and a mix between hardware and software functionality. In particular, as defined by IEEE, embedded systems are small computational devices integrated into larger products to perform some of the overall product requirements. Consequently, they are as per nature severely resource constrained and must in general satisfy dependability (reliability, safety, etc.) and real-time (execution time, response time, etc.) demands. As a result, supporting the management of extra-functional properties (EFPs) and guarantying their correctness is as important in those systems as their functional counterpart.

Component-Based Software Engineering (CBSE) is an approach that has already been proven successful for alleviating system complexity, mainly regarding functional aspects. Much less attention has been paid to extra-functional properties [20] and although that topic has gradually gained interest within the CBSE community as shown in many recent works [23, 15, 4, 2, 24, 12], no standardized solution has emerged yet for the specification, management and assessment of extra-functional property values. As highlighted in [8], a main reason behind this state-of-fact is that extra-functional properties are highly heterogeneous and most of their values are context-sensitive, i.e. dependent upon factors such as the overall system architecture, the usage profile, the specific hardware of the targeted platform and/or even upon the value of other properties.

This context-sensitivity of EFPs makes them difficult to handle in a reusability context. Since components are intended to be reusable in different systems, the value of their associated extra-functional properties must remain valid in these contexts. To cope with this issue, either their evaluation is often simply postponed to late development phases or over-estimations are introduced to ensure the validity of the property in different contexts.

This is a major problem in embedded system development for which safety and real-timeliness concerns are dependent upon the accuracy and tightness of the extra-functional values with regards to their actual value. Hence, this calls for solutions to manage extra-functional properties and enable refining their values in a similar manner as what is done for functional ones, i.e. assessed and gradually revised the values throughout the development process (starting from early

estimates to more accurate values). This is a non-linear process that requires to consider multiple values with possibly different evaluation context.

In a previous work [21], we proposed a framework, called the attribute framework, tailored for component model development that enables specifying and managing EFPs in a uniform way. One particularity of this framework is to sustain multiple values for a given extra-functional property, hence supporting multiple contexts. In this paper, we propose a method to refine the values of these extra-functional properties. The method relies on the assumptions that the refinement of extra-functional properties is a parallel activity to functional refinement and that the functionality of the component under study is not changed during the extra-functional refinement. As a basis for the work, we re-used the attribute framework together with ProCom [22], a hierarchical component model for distributed embedded system that enables having typed hierarchical components consisting of instances at n-levels of nesting.

The rest of the paper is organized as follows. Section 4 presents our case study. Section 4 describes the background information. Section 2 describes the challenges that are faced in the refinement of extra-functional properties in hierarchical component model. Section 3 describes the approach. Section 7 evaluates the work and Section 9 introduces the related work before the conclusions in Section 10.

2. MAIN CHALLENGES

Following the CBSE's principles in presence of a hierarchical component model, a system can be constructed out of components (the composites), implemented as assembly of component instances. This is a recursive process that allows a component instance to be in its turn an instance of a composite. As a result, a hierarchical component model generally implicitly leads to have multiple instantiation levels, i.e. a hierarchy of component instances. Looking at extra-functional properties, this raises the question of what the influence of the multiple instantiation levels on the values of the extra-functional properties is.

Following a common assumption, the value of an extra-functional property specified for a component type must also hold for all its instances as it is the case for the queryLanguages property shown in Figure 1. Some properties such as the Worst-Case Execution Time (WCET) can be smaller in a more constrained environment (see Figure 1. For example, a smaller value range on some input parameters would remove possible execution paths in the component's behaviour possibly leading to greater WCET. EFP values of instances can benefit from the knowledge (architectural, hardware, etc.) of the composite they are enclosed in. This means that it should be possible to refine the EFP value, defined for a component type, on one or several of the component instances. The question is in that case how to determine (1) what are the properties that can be refined on the instances, (2) how this refinement can be done and (3) if there is some constraints associated with this refinement. For example, the value of the WCET defined on the DB component type should bigger or equal to the value specified in each instance, since the value defined on a type denotes a upper bound that should hold for all instances. The reciprocal also applies: what are the influences of EFP values specified on component instances only with respect to the component types.

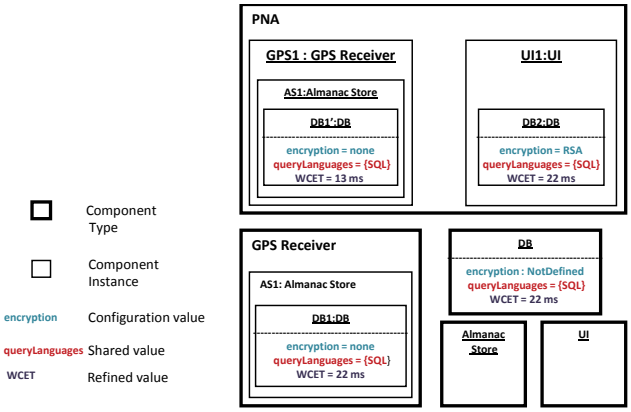

Figure 1: Refinement examples

In general, available tools to compute EFPs do not know how to manage EFPs with multiple values. As an example, a WCET analysis assumes that there is only one WCET value for each component. While CBSE helps to manage complexity, managing multiple EFP values introduces a new complexity level. This explains why EFPs are generally assumed to have at most one value in most of component models. The related challenge is how to allow multiple context sensitive values for EFPs while reducing the complexity to manage them for analysis and users.

