Evolution Management of Extra-Functional Properties in Component-Based Embedded Systems

Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque, Séverine Sentilles
Mälardalen Real-Time Research Centre
Mälardalen University
Västerås, Sweden
{antonio.cicchetti, federico.ciccozzi, thomas.leveque,severine.sentilles}@mdh.se

ABSTRACT

As software complexity increases in embedded systems domain, component-based development becomes increasingly attractive. A main challenge in this approach is however to analyze the system's extra-functional properties (such as timing properties, or resource requirements), an important step in a development of embedded systems. Analysis of such properties are computational and time consuming, and often difficult. For this reason reuse of the results of the analysis is as important as the reuse of the component itself, especially in case of modifications of the context in which the component is used. This paper presents concepts and mechanisms that allow to automatically discover whether a property value is still valid when related components evolve: a value context language is proposed to formally define the validity conditions and identify possible threats.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Design, Measurement, Theory

Keywords

Evolution management, Component Based Software Engineering, Model Driven Engineering, Context, Extra-functional properties, Impact analysis

1. INTRODUCTION

Nowadays, the majority of computer systems are embedded in a variety of products, such as aircrafts, cars, consumer electronics, etc., to control their main functionality or to provide additional functionality. As expected, embedded systems face the general trend of an ever increasing complexity which tends to make their design, development, and maintenance more and more intricate. This issue is even more evident when particular extra-functional properties have to be guaranteed, like for instance in embedded real-time systems (ERTS) [10], that require tasks to be executed within a given amount of time.

Model-Driven Engineering (MDE) and Component-Based Software Engineering (CBSE) can be considered as two orthogonal ways of reducing development complexity: the former shifts the focus of application development from source code to models in order to bring system reasoning closer to domain-specific concepts; the latter breaks down the set of desired features and their intricacy into smaller sub-modules, called components, starting from which the application can be built-up and incrementally enhanced [17].

Despite the raising of the abstraction level at which the problem is faced and/or its decomposition into sub-issues, several aspects of such complex systems development remain still open research questions. Notably, quality attributes are typically crosscutting the functional definition of the application, making compositionality of analysis results for each component to hold only under particular conditions [6]. Moreover, each system's property demands for corresponding theories precisely defining how such a property can be specified, the kind of values it can assume, the manipulations it can undergo, and the impact a particular class of changes entails to the system. Such issues are critically relevant when considering evolution and maintenance activities, since changes operated on the current version of the application should preserve (or possibly improve) its characteristics.

In this paper we exploit the interplay between MDE and CBSE techniques to alleviate the issues related to quality attributes and their evolution. In particular, we introduce a new language for the description of the computational context in which a given property is provided and/or computed by some analysis. Then, whenever such context is modified we detect the operated revisions and, based on a precise representation of differences between old and new versions we analyze the impact caused on the attribute values. In this way, it is possible to provide a development environment with a validation feature, able to detect evolution issues as early as possible in the system lifecycle. The approach has been plugged-in in an existing framework that enables the definition of quality attributes for components. In this way, the framework has been provided with attribute evolution management capabilities including validation features, and allowed to validate this proposal against a case study.

The paper is organized as follows: Section 2 discusses the

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

motivations underlying this proposal together with existing approaches in related research areas; moreover, Section 3 introduces a running example to illustrate the current support to quality attributes specification and the corresponding impacts due to their evolutions on the system. Section 4 provides an overall view of our approach to deal with extra-functional properties evolution; then, Section 5 and Section 6 detail the relevant technical aspects enabling impact analysis support. The validation is described in Section 7, while Section 8 concludes the paper and presents our future works.

2. MOTIVATION & RELATED WORKS

One of the main challenges still open in CBSE is quality attributes management [6]; this is a two-fold issue: (i) how to express the properties (in which language and where), and (ii) how to perform their computation. In general, static analysis computes composite component properties from the ones owned by its subcomponents. The related computation cost is usually high, thus leading to perform analysis not in a systematic manner but rather only whether unavoidable. However, real-time requirements, severe resource limitations, and safety concerns, make a mere reuse of components from their functional perspective not applicable. System developers must also be able to predict, at an early stage, the impact of using a component on extra-functional system properties, e.g., timing and resource usage. When a component is reused, information stored along with it should be general enough to be reused in several different contexts, while at the same time providing as detailed information as possible in each specific case.

An active research field is devoted to analysis and detection of possible problems caused by revisions of a system, typically known as change impact analysis. The book in [1] offers a survey on available solutions related to change impact analysis at source code level. However, as noticed in [18] the increasing complexity of software systems requires the analysis and detection of possible problems triggered by a given evolution to be at the same level of abstraction of system design. In fact, in that way it is possible to ease the impact analysis task and in the meantime anticipate it at the early stages of the development, when modifications are more cost-effective.

In [7, 18] the authors discuss similar techniques to detect problems related to system architecture revisions: each time a modification is performed at architectural level, the designer can exploit slicing and chopping methods to synthesize the corresponding effects on the current code of the specification. Impact analysis support is extended to the development environment in [8], where design documents are considered as first class artifacts: whenever a modification is operated on them, corresponding impact on the existing implementation is automatically derived. In [12] the author proposes a framework to validate user's functional requirements by animating the available architectural description of the system. Finally, in [16] the authors describes an approach for assisting the software maintainer during an evolution activity on his/her demand based on the definition of bindings between formal descriptions of architectural decisions and their targeted quality attributes.

Another field of research is devoted to modeling and analyzing adaptations of a system at runtime due to environmental context variations. In [9] the authors introduce the concept of modes, that is structural constraints that drive a (sub)system configuration at runtime, impacting on its architectural arrangement. Modes are related to scenarios that abstract contexts of usage, entailing that system reconfigurations are induced by scenario variations. Such work is extended and improved by [2], where an approach is proposed for performance modeling and analysis of context knowledgeable mobile software systems. In particular, runtime reconfigurations are also interconnected to positioning and users.

