Product Line Architectures for Embedded Real-Time Systems

Anders Wall, Kristian Sandström and Christer Norström
Department of Computer Engineering
Mälardalen University
2000-12-18

`Technical report`

Abstract

In this paper, we propose the use of product line architectures in order to shorten time to market, cut cost, and to reduce maintenance when developing embedded real-time systems. A development process that supports such an approach, taking the temporal domain into account is also outlined. Moreover, a case study was performed in cooperation with the automotive industry where the product-line architecture approach has been successfully applied. Although the architectural language used was sufficient for their specific product-line, considerations about future development indicate that the description language must support more flexible constructions. We suggest several mechanisms for modeling functional, as well as temporal flexibility in the architectural description.

1 Introduction

Today the trend in computer-based products, such as cars and mobile phones, is shorter and shorter lifecycles. As a consequence, time spent on development of new products or new versions of a product must be reduced. One solution to this emerging problem is to *reuse* code and architectural solutions within a product family. Besides shortening development time, properly handled reuse will also improve the reliability since code is executed for longer time and in different contexts [15]. However, reuse is not trivial when applying it to real-time systems since both functional behavior as well as the temporal behavior must be considered.

In this paper we describe how to employ the concept of product line architectures in order to shorten the development time, cut cost, and reduce maintenance for a specific class of systems, embedded real-time systems [1][2][3]. The use of the product line concept in the context of control systems for construction equipment vehicles has been studied. Since several different models of each family of construction equipment vehicles exist, architectural solutions and components may potentially be reused among different models

We define a *product line* as a set of software products that share a common technology platform as well as having common functionality. A software architecture that constitutes the base on which all products in the line are built, is called a *product line architecture* (PLA). For each product a *product architecture* (PA) is derived from the PLA upon *product instantiation*. When instantiating a product, the PLA is tailored via different techniques such as parameterization, and by populating the architecture with components. The PLA approach has been used for soft real-time systems [4], where the timeliness is of less, or no, importance compared to real-time systems in general. However, in systems where the temporal correctness is of vital importance for the reliability of the product, the temporal requirements and the temporal behavior must be included in the PLA. These temporal requirements must not be violated when instances of products are created.

We propose a design process suitable for developing product lines for real-time systems. The process starts in a requirement-capturing phase where the requirements from all products in the line are collected. Communalities in functional- and temporal requirements among the products will be considered when the actual PLA is designed. The PLA is then analyzed. The objective of analyzing the PLA is to gain confidence in that the PLA is flexible enough to be a base on which all products can be realized without violating any temporal constraints.

To enable the use of a PLA and derivation of product architectures from the PLA we need a design language. Such a language is often called *architecture description language* (ADL) [3][5]. The ADL should have a precise syntax and semantics to enable architectural analysis, including analysis of, for instance, performance, maintainability, flexibility, and temporal properties. Moreover, the ADL must facilitate constructions for modeling of flexible components. The flexibility mechanisms specified on components constitutes the variation points that are used when product architectures are derived from the PLA, i.e. when the PLA is tailored for a particular product

David Stewart et al has addressed the area of reusable components for embedded systems and how to create a framework for building systems based on components [13][14]. They have introduced a component model that provides flexibility in component behavior, hence reusability, through parameterization. In this paper, component reuse is accomplished by using additional techniques, not only through parameterization.

The contributions of this paper with respect to embedded real-time systems are:

- An outline of a development process, focusing on the special considerations that must be taken into account when designing a PLA for embedded real-time systems
- An industrial case study of the use of a PLA for construction equipment.
- Constructions that must be supported by an ADL in order to be suitable for describing product line architectures for embedded real-time systems.

The rest the paper is organized as follows. Section 2 discusses the development of a PLA for embedded systems. In Section 3 we present a case study from the automotive industry where the PLA approach has been used. Furthermore, in Section 4 we discuss mechanisms providing specification of flexible architectures. Finally we conclude the paper in Section 5.