Many works propose to annotate system and component model with metadata to describe EFP values. A survey can be found in [13]. However, extra functional properties are not independent. So, the consistency of attribute values should be ensured. This lack of consistency management leads in practice to lower the confidence of reliability of EFP values and to force designers to execute all EFP value computation after each modification or when a component is reused. Component based approach has been recognized as a promising way to improve reuse and quality in embedded systems. While efficiency of component functional description reuse has been proved, reuse of extra functional properties still need to be improved.

3. APPROACH OVERVIEW

Taking the above mentioned challenges into consideration, our objectives are:

[Objective 1] *Refinement of EFP values*
Allowing and controlling the refinement of EFPs values according to the usage context of the related component, i.e. enabling deriving EFP values between component type and instances in a controlled manner;

[Objective 2] *Consistency between EFP values*
Checking and ensuring the consistency of EFPs values whenever it is possible;

[Objective 3] *Usage Transparency*
Hiding the versioned nature of EFPs to the analysis tools and tool users.

While our main objective is to support EFP value refinement, we would like to make the approach as generic as possible. That is why we propose to support EFP value refine-

ment using type specialization and instantiation paradigms. To ensure the genericity of our approach, an object (component or any model element) can be refined by creating a new object which is an instance of the original one (e.g. a component instance), or which is a subtype of the original one (e.g. a component that extends another component). The original and the refined object coexist in the workspace. We choose the following definition for a refined object: *"an object is a refinement of another object if all information defined by the original object is still valid for the refined object"*. From a certain point of view, they cannot be distinguished. The original object is an abstraction of the refined one. Several objects could be refinement of the same object. It enables to have multiple variants of an object which can also have been refined by a set of objects.

We base our work on the following assumptions:

- *Assumption 1*
  A system is designed using a hierarchical component model where a composite component is an assembly of component instances;
- *Assumption 2*
  Extra functional properties (EFPs) are defined as annotations on model elements;
- *Assumption 3*
  Multiple values of EFPs can be defined and there is means to distinguish between them (as with metadata for example);
- *Assumption 4*
  All actors used the same tool.

ProCom together with the attribute framework and PRIDE [6] follow all these assumptions. Hence, we follow an MDE approach using ProCom component model which can be summarized as:

- Introducing metaclasses to support multiple levels of instantiation with strong typing of EFPs and their metadata;
- Defining and interpreting explicit inheritance and derivation policies for attributes;
- Providing automatic selection mechanism based on user preferences and task context to let the user work in a mono version workspace.

4. BACKGROUND WORK

4.1 The ProCom Component Model

To address the key characteristics and concerns of embedded system development, i.e. providing support for the complete design-verification-deployment cycle, distribution, extra-functional properties and code reuse, we have developed a domain specific component model called ProCom [22].

One of the particularities of ProCom lays in its notion of rich design-time typed components, where components are seen as the collection of all the artifacts needed or produced during the development of this system element. This encompasses artifacts such as source code, early design models, test results, architectural models, and more specifically, of analysis models, behavior models, and their corresponding results. Reusing a component means reusing not only its concrete realization, but the whole collection of artifacts.

Another of its particularity is to consider the need for the design of a complete system consisting of both complex and distributed functionalities on one hand, and small low-level control-based functionalities on the other. As a consequence, ProCom is structured into two hierarchical layers, each layer dedicated to a specific type of functionality. The upper layer, called ProSys, is intended for modeling distributed, complex, active and concurrent subsystems, communicating via asynchronous message passing. The lower layer, called ProSave, serves for modeling of non-distributed, passive and small units of functionalities, closer to tasks or control loops. The connection between layer is done through modeling ProSys component out of ProSave components. For more details, see [7].

4.2 The Attribute Framework

Although dedicated to distributed embedded systems and developed to facilitate evaluating extra-functional properties, extra-functional property value specification is not an intrinsic part of ProCom. This specification is done through the attribute framework metamodel, which derived extra-functional properties artifacts are integrated to the rich component definition.

The attribute framework [21] provides a systematic way to support the management and integration of extra-functional properties during the development of a component or a system. In it, extra-functional properties are represented by attributes with a unique identifier and several values. The complete list of the attribute types that are available during the development is stored in an attribute registry together with the specification of each attribute, that is (i) the list of entities to which this attribute can be attached, and (ii) the valid format for its values (e.g. integer, interval, model, etc.). Providing that it is authorized by its specification, an attribute can be associated with any entity of a component model such as component, service, port, connection or even component instance.

5. SUPPORT FOR MULTI-LEVEL INSTANTIATION

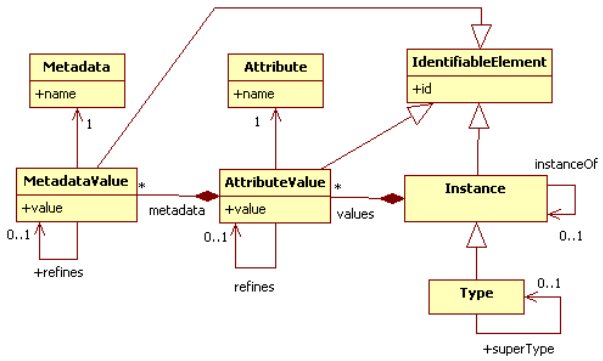

Figure 2: Simplified view of the attribute metamodel

First, in order to ensure consistency of refined EFPs, we need to track which objects are refined. That is why we

changed the ProCom metamodel to make explicit instantiation and specialization. To do so, we have modified the core part of the attribute meta-model in introducing *Instance* and *Type* meta classes (see Figure 2). All model elements inherit from one of these metaclasses. Every model element has a global unique identifier. Refined objects and the related original object must be at least distinguishable by their id. PRIDE generates this id for every model element. You can observe that any model element (which is an *Instance*) can be associated to a set of attribute values with their related metadata.