The main distinction of our goal with respect to impact analysis approaches is the purpose of the analysis itself: we aim at synthesizing system evolution side effects as early as possible, by providing the developer with a validation tool showing the impact of the *ongoing* adaptation to the current application, hence before the changes themselves are committed. Whereas, usual impact analysis methodologies consider the revision phase as completed, thus presenting to the user the propagation of the *completed* adaptation to the existing system. As a consequence, the process itself is moved from the file repository back to the IDE supporting the system design; in this way, modification analysis can be performed at higher levels of abstraction and are no longer limited to file- or text-based reasoning.

On the other hand, we do not deal with adaptations at runtime, or in other words we cope with different kinds of adaptations: in the former case a system is provided with different operational modes and corresponding configurations to make it self-adaptable to different operating conditions. Whereas, in our case one or more components have to be reused in different scenarios, i.e. other systems and/or architectural arrangements that could affect and invalidate the preconditions by which component attributes have been asserted.

Our approach is based on the specification of component attributes as *context sensitive*, that is, each property value is specified together with a set of preconditions that must hold in order to keep the assertion valid. In this way, whenever a component is put in a different context it is possible to check if its attributes can be considered still reliable and hence usable to update the overall system properties.

In the following, our proposal is detailed by illustrating how we made it possible to describe components quality attributes and their validity conditions. Then, we discuss how to exploit such information to detect conditions by which assertions on components are no more valid in new system composition scenarios.

3. A CASE-STUDY: THE ACC SYSTEM

In order to illustrate our approach, we use the Advanced Cruise Controller (ACC) system. Such system extends the basic cruise control functionality of keeping a constant speed of the vehicle with the following features:

- Automatically adjusting the vehicle's speed to keep a constant distance with the car in front;
- Adjusting the vehicle's speed to the speed limit provided by the traffic signs;
- Providing emergency brake assistance to avoid collisions.

Following the component-based approach, the ACC system is developed as an assembly of components complying to a specific component model. Let us consider that an ACC system contains: a *Speed Limit* component to compute the

maximal authorized speed with regards to road signs and user's desired speed, an *Object Recognition* component that evaluates the distance to an obstacle in function of the vehicle's current speed, a *Brake Controller* to detect whether slowing the vehicle down is required, a *Controller* in charge of computing the necessary acceleration or braking forces that must be applied and *Logger HMI Output* to keep a trace of the events.

3.1 The ProCom Component Model

We have previously developed a domain-specific component model [15], called ProCom, especially designed to support the specific development concerns of distributed embedded real-time systems. The ACC example is a typical illustration of such category of systems. One of the main objectives behind ProCom is to facilitate analysis of the extra-functional properties typically found in embedded systems such as resource usage, timing and dependability — in particular for early analysis. In ProCom, components are rich design units that aggregate functional interfaces, modeling, analysis and implementation artifacts.

To comply with ProCom, the ACC system introduced above can be modelled as a ProCom composite component as illustrated in Figure 1. This type of component follows a pipes-and-filters architectural style that separates data and control flows. Data input and output ports are denoted by small rectangles whereas trigger ports are denoted by triangles. For more information about ProCom, refer to [3]. PRIDE[1] tool supports development of systems using ProCom component models. We take the hypothesis that both designers and developers use this tool to develop their system.

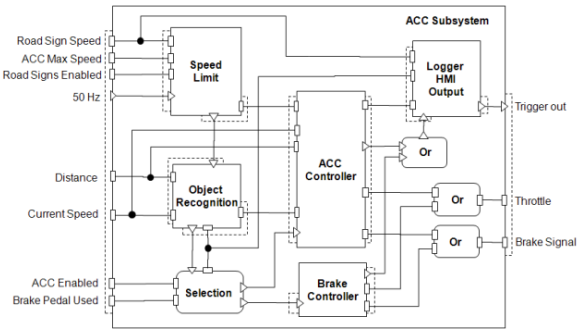

Figure 1: ProCom Architectural Description of the ACC

3.2 The Attribute Framework

As stated before, our objective is to provide a way to evaluate whether a system evolution preserves the validity of extra-functional properties or not. However, ProCom does not per se manage extra-functional properties specification. This is done through the *attribute framework* [14] that is part of the PRIDE tool, which enables annotating any ProCom element (from coarse grain such as composite component to fine grain such as a data port or a connection) with multiple context-aware values of quality attributes. The attribute semantics is formally defined through several properties including the types of element that can be annotated and the data type for the value.

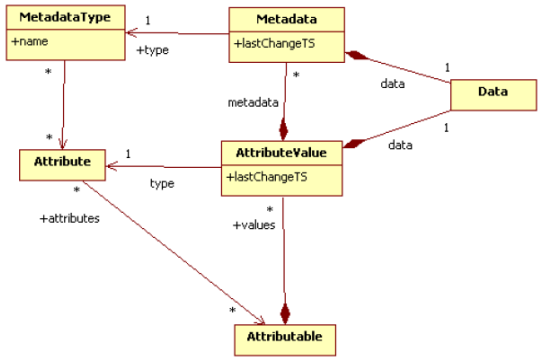

Figure 2: Attribute Framework Metamodel

As depicted in the metamodel in Figure 2, each attribute can have multiple values depending on the considered context (e.g., a specific target platform). Each value can be considered as a possible alternative (variant). Different values of a same attribute constitute versions of each other. While it is interesting to have several alternative values, we must be able to distinguish them. Without meta information, analyses would not able to select the right value to be used. Therefore values are annotated with typed *Metadata*. The semantic of the metadata types is formally defined in a similar way as it is for the attribute types. This includes the attribute to which this metadata can be associated with and the data type for the metadata value. This means that, as for attributes, metadata can have different formats.

