Towards an Impact Analysis for Component Based Real-Time Product Line Architectures

Anders Wall
Mälardalen University
Mälardalen Real-Time research Centre
Dept. of Computer Engineering
awl@mdh.se

Magnus Larsson
ABB Automation
Technology Products
mlo@mdh.se

Christer Norström
Mälardalen University
Mälardalen Real-Time research Centre
Dept. of Computer Engineering
cen@mdh.se

Abstract

In this paper we propose a method for predicting the consequences of adding new components to an existing product line in the real-time systems domain. We refer to such a prediction as an impact analysis. New components are added as new features are introduced in the product line. Adding components to a real-time system may affect the temporal correctness of the system. In our approach to product line architectures, products are constructed by assembling components. By having a prediction enabled component technology as the underlying component technology, we can predict the behavior of an assembly of components. We demonstrate our approach by an example in which temporal correctness and consistency between versions of components is predicted.

1. Introduction

Applying the concept of product-line architectures (PLA), is one way to achieve component reuse and benefits from component-based development. A PLA from a software system's perspective is a common architecture, a set of common strategies, tools, and methods that are shared among several different products within a particular domain [1,2]. Thus, not only components are reused, but also the architecture and the design strategies that initially were chosen. Examples of such strategies are strategies for adding new features to an existing PLA and strategies for providing variability. A product line consists of different products that are distinguished by different features but they also share a set of common features. Typically, features realize a set of functional, and non-functional requirement (e.g. quality of services, temporal constraints, etc.). Variations in features may be obtained in different ways, e.g. applying variations in a flexible software architecture, parameterization of existing components, by using different implementations of components. In a PLA it is more likely that the software architecture is a constant, while flexibility is achieved through component variations. New functional, or non-functional requirements will be implemented by adding new components or by using different variants of existing components.

The flexibility is not only specified in the functional domain. Also non-functional properties may be subject for variability. For instance, in the real-time systems domain we are interested in the temporal behavior of a system as it is considered correct only if it performs correct function at correct time, i.e. *temporal correctness*. Consequently, by adding the temporal domain we must not only manage functional flexibility but also temporal flexibility. For instance, the frequency with which a particular component executes may vary between a high-end product and a low-end product due different demands from the controlled process.

One of the main problems in constructing and maintaining a PLA is to express and verify product properties derived from the properties of the individual components. To be able to predict the product properties from the component properties, we define a *prediction-enabled component technology* (PECT) similar to the one proposed in [3]. In a PECT there are both a *constructive model* and an *analytical model*. Examples of such analytical models on a component are different temporal attributes such as the frequencies with which a component executes and the version dependency among components. While a constructive model deals with operational (functional), properties, analytical model describes non-functional properties. From the predictability point of view, obtaining new functional features of the products is straightforward as they come directly from the functional properties of components. On the opposite, the non-functional properties of products are hard to predict. For example, adding components with new functional features may degrade the quality of services of a product and, consequently, affect the temporal correctness. Moreover, a product line strategy can be focused on product families with the same functional properties, but different non-functional properties, e.g., scalability, flexibility, and safety. For this reason, the ability to derive non-functional product properties from the properties of the components plays a significant role for PLA. Furthermore, as developing software products according to the PLA approach is based on reuse and repeatable processes, the findings and measurements from previously developed product versions can be taken as input to the method proposed in this paper which may give more accurate predictions.

In this paper we present a component concept that provides means for performing an *impact analysis*. The aim of impact analysis is to predict the consequences of altering a system, i.e. adding new functionality or changing existing components. This is especially important in a PLA perspective where components and architectures are reused and customized for different products. The analysis is based on the concept of PECT, which is integrated into our component model developed for use in real-time product line architectures. We demonstrate the analytical models by an example showing how they can be used to derive properties of an assembly and analyze the impact of, e.g. adding new features to a product. However, the intention of this work is to provide a framework in which analytical properties can be added to the model such that any interesting property of an assembly can be expressed and analyzed. In particular we illustrate our approach by presenting how two different non-functional properties, *temporal correctness*, and *version consistent*, can be analyzed.

The remainder of the paper is outlined as follows: Section 2 gives an introduction to components and assemblies of components[3,4]. Section 3 elaborates on the different properties of assemblies. Section 4 discusses the concept of impact analysis, and finally Section 5 concludes the paper.

