©2077 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Elastic Components:
Addressing Variance of Quality Properties in Components

George Kakarontzas
Dept. of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece, and
Dept. of Information Technology and Telecom.
T.E.I. of Larissa
Larissa, Greece.
gkakaron@teilar.gr

Panagiotis Katsaros and Ioannis Stamelos
Dept. of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece.
{katsaros,stamelos}@csd.auth.gr

Abstract

The quality properties of a software component, although verified by the component developer and even certified by a trusted third-party, might very well be inappropriate for the requirements of a new system. This is what we call the quality mismatch problem: the mismatch between the quality requirements of a new system with the quality properties exhibited by the components that we want to use for its development. This work contributes to the understanding of the quality mismatch problem between component properties and component-based systems requirements. To solve this problem we introduce the concept of elastic components. An elastic component is an open-ended hierarchy of the same pure component with variants that differ between them to the quality properties that they exhibit. We present a quality-driven design approach that can be effectively applied for the design and implementation of elastic components.

1 Introduction

In Component-Based Software Engineering (CBSE) we distinguish between two largely independent activities [1]:

1. Development for reuse or Component Development: This is the activity of component development, in which software components are developed to be reused in several future software products

2. Development with reuse or Component-Based System Development: This is the activity in which software systems or parts of these systems are built by integrating prefabricated components.

In a previous article [2], we have described TactPC (Tactic-Driven Process for Component design), a process used during component development, in which software components are designed using a model-based testing approach. In TactPC, the component under development and the component architectural dependencies are modeled in a formal specification language (e.g. Abstract State Machines Language – AsmL) and then model-based testing is used to discover quality defects. The defects are then mitigated with the application of tactics (e.g. some of the architectural tactics from [3, 4]) both inside the component's assembly model and at the component's context dependencies. The process is iterative and is repeated as many times as necessary to address all quality-related concerns.

An important issue that arises with a methodology such as TactPC is the issue of quality adaptation of components to new systems. As we elaborate further on Sec. 2, the *quality mismatch* between assumptions made during component development and quality requirements of systems during component-based system development, is unavoidable if the component's quality properties are fixed. In this work we try to address this issue by introducing the concept of *elastic components*: components which provide the means for adaptation of conflicting quality properties for the satisfaction of system quality requirements during system design.

In the rest of the paper in Sec. 2 we provide the conceptual definition of elastic components and motivate their use by elaborating on the quality mismatch issue. Then in Sec. 3 we discuss the design of elastic components. A short example demonstrating elastic component development is provided in Sec. 4. Related work is presented in Sec. 5. Finally in Sec. 6 we provide future research directions and conclude.

33rd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2007)
0-7695-2977-1/07 $25.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

2 Quality mismatch and the concept of elastic components

Quality properties are properties like performance, availability, usability etc. Quality is considered orthogonal to a system's functionality [3]: we can achieve several different quality levels in relation to the same system function. Functional requirements are those requirements that describe what the system must do in order to function correctly (deliver the required result). To quote the ISO/IEC 9126-1 standard quality model [5], functionality is *"The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions"*. Quality requirements are usually linked to functional requirements and constrain the accepted solutions in relation to a quality property. An example of a quality requirement could be that "The processing of order requests would always terminate in no more than 2 seconds" in which the 2 second constraint is a quality requirement linked to the functional requirement of order processing. We observe that quality requirements are essential for the system in order to satisfy its functional requirements. This is why they are considered "requirements" and not merely "good-to-have". They are sometimes mentioned separately simply because they concern quality properties, although in our view this is a questionable practice since a developer might overlook them or consider them as not as essential.