During the development, the system developer is responsible for ensuring extra-functional properties such as worst case execution time (WCET), which can be computed by analysis. For example, when the implementation of the Speed Limit component changes, the developer performs a source code analysis to compute a new WCET estimation for this component; let us suppose it to be greater than the previous value. The developer wants to know if this change has an impact on the overall system properties. This requires to track the system evolution focusing on the differences between the old system (used to compute the system properties) and the new one. In the ACC example, the WCET of the Speed Limit component is used to compute the overall WCET of the system. Hence, this evolution may have a significant impact on the system properties. The impact analysis needs to be aware of this relationship to compute the overall WCET validity. This means that the values used for this computation must be known. In general, the developer does not have the expertise on how extra-functional properties are calculated. This implies, for example, that s/he may not know how to derive the impact of a change as for the WCET of the Speed Limit component. The WCET analysis expert is the only capable to determine the dependencies between attributes values, and possibly the impact of a change on a overall system property.

In this context, our approach aims at formally specifying the dependencies for such type attributes. As a consequence, a context is attached to every attribute value. Figure 3

[1]The approach presented in this paper has been integrated in PRIDE (ProCom IDE) available from: `http://www.idt.mdh.se/pride`

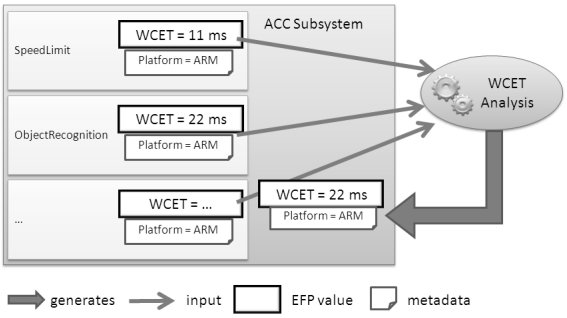

Figure 3: WCET Analysis example

presents the WCET analysis. It computes the WCET of a composite component from the WCETs of its subcomponents for a specified targeted platform. In the example, the WCET attribute value of the ACC Subsystem is derived from the WCET values of Speed Limit, Object recognition, ACC Controller, Selection, Logger HMI Output and Brake Controller components.

4. PROPOSED APPROACH

In order to be able to reuse components in the embedded real-time systems world, the developer should follow an approach which allows to predict, early in the development process, the impact of a component on extra-functional system properties. Therefore a component should store its information in a manner general enough to make it reusable in other different contexts, while at the same time providing as detailed information as possible in each specific case. One way to accomplish this is to use parametric attributes, i.e., attributes specified as a function over context parameters rather than as a single value. The context may include targeted platform, precision, information about how the value has been computed. Attribute values become complex formulae with many context parameters. Unfortunately, computation of such formulae is complex thus preventing their usage in common practice [4]. As a consequence, generally attribute values are valid for a specific context and they cannot be reused in a different context without precise information about it.

As clarified above, a context is characterized by a set of attributes related to system components, each of them pertaining to a particular aspect of the system, such as timing, fault tolerance, behavior, power consumption, and so forth. By following the MDE vision, which encourages to use a specific language for each concern, attribute values would define the system's characteristics using different models expressed in different languages. However, it is worth noting that such an approach would demand consistency across the different models to be preserved; consequently, we provide a corresponding solution to keep the coherence across the different artifacts involved in the process.

The proposed approach aims at allowing safe, fast and automatic property value validity evaluation. We take the hypothesis that multiple values are available for extra-functional properties. The main goal is to provide checking of attribute values validity when the system evolves and the followings are the evolution scenarios we have identified:

- [Scenario 1] The targeted platform changes;
- [Scenario 2] An estimated value is validated by measurements;
- [Scenario 3] The architecture of the system changes. It may include addition, removal of components, modification of their interface, connection changes;
- [Scenario 4] The component implementation is modified. For example, corresponding source code is improved;
- [Scenario 5] The functional specification of a component is modified;
- [Scenario 6] The extra-functional specification changes.

The approach workflow is depicted in Figure 4: our solution gives the possibility to model and store the context of each extra-functional property, while the evolution of the system is tracked by means of effects to the context. In particular, two or more context models representing the embedded system in different points in time, are given as input to the differences calculation engine which outputs the detected manipulations to an appropriate difference representation approach. Starting from this result, the context evolution management technique is able to check the validity of existing property values and in case of invalid values, to produce possible impacts *estimations*. To summarize, the

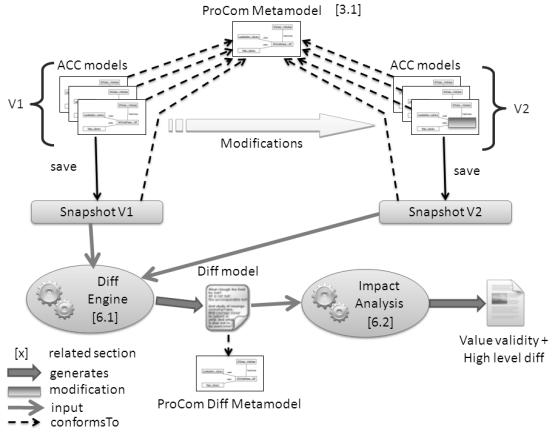

Figure 4: Impact Analysis Workflow

different steps to be performed to evaluate derived property validity are:

- [Step 1] The attribute value evolution is tracked and the attribute modifications are recorded;
- [Step 2] The attribute value context is automatically defined during analysis time. It includes the values taken into account during the related value computation;
- [Step 3] The system model which has been used to perform the considered attribute value computation is retrieved from the repository or from a saved snapshot (from the local history);
- [Step 4] The differences between the old system model and the new one are computed;
- [Step 5] The validity conditions are evaluated. An error report is associated to the attribute value in case of no more valid value.

The prerequisites for our approach are:

- [Requirement 1] The attribute value evolution must be tracked;

- [Requirement 2] The attribute value computation context should be defined;
- [Requirement 3] System models are versioned in a local and a shared repository;
- [Requirement 4] The attribute validity conditions must be defined as precise as possible to be able to compute accurate impact of changes;
- [Requirement 5] A specific algorithm for differences calculation must be defined for complex attributes.