2 Product line based development

In this section we discuss the development process in which a PLA for embedded real-time products is constructed. The process is iterative and includes architectural analysis of properties that are of vital importance for a PLA, e.g. flexibility. Moreover, the derivation of products from a PLA is dealt with. The design process proposed in this paper is shown in Figure 1 where the process is divided into *requirements capturing*, *PLA development*, and *product development*. The proposed design process will be further described in this section.

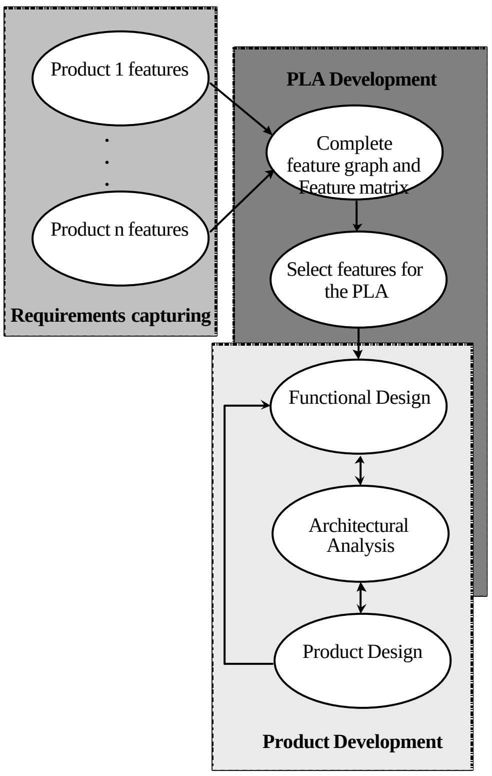

Figure 1 The process of developing a product line based on a PLA.

2.1 Developing a product line architecture

Developing a PLA is done either in an *evolutionary* or *revolutionary* way [2]. The evolutionary approach to PLA design is conducted by a generalization of existing products, whereas in the revolutionary approach, the common architecture is developed, rather than extracted from existing implementations. Independent of whether the evolutionary or the revolutionary approach is taken, the first step when developing a PLA is to capture the requirements for every product in the product line. One way to do this is to organize and group required functionality into *features*. We consider features to be functional entities seen from a stakeholder's perspective. For instance, the function for adding items to an address book in an ordinary cellular phone can be a feature. A feature typically groups a set of requirements and therefore it also simplifies the requirements handling. At this point, we must consider all features in the product line. The products and their features make up a matrix, the *feature matrix* for the product line. The feature matrix shown in Table 1 has *m* features distributed over *n* products. A position in the matrix marked with X indicates that the feature in that column is present in the product of that line.

	Feature 1	Feature 2	Feature 3	Feature m
Product 1	X	X		X
Product 2		X	X	X
Product 3			X	
Product n		X		X

Table 1 A feature matrix

Features are rarely independent from each other. A feature can for instance depend on other features in order to deliver the desired functionality. Another example of a relation between features is the mutually exclusive relation, implying that only one of the related features can appear in the final product. If mutually exclusive features must co-exist in the product, effort has to be made to resolve the conflict. Features in real-time systems will also exhibit dependencies related to the temporal domain. For instance, consider the lock-free break feature and the anti-slid feature in automotive vehicles. Both features need information about the wheels' velocity, thereby having a shared temporal dependency related to the freshness of the sensor data. Such relations on features are specified in a *feature graph*. In Figure 2, a feature graph with three features and two relations is depicted.

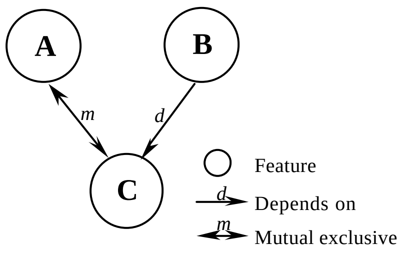

Figure 2 A feature graph

The feature matrix and the feature graph constitute the basis for deciding what features to include in the PLA. Typically, features that are common among a majority of the products in the product line are included. Consequently, the PLA may provide features that are superfluous for a specific product. Identifying the commonality among the features for real-time systems is more complicated since we also have to take the temporal domain into consideration.