Quality requirements create an interesting dichotomy between the two CBSE activities of component development and component-based system development. From a component developer standpoint, they are *absent*: the component developer cannot assume a particular system in order to really answer the question whether a particular quality property is essential for the system, and therefore a system's quality requirement since without it the system wouldn't function correctly, or is it just good to have, and therefore merely enhances system's quality. If for example the component computes r then what are the performance requirements for this? We can't really answer this question unless we have a specific system in mind. For component-based system development on the other hand, the quality requirements are known. The component-based system developer knows exactly what the performance requirements for r's computation are. If we assume that the component should deliver r in t seconds or otherwise the system would fail, then if a given component c_a delivers r in t_a seconds and $t_a < t$ the component satisfies the quality requirement. The difference $t - t_a$, assuming that faster is better for the given system (which might not always be the case e.g. real-time systems), is a performance enhancement. A component c_b that delivers r in t_b seconds with $t_b < t$ would also satisfy the quality requirement, but would be of poorer quality if $t_a < t_b$. The discussion so far, might lead to the conclusion that the component developer should try to enhance all quality properties as much as possible. This approach however would be overly simplistic, since quality properties are often conflicting. Consider for example performance and availability. Higher availability is usually achieved with redundant components and computations. But redundancy has a cost in performance since, for example updates should take place in all redundant copies. This situation, somewhat simplified, is depicted in Fig. 1 in which the functional and associated quality requirements are depicted as a gap in a system.

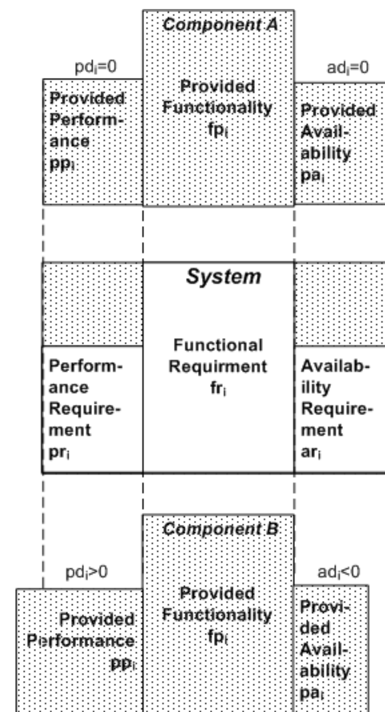

Figure 1. Component quality tradeoffs and their relationship to system requirements.

In Fig. 1 we have a functional requirement fr_i for a system in which the component is to be placed. The component was developed independently by a component developer at some earlier time. The system functional requirement is linked with a performance requirement pr_i and an availability requirement ar_i. We emphasize that these quality requirements are unknown to the component developer since they are part of a future system description. The component developer simply tries to score as high as possible on both or uses some estimates based on an imaginary system or in some cases in an actual system under development for which this component is developed. However since these requirements are conflicting the component developer is forced to proceed to a rather uninformed or system-specific

decision, for the provided performance pp_i and the provided availability pa_i. The decision might be to balance performance and availability as in the component A in Fig. 1, in which the differences in provided performance and availability from the required performance and availability (pd_i and ad_i respectively) are zero, and therefore the component happens to satisfy exactly the quality requirements of the future system. If the component developer however, decided to balance performance and availability as in the component B, favoring performance over availability, then although the component would satisfy system's performance requirements since $pd_i > 0$, it would fail to satisfy system's availability requirements since $ad_i < 0$.

The important observation here is that the balancing of quality properties for a component cannot possibly be a perfect match for all future systems that the component will be used and therefore fixing these properties to specific values during component development is an extremely limiting factor for the component's suitability. To avoid this seemingly unavoidable *quality mismatch* between components and component-based systems we introduce the concept of *elastic components*.

> An elastic component is a family of components in which a base component called pure is refined by a number of variant components. Variants are produced through quality-driven transformations of the pure component. Variants provide the same essential functionality with the pure component but differ on a number of quality properties which they improve.

Before proceeding to the design of elastic components in Sec. 3, we make the following important observations: (1) In the literature there is a distinction between *execution quality attributes* and *evolution quality attributes* (e.g. [6]). Execution quality attributes are attributes that are observable at system runtime (e.g. performance, security, availability, usability etc.). Evolution quality attributes are observable in the development and maintenance phases of the system lifecycle (e.g. flexibility, reusability, testability etc.). In this work we consider only execution quality attributes. (2) We don't try to evaluate components in isolation to their assemblies. We accept the argument in [7] that this is meaningless. We are concerned instead with quality adaptive components that can be used in the context of a system evaluation method. (3) Finally, elastic components are a concept as we explained in this section. As a concept they are independent from the design methods and implementation techniques that can be used for their development. The design and implementation techniques proposed in this work, is only one possible way but other methods might also be appropriate.