5.1 Introducing Refinement By Instantiation

The type-instance design pattern is often used in modeling languages to allow specifying information (in the type) that will be shared by a set of objects (in the instances). There is an implicit conformity between instances and their type. Object-oriented programming languages use heavily this pattern in which a class defines a set of attributes and methods that all object which are instances of this class will inherit. In such languages, conformity is checked at compilation time and at runtime. In general, an instance cannot be a type, which limits the number of instantiation levels to one.

In our case, we want to allow to refine an object as many times as necessary. In this case, the number of instantiation level is not limited. That is why a *Type* inherits from *Instance*. It becomes possible to have instances which are also types enabling refine them with their instances.

To have explicit refinement traces, an instance is linked to his type thanks to an *instanceOf* link. While typical instantiation forces instances to value the attributes defined by the related type, we do not ensure such property because EFPs do not need this strict conformity as they may be defined after several refinement. In order to facilitate evolution management, we choose to forbid an instance to change its parent after creation time. In other words, the *instanceOf* link destination is defined at the creation time of the source element.

5.2 Introducing Refinement By Specialization

In object oriented languages, a class can be the specialization of zero, one or many other classes. A child class inherits all information from the parent ones except some of them such as their names. It refines its parent class by adding new information (new attribute and methods). We choose to manage only simple inheritance where a class can at most inherits from another class. To have explicit refinement traces, a type is linked to his parent type thanks to a *superType* link. As with instantiation, we choose to forbid a subtype to change its super type, i.e. to point to another type, after creation time.

5.3 Application To ProCom

Figure 3 shows a simplified view of the ProCom metamodel. One original part of the ProCom meta model is the fact that a component instance is also a component (*CompInstance* inherits from *Component*). It is considered as a component type which is used in a specific context: its enclosing composite. This means that a component instance is a refinement of the related component.

We defined *Component* class as a sub class of the meta class *Type* in order to be able to refine component using

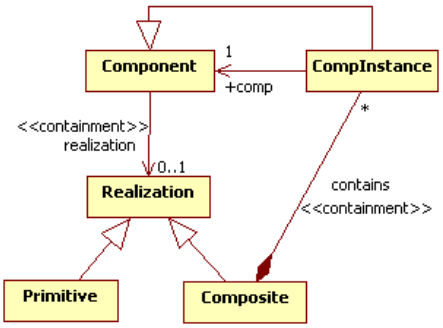

Figure 3: Simplified view of the ProCom metamodel

specialization. All others classes are defined as sub classes of the *Instance* meta class. In order to apply the same approach to other component models, we have only to define for each class if it inherits from *Type* or *Instance*.

5.4 Instance Creation

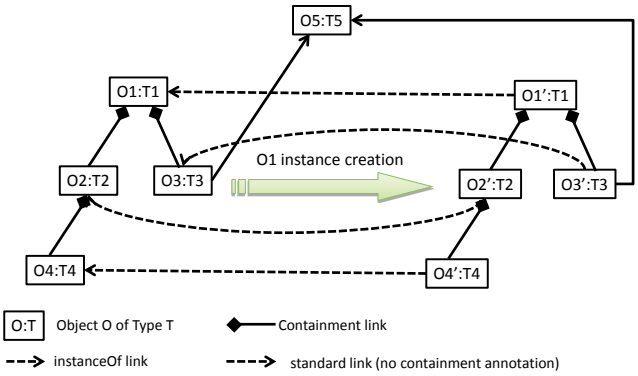

Figure 4: An instantiation example

We used Eclipse Modeling Framework (EMF) to manage our ProCom models. In EMF, a relationship R from a class A to a class B can represent a whole/part relationship if it is associated to a containment annotation. This annotation defines a strong life cycle dependency between the source element S and destination elements Di. An element Di cannot exist without its parent S. As a consequence, deleting S forces removal of Di elements. An important property is that one element Di must have at most one parent element.

Our algorithm to create an instance preserves this property. It can be summarized as cloning every model element which is contained (following EMF containment links) in the object to instantiate, and to keep same reference values if destination model elements have not been cloned. So, containment relationships guide our instantiation mechanism. The cloned object is associated to a new ID and is linked to its corresponding parent using *instanceOf* reference. For all shared EAttribute and EReference definition (same attributes and outgoing relationships defined for both classes), the attribute or reference value is copied or cloned. As we want to manage EFP inheritance, the EFP values are not copied but they are computed at access time.

Figure 4 shows an example of the creation of an instance

of the model element O1. All contained elements, i.e. the transitive closure of containment, which includes O2, O3 and O4 model elements, are instantiated. O5 is not instantiated as it is not contained in O1 and all links pointing to O5 have been cloned on the related created instances: in this example, O3' is linked to O5.

An algorithm based on same principles is used to synchronize modifications on refined objects and their related original objects.