With regards to the different evolution scenarios we aim at supporting, the following changes occurs in the attribute values (see Figure 2). When an existing value comes from a new target platform (1), the platform metadata of this value changes. When an attribute value moves from estimation source to measurement source (2), the source metadata changes. If the value is different then a new *AttributeValue* is added with the corresponding metadata. *InterfaceContent, compositeContent* and *primitiveContent* predefined attributes represent respectively the service and port definition of the component, the composite component architectural model and the component implementation source code. An architectural modification (3) generates a more recent *lastChangeTS* for the *compositeContent* attribute value. An implementation modification (4) is represented as a more recent *lastChangeTS* for the *primitiveContent* attribute value. A functional specification modification (5) leads to a more recent *lastChangeTS* for the *interfaceContent* attribute value. Finally, any extra-functional specification modification is represented as value data change.

5. EXTENDING THE ATTRIBUTE FRAMEWORK WITH CONTEXT MODELING

5.1 Computation Context

To provide support for the aforementioned approach, the attribute framework metamodel needs to be modified to enable context modeling and context evaluation. Figure 5 presents the changes performed in the attribute framework metamodel.

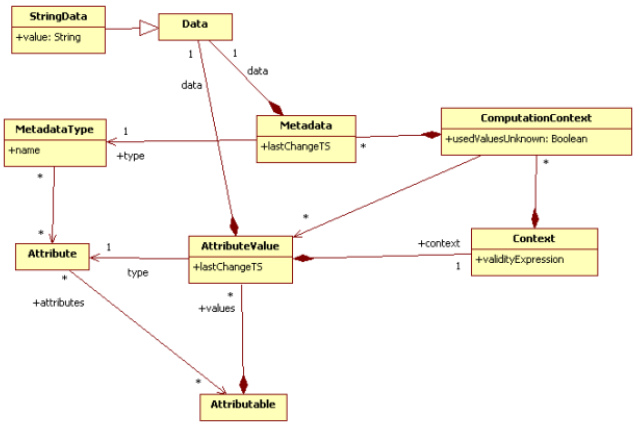

Figure 5: Context Metamodel

First, each model element is identified by a global unique identifier to facilitate model element evolution tracking. Then, since the attribute framework should be able to track the evolution of each attribute value (Requirement 1), it is necessary that every modification to a value is done through the attribute framework and tagged with a last modification time-stamp *lastChangeTS*. In case of attributes whose values refer to files, the value is considered to be changed when the related file is modified or when the value references a new file.

As a value may have been computed from different analyses and/or can be valid for multiple contexts, we represent the different computations by means of the *ComputationContext* concept. When possible, the computation context defines the attribute values used to compute this value (Requirement 2).

As mentioned before, this definition is usually not decided by the user. We propose to automate the computation of used values when analysis is performed. For this purpose, we provide an *analysis framework* which allows to integrate new analyses to PRIDE. As attributes, analysis are partially modeled (Figure 6). Their definition includes the types on which they can be performed and a function computing the result validity conditions. Thanks to this integration,

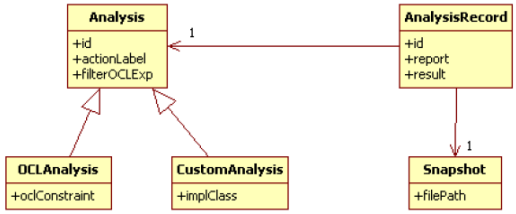

Figure 6: Analysis Metamodel

PRIDE is able to trace any read and written attribute value during the analysis. A computation context is automatically created for each new attribute value which defines all the corresponding read values as they are used. Each performed analysis is recorded and a snapshot of the current ProCom model is attached to it (Requirement 3). Hence, the context of each derived attribute value coming from analysis is automatically computed. However, the context of user defined attribute values must be manually specified. A *ComputationContext* is created by default for user defined values and it may specify, through the *usedValuesUnknown* property, that the used values are unknown.

5.2 Computation Context Language: Expressing the Validity Conditions

Validity conditions are attached to the computation context (Requirement 4). They formally define when the corresponding value is valid. In general validity conditions considering only the current system version can not be defined in a simple way. That is why we consider validity conditions which define what the modifications that invalidate a value are. They are expressed as an Epsilon Validation Language (EVL) [11] expression, which is similar to Object Constraint Language (OCL) [13]. Our objective is to have enough information within the context to be able to evaluate whether a value is still valid or not after an evolution. We identified the following condition categories:

- Value validity: Derived value validity depends on no data changes of used values.

- Metadata validity: Derived value metadata can depend on used value metadata.

Looking at our example, ACC Subsystem WCET is computed from the other subcomponents' WCET (e.g. Speed Limit's WCET), so for instance a modification of Speed Limit changes the ACC Subsystem WCET. The computed WCET is still valid only if a subcomponent WCET has been reduced. In this case, the overall WCET should be recomputed but it can be considered still valid since it represents an upper bound. Figure 7 shows how to define this validity condition. As in OCL, EVL constraints are related

```
constraint ValueIsStillValid {

  check : (self.usedValuesUnknown) or ((not self.usedValuesUnknown)
      and self.satisfies("UsedValuesHaveNotChanged"))

  message : 'ComputationContext ' + self.id + ' is no more valid'
}

constraint UsedValuesHaveNotChanged {

  guard : not self.usedValuesUnknown

  check : self.usedValues.forAll(v|not v.hasIncreased())

  message : 'ComputationContext ' + self.id +
      ' is no more valid due to an used value changed'
}
```

Figure 7: WCET Value validity Conditions

to a context. The analysis framework allows the analysis provider to define the validity conditions of the automatically created computation context and its related metadata. The element on which the analysis provider specifies validity conditions defines the EVL constraint context. A library which provides operations to consult the differences between the computation time and the current moment is automatically imported. For example, *hasIncreased* operation returns true if the corresponding attribute integer value has been increased. A default value validity condition is defined by PRIDE and it specifies that a value becomes invalid when one of the used values changes. It allows to benefit from the validity evaluation without defining validity conditions. It is convenient when no analysis expert is available. Even if the ACC Subsystem WCET computation is the same, the platform metadata may be no longer correct if a sub-component's WCET has been measured for a different platform. Figure 8 presents the expression used to specify this validity condition. We can observe that meta-

```
constraint IsStillValid {

  guard : self.getComputationContext().
      satisfies("ValueIsStillValid")

  check : self.getComputationContext().forAll(value|
          not value.metadataHasChanged(self, "plateform"))

  message : 'Platform metadata is no more valid due to ' +
      'a subcomponent platform metadata change'
}
```

Figure 8: WCET Platform validity Conditions

data validity is part of the value validity. Thus, the Platform validity condition *IsStillValid* is checked only if the WCET value validity conditions have been checked. It is *self.getComputationContext().satisfies("ValueIsStillValid")* statement that defines this dependency. The metadata are as important as the value itself. They define precisely in which context the value can be used, that is to say when it is valid. As for values, a default validity condition is defined. It specifies that a metadata becomes invalid when any metadata of used value is modified.