3 Design and implementation of elastic components

The components presented to the customers should not be "frozen" entities, with fixed quality properties. Instead the client should be able to select a general component that satisfies the required functionality first, and then specialize the component by traversing a hierarchy of possible components extending the general component with specific non-functional enhancements that interest the client. This selection hierarchy is depicted in Fig. 2, where the general component is called *pure* and the different variations of it are called *variants*.

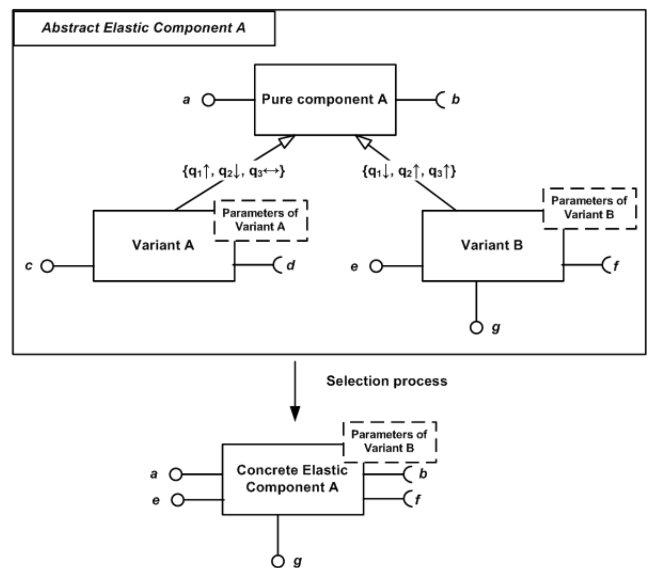

Figure 2. Elastic components as a selection process deliverable.

Each variant affects a number of quality properties (e.g. q_1,q_2, and q_3). Symbols $\uparrow$, $\downarrow$ and $\leftrightarrow$, stand for improved, diminished and indifferent effect on the quality property. Each variant comes with additional architectural requirements that might be necessary for the quality enhancement. Also each variant is parameterized to further allow the tailoring of the variant in the new system. Additional services and dependencies are depicted as additional provided and required interfaces. During the selection process the client consults the quality properties of each variant as well as the architectural dependencies to find the most appropriate component variant for the application in hand. The selection process is then the creation of an appropriate variant component with the required quality properties and architectural dependencies. The selection process is essentially a composition of the interfaces and parameters from the hierarchy, that looks promising in satisfying the client's re-

0-7695-2977-1/07 $25.00 © 2007 IEEE

quirements. Conceptually the component doesn't exist prior to its selection, although in practice certain popular variants might be already available. Notice also that the client might select the pure component, without any quality enhancements, in case that the client is not interested in any of them.

The refinement hierarchy depicted in Fig. 2 seems inevitable if we want to have any chance in achieving true reusability of components in many applications. However in order to be successful the process for the creation of variants should be easy and automated as possible. For this we must carefully address both the design and the implementation of elastic components.

In Fig. 3 we can see the basic classes for the design of elastic components.

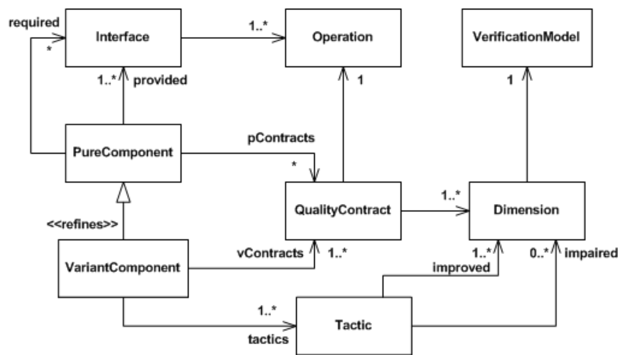

Figure 3. Class diagram for the elastic component design.