2. Components and Assemblies

In order to enable analyze properties of component-based products, we must have means for specifying analytical properties of components and identify synchronization and communication between them. Different component models specify this to different extent. Most of them do not treat non-functional properties. Our component model is based on the port-based object approach in which components are connected to each other by data ports that constitutes a components *data interface* [5]. This component model extends the expressiveness of port-based objects and is presented in a simplified manner hereinafter. For a more detailed description we refer to [6].

In Figure 1, our component meta-model is depicted in UML-fashion. Components have in and out ports which resembles the data interface. Also, a component encapsulates services, which provide the actual functional behavior. Besides having data interfaces, defined by their ports, components in the framework have two additional interfaces, *control interface*, and *parameterization interface*. The execution of, and synchronization among components is controlled through its control interface by associating a task to the interface. A task provides a thread of execution that is defined and restricted by a set of attributes, e.g. priority, frequency. A task in our framework can be based on any task model defined by the used real-time operating system (RTOS). A task is a runtime mechanism and hence, it is a constructive part of a component. However, note that some of the attributes of a task are required when, together with some analytical properties, analyzing temporal properties of an assembly. The parameterization interface defines the points of variation of a component's behavior.

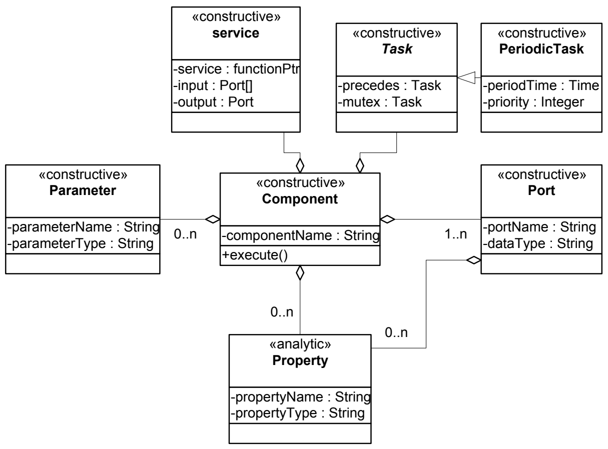

Figure 1. The component model

The property class that is stereotyped as analytic provides the information needed by the different analyses we are interested in performing on an assembly. We will refer to such a property as an *analytical property*. An analytical component property usually does not have a correspondence in a component instance. A typical example of such a property would be the execution time of a service of a component. The execution time is derived from the source code, or by measurements, for the purpose of modeling and analysis of a system and has no correspondence as such in the runtime. The *analytical model* of a component is defined by its analytical properties.

For further discussions we need definitions of certain terms in our component model. In this model we shall emphasize the real-time properties. Formally we define the constructive part of the component model depicted in Figure 1 as:

Definition 1. A component c is a tuple $\langle f, P, I, O, C, s_c \rangle$, where f is the service encapsulated by c, P is the set of parameters, I is the set of in-ports, O is the set of out-ports, C is the control interface and s_c is the state of component c. □

A component's state is updated by the service within a component and remains in between consecutive executions of a component.

An assembly is a specific configuration of a set of components that also defines the components interconnections. The union of all its component's states gives the state of an assembly. Formally we define an assembly as:

Definition 2. An *assembly A* is a tuple $\langle C(A), R^*\rangle$, where $C(A) \subseteq C$ is the set of components in A, and R^* is the set of relations valid between $C(A)$ in A, and C is a set of all components encapsulated in the product □

Note that an assembly does not necessary corresponds to a product. While in some cases we are interested in properties of the product, in some cases we may want to analyze properties of a sub-part of the complete product. In both cases we will refer to an assembly. An assembly is only a conceptual- and analytical view of a complete product that exists for the analysis of a particular property, and has not necessarily a constructive correspondence.

In order to construct an assembly, we must be able to connect components with each other via some relation. In our definition of an assembly we have three kinds or relations among components that belongs to the set R, *precedence*, *mutual exclusion* (mutex), and *data-flow connections*.