The features that are not part of the PLA are considered product specific. However, the PLA must be flexible enough to incorporate product specific features upon product instantiation. Note that a PLA for real-time systems must provide sufficient flexibility in the temporal domain as well. The features that have been chosen to be part of the PLA are documented in a PLA feature graph. The PLA feature graph is an aid for the product instantiation process, discussed in Section 2.3, as well as in evolutionary design of the products and for maintenance of the PLA itself.

When the scope for the PLA has been decided, the features should be mapped to components that, together with their interrelations, constitute the actual architecture. This part of the development process is referred to as *functional design*.

When performing functional design of a PLA, the designer must take into consideration that features may have different implementations for different products. Depending on how implementations differ, the correct mechanisms for obtaining the desired flexibility must be selected. In Section 4, some mechanisms for flexible architectures are discussed. For real-time systems we also have to consider temporal behavior. A typical example of how the temporal requirements influence the functional design is the following. Consider features that are functionally equivalent between a set of products. If these products will run on different infrastructures, i.e. operating system and hardware platform, the functionality may be partitioned among components in different ways to fulfill the timing requirements. In the high-end product we can partition a feature in such a way that it will be easy to maintain while in a low-end product we have to make an architecture that is focused on performance to be able to fulfill the timing requirements.

After the functional design the PLA must be analyzed in order to secure that the architecture is flexible enough to facilitate all products in the product line. If not sufficiently flexible the architecture must be transformed. Thus, iterations between analysis and functional design are required. Since our focus is on real-time systems we would like to gain confidence in that the architecture is sufficiently flexible to be used in all products in the product line without violating the temporal constraints.

2.2 Product line architecture analysis

Architectural properties that typically can be analyzed are performance, reliability, temporal, maintainability, reusability, and testability [6]. In this section we will focus on early analysis of temporal properties of an architecture. Discovering that the real-time requirements cannot be fulfilled in a late stage of a product development often implies a delay in the release of the product on the market. Furthermore, such problems are often handled by ad-hoc optimizations, which in turn decreases maintainability and reusability. Positive experiences from making early analysis of temporal behavior using architectural descriptions and estimates of temporal properties have been reported in [7].

By making early analysis of the temporal behavior, for each product, based on the PLA and the product specific features, we can extract the following information.

- Traditional real-time measurements such as system utilization, response time for each feature, distribution of response times for a specific feature, and jitter information for features. However, this information is not only used for schedulability analysis, it is also used for analyzing the flexibility of the PLA with respect to implementation constraints. For example what will happen if a non-implemented component will use more resources than estimated in the architectural design.
- Robustness with respect to internal errors and erroneous assumptions about the environment. Typical analysis is based on “what if”-questions, examples include:
    - What will happen if a specific component slightly overruns its time budget?
    - What will happen if events from the environment are generated in a higher frequency than assumed?

To facilitate temporal analysis on the level of abstraction provided by the PLA, we must have information about the temporal behavior of the components that define the PLA. Information about the temporal behavior can in principle be attained in two different ways, namely by earlier implementations of the same or similar functions or by intelligent guesses of the temporal attributes for completely new components. Examples of temporal attributes are execution times of components and period times. The execution time for a component is normally measured on the implemented component while the period time for a component is derived from the requirements. However, if we do not know the execution time for a component an estimate has to be made. This estimate is later used as an additional implementation requirement for the component.

The closer we get to the architecture for the specific products, the more confidence we get in the predictions. Depending on the confidence in the temporal information, different types of analysis can be performed, such as:

- Simulation, which can be used for exploring the system behavior when the confidence for the different parameters are low or when the system will operate in dynamic environment where the load is hard to estimate [8].

- Mathematical analysis which can be used when the product line architecture or parts of it has estimates of execution times, temporal properties such as period time, and the synchronization between the components specified. Mathematical analysis that can be used includes fixed priority scheduling [9][10] and pre-run-time scheduling [11] theory. Especially different kinds of sensitivity analysis are of interest to predict the remaining capacity for product specific features [12].