In specific cases of newly invalid value, it is possible to resolve it by a fast computation. If one of the sub-components' WCET moves from 10 to 20 ms, the overall WCET is lower than or equal to the previous value with an addition of 10 ms. Since the static WCET analysis is heavily time and resource consuming, when the developer reuses an existing component without the corresponding WCET analysis tool, it could be useful to use this approximative WCET. We reuse the EVL quick fix feature to provide such functionality. Fix statements can be added to the value validity condition to quickly resolve invalid values. Figure 9 shows fix statement to add to the *UsedValuesHaveNotChanged* constraint in order to provide an invalid WCET resolution.

```
fix {
    title : 'Increase WCET as the sucomponent wcet ' +
        'have been increased'
    do {
        var valueToUpdate := self.getAttributeValue();
        for (usedValue in self.usedValues) {
            var diff := usedValue.getNew().getIntValue()
                - usedValue.getOld().getIntValue();
            if (diff > 0)
                valueToUpdate.addInt(diff);
        }
    }
}
```

Figure 9: WCET Value validity fix

6. MODIFICATION IMPACT ANALYSIS

In this section we describe our approach to the evolution management together with the evaluation of quality attributes value validity, computed by analysis, when the information used to perform it evolves. The computation of the evolution of a context model is calculated in terms of model differencing between an initial context model and its modified version (Step 4 in the approach presented in Section 4). The first step of calculating the differences is followed by their representation which gives a structured result of the calculation to be reused for further validity evaluation of the context evolution.

6.1 Difference Computation

As previously mentioned, each system's property demands to precisely define the impact a particular class of changes entails to the system. This issue is crucial when considering evolution activities, since changes operated on the current version of the application should preserve (or possibly improve) its characteristics. In order to define such evolution impact, a differences calculation task has to be performed; this is carried out by applying a set of rules defined using the Epsilon Comparison Language (ECL) [11] to the two models (i.e. initial and modified). These rules aim at identifying matching elements between the two models by comparing the elements unique identifiers; correctness by construction is ensured by the use of such identifiers, which also remarkably simplify the matching task. Using identi-

fiers allows to correctly find the two corresponding elements between initial and modified version of the model and explore their properties to detect possible changes. Once two matching elements (e.g. *ComputationContext* in Figure 10) are identified, further matching conditions are defined within the ECL rules body in terms of Epsilon Object Language (EOL) expressions and applied to the properties of interest (e.g. *validityExpression* in Figure 10) of the matching elements in order to catch possible changes. In the case of a changed property, its name and the corresponding element identifier are stored in an apposite structure called *matchTrace* and provided by ECL in order to be reused for representation issues (Figure 10). Modified elements are not

```
rule ComputationContext
    match l:left!ComputationContext
    with r:right!ComputationContext{
        compare : l.id = r.id
        do{
            if(not (l.usedValuesUnknown =
                        r.usedValuesUnknown)){
                var mod : String;
                mod = 'usedValuesUnkwown';
                matchInfo.put(l.id,mod);
            }
            if(not l.validityExpression.matches
                        (r.validityExpression)){
                var mod : String;
                mod = 'validityExpression';
                matchInfo.put(l.id,mod);
            }
        }
}
```

Figure 10: ECL rule for ComputationContext elements comparison

the only differences we want to catch; deleted and added elements have also to be identified in order to produce a complete view of the evolution which involved the initial context model. In other words, in order to automatically generate the difference model starting from the computation carried out by our defined set of ECL rules, the following two tasks are needed: (1) generation of changed/original element pairs which represent the modifications to the initial model elements and (2) generation of elements added in the modified version of the model and elements deleted from the initial version. Once the differences have been computed, they need to be stored for further analysis and re-use. For this purpose we use difference models which conform to a difference metamodel automatically derived from the initial ECORE context metamodel (see Figure 5) through the ATL transformation described in [5]. The transformation takes the input metamodel and enriches it with the constructs able to express the modifications that are performed on the initial version of a given model in order to obtain the modified version: additions, deletions and changes. These constructs are defined in terms of metaclasses which specialize the corresponding original metaclass (e.g. *AddedComputationContext*, *DeletedComputationContext*, *ChangedComputationContext* for the metaclass *Context*). For instance, an added or deleted ComputationContext is represented by respectively creating an AddedComputationContext or DeletedComputationContext while, in order to represent a changed context, the creation of a ChangedComputationContext element is not enough. In fact a ComputationContext element has also to be created for representing the ComputationContext element in the initial model which has been affected by changes in the modified model. As previously mentioned, the differences representa-

```
//Merge Changed ComputationContext Element
rule MergeComputationContext
    merge s : left!ComputationContext
    with t : right!ComputationContext
    into m : diffModel!ComputationContext{
    m.id := t.id;
    m.usedValuesUnknown := t.usedValuesUnknown;
    m.validityExpression := t.validityExpression;
    diffModel!DifferenceModel.allInstances().first()
                        .differenceElements.add(m);
    //creation of the original element
    if(s.isChanged()){
        var p = new diffModel!ChangedComputationContext;
        p.id := s.id;
        p.usedValuesUnknown := s.usedValuesUnknown;
        p.validityExpression := s.validityExpression;
        //assign the changed element to the original
        p.updatedElement := m;
        //manage the composition
        diffModel!DifferenceModel.allInstances().first()
                        .differenceElements.add(p);
    }
}
```