5.5 EFP Inheritance Policies

First of all, we take the hypothesis that the refined objects cannot exist without their original objects. This hypothesis is ensured by PRIDE which forces to import original objects and deletes refined objects when the original object is removed. As values relationship is a containment relationship, an attribute value can only be associated to exactly one model element. From a conceptual point of view, a refined object must be able to have his own attribute values which may refine original values. Value refinement is tracked using refines links (see the "refines" relationship in Figure 2). As other model elements, attributes and metadata have ids. We choose to ensure that refined values are the same if no modification has been performed on one of ancestor in the inheritance model element hierarchy. In order to facilitate evolution management of refined values, attribute values are the same all the time if no modification has been performed on one of ancestor in the inheritance model element hierarchy. In particular, it ensures that refined value ids does not evolve if it is not necessary. It considerably simplify evolution management as it ensures that refines links always points to the original value even if the original evolves except. In addition, if the original value is deleted, the refined value must be deleted.

As already said previously, the inherited attribute values are computed each time you access to the attribute values of a model element. This computation is guided by inheritance policies of attributes. We have defined *three different attribute value inheritance policies*:

– **final**
  the value of a **final** attribute is always inherited and can only be modified on the original object where this value has been defined;

– **override**
  the value of a **override** attribute is inherited by default but can be overriden by the user. Additionally, OCL constraints can be specified to check the consistency with the parent value;

– **notInherited**
  a **notInherited** attribute value is never inherited.

In order to be able to define an attribute value on a model element, its definition must be available for the considered model element. It explains why we also need an attribute definition inheritance mechanism. We have defined *two attribute definition inheritance policies*:

– **inherited**
  the definition of an **inherited** attribute is always inherited;

– **notInherited**
  the definition of a **notInherited** attribute is never inherited.

Identifier	Def. Policy	Value Policy	Constraint
Vendor Name	notInherited	notInherited	none.
Acquisition Time	inherited	override	originalValue >= refinedValue.
Response Time	inherited	override	originalValue >= refinedValue.
WCET	inherited	override	originalValue >= refinedValue.
Static memory	inherited	final	none.

Table 1: Example of attribute inheritance policies related to PNA example.

The value inheritance and attribute definition inheritance policies are not independent. **Final** and **override** value inheritance policies require that the attribute is defined. This is the reason why these policies imply setting the **inherited** attribute definition inheritance policy. The inheritance policies may differ for specialization and instantiation. In order to manage these specific cases, different inheritance policies can be defined for specialization and instantiation. Table 1 gives some examples of possible attribute inheritance policies. the Vendor Name attribute is not inherited at all. The attribute inheritance policies may be arbitrary defined. However, defining specific inheritance policies such as override can ensure that some properties computed for the original object are still valid with the refined object. As an example, if WCET attribute is defined as inherited and override with the constraint that refined value cannot be greater than the original value, it ensures that all WCET computation done using refined object is valid even if it may not be as precise as it is possible.

6. VIRTUAL WORKSPACE

While having value for each specific usage context seems to be convenient, the number of possible configurations of selected attribute values is too big to be manageable by hand. In particular, tools such as static analysis cannot work on multiples attribute value versions. That is why an automatic or semi-automatic selection mechanism should be available for the users.

To be able to select the most appropriate value, we need to be able to distinguish the differences between the different values of a same attribute. In order to be able to process automatically this selection, metadata must be strongly typed. Using many case studies, we have observed that the considered attribute values depend upon the current activity the user wants to perform. We propose to let user define what his current task is and in particular, what the attribute values he is interested in are for his current task. Based on the user's preferences, he will have access only to the attribute values that match his expectations. While the workspace contains all attribute values, only some of them are visible from the user. We call this view of the workspace a virtual workspace. Figure 5 shows the meta model of a virtual workspace. While multiple user tasks may be defined, there is at most one defined as the current one at a given time and each task has its own selection conditions for attribute values. We propose a selection language based on OCL expressions which allow to specify the attribute values which must

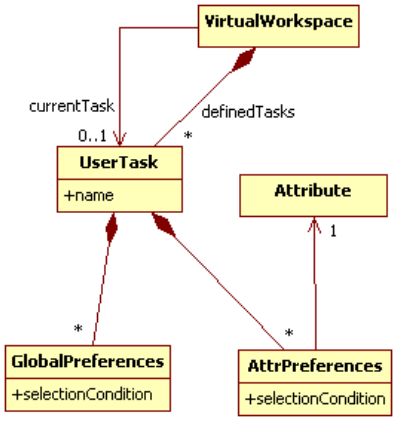

Figure 5: VirtualWorkspace meta-model

be considered. The grammar of this language is described in Listing 1. A selection condition expression (SelectCond) is a list of selection expression (AtomCond). The selection expressions are evaluated in the order of their definition until one of them results in at least one selected attribute value. A selection expression defines mandatory constraints and optional ones. The evaluation of such expression tries first to select values which conform to mandatory and optional constraints. If there is no such value, the selection is performed by considering only mandatory constraints. A constraint (AttrCst) is defined as an OCL logical expression assuming that the OCL context is the AttributeValue class.

Listing 1: Selection Expression Grammar

```
SelectCond => AtomCond [ or SelectCond ]
AtomCond => mandatory( AttrCst )
            [ and optional( AttrCst ) ] |
            optional( AttrCst )
AttrCst => an OCL expression
```

Listing 2 presents an example where a user defined that all attribute values must be related to FreeRTOS platform and that measured values are preferred upon estimates. It allows to ensure that all considered EFP values are related to the same targeted platform. Acquisition Time values must be lower than or equal to -10 according to the user defined preferences.