Based on the result of the temporal analysis of the system different actions may be taken. If the analysis indicates that the requirements are fulfilled then the PLA is accepted. On the other hand, if the analysis indicates that the requirements cannot be fulfilled we have several options. The options are listed in the order of increasing cost, i.e., the required effort for applying them:

- Modify the derived temporal attributes, e.g., deadline and period time. There are many ways to derive the temporal attributes of the PLA from the requirements, which may give different analysis results.
- Transform the architecture. For example, an architecture that is designed for portability could experience a trade-off situation, where we have to decrease the portability to gain real-time performance.
- Renegotiate the requirements. If none of the options above are possible we have to renegotiate the requirements with the stakeholders.

So far the analysis provides a rough foundation for deciding whether the PLA can be used or not. The next step is to derive product architectures from the PLA and analyze each product architecture, in order to assure that all products fulfill their temporal requirements.

2.3 Product design based on a Product Line Architecture

Requirements for the PA have already been addressed during the construction of the PLA. However, at that point the focus was on the commonality between several products belonging to the product family. The process of developing a product focuses on product specific requirements and functionality not covered in the PLA, i.e., the PLA must be tailored to cover the requirements of the product at hand.

The first step is to merge the product specific features with the features provided by the PLA consistently, i.e. merging the product and the PLA feature graphs. In this process conflicts can occur, e.g. overlapping features, superfluous features, temporal discrepancies [2].

When the feature conflicts have been resolved, the concrete product architecture can be derived from the PLA. The derivation involves *architecture pruning*, i.e. removing superfluous parts of the architecture and extending the architecture with product specific components. Although architectural analysis has been performed at the PLA level, the resulting product architecture provides more details. Thus we can make a more detailed analysis of the architecture with respect to, for instance, the temporal correctness.

The remaining step until we can release the product is to implement the components that make up the architecture in such a way that the temporal requirements are fulfilled. This step also includes component testing, integration, and verification of the integration, which have been covered elsewhere when considering temporal estimates [7].

3 Case study: Volvo Construction Equipment

Volvo CE has built a distributed real-time system to control mainly the automatic gearbox, the brakes and the hydraulic systems in a product line of construction equipment. This section describes this existing product line architecture, together with some reflections.

3.1 Method and system description

There are five products in the product line and each product has five computer nodes. All the five products are different sized construction equipment vehicles.

The requirements for a new generation product line were brought to the development department. These requirements were evaluated to derive the wanted features and the temporal requirements. The construction of the PLA was in this case done strictly in an evolutionary way. There was products and application knowledge already at the scene. Because of this, the process of extracting the features was quite painless. The feature matrix did to some extent exist already in the minds of some of the experienced developers. Examples of features in this product line are the control of brake fluid pressure and automatic shifting of gears. The development process was essentially equivalent to the process proposed in Figure 1.

The mapping of features onto components, and the temporal attributes, is specified in configuration files. A configuration tool uses the configuration files in order to schedule the system and produce configuration data for the operating system. Moreover, the synchronization and communication between components are specified in these files. Thus, the configuration files constitutes the architectural description.

In order to implement components with a correct temporal behavior, time budgets were estimated. A time budget is a specified maximum time that a component can execute to perform its function. The fact that this was done before the actual code implementation could have been a problem, but based on experiences from earlier products the estimates turned out to be quite good.

Each node has separate configuration files and source code files. These files, however, are common between products in the product line. All products have identical hardware and software architecture and accordingly the configuration and source code files are identical for the products. Hence, the executable for each product is identical to its product sisters. This can be done since the nodes have identical computer hardware.

The unique behavior of each product is instead defined by *datasets*. A dataset is simply a set of parameters that control the program flow. The program then checks the parameter values and calculates output accordingly. There could, for instance, be a conditional invocation of a function depending on a parameter value.

The temporal properties of this PLA would require great effort to test if we had to test each product/node independent of the next. We would then have to test all the nodes in all the products and see that all temporal constraints are intact. By designing the system in a certain way, the tests could be simplified. The system was constructed by starting with the most advanced model. The most advanced model is assumed to conform a superset of the functions of all systems, i.e. its tasks should be the hardest to schedule. Successful testing of the most “advanced” model indicates that the simpler models also will perform correctly time wise. Note that this does not cover the testing of the systems functionality. Functionality was tested separately.