Each pure component provides a number of provided interfaces and might require a number of required interfaces. Interfaces are sets of operations. In addition each pure component provides some quality contracts that include the dimensions of interest. Dimensions are application domain specific quality properties, such as timing, latency, etc. Dimensions of the same type are comparable to each other. Also each dimension that is mentioned in a quality contract can be verified by using a model. The model describes the method that the component developer used to determine the value of the dimension. For example the model might describe that the dimension value was determined by simulating the component and its environmental assumptions and provide details for the simulation. A variant component is obtained by applying a number of tactics in the pure component. Tactics are also associated with the dimensions that they improve. This results in a variant component that has improved quality contracts in these dimensions, but might have impaired quality contracts in other dimensions. In any case the component developer after applying the tactic uses the same model that was used for the pure component's contracts, to produce a new set of contracts for the variant component in which at least one dimension will be improved and zero or more dimensions might be impaired.

Tactics are reusable quality aspects that can be composed with pure components for the creation of variants. They might be architectural tactics [3, 4] which are specific design alternatives that can be applied during software architecture design to improve certain quality properties, design patterns [8], or more domain specific techniques such as real-time design patterns [9]. All these techniques, which we collectively call *tactics*, are indispensable for the design of systems with the general goal of improving system quality and are widely applied in the context of system development in which a system is analyzed, designed and implemented from scratch. In our work we apply them in the context of component development, in which components are developed independently from the systems, to introduce the required variability in the main functionality of the components, essentially allowing the tailoring of components to different applications. Tactics can have: (a) *component scope*, affecting only the internals of a component while leaving the interface of the component unmodified, (b) *component interface scope*, changing the interface of the component and (c) *assembly scope*, requiring modifications that span multiple components and complex interaction patterns between the collaborating parts. Although component scope tactics are preferable since they do not introduce any additional dependencies, in general what is required is a precise documentation of components that will allow their use in new systems.

For documenting tactics in a systematic way we can use the notation suggested in [10]. In this work Ivar Jacobson and Pan-Wei Ng, propose a method that respects the separation of concerns all the way from requirement specification with use-cases to the implementation of the system with Aspect-Oriented Programming (AOP). In fact each use case *is* a crosscutting concern, since its design and implementation will require use case specific code to appear in many different objects. The authors capture these concerns in *use case slices*, a new modularity unit, which contains *collaborations, specific classes and specific extensions*. Collaborations contain class diagrams, interaction diagrams etc. needed for the realization of the use case. Specific classes are classes not needed by other use cases in a system but are specifically introduced for the use case captured in the use case slice. Finally extensions are extensions of existing classes with operations to specifically support the use case modeled by a use case slice. On particular interest to us are *extension use cases*. These use cases represent additional steps during the execution of a use case path, that are triggered by some condition (e.g. the calling of a method). The base use case that they extend can operate and provide the functionality without the extension, but with the exten-

sion it provides some additional functionality that modifies the basic functionality in some way. We can model extensions separately and then weave them in specific points at the base use case execution path. For tactics modeling we can use the use case slice concept in its abstract format. By this we mean that the tactic defines the participating roles instead of specifying interface methods that it applies to by naming them. An abstract tactic can be instantiated with a concrete tactic, by binding the roles to specific components, and interface operations.

For the implementation of variants we currently use Aspect-Oriented Programming (AOP) with pointcuts defined on methods of the component interface. The tactic is implemented as an abstract aspect, that can be instantiated to provide a more specific aspect instance for a given component. The implementation of tactics as abstract aspects brings similar advantages to those for the implementation of design patterns as abstract aspects in [11], namely (i) locality since the code of the tactic is localized in the tactic aspect, (ii) reusability since the abstract tactic can be instantiated in different tactic instances, (iii) composition transparency since tactic instances are independent, and (iv) (un)plguggability which allows the introduction and removal of tactics from the component at will. In addition to AOP we also plan to evaluate other available variability introduction techniques such as inheritance and aspect-component composition.