Listing 2: Selection Expression Example

```
Global Preferences:
 mandatory( self.metadata(''platform'').
     stringValue() = 'FreeRTOS' ) and
 optional( self.metadata(''measured'').
     booleanValue() = true )

Acquisition Time Attribute Preferences:
 mandatory( self.metadata(''minTemperature'').
     intValue() <= -10 )
```

The selection expressions are merged and transformed to an equivalent OCL expression using "if then else" and "exists" constructs. It allows to use OCL runtime to evaluate our selection expression. As analysis access to attribute values thanks to the attribute framework which evaluate the selection expressions to present only matching values, the analysis as the user work in a mono version workspace. In our example, only 7 min value for acquisition time attribute will be visible for the user.

The system does not ensure that only one value is selected. If selection expressions lead to multiple selected values for the same attribute, a warning is logged and a non deterministic selection is performed on this value set.

7. EXEMPLIFICATION

7.1 The GPS Example

To illustrate ideas and results, we use the example of the Personal Navigation Assistant (PNA) system. A PNA relies on the Global Position System (GPS) to provide aid to navigation functionalities such as computing the best routes between two cities, distance and time to arrival, current speed, direction, etc. A GPS is composed of a group of 24 Earth-orbiting satellites periodically sending information to GPS receivers that calculate their Earth-based geolocation. In common language, GPS refers to the GPS receiver devices only. Likewise in this paper, we focus on the GPS receiver part of the PNA.

A GPS receiver is a device able to determine its location on Earth through a trilateration calculation method that requires the exact position of at least three satellites. With three satellites, a GPS is able to estimate its 2D-position (longitude and latitude) whereas with four satellites, it can also compute its altitude. The more satellite positions the receiver get, the more accurate is the position calculation. For example, most of today receivers, such as the Garmin G18 [10], tracks simultaneously up to twelve satellites for better results. Other type of receivers includes multiplexing channel receiver that can only follow one satellite at a time, thus forcing them to switch rapidly between the satellites being tracked at the cost of time and precision.

In order to know the satellite's position precisely, the GPS receiver must be fully aligned with the signal of the satellite being tracked. To enable satellite's position discovery, the GPS receiver uses a clock to have the current time and an almanac containing the supposed positions of a satellite at a given time.

To create the PNA, the GPS receiver is associated to a navigation processor (Navigation System) that computes the navigation data (current position, direction, current speed, etc.) and a graphical user interface that enables displaying the device's geolocation data on maps, together with the navigation data and other information such as distance and time to destination, point of interests, etc.

Hence, the PNA system (illustrated on Figure 7) is developed out of four ProSys components (the *GPS Reveiver*, *Power Management*, *Navigation System* and *UI*) since a PNA installed in a car could be distributed, i.e. having its central computation unit in one part of the vehicle while the signal receiver units would be located closer to the roof for better reception.