Precedence and mutual exclusion specify the synchronization among tasks that controls the execution of components. Formally we define precedence and mutual exclusion as:

Definition 3. A precedence relation, $\rightarrow$, is a binary, transitive relation among a pair of tasks $\langle \tau_{i,}\ \tau_j\rangle \in T \times T$, such that, if $\tau_i \rightarrow \tau_j$, then τ_j may start its *n:th* execution earliest at the end of τ_i's *n:th* execution where $i \neq j$ and n is the number of invocations of $\tau_{i,}$ and τ_j. □

Definition 4. A mutual exclusion relation, $\otimes$, is a binary, symmetric relation among pair of tasks $\langle \tau_{i,}\ \tau_j\rangle \in T \times T$, such that if $\tau_i \otimes \tau_j$, then neither τ_i nor τ_j is permitted to execute while the corresponding party, or a transitively related party is executing and $i \neq j$. □

Besides synchronization, we can also specify data-flow relations among components in an assembly. Data-flow connections specify the data that are exchanged between components in an assembly through their ports. We define the data-flow relation as:

Definition 5. A data flow connection =, is a binary, anti-symmetric relation among pair of ports on components, $\langle c_i.i_x\ c_j.o_y\rangle \in C.I \times C.O$, such that if $c_i.i_x = c_j.o_y$ then c_i's in port i_x is connected to c_j's out port o_x. □

3. Properties of an Assembly

The intention of our work is to provide a framework in which new properties of an assembly could be taken into consideration and predicted for the purpose of analyzing the impact that the introduction of a new component in the system have. The general idea is that if the model has to be extended with a new predictable property, new analytic properties can be defined and new *property theories* be developed. The property theory defines how a particular property of an assembly is calculated, e.g. theories for verifying the temporal correctness. For instance, if we require an assembly to be type correct, i.e. the types of connected data ports are correct, we must add a method for checking this property and doing so require an analytical property on data ports which carries the type information. Furthermore, we are using the prediction technologies in a product line perspective, i.e. we will discuss properties that are important when developing and maintaining product line architectures.

There are several realistic scenarios describing activities that a product line may undergo during its lifetime. We have not identified all possible scenarios but highlighting some relevant cases and propose examples of properties that are interesting from their perspective.

Scenario 1: New features will eventually be added to a product line or a specific product within the product line. This new feature might be implemented by a set of new components as well as new versions of old components already existing as part of the reusable assets in the product line. Doing this, there is a potential risk that components could end up being incompatible with components already used in the product, both with respect to version and variants. This scenario is also related to maintenance of a product that may alter the characteristics of a particular component. This change of characteristics is possibly acceptable for one particular product, but what are the consequences in the rest of the product line?

Scenario 2: As we operate in the real-time systems domain, we are also interested in predicting the temporal behavior of an assembly. Adding component to-, or changing components in a product or product line, may violate the temporal constraints in the system. The reason for violating the temporal constraints could be an over-utilization of the available resources in the system architecture. A big share of existing real-time systems are embedded systems, thus resources are usually limited.

Scenario 3: When an assembly of components is composed it is of importance to be able to predict if all component interactions are type correct. In a port based component model the components read the outputs from other components at the start of the execution. If the output type is not the same as the type of the input then we have a fault which can lead to a failure of the system.

Hence we want to predict if an assembly is type correct before deploying it.

The scenarios discussed above also apply to the assembly of a new product, based on pre-existing reusable components. We have to make sure that the product is feasible both with respect to the functional behavior and the temporal behavior.

We will refer to the analysis of relevant properties of assemblies in a product line prospective as *impact analysis*. Thus, we want to analyze the impact of a change, e.g. installing new features in a product, maintaining existing components, construct a completely new product based on reusable assets within the product line.

To illustrate predictability of assemblies for the specified component model, we shall discuss two concrete examples of assembly's properties from a real-time product line's point of view: *consistent*, and *end-to-end deadlines*. These properties are of completely different nature. *Consistent* is typically a property of a complete product. End-to-end deadline only concerns a subset of components in a complete product assembly. Moreover, there can be several end-to-end deadline requirements within the same assembly with respect to a subset of components from the full assembly.

3.1 The end-to-end temporal property

The second example of properties is related to temporal constraints. The temporal correctness is of vital importance in the real-time systems domain. Moreover, the temporal requirements on a real-time system are seldom presented in terms of the temporal attributes provided by the RTOS or as simple deadlines for individual components. Typically they are considered on a higher level; for instance jitter constraints for the control performance, end-to-end deadlines, response times, etc. Designing a real-time system is partly a matter of transforming such high-level temporal requirements to the attributes available in the task model at run-time, typically considering priorities and frequencies. In our approach the high-level temporal requirements are specified as properties on an assembly, e.g. end-to-end deadline, and the implementation of those requirements, e.g. frequencies, priorities, execution times, are specified as analytical properties on components.