Figure 11: EML rule for ComputationContext

tion results in a difference model conforming to the difference metamodel. This model represents the computed differences in terms of added, deleted and changed elements. These elements are generated by using two different languages provided by Epsilon which are, respectively, Epsilon Merging Language (EML) for the creation of changed elements and Epsilon Transformation Language (ETL) for the generation of added and deleted elements. Defining EML rules allows to transparently take the ECL result structure from the differences computation and use it for the generation of changed/original element pairs (Figure 11). If the

```
//Added ComputationContext Element
rule AddComputationContext
    transform s : right!ComputationContext
    to t : diffModel!AddedComputationContext {
    guard : not tempAdd.includes(s)
    t.id := s.id;
    t.usedValuesUnknown := s.usedValuesUnknown;
    t.validityExpression := s.validityExpression;
    diffModel!DifferenceModel.allInstances().first()
                    .differenceElements.add(t);
}
//Removed ComputationContext Element
rule DelComputationContext
    transform s : left!ComputationContext
    to t : diffModel!DeletedComputationContext {
    guard : not tempRem.includes(s)
    t.id := s.id;
    t.usedValuesUnknown := s.usedValuesUnknown;
    t.validityExpression := s.validityExpression;
    diffModel!DifferenceModel.allInstances().first()
                    .differenceElements.add(t);
}
```

Figure 12: ETL rules for ComputationContext

ECL result structure contains information about a modification which affected the element, the changed element is created and linked to the original within the rule's body by

using EOL expressions. Once the first task is completed, a set of ETL rules are in charge of creating added and deleted elements (Figure 12); after the completion of the process, the difference model (Figure 13) will show the results in a structured manner, ready to be used for the validity evaluation. Among the others, a changed/original pair of ComputationContext is highlighted in the center of the picture. In

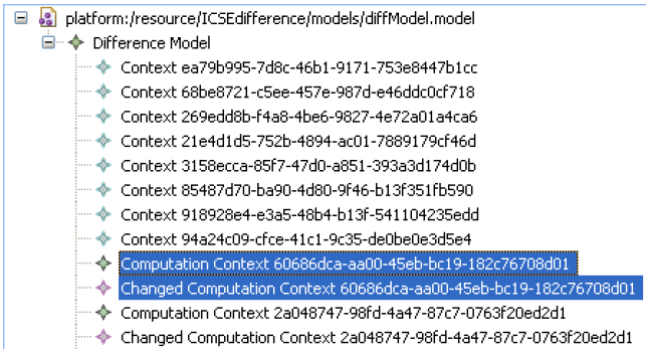

Figure 13: An excerpt of the difference model related to the ACC example

Figure 14, we can see the properties of the original ComputationContext element and its changed version respectively on the top and bottom of the figure. The elements have the same unique identifier since they represent the evolution of the same object, while the different value of the property *UsedValuesUnknown* represents the change which made the original element to evolve. Moreover, the link between original and changed elements is maintained by the property *updatedElement* in the ChangedComputationContext element.

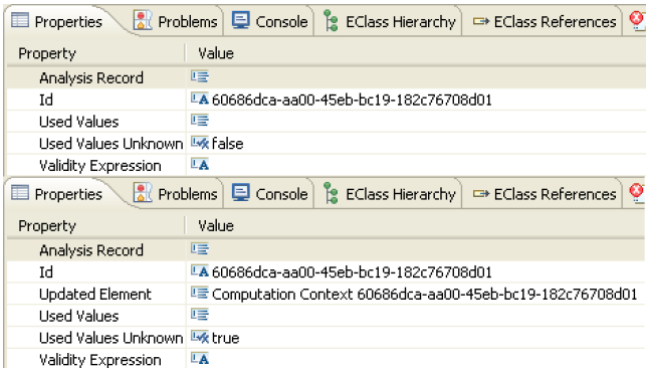

Figure 14: Evolution of the ComputationContext properties

6.2 Validity Evaluation

The objective of the proposed approach is to evaluate the validity of components' extra-functional properties consequently to a context evolution. Figure 15 shows PRIDE with its component explorer (a), architectural editor (b), attribute view with all the defined attributes (d), parametric WCET value editor (c) and analysis record view (e) with all the performed analysis and respective results.

The analysis menu provides all the analysis related actions which can be performed on selected elements; analysis report and results are available on the created analysis record. The validity evaluation is implemented as a specific analysis that can be applied on any element (Step 5 in the approach presented in Section 4). Firstly, it generates a snapshot of the current ProCom models; in order to lighten the size of such snapshot, no attribute values representing files are included. Moreover, a further optimization consists in avoiding the inclusion of components which are not related to the values to be validated. For instance, an enclosing component does not have influence on subcomponents extra-functional properties while they are usually not independent from the enclosing composite component.

Then, for each computation context of considered attribute values, we launch the difference computation between the snapshot stored with the computation context and the current snapshot. We transform and merge the related EVL expressions to obtain an EVL file to be applied to both the difference model and the two snapshots. The transformation adds context to the EVL expression and adds an instance filter to apply the constraints only to the expected elements. Eventually, results are collected and merged to compute the analysis report. It is worth noting that the underlying mechanism to deal with difference detection and encoding is completely transparent to the PRIDE user (i.e. the system developer); as aforementioned, new quality attributes would require corresponding value specifications, evolution forms, and validity impacts. In turn, such addition would entail a refinement of the current detection/representation mechanism that, once realized, would be provided through the PRIDE tool as a new analysis.

In Figure 16, a validity evaluation which has resulted in an error is presented. PRIDE uses the fix defined in the validity conditions and allows the developer to apply them. In the example, the WCET value can be fixed by invoking "*Increase WCET as the sub-component's WCET have been increased*" action. As previously claimed, the validity evaluation analysis cost is negligible. During our experiments, the analysis execution time has never been greater than 3 seconds to check validity of three extra-functional properties for the ACC Subsystem which is constituted of more than 400 model elements for the functional definition and more than 800 model elements for the extra-functional properties. In comparison, measurement based WCET analysis needs to produce system code, compile it, deploy it and monitor its execution; even disregarding time costs issues, these tasks may also require a certain degree of expertise.