4 An example from the real-time control problem domain

In this section we provide an example elastic component from the real-time control systems domain. Before proceeding we provide a short introduction to the problems faced in the problem domain. In any (embedded) realtime control system, a control task is a granule of computation treated by the scheduler as a unit of work to be allocated processor time, or *scheduled*. Tasks execute in parallel with other tasks, including other control tasks. The system has to be able to predict the evolution of the running tasks and guarantee in advance that all tasks have bounded resource and processing time requirements and are allocated sufficient processor time to complete by their *deadlines* (predictability). This concern is undermined by outstanding market requirements like reduced time-to-market, which result in a strong trend to use commercial-off-the-self (hardware and software) components. In order to guarantee predictable behavior we usually use off-line design methodologies that aim to maximize the determinism. Also the constant demand for reduced development costs yields systems that are subject to resource constraints such as limited CPU speed, and limited memory and network bandwidth of the underlying platform. Thus, in addition to adequate predictability a tradeoff goal is to aim in more efficient use of the available resources (improved efficiency).

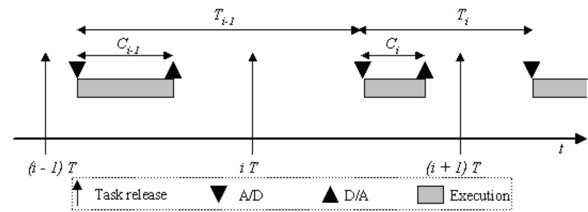

Figure 4. The basic control loop task model.

Controllers are assumed to be periodic tasks consisting of three parts: input data collection (A/D), control algorithm computation, and output signal transmission (D/A). Figure 4 introduces the critical timing parameters of the *basic task model* for periodic controller tasks (task period T equal to the task's *relative deadline*). The *task release time* is the time when the task is woken from its sleep. The *sampling interval* T_i is the time interval between two subsequent A/D conversions. The *computational delay or input-output latency* C_i is the time interval between a couple of A/D and D/A conversions. Simulation experiments [12] show that if the *control performance* is measured by some appropriate cost function say J (e.g. the stationary variance of some target control variable), it degrades and the control loop becomes unstable ($J \to \infty$), when the average value of the delay $C \to T$. In general, we can compensate a too large computational delay by decreasing the speed of the controller system. The *sampling jitter* is the variation in sampling interval and the *computational delay jitter (input-output latency jitter)* is the variation in C_i. Simulation experiments show that control performance also degrades as the jitter level increases. Thus, ideally the sampling jitter should be zero and the computational delay should be jitter-free.

Today control systems basically operate in highly dynamic environments, where the application characteristics are not fixed a priori and also, when a new control task is admitted this raises a need for readaptation.

A *task overrun* may occur if some control task does not meet its design specifications during run-time, e.g. if the execution time of the control algorithm is getting larger than the predicted worst-case execution time (*execution overrun*) or because input samples arrive more frequently than expected (*activation overrun*). Overruns cause a *transient overload condition* because they affect the scheduling of other concurrently running tasks in a way that depends on the applied scheduling algorithm (*static or dynamic priority* scheme). In general, a task overrun does not necessarily cause an overload, but a large unexpected overrun or a sequence of overruns can cause very unpredictable effects on the system, if not properly handled by an adaptive scheduler.

33rd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2007)
0-7695-2977-1/07 $25.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

The example given in the sequel introduces an elastic *EDF (Earliest Deadline First) scheduler component* for flexible and adaptive real-time control applications, where some of their constraints can be occasionally lost thus decreasing the provided control performance without causing critical system faults.

Control tasks' schedulability is not successfully guaranteed off line if systems suffer from sudden variations in computational load and overloaded situations and worst-case execution times have been underestimated. The variants of the example are derived from the related bibliography ([13, 14, 15]). Each variant preserves the system's predictability by different tactics that result in prominent impacts on *control performance*, *efficiency in the use of available CPU time* and *incurred overhead*.