A concrete example of a temporal property is *end-to-end deadline*. An end-to-end deadline, *A.e2e*, specifies a temporal requirement on a set of components. It defines the maximum distance between an input stimuli and the output response given. Typically, the end-to-end property requirements in hard real-time systems must be met, while in soft real-time systems a particular confidence of meeting the requirement may be sufficient. Statistical verification of a prediction theory can be performed to show how reliable the prediction actually is, e.g. the confidence in the estimated worst-case execution time.

Verifying that a temporal property of the assembly is feasible, we verify that our temporal implementation is correct. However, this verification is correct under the assumption that all prerequisites are correct (For example, the execution time of a component, which is a component property). Consequently, the correctness of a property of an assembly depends on the confidence we have in analytical properties. The concept of credentials as presented in [7] includes a notion of confidence associated with a component property. The execution time can be statically analyzed given the source code, or empirically measured in runtime [8]. Empirical validation of the prediction theory is also needed to prove the soundness of the theory.

Figure 2 shows an example where four components have been instantiated from the model presented in Figure 1. The infrastructure in which those components will execute (the RTOS) has a scheduling policy based on fixed priorities. The task model consequently specifies the level of priority and the frequency of each task. When defining an assembly we also must specify how the assembly is build. There are not only the properties of the components that determine the properties of an assembly, but also the assembly architecture; we must define how the assembly is built. For example, in a pipe-filter architecture the dataflow between components (i.e. the precedence relations) must be specified. In this example we define the precedence property and ports connections. We also add an analytical property that specifies how many times components are supposed to be executed.

Component c_1 has two preconditions, the first one express the precedence relation and the second the connection of ports.

The figure shows four components where c_1 reads the out ports of c_0 and c_2, c_3 reads the out ports of c_1. c_0 precedes c_1 and c_1 precedes c_2, while c_3 can execute independently (i.e. $c_0 \rightarrow c_1$ and $c_1 \rightarrow c_2$). Below is the components described according to definition 1:

$$\begin{aligned}
c_0 &= \langle f, P_0, \emptyset, \{o_1\}, f(\emptyset, \{o_1\}), \tau_0, s_0\rangle \\
c_1 &= \langle g, P_1, \{i_1\}, \{o_2, o_3\}, g(\{i_1\}, \{o_2, o_3\}), \tau_1, s_1\rangle \qquad (1) \\
c_2 &= \langle h, P_2, \{i_2\}, \{o_4\}, h(\{i_2\}, \{o_4\}), \tau_2, s_2\rangle \\
c_3 &= \langle x, P_3, \{i_3\}, \{o_5\}, x(\{i_3\}, \{o_5\}), \tau_3, s_3\rangle
\end{aligned}$$

There are many views of one assembly depending on the relations of components. In our example we have two views, one is for precedence of components and another that shows how the components are connected through ports. The assembly in our example according to definition 2 is:

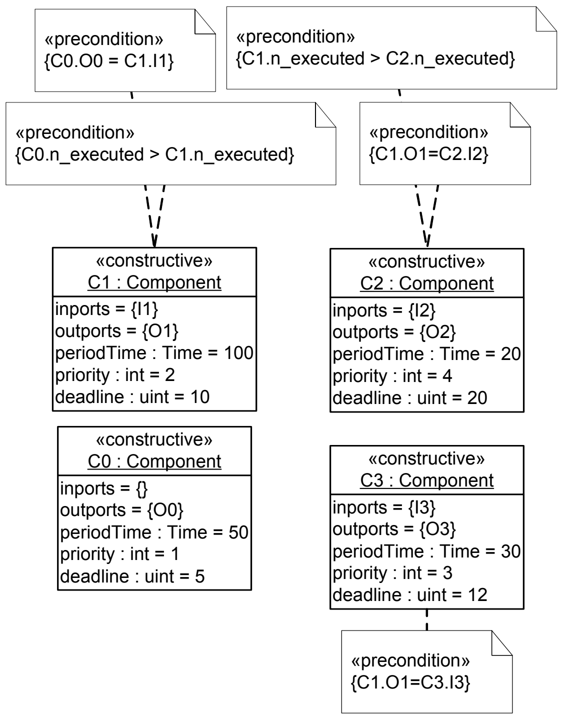

Figure 2. Four components with precedence and connection relations specified using constraints

$$A = \langle \{c_0, c_1, c_2, c_3\}, \{R_{precedence} = \{c_0 \to c_1, c_1 \to c_2\}, \{R_{Connection} = \{(o_1, i_1), (o_1, i_2), (o_2, i_3)\}\}\rangle. \quad (2)$$