7. APPROACH VALIDATION

On a dual core CPU with 2,79 GHz and a memory space of 3,48 GB of RAM, depending on the complexity of a composite component (number of components and parallel control paths), parametric WCET computation can range from 2 minutes to 30 minutes. This analysis has been used to compare time needed for our validity checking analysis and WCET computation on the ACC example (see Figure 1). The experiments consisted of performing the whole WCET computation and comparing it to our validity checking analysis. This comparison has been performed for each evolution scenario testing two cases: the first one invalidates the previous WCET value, while the second one preserves the validity of the related value. The local repository has been used for all these experiments, thus avoiding delays due to checkout time to retrieve old versions. Table 1 shows the

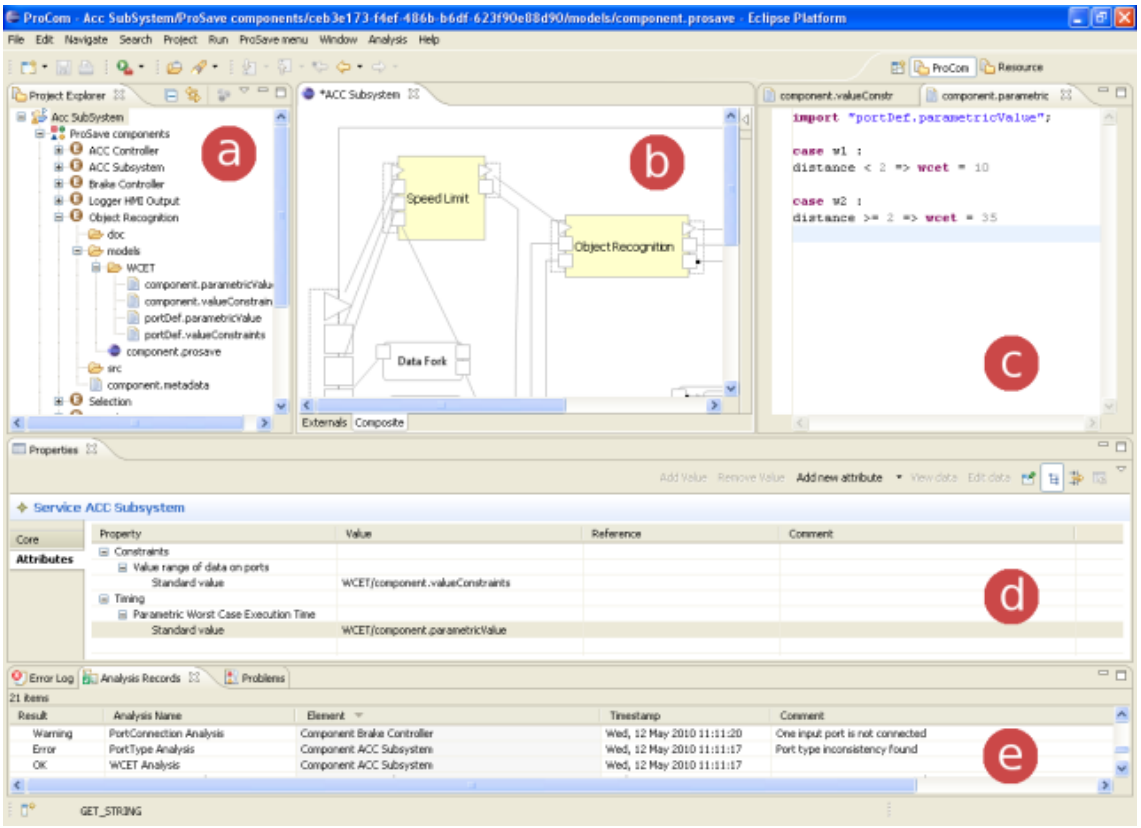

Figure 15: Attribute Management in PRIDE

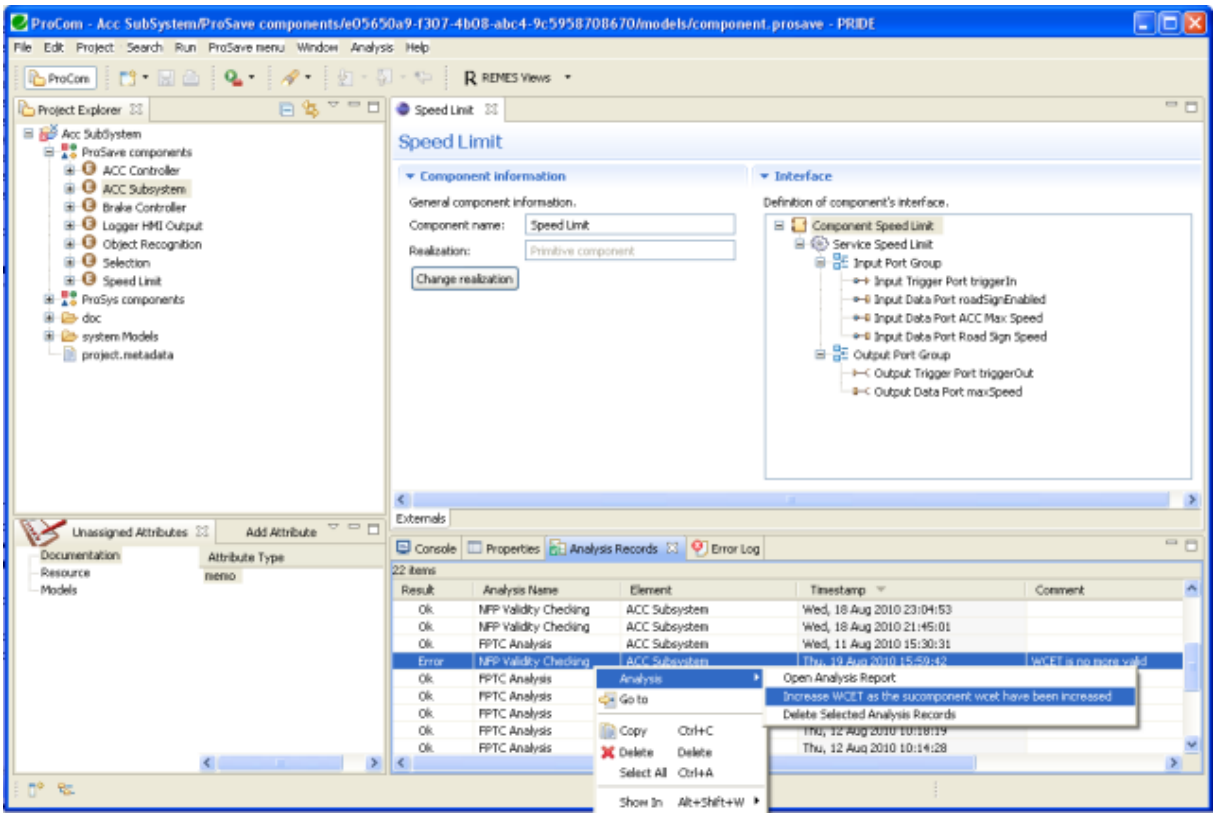

Figure 16: Validity Management in PRIDE

tested cases, the WCET computation average time, and the WCET value validity analysis computation time for each of the defined scenarios. First of all, the language used to define

Scenario	Checked Cases	WCET	Value Validity
1	Valid and Invalid	13,62 min	1,73 s
2	Valid	13,78 min	1,87 s
3	Valid and Invalid	13,06 min	2,56 s
4	Invalid	16,63 min	2,12 s
5	Invalid	15,36 min	2,45 s
6	Valid and Invalid	14,55 min	1,62 s

Table 1: ACC System Analysis Computation Time

validity conditions is expressive enough to allow detection of cases where value validity is preserved and where not. If the platform is preserved, the value validity checking is very fast (about 5s), especially if compared to the WCET computation (13 minutes). The precision of the impact estimates is limited by information precision defined by attribute and analysis provider including validity expressions and complex attribute differentiation algorithms.

It is worth noting that, even if quality attributes not always demand for such heavy analysis to be performed, they usually need external tools and complex environment setups. By adopting the approach illustrated in this work, such steps can be avoided whenever a value is checked and confirmed as still valid after an evolution of the considered system.

8. CONCLUSIONS

Management of extra-functional requirements in embedded systems development presents intrinsic difficulties due to distribution and componentization factors. This paper proposes a possible solution to alleviate the issues arising when managing extra-functional properties in evolving scenarios by exploiting the interplay of MDE and CBSE techniques. In particular, we aim at anticipating the impact analysis of the changes operated on the application as early as possible, that is by detecting modifications at the modeling level and providing corresponding validation responses.

Since we applied generic MDE methods, the proposed approach can be considered as generic and reused for other component models. Moreover, additional validation constraints and corresponding analysis can be *plugged-in* the development environment given the abstraction level at which the computational context and its evolutions are represented.