The *job skipping scheduling approach* [13] is an "open loop" tactic where once schedules are created they are not "adjusted" based on continuous feedback. The associated *Skip-Over Algorithms* permit skips in periodic tasks thus making better use of the available CPU time in order to achieve a feasible schedule in systems that would otherwise be overloaded. The maximum number of skips for each $task_j$ is controlled by a specific parameter associated with the task. In particular, each periodic $task_j$ is characterized by a worst-case computation time C, a period T, a relative deadline D equal to its period and a skip parameter $S(2 \leq S \leq \infty)$, which gives the minimum distance between two consecutive skips. The skip parameter can be viewed as a task-specific Quality of Service (QoS) metric (the higher S, the better the quality of service). Every instance of a periodic task with skips can be red or blue. A red instance must complete before its deadline; a blue instance can be aborted at any time. When a blue instance is aborted, we say that it was skipped. The fact that $S \geq 2$ implies that, if a blue instance is skipped, then the next $S - 1$ instances must be red. On the other hand, if a blue instance completes successfully, the next task instance is also blue. The *Blue When Possible (BWP) EDF scheduler* shown in Fig. 5 schedules blue instances whenever there are no released red tasks to execute. Red instances are scheduled according to EDF.

Feedback Control - U EDF (FC-U EDF) adopts the feedback control scheduling tactic and it is one of the scheduling algorithms proposed in [14]. The employed architecture (Fig. 6) forms a feedback control loop composed of a *monitor*, a *controller*, a *QoS actuator* and a basic scheduler. Each task has N QoS levels ($N \geq 2$), which in the simplest case of only two levels these correspond to the rejection and the admission of the task. Each QoS level m ($0 \leq m \leq N - 1$) of $task_j$ is characterized by: (i) the period $T[m]$, (ii) the estimated (not the worst-case) execution time $EC[m]$, (iii) the relative deadline $D[m]$, which in the shown component is considered to be equal to $T[m]$ and (iv) the value $V[m]$ that $task_j$ contributes if it is completed at QoS m before its deadline $D[m]$. Every QoS level contributes a value $V[0]$ if it misses its deadline. In the described architecture the monitor measures the controlled variable, i.e. the actual CPU utilization $U(k)$ and feeds the samples back to the controller at every sampling instant k. The controller compares the utilization reference parameter U_S with the corresponding controlled variable to get the current errors and computes a change - called *control input* - to the total estimated requested utilization based on the errors. The QoS actuator dynamically changes the total estimated requested utilization at each sampling instant according to the control input by adjusting the QoS levels of tasks. This is accomplished by the use of a QoS optimization algorithm that aims in maximizing the system value.

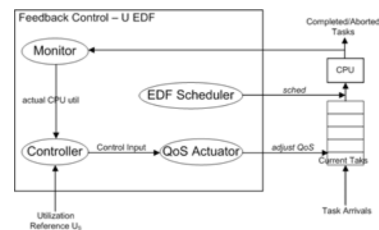

Figure 6. The feedback control scheduling architecture.

The FC-U EDF scheduler guarantees that the deadline miss ratio (number of deadline misses divided by the total number of completed and aborted tasks) is 0 in steady state, if the input reference U_S is lower than the schedule utilization threshold. In EDF this threshold is 100% for a task set with independent and periodic tasks. Although U_S cannot be very close to 100% (to avoid potential saturation on the control performance) it is still possible to perform in utilization levels well above 90%. If we take into account that Basic EDF and BWP EDF guarantee schedulability on the basis of pessimistic estimations for the task execution times, this justifies why the published theory and simulation results show that the FC-U EDF scheduler makes more efficient use of the available CPU time. Of course, this happens at the cost of additional processing overhead (Fig. 5).

The last mentioned EDF variant views each task as flexible as a spring with a given rigidity coefficient and length constraints. More precisely, the so-called Elastic EDF scheduler [15] treats the utilization of served tasks as an elastic parameter, whose value can be modified by changing their periods. Each task is characterized by five parameters (Fig. 5): a nominal period T, an estimation C for the task's worst-case execution time, a minimum period T_{min}, a maximum period T_{max} and an elastic coefficient $e \geq 0$, which specifies the flexibility of the task to vary its utilization for adapting the system to a new feasible rate configu-