One view of the assembly is the one

$$A_{Precedence} = \langle \{c_0, c_1, c_2, c_3\}, R_{Precedence} \rangle. \quad (3)$$

The other view is

$$A_{Connection} = \langle \{c_0, c_1, c_2, c_3\}, R_{Connection} \rangle. \quad (4)$$

We shell analyzed a high-level requirement of the assembly, namely end-to-end deadline, *A.e2e*.

An end-to-end deadline constraint can be defined as a property on the assembly A.e2e which can be calculated as

$$A.e2e = \mathrm{Max}(\mathrm{ResponseTime}(c_2), \mathrm{ResponseTime}(c_3)) - \mathrm{EarliestStartTime}(c_0). \quad (5)$$

An end-to-end deadline is consequently constraining the maximum time interval between start of the first component in an assembly and the finish of the last component in the assembly. Formally we define A.e2e in a general expression as:

Definition 6. An end-2-end property of assembly *A*, *A.e2e*, is $A.e2e = Max(R(c_i)) - Min(StartTime(c_j))$,where $c_i, c_j \in C(A)$, $Max(R(c_i))$ is the maximum response time of c_i, and $Min(StartTime(c_j))$ in the minimum earliest start time of c_j. □

Calculating the response time of components based on the attributes provided in a fixed-priority based RTOS is done with *response time analysis* [9]. However, different methods must be utilized if a different scheduling policy is provided by the RTOS, e.g. earliest-deadline-first. Thus, the definition of a particular property may vary due to mechanisms provided by the infrastructure in which the system will execute.

In our particular example we are using fixed priority scheduling in which we calculate the response time of component c_i, $R(c_i)$, as:

$$R^{n+1}(c_i) = c_i.wcet + B(c_i) + \sum_{\forall c_j \in hp(c_i)} \left\lceil \frac{R^n(c_i)}{c_j.T} \right\rceil c_j.wcet \quad (6)$$

,where B is the blocking time, $hp(c_i)$, is the set of components having tasks with higher priority than component i, and c_j.wcet is the worst-case execution time of component c_i.

The earliest-start time can also be calculated with equation 6 by assuming that all components execute as fast as possible, i.e. with their *best-case execution time* (bcet). Furthermore, the start time will be approximately equal to the response time if we assume an execution time equal to zero of the component whose earliest start time is subject for the analysis.

The end-to-end property is a typical example of a property that may be defined on only part of a complete product. In

Figure 2 it can be seen that c_0, c_1 and c_2 are connected with the precedence relation but c_3 can execute anytime when in the ready queue. It is of importance to be able to calculate the e2e property for c_0, c_1 and c_2 only. Our proposal is that the property shall be defined for parts of the assembly with respect to a relation. In our example we can say that c_3 is independent from the other components with respect to precedence. Hence A.e2e over $\{c_0, c_1, c_2\}$ can be calculated with the response time of c_2. By having this notation it is possible to define properties that reflects parts of the assembly.

As discussed above, different task models will affect the set of analytical properties on components and how temporal properties of assemblies are calculated. Equation 6 shows how to calculate the response time for a system with periodic tasks and static priorities. However, if systems are event based and uses the earliest-deadline first scheduling algorithm new theories for verifying the temporal behavior are required. Thus, components, assemblies and the execution model affect the property theory. Hence, each of these has to be defined before we start reason about temporal properties of assemblies.

3.2 The version consistency property

In a product line approach the handling of consistency is a 2-dimensional problem. A component in a product line may be compatible with- or dependent of several different variants of other components. For instance, A GUI component for an embedded system could differ between products in a product line, e.g. high-end products with a color display and low-end products with monochrome displays. The color display and the monochrome displays are variants of the same feature, i.e. the feature of presenting information graphically to a user of the system. In turn, there can exist several versions of every variant of a component. Typically new versions emerge from error corrections and from new functionality being added.