3.2 Turn out

One benefit from designing the system to run the same executable for different products is that the whole product line needs to be scheduled only once. This could, however, also be a drawback, if we have a wider span of functionality between low- and high-end products. The low-end product would have to include hardware capable of executing the high-end program, even if most of the functionality is disabled. Given enough separation between low- and high-end products, this could prove costly in terms of memory, I/O and CPU. This could also be too expensive in product lines where large amounts of units will be produced or even in small series products where system resources are scarce. For instance, mobile phones are large series products where it would prove to expensive to include even the least unnecessary hardware. For a PLA of quite similar products, these extra resources would probably be acceptable.

The PLA in this case is the set of five different construction equipment products. Ideally, one would want to include other products or product lines to be able to reuse components that have already been developed.

In this respect, the mentioned method might not be perfect. The method is inflexible when we want to add products that include more functionality than the most advanced product in the product line.

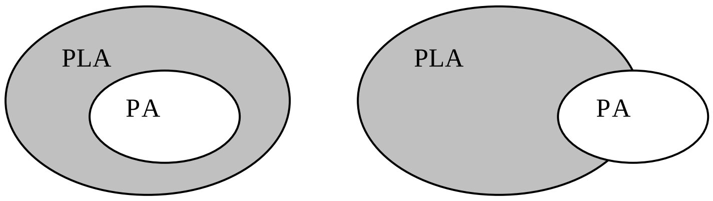

Figure 3 The relation between the functional sets of PLA and PA respectively

Our PLA of construction equipment can be illustrated by the leftmost figure in Figure 3. All product architectures are derived within the boundaries that the PLA define. That gives the advantages, discussed above, when testing the temporal behavior of the products. On the other hand we are somewhat limited to in developing new products. The PLA must be changed altogether if a product with more features is to be developed.

The ADL used in this project only allows parameter-based derivation of PA's. This requires a very homogenous product line. Instead, a design language with support for greater flexibility in expressing differences in different products is desirable.

4 Mechanisms providing Flexible Architectures

As discussed in Section 3.2, we need mechanisms, apart from parameterization, to obtain flexible architectures. Consequently, we need a design language that supports flexibility in component behavior, as well as flexibility in the systems architecture. We will refer to flexibility as optionability and variability. Optionability is the absence or presence of functionality, whereas variability is the possible variation in functionality. In real-time systems, there is also a need for variation of the temporal behavior, e.g. the period time of a control-loop might vary between products in the same line.

An architectural description language (ADL), for product line architectures should not only have language primitives necessary for describing the structure of a software system. Such a language must also restrict, and guide the process in which an actual product is instantiated, to make sure that the PLA is not corrupted during implementation. Consequently, the primitives that describe the functional- and temporal variations must have a well-defined semantics. In Section 4.1 we propose language primitives needed for an architectural description language in order to describe variability and optionability in architectures for real-time systems.

4.1 Language primitives for variability and optionability

In this section, language primitives supporting the specification of variability and optionability for product lines architectures for real-time system is discussed. The primitives that will be discussed are *components, interfaces,* and *tasks.*

A *component* is a computational unit having a data interface and a control interface. Moreover, several components can be composed to a composite component. The interface of a component should be a separate language primitive so a specific interface can be used for different components as long as each component implementing the interface conforms to it.

The *data interface* of a component represents the data-flow to- and from the component and the *control interface* represents the control-flow. The control flow is defined by associating one or more tasks to a component. A *task* provides the thread of execution of a component. Further, one task can control several components, which is beneficial when for example two components should be executed serially, and several tasks can control one composite component which includes several parallel sub-components. A task also defines the temporal behavior of a component and can be either periodic or aperiodic.

The temporal parameters in the control interface are of two kinds; the ones derived from the requirements such as deadline and release time of the task controlling the component, and the execution time of the components controlled by the task. However, the execution time of a component has to be specified for the component implementing the functionality and for a specific hardware platform. Such specification can either be based on real assessments or by intelligent guesses, as discussed earlier. In the latter case an intelligent guess becomes an implementation requirement of the component.