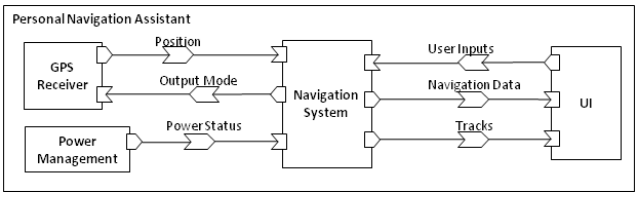

Figure 7: a PNA system modelled in ProCom

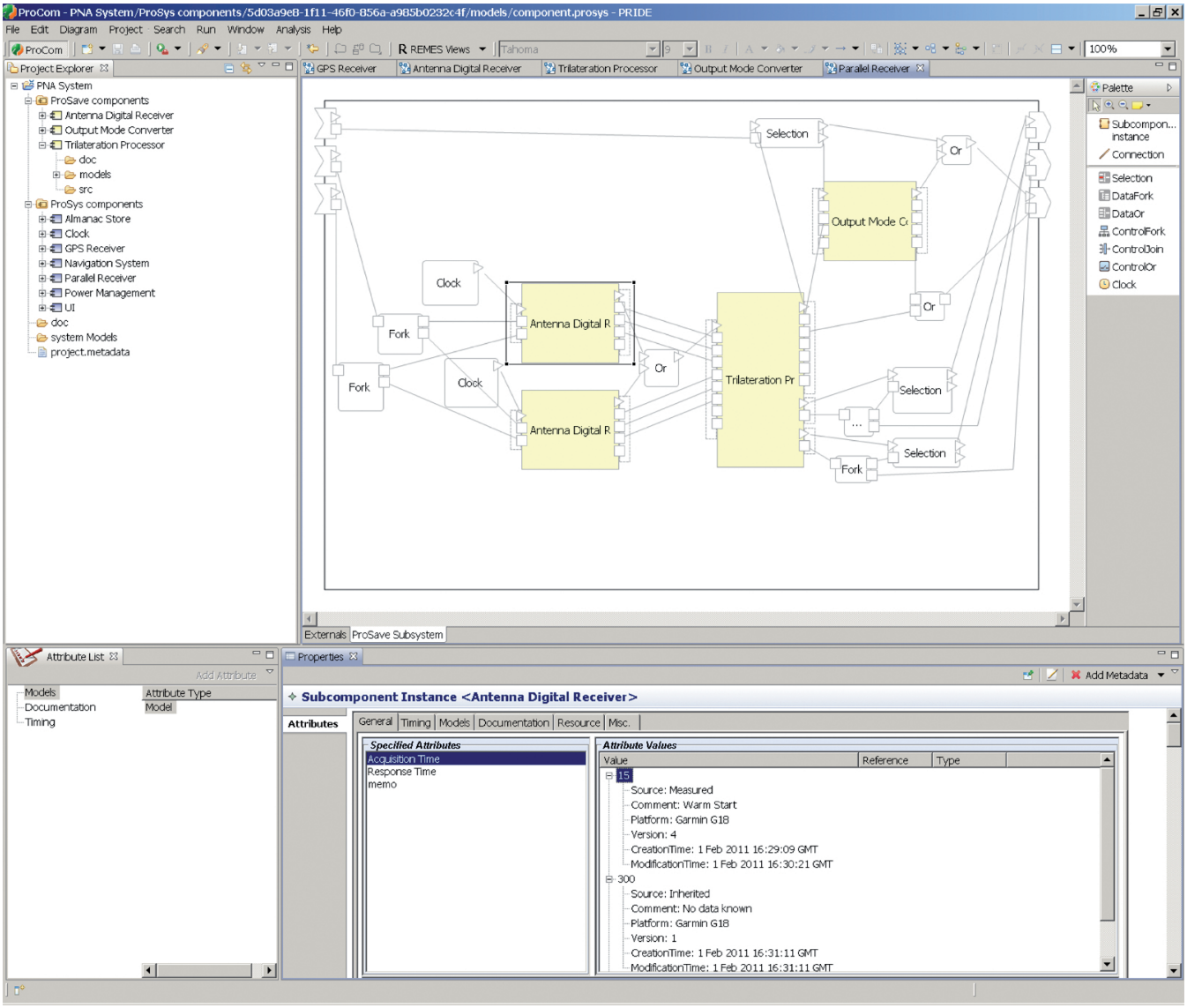

Figure 6: Screenshot of Pride showing different attribute values associated to the Antenna Digital Receiver

Looking closer at the *GPS Receiver* component shown in Figure 8, it is a composite ProSys component that consists of the *Clock* and *Almanac Store* ProSys components to help the GPS receiver to faster locate the satellites on start-up and a *Parallel Receiver* component that simultaneously tracks up to twelve satellites to compute the geolocation of the device.

In that component, the *Parallel Receiver* is a primitive ProSys component built out of ProSave components as shown on Figure 9. It consists of twelve instances of an *Antenna Digital Receiver* ProSave component, a *Trilateration Processor* and an *Output Mode Converter*. For readability purpose, only two instances of the Antenna Digital Receiver are depicted on Figure 9. The Antenna Digital Receiver is in charge of the synchronization with the satellite's signal and get the satellite location. The Trilateration Processor computes the actual position of the devices and if activated, the Output Mode Converter converts the position into a different format. The communication between these components follows a pipes-and-filters architectural style separating data flows from control flows. Data input and output ports are denoted by small rectangles whereas trigger ports are triangles. Moreover, the antenna digital receivers are periodically activated every ten seconds. Once one of the Antenna Digital Receiver has terminating its computation, it activates the trilateration processor.

As listed in Table 2, several attributes can be used to analyse the PNA system such as the acquisition time attribute which specifies the amount of time that a component requires to correctly receive the satelitte's position signal. Such an attribute is dependent upon the platform (parallel receiver vs. multiplexing channel receiver), therefore it is important to use the metadata Platform to know the context in which this value is valid.

Table 3 shows attribute values defined for the GPS Receiver component.