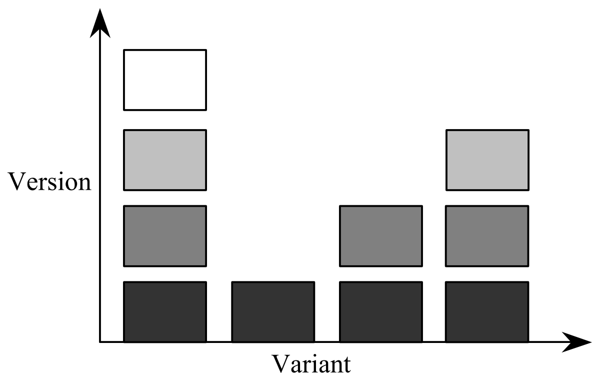

Figure 3. The 2-dimensional version-variant concept

A version of a component can be defined by having an analytic property on the component. Also dependencies between components are express through such a property. In our model we allow a component to depend on several different variants of a component but only one distinct version of each variant.

The consistent property, *A.consistent*, is related to a capability to predict consistency of an assembly. An assembly is considered consistent if the versions of each component are correct according to the specification of a product in the product line. The specified features of a product determine which components, and in particular which components version should be included in a product. To be able to guarantee consistency we need to specify what versions of components a product depends on.

This idea of having version dependencies is very similar to how .NET assemblies use meta-data to describe dependencies to other assemblies [10]. Dependencies can be expressed and assured using OCL constraints for the components. A new constraint has been added to all components that state how the dependencies shall be evaluated and regarded analyzing the assembly.

For the purpose of predicting variant- and version consistence on an assembly, we must introduce the analytical property depends on a component, *c.depends*. The property *c.depends* is a set containing all components and their variant and version, which component *c* consistently can be assembled with. A tuple <C, variant, version> identifies a variantand version of a component.

In many component models multiple versions of the same component may not coexist. In those cases there is a risk that components are assembled in an inconsistent way, by means of having the assembly include two or more different versions of the very same component. It is desired to prevent such invalid assemblies by being able to predict whether an assembly is consistent or not.

The consistency of all variants and versions in an assembly can be calculated with the following formula. The property consistent is of type boolean.

Definition 7. An assembly A is variant- and version consistent, *A.consistent* if:

$$A.consistent = \forall \langle\langle c_i, variant, x\rangle, \langle c_i, variant, y\rangle\rangle \in V \times V: x=y$$

,where $V = \bigcup_{c_i \in C(A)} c_i.depends$, $c_i \in C(A)$, *variant* is a component variant and *x,y* are versions. □

That is, the assembly is consistent if a component does not appear twice with different version in the union set of all dependencies.

4. Impact Analysis

Before the new component is added we want to predict the impact it has to the system. For instance we want to calculate *A.consistent* and *A.e2e* over $\{c_0, c_1, c_2\}$ and $\{c_3, c_4\}$. We refer to such an analysis as *impact analysis*.

The e2e property, or any other temporal property of an assembly, may be affected by adding new components to a product. Assume, for instance, a fixed-priority scheduled system. The majority of the commercial available RTOS belong to this class. Moreover, assume that priorities are assigned to tasks according to the deadline-monotonic algorithm, i.e. the task with the shortest deadline is assigned the highest priority. Adding a component that has a unique deadline in such a system may require the rest of the system to undergo a new priority assignment, unless it has the latest deadline. Consequently, it is important to formalize the algorithm or strategy used for priority assignment as a property of an assembly. If such formalization does not exist, evolution and maintenance of the system may become expensive. Note that adding a component with lower priority than all existing components is no guarantee for a temporal correct system. Such a component can still affect the temporal correctness through, e.g. shared resources resulting in priority inversion.

In order to predict the need for reassigning priorities in a fixed-priority system we introduce a boolean property on an assembly that the pre-existing priority assignment still will be valid after adding a new component, *A.priority*. The theory for this property varies according to the strategy for assigning priorities; just as the theory A.e2e varies depending on the scheduling policy.

Adding a component c_i will not affect the priority assignment according to deadline-monotonic if:

Definition 8. The priority assignment of an assembly *A* with a correct priority assignment, is still valid, A.priority when adding component c_i if:

$$A.priority = \exists c_j \forall c_k \in C(A): c_i.d = c_j.d \vee c_i.d < c_k.d$$

,where *c.d* is the deadline for the task that controls the execution of componet *c*. □

We illustrate the problem of adding a new component to a product line by continuing the example in Section 3.1. We introduce a new component c_4 which is dependent on the execution of c_3 and the output from c_3 and c_2. Such a component is presented in Figure 4. The component c_4 also express its version relation to other components. Component c_4 depends on a particular version of c_3. The dependencies are expressed using a precondition that asserts that the correct version of c_3 is in c_4's list depends on.