Furthermore, specification of synchronization can be achieved by using semaphores for mutual exclusion relationship between different components and signals can be used to trigger the execution of another component.

In the control interface of a composite component the temporal parameters that can be altered. As an example consider an interface for a multirate feedback controller component, composed by a sampler component, a controller component, and an actuator component. Such an interface can specify period times for each of the different sub-components as well as control jitter constraints.

When defining an interface for a component that is not yet implemented, and for which we suspect that the implementation can include several sub-components, we can specify a utilization bound for each platform it shall run on.

In Figure 4, χ_A and χ_B are components and ι is an interface for a component that, at least, must produce the output that χ_B requires. Two tasks, τ_A and τ_B, control the execution of the components. Task τ_A is periodic and executes χ_A and χ_B in a predefined order whereas τ_B is aperiodic and controls the execution of the component that implement the interface ι.

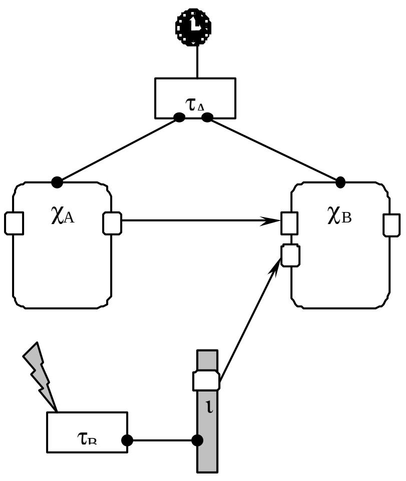

Figure 4 An example of our architecture description

4.2 Variability

Product-line architectures for real-time systems can be varied functionally and temporally. Variations in component behavior and the systems architecture itself constitute the functional variation, whereas the point of variation in the temporal behavior is tasks and their temporal attributes.

We have identified three possible techniques for obtaining variability on components. Basically, the techniques describe how interfaces and implementations alters a components behavior between products:

- The same interface, but different implementation
- Different interface and implementation
- The same interface and the implementation is varied by parameterization

In order to describe the techniques listed above, we use the language primitives discussed in Section 4.1, i.e. components and interfaces. To model a construction having the same interface but different implementations, we use the interface language primitive. Thus, in the architectural description only the interfaces are present. Each product instance of the architecture then has to implement the behavior by providing one or more components that comply with the given interface description.

If parts of the interface, as well as the implementation of the interface are different between products in the product line, we cannot describe this with the interface language primitive. For this purpose, we introduce a language primitive called *abstract component*. An abstract component indicates that the implementation of the component is tailored for each product. Note that, although the component is abstract, several products, but not all, can share the same implementation of it.

The last identified technique for obtaining variation in components is *parameterization*. Here the same implementation is used throughout the complete product line, but the components' behavior is controlled by parameters. Consequently, the implementation must take all possible behaviors into account when the component is designed. Different behaviors are not necessarily equivalent with different control-flow in the implementation; it can also be related to constants used in the calculations. For instance, a PID controller component could be provided with the proportional gain, the integration time, and the derivative time constants upon instantiation.

When the systems architecture is varied between product instances, the product-line architecture describes how the system is to be organized. As an example, consider a fire-alarm system where the number of smoke detectors varies depending on the size of the building in which the system resides. The smoke detectors in a product-line architecture for the fire-alarm system are described as a *multiple component*, i.e. the exact number of smoke detectors is decided upon instantiation of the product. In Figure 5, the sensor is a multiple component that represents *one*, up to *n* instances connected to the control component.

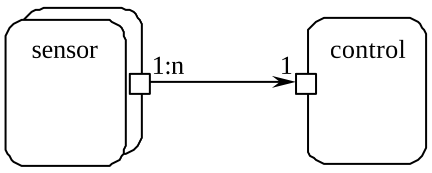

Figure 5 A Multiple component

As discussed in the ingress of this section, real-time systems can also be varied through their temporal behavior. The control interface of components provides this variability. By changing the tasks that control a component through the control interface, the temporal behavior of that component is varied. For instance, the execution strategy and temporal attributes such as the period time can be altered. If such changes are made to a task, the temporal correctness of the system must be re-verified.