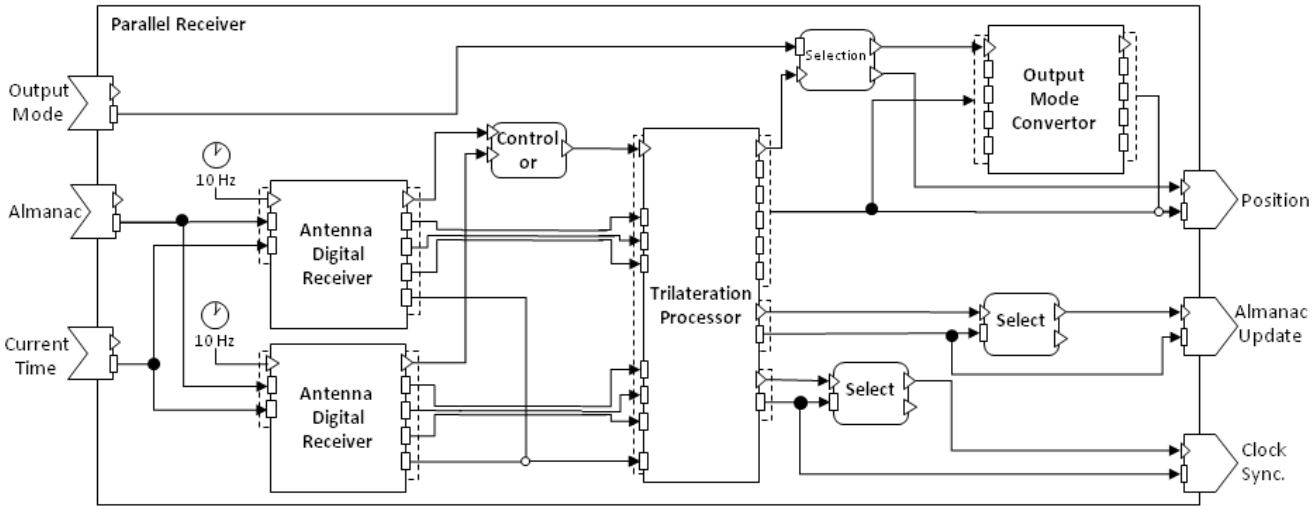

Figure 9: A simplified version of a ProSys primitive receiver

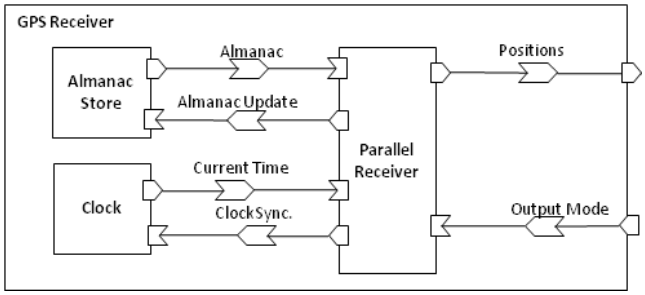

Figure 8: Model of the GPS receiver as composite ProSys components

Table 2: Examples of attribute specification related to the PNA example.

Identifier	Attributables	Data Type	Doc.
Acquisition Time	Component, Instance	Int	Time (in ms) to acquire a signal
Response Time	Component, Instance	Int	Time (in ms) to react to a given input
WCET	Service	Int	The maximum number of clock cycles the service can consume before terminating.
Static memory	Component, Instance	Int	The amount of memory (in kB) statically allocated by the component or subsystem.

Table 3: GPS Receiver Acquisition Time Attribute Values

Value	Min Temperature Metadata value	Measured Metadata value	Platform Metadata value
43 s	0	true	FreeRTOS
32 s	0	true	Linux
5 min	-15	false	FreeRTOS
7 min	-25	true	FreeRTOS
5 min	-15	false	Linux

8. OUTCOMES

The approach has been implemented in Pride, the ProCom IDE. The attribute framework has been redesigned to introduce strong typing of metadata and the possibility to define the inheritance policies. You can see attribute values defined for our *Parallel Receiver* component on Figure 6.

We designed several systems including an Advanced Cruise Control system, an automatic truck and the example presented in this paper using Pride. In early Pride releases without refinement support, users were lacking support for EFP inheritance and consistency checking.

While Model Driven Engineering allows to reason at a higher abstraction level, the model itself can also contain multiple abstraction levels. In our work, we have shown how a component instance can be seen as a refinement of the related component. The idea of such refinement support started from observation of the system synthesis needs where each component instance has its specific optimization property depending on its usage context. In particular, two different instances of the same component may be optimized in different ways depending on their usage context. We have also observed this need with fault tolerant properties.

The control of the EFP modification is not possible if hypothesis 4 is not fulfill. It ensures that all modifications on the model is performed thanks to Pride, especially that each time EFP values of model element are accessed, the inherited values have been recomputed and that EFP refined values are still consistent with their original value. Otherwise, Pride shows a warning and list only the original value.

On one hand, while the selection expression language seems to be able to define all selection we need, some expressions

could be quite complex such as selecting the most recent value that follow a certain constraint. We investigate to introduce some keywords for the most used constraints. On the other hand, the language is easy to learn and to use as constraints are OCL expressions.

9. RELATED WORK

Within the last decade, extra-functional properties have gradually gained importance in system development to be recognized as an indispensable counterpart to functional properties for establishing system correctness. As a consequence, several works have been proposed to specify either extra-functional properties in general [9, 1, 18, 24, 13] or focusing on a dedicated subset of extra-functional properties such as temporal, performance or resource-related properties [23, 16, 5]. However, none of the above approaches considers multiple values for EFPs. As a consequence, they only support predefined value metadata. The refinement of extra-functional property values is often done in ad hoc manner, without tracking changes nor ensuring the consistency between changes. Furthermore, to the best of our knowledge, there is little work in the CBSE community concerned with studying the relationships and impacts of multiple levels of component instantiations on EFP values.

On the other hand, in the modelling domain, several approaches [11, 19, 17, 3] have been proposed to manage multiple instantiation levels together with the related attributes at each level. Note that in that specific context the term attribute refers to field or static variables that can be instantiated at a given level of instantiation and not to extra-functional properties. These approaches address the similar problem of having to deal with multiple levels of instantiation, which requires to be able to determine (1) whether an element in a higher instantiation level influences elements beneath that level and (2) what the impacts of that influence are. In difference to those approaches, our approach is not concerned with instantiation at run-time of elements (objects or variables) but with the impact of the instantiation hierarchy on the values of extra-functional properties, that is with refining the extra-functional property values within a static hierarchy of component instances. Also, the potency concept proposed in [3] is not applicable in the context of our work, since determining the level of instantiation at which an extra-functional property value would be applicable seem to be a rather difficult decision to make at the development of a component type or at the specification of an extra-functional property. However, one particularity of the potency concept is to enable the framework to know which values should exist at which level of instantiation. Similarly in our approach, we derive only values with suitable inheritance policies. Finally a main difference between these approaches and our work lays in the fact that we have constraints to guaranty the coherence in the refinement process.