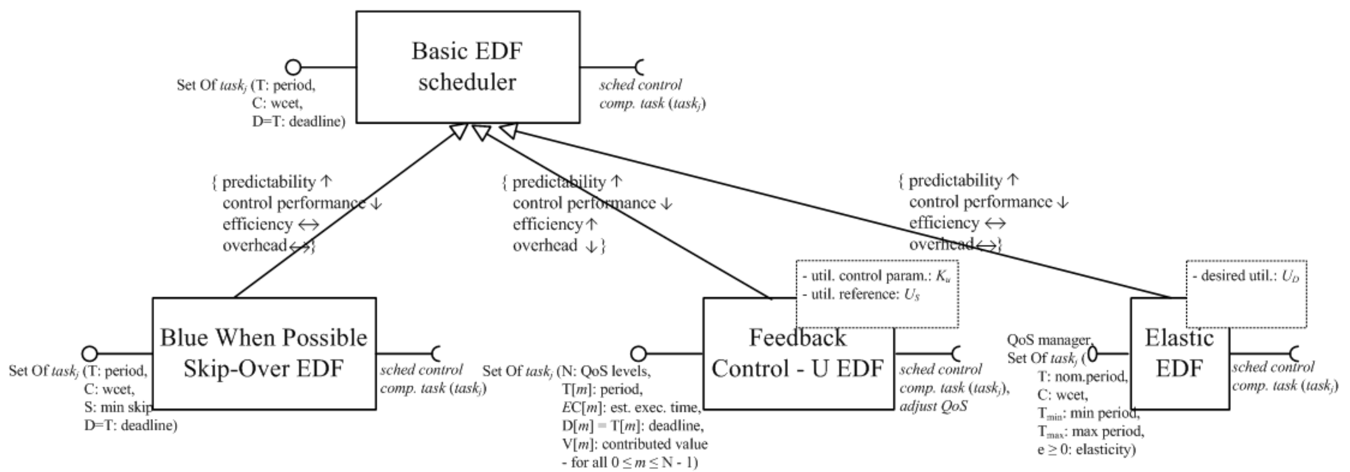

Figure 5. An elastic EDF scheduler component.

ration. The greater e, the more elastic the task. The elastic scheduler mechanism is based on a high priority task, the *QoS manager* that is activated by the other tasks when they are created or when there is a need to change their period. Whenever activated, the QoS manager calculates the new periods such as to succeed the desired utilization level U_D and changes them atomically. The described solution preserves the system's predictability at the desired level without incurring significant processing overhead as in the preceding feedback control scheduling architecture.

5 Related Work

In [16] components are also accompanied by contracts expressed in CQML+. The approach is that the component realtime container ConQoS RT enforces the component quality contracts in runtime and supports dynamic adaptation by using different versions of components. Components can be Composite Components [17], which are components that modify their internal structure to support different quality requirements. A composite component, supports variance during runtime by reconfiguration of its internal structure. Internally, glue code and aspect operators are used to modify the structure of the component with components in a component repository. Composite components are similar to our elastic components, with the main difference being that we don't modify the internal structure of the component in runtime, but instead provide more flexibility through a tactic-driven process (which requires some human intuition) during design time. This doesn't exclude the use of our elastic components in runtime: different variants can be deployed in a container that matches application requirements to the appropriate variants similar to ConQoS RT. We view the two approaches, of runtime internal reconfiguration and of design-time tactic-driven variation, as complementary research tracks.

Tactics are inspired by [3, 4] which describe architectural tactics as a means to achieve quality requirements during architectural design.

Reasoning frameworks were proposed in [18] as the vehicle for systematic reuse of analytical theories that can be used for the evaluation of software architectures. We note that the so-called models of our approach could be reasoning frameworks. We are strongly interested in tradeoff analysis metrics and models like the ones introduced in [19].

Another work using aspects for component adaptation of realtime embedded systems is described in [20]. Their approach is similar to ours in that aspects are applied to interface operations and advice internal mechanisms used by these operations. Aspects however are not related directly with quality improvements or impairments: they are not tactics in the sense described in this work. Therefore the determination of appropriate methods for improving qualities is left unspecified. Also the adaptation should maintain the same interface which can conceivably be restrictive in some cases.

6 Conclusions and Future Research Directions

In this work we presented the quality mismatch problem between components and component-based systems. This mismatch severely affects reuse and adaptation of components in new systems and requires an effective solution. We view our elastic components as a means to address this problem. Essentially our approach suggests that having just one fixed component for all systems is very restricting. Instead we proposed a decomposition of the component to functional and quality parts, that we call pure components

IEEE COMPUTER SOCIETY

and tactics respectively. The quality parts can be composed with the pure components to produce new variants satisfying the requirements of new applications. Feedback from the component-based system development can be fed back in the component development process to yield new reusable quality parts and new variants, in an ever growing hierarchy of components. The whole hierarchy is the elastic component.