We are currently investigating the possibility to extend our solution with change propagation features. As noticed in [8], complex systems usually rely on relationships and dependencies between their main sub-systems; therefore, impact analysis should also foresee the side-effects caused by the propagation of changes to interrelated artifacts. Consequently, further improvements of our solution should include the estimation of new values (or ranges of values), able to re-establish a valid condition whenever it is violated.

Acknowledgments

This research work was partially supported by the Swedish Foundation for Strategic Research via the strategic research center PROGRESS and the CHESS project (ARTEMIS JU grant nr. 216682).

9. REFERENCES

[1] R. S. Arnold. *Software Change Impact Analysis*. IEEE CS, 1996.

[2] L. Berardinelli, V. Cortellessa, and A. Di Marco. Performance modeling and analysis of context-aware mobile software systems. In *FASE*, volume 6013 of *LNCS*, pages 353–367. Springer Berlin, 2010.

[3] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. Vulgarakis. ProCom Reference Manual, v1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[4] S. Bygde, A. Ermedahl, and B. Lisper. An Efficient Algorithm for Parametric WCET Calculation. In *Proceedings of RTCSA 2009*, pages 13–21, 2009.

[5] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *JOT*, 6:165–185, 2007.

[6] 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 *LNCS*, pages 257–278. Springer Berlin, 2005.

[7] T. Feng and J. I. Maletic. Applying dynamic change impact analysis in component-based architecture design. In *Proceedings of SNPD*, pages 43–48, 2006.

[8] J. Han. Supporting impact analysis and change propagation in software engineering environments. *Int'l Workshop in Software Technology and Engineering Practice*, page 172, 1997.

[9] D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for software architectures. In *Software Architecture*, LNCS, pages 113–126. Springer Berlin, 2006.

[10] 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 ICSE*, 2009.

[11] D. Kolovos, L. Rose, and R. Paige. *The epsilon Book*. http://www.eclipse.org/gmt/epsilon/doc/book/, 2010.

[12] S. Looman. Impact analysis of changes in functional requirements in the behavioral view of software architectures, August 2009.

[13] OMG. OCL 2.0 Specification, 2006. OMG Document formal/2006-05-01.

[14] S. Sentilles, P. Štěpán, J. Carlson, and I. Crnković. Integration of Extra-Functional Properties in Component Models. In *Proceedings of CBSE*, 2009.

[15] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković. A Component Model for Control-Intensive Distributed Embedded Systems. In *Proceedings of CBSE*, pages 310–317. Springer Berlin, 2008.

[16] C. Tibermacine, R. Fleurquin, and S. Sadou. On-demand quality-oriented assistance in component-based software evolution. In *Component-Based Software Engineering*, volume 4063 of *LNCS*, pages 294–309. Springer, 2006.

[17] M. Törngren, D. Chen, and I. Crnkovic. Component based vs. model based development: A comparison in the context of vehicular embedded systems. In *SEAA'05*. IEEE.

[18] J. Zhao, H. Yang, L. Xiang, and B. Xu. Change impact analysis to support architectural evolution. *Journal of Software Maintenance*, 14(5):317–333, 2002.