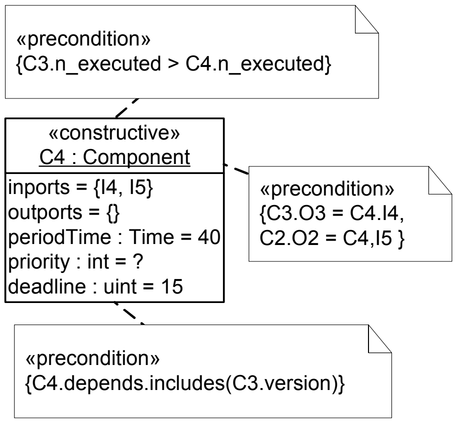

Figure 4. A new component c4 is added to represent a new feature of a product.

Applying the property theory for A.priority indicates that the priority assignment currently existing in the assembly must be revised as the new component has an unique deadline that is shorter than the deadline for component c_2. Note that the priority of a task only can be decided in relation to every other task in a system.

In a similar way we can define, and apply, any other important property theory in order to analyze the impact of adding a new component to a system.

5. Conclusion

In this paper we have proposed the use of a prediction-enabled component technology for developing and maintaining component based product line architecture in the real-time system's domain. We have extended an existing component model with analytical models that specifies the properties needed for predicting the different properties of a component assembly. As examples of properties that are interesting from a real-time product line architecture's point of view, we define the end-to-end deadline property and the type consistent property.

We have used the concept of impact analysis. In the impact analysis the effect of introducing new components in a product line architecture is predicted. The new components could be due to the introduction of new features in the product line or maintenance of existing components that potentially alter the characteristics of a component.

The ideas are presented in the paper as concrete examples of two properties on assemblies. However, the presented methodology is supposed to be the base to a general framework in which new assembly properties could be included as the need for them emerges. As a consequence of introducing a new assembly property, new analytical properties on the components may be needed.

As future work we will develop the property theories presented in this paper further as well as the framework concept. As a base for this work we will implement the component model and provide a tool for specifying and analyzing systems based in the component model. Such a tool should support the framework ideas. Thus, it must provide means for extending the component's analytical model with new analytical properties and to define new property theories on assemblies.

Another application of the ideas presented in this paper is a principle for handling dynamically configurable systems. Consumer-products such as cellular phones may be configured/customized by the consumer himself, e.g. by downloading a new feature to the phone. Thus, the end-customer assembles products based on a product line architecture. By distributing the analytical model together with the constructive software, the system itself can predict the impact the new feature will have on the system. Based on such an analysis the system can decide whether to accept the new product as valid or not.

6. References

[1] Bosch J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.

[2] Clements P. and Northrop L., *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[3] Hissam, S. A., Moreno, G. A., Stafford, J., and Wallnau, K. C., Packaging Predictable Assembly with Prediction-Enabled Component Technology, report Technical report CMU/SEI-2001-TR-024 ESC-TR-2001-024, 2001.

[4] Wallnau K. C. and Stafford J., "Ensembles: Abstractions for A New Class of Design Problem", In *Proceedings of 27th Euromicro Conference*, 2001.

[5] Stewart D.B., Volpe R.A., and Khosla P.K., Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects, *IEEE Transaction on Software Engineering*, volume 23, issue 12, 1997.

[6] Wall A. and Norström C., "A Component Model for Embedded Real-Time Software product-Lines", In *Proceedings of 4th IFAC conference on Fieldbus Systems and their Applications*, 2001.

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

[8] Lim S.S., Bae Y. H., Jang C. T., Rhee B. D., Min S. L., Park C. Y., Shin H., Park K., and Ki C. S., An Accurate Worst-Case Timing Analysis for RISK Processors, *IEEE Transaction on Software Engineering*, volume 21, issue 7, 1995.

[9] Audsley N.C., Burns A., Richardson M. F., Tindell K., and Wellings A. J., Applying New Scheduling Theory to Static Priority Preemptive Scheduling, *Software Engineering Journal*, volume 5, issue 8, 1993.

[10] Thai T. and Lam H., *.NET Framework*, O´Reilly, 2001.