The proposed process assigns the various responsibilities to the right parties. System developers have a goal to create correct systems. With elastic components they have more information than what is usually available, and can use this information to discover if a component provides the required quality properties. They can contact the component vendor to provide a new variant if one is unavailable. The component vendors have a quality-driven reuse process in place to quickly produce a new variant if one is requested, and also have a very good incentive in doing so: *expanding their market*.

Future work includes:

1. Tool support for the creation of elastic components coupled to a repository of reusable tactics as aspects.
2. The formal definition of tactics as well as their composition with components, which is essential for both tool support and potential expansion of the method.
3. Delivery of a set of tactics for component-based embedded realtime systems, which seems to be a demanding CBSE application area.

References

[1] H.-G. Gross: "Component-Based Software Testing with UML", Springer, 2005

[2] G. Kakarontzas and I. Stamelos: "A Tactic-Driven Process for Developing Reusable Components", 9th International Conf. on Software Reuse, LNCS 4039, pp. 273–286, 2006

[3] L. Bass et. al.: "Software Architecture in Practice, 2nd ed.", Addison-Wesley, 2003

[4] F. Bachmann et. al.: "Deriving Architectural Tactics: A Step Toward Methodical Architectural Design", Tech. Rep., CMU/SEI-2003-TR-004, March 2003

[5] ISO/IEC 9126-1: "Software Engineering - Product Quality - Part 1: Quality Model". ISO/IEC Standard, ISO/IEC 9126-1:2001(E), 2001

[6] M. Mari and N. Eila: "The Impact of Maintainability on Component-Based Software Systems", 29th EUROMICRO Conference, IEEE, 2003

[7] K. Wallnau J. A. Stafford: "Dispelling the Myth of Component Evaluation", in Building Reliable Component-Based Software Systems, I. Crnkovic M. Larsson (eds.),pp. 157-177, Artech House, 2002

[8] E. Gamma, R. Helm, R. Johnson and J. Vlissides: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1997

[9] B. P. Douglass: "Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems", Addison Wesley, 2002

[10] I. Jacobson and P.-W. Ng: "Aspect-Oriented Software Development with Use Cases", Addison Wesley Professional, 2004

[11] J. Hannemann and G. Kickzales: "Design pattern implementation in Java and AspectJ", OOPLSA'02, ACM Press, 2002.

[12] A. Cervin: "Towards the integration of control and real-time scheduling design", Licentiate Thesis, Lund Institute of Technology, May 2000

[13] G. Koren and D. Shasha, "Algorithms and complexity for overloaded systems that allow skips", IEEE Real-Time Systems Symposium, December 1995

[14] C. Lu, J. Stankovic, S. Son and G. Tao, "Feedback control real-time scheduling: Framework, Modeling and Algorithms", Real-Time Systems 23, pp. 85-126, 2002

[15] G. Buttazo et. al., "Elastic scheduling for flexible workload management", IEEE Transactions on Computers 51 (3), pp. 289-302, 2002

[16] H. Härtig et. al.: "Enforceable component-based realtime contracts: Supporting realtime properties from software development to execution", Real-Time Systems Journal, vol.35, no.1, pp. 1-31, Springer, 2007

[17] Steffen Göbel: "Encapsulation of structural adaptation by composite components", Workshop on Self-healing systems, pp.64-68,ACM Press, 2004.

[18] L. Bass et. al.: "Reasoning frameworks", CMU/SEI-2005-TR-007, 2005.

[19] P. Katsaros et. al.: "Performance and effectiveness trade-off for checkpointing in fault-tolerant distributed systems", Concurrency and Computation: Practice and Experience, 19(1), pp. 37-63, Wiley, 2007

[20] A. Tecanovic et. al.: "Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software", Journal of Embedded Computing, October 2004.

0-7695-2977-1/07 $25.00 © 2007 IEEE

IEEE COMPUTER SOCIETY