4.3 Optionability

Optionability is concerned with the absence or the presence of functionality in a system. We define components to be the smallest software entity subject to optionability, i.e. we can only add or remove complete components. Components in our description language that, for some products in the line, can be removed have a property called optional. The removal of a component could result in partial use of an interface in the product-line architecture. As a consequence of adding or removing components, the temporal domain is affected. For instance, tasks are removed, added, etc.

5 Conclusion

In this paper, we propose the use of product line architectures in order to shorten time to market, cut cost, and to reduce maintenance when developing embedded real-time systems. A development process that supports such an approach, taking the temporal domain into account is also outlined. Moreover, a case study was done in the automotive industry where the product-line architecture approach has been successfully applied. Although the architectural language used was sufficient for their specific product-line, considerations about future development indicate that the description language must support more flexible constructions. We suggest several mechanisms for modeling functional, as well as temporal variability in the architectural description. Thus the architectural description language must define language primitives that model the mechanisms.

As future work we will formally define a component model that complies with the product-line architecture concept. Such a component model is a necessary prerequisite in order to construct a complete description language for product-lines. Moreover, the description language must facilitate the specification of architectural flexibility, temporal requirements, communication and synchronization with well-defined semantics. If such a language exists, we can develop automatic and semi-automatic methods for architectural analyses. Besides analyzing the temporal properties we are also interested in other quality attributes and especially how they relate to the temporal domain.

6 References

[1] D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson. Applying Software Product-Line Architecture. IEEE Computer, Volume 30, Nr 8. Aug., pp 49-55, 1996

[2] J. Bosch, Design and Use of Software Architectures, Addison-Wesley, ISBN 0-201-67494-7, 2000

[3] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, ISBN 0-201-19930-0, 1997

[4] J. Bosch, Product-Line Architectures in Industry: A Case Study, in Proceedings of the international conference on Software engineering, pp. 544 – 554, 1999

[5] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, ISBN 0-13-182957-2, 1997

[6] R. Kazman, G. Abowd, L. Bass, P. Clements, Scenario-Based Analysis of Software Architecture, IEEE Software 1996.

[7] Christer No rström, Kristian Sandström, Mikael Gustafsson, Jukka Mäki-Turja, and Nils-Erik Bånkestad. Findings from introducing state-of-the-art real-time techniques in vehicle industry. In industrial session of the 12th Euromicro Conference on Real-Time Systems, Stockholm, Sweden ,2000.

[8] Markus Lindgren, Hans Hansson, Christer Norström, and Sasikumar Punnekkat. Deriving Reliability Estimates of Distributed Real-Time Systems. In Proceedings of RTCSA2000 Cheju Island, South Korea , December 2000. IEEE Computer Society.

[9] N. C. Audsley, A. Burns, M.F. Richardson, K. Tindell, and A.J. Wellings. Applying New Scheduling Theory to Static Priority Preemptive Scheduling. Software Engineering Journal, 8(5):284{292, September 1993.

[10] A. Burns. Preemptive Priority Based Scheduling: An Appropriate Engineering Approach. Technical Report YCS 214, University of York, 1993.

[11] J. Xu and D. L. Parnas. Priority scheduling versus pre-run-time scheduling. Real-Time Systems Journal, 18(1), January 2000.

[12] Sasikumar Punnekkat, Rob Davis, and Alan Burns. Sensitivity Analysis of Real-Time Task Sets. Asian Computing Science Conference, Kahlmandu December 1997.

[13] David B. Stewart. Software Components for Real Time. Embedded Systems Programming VOL. 13 NO. 13 December, 2000

[14] Stewart, D.B., R.A. Volpe, and P.K. Khosla, Design of dynamically reconfigurable real-time software using port- based objects, IEEE Trans. on Software Engineering, v.23, n.12, Dec. 1997.

[15] N. E. Fenton, S. L. Pfleeger, Software Metrics: A rigorous & practical approach, International Thomson Computer Press, ISBN 0-534-95600-9, 1996