With regards to our usage transparency objective, our work relates to Mylyn [14]. Mylyn is an Eclipse extension that facilitates multi-task activity by hiding unneeded information such as unused projects. It allows to switch from an activity to another one without restarting Eclipse. Mylyn attachs task information to artifacts to easily recognize what are the information related to a specific task. Our approach differs from the fact that our workspace view is dynamic in the sense that selection condition can evolve over time and that elements are not directly associated to a task. A virtual workspace is described by intention instead of by extension.

10. CONCLUSIONS

We have presented in this article how to support and control refinement of extra functional properties in hierarchical component models by introducing attribute inheritance mechanism based on explicit inheritance and refinement policies. New extra functional property and analysis can benefit this refinement support without modification on the environment. Our contributions includes

- the demonstration that hierarchical component models needs support of multi-level instantiation to support component refinements;
- the definition and the implementation of a multiple level instantiation mechanism;
- to provide an inheritance attribute definition and value mechanism that interprets fine grain explicit inheritance policies;
- to propose a new concept called virtual workspace that allows users to work in a mono version workspace.

While this work has been applied on the ProCom component model, it can be generalized to other hierarchical component models. We focus our work now on finishing the implementation and the validation of the virtual workspace concept. Our future work will intend to provide support for other explicit evolution policies including versioning ones for extra functional properties. We are also investigating different mechanism to allow designers to work at the most convenient abstraction level depending on their goals.

11. ACKNOWLEDGMENTS

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS.

12. REFERENCES

[1] J. Ø. Aagedal. Quality of Service Support in Development of Distributed Systems. *PhD thesis*, Faculty of Mathematics and Natural Sciences, University of Oslo, 2001.

[2] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli. The SAVE Approach to Component-Based Development of Vehicular Systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

[3] C. Atkinson, M. Gutheil, and B. Kennel. A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Transactions on Software Engineering*, 99(RapidPosts):742–755, 2009.

[4] S. Becker, H. Koziolek, and R. Reussner. Model-Based Performance Prediction with the Palladio Component Model. *the 6th international workshop on Software and performance*, 2007.

[5] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82:3–22, January 2009.

[6] E. Borde, J. Carlson, J. Feljan, L. Lednicki, T. Leveque, J. Maras, A. Petricic, and S. Sentilles. PRIDE, an Environment for Component-Based Development of Distributed Real-time Embedded Systems. *WICSA*, 2011.

[7] T. Bures, J. Carlson, I. Crnkovic, S. Sentilles, and A. Vulgarakis. ProCom - the Progress Component Model Reference Manual, version 1.0. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[8] I. Crnkovic, M. Larsson, and O. Preiss. Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes. In *Architecting Dependable Systems III*, volume 3549 of *Lecture Notes in Computer Science*, pages 257–278. Springer Berlin, 2005.

[9] X. Franch. Systematic Formulation of Non-Functional Characteristics of Software]. In *Proceedings of the 3rd International Conference on Requirements Engineering: Putting Requirements Engineering to Practice*, pages 174–181, Washington, DC, USA, 1998.

[10] Garmin. GPS 18 Technical Specifications (190-00307-00), Rev. D. Technical report, June 2005.

[11] R. C. Goldstein and V. C. Storey. Materialization. *IEEE Trans. on Knowl. and Data Eng.*, 6:835–842, October 1994.

[12] L. Grunske. Early quality prediction of component-based systems - a generic framework. *J. Syst. Softw.*, 80:678–686, May 2007.

[13] K. Jezek, P. Brada, and P. Stepán. Towards context independent extra-functional properties descriptor for components. *Electronic Notes in Theoretical Computer Science*, 264(1):55 – 71, 2010.

[14] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 1–11, New York, NY, USA, 2006.

[15] J. E. Kim, O. Rogalla, S. Kramer, and A. Haman. Extracting, Specifying and Predicting Software System Properties in Component Based Real-Time Embedded Software Development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.

[16] M. Mohammad and V. Alagar. Tadl - an architecture description language for trustworthy component-based systems. In *Proceedings of the 2nd European conference on Software Architecture*, ECSA '08, pages 290–297, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] B. Neumayr, K. Grün, and M. Schrefl. Multi-level domain modeling with m-objects and m-relationships. In *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling - Volume 96*, APCCM '09, pages 107–116, Darlinghurst, Australia, Australia, 2009.

[18] Object Management Group. A UML Profile for MARTE, Beta 1, August 2007. Document number: ptc/07-08-04.

[19] J. Odell. Power Types. *JOOP*, 7(2):8–12, 1994.

[20] H. Schmidt. Trustworthy components–compositionality and prediction. *Journal of Systems and Software*, 65(3):215 – 225, 2003. Component-Based Software Engineering.

[21] S. Sentilles, P. Stepan, J. Carlson, and I. Crnkovic. Integration of Extra-Functional Properties in Component Models. In I. P. Christine Hofmeister, Grace A. Lewis, editor, *12th International Symposium on Component Based Software Engineering, LNCS 5582*. Springer Berlin, LNCS 5582, June 2009.

[22] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic. A Component Model for Control-Intensive Distributed Embedded Systems. In M. R. Chaudron and C. Szyperski, editors, *Proceedings of the 11th International Symposium on Component Based Software Engineering*, pages 310–317. Springer Berlin, October 2008.

[23] K. C. Wallnau. Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC). Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon, 2003.

[24] S. Zschaler. Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof. *Software and Systems Modelling (SoSyM)*, 9:161–201, Apr. 2009.