

Managing quality of service during evolution using component contracts*

J. Gorinsek, S. Van Baelen, Y. Berbers, K. De Vlaminck
K.U.Leuven, Department of Computer Science,
Celestijnenlaan 200A, 3001 Leuven, Belgium
E-mail: {jorisg | stefanv | yolande | kdv}@cs.kuleuven.ac.be

March 10, 2003

Abstract

In this paper we propose a methodology to guarantee resource contracts and increase reliability across updates in embedded soft real-time systems. We present the road map to a solution based on the verification of component updates using update contracts. In our system the behavior of all components is modeled in detail. Relying on a resource aware component system which has full control over all resources we will be able to guarantee that an update doesn't cause resource constraint violations for components that are not updated. We also lay out a tool which helps the software developer in estimating the maximal resource consumption of a certain update and determining the impact of a that update on the running program.

Keywords: component based development, embedded systems, software evolution, contract based development, resource-aware components, QoS monitoring, resource broker

1 Introduction

In theory, dynamic software updates are a god sent gift for embedded software developers: dynamic updates allow for easy bug fixing, adding functionality on demand (for instance adding an encryption component to your Bluetooth enabled PDA when you enter an untrusted network) or updating firmware without recalling all devices on the market. In practice however, we argue that strong verification is needed to guarantee system reliability and quality of service across updates.

While dynamic software adaptation introduces great benefits for embedded systems, it also imposes a severe threat on the reliability and quality of service aspects of the system. Each modification to the code of a component (for example changing an algorithm by a faster, but more memory consuming version) will change the resource consumption and QoS properties of that component. Since resources in embedded system are typically scarce, changing one component may impact the complete running program. In an ideal scenario, the impact of this component update will not cause any problems. However, we believe that for the majority of updates, problems will occur in the form of missed deadlines or even corrupted systems during or after the update.

Our pessimism is rooted in the fact that even though the impact of the component update on the running program may be minor, the updating itself requires a significant amount of resources since it can require among others having two versions of a component in memory

*The described work is part of the EUREKA-ITEA project EMPRESS (<http://www.empress-itea.org>), and partly funded by the Flemisch government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders)

at the same time while transferring the internal component state from the old to the new version. Therefore, even if the new component will fit seamlessly into the running program, the updating process itself will most likely cause the program to violate its timing or memory constraints.

In order to avoid the problems mentioned above we need to have reliable information on the resource consumption of a new component version and of the update process itself. Using this data we can determine the impact of a given update on the running program and evaluate if that update can be performed without causing the running program to violate its resource constraints.

In this paper we focus on the quality of service aspects of evolution, not the technical details of the evolution of components itself. For the more technical aspects of the evolution process itself we refer to [10, 11]

2 Modeling component behavior

In order to be able to estimate what the impact of the update of a component will be on the program to be updated, we need to know exactly how this component behaves. Not only do we need to know how that specific component behaves, we also need to have information on how its neighbors behave and how all these components interact. In this section we present a brief overview of how we will model component behavior and component interaction using contracts [2, 4].

We distinguish three types of contracts:

Component contracts: Each component has a component contract associated to it. This contract specifies the minimum and maximum amount of resources this component requires. Upon deployment a component negotiates with the component system about the amount of resources it will actually receive. This quota of resources can be dynamically re-assigned when necessary. The contract interval boundaries enable a graceful degradation of the *quality of service* of a component while remaining fully functional. An example of this technique is a video decoder component which can use less CPU cycles by occasionally dropping frames or degrading image quality. This kind of contract can be seen as contract between a component and the component system in which it runs.

Intercomponent contracts: Each component has an *intercomponent contract* which details the outgoing messages that are triggered per message it provides in its interface. This way we can easily trace paths through the system without need to inspect the code itself. Also, using the intercomponent contracts of two communicating components we can extract a protocol that details how these two peers exchange messages. Also, for each interface a QoS specification is built using the information provided in the intercomponent contracts and the QoS specifications of connected components which specifies timing constraints on each of the methods specified in that interface.

Update contracts: An update consumes a very significant amount of resources. Update contracts describe the maximum amount of resources an update may spend. This conservative estimation takes into account the updating method to be used, such as the manner in which state transfer will be performed, the intermediate data that will have to be saved, etc. This kind of contract can be seen as contract between the update service and the component system in which it fulfills the update.

We will employ contracts on two levels as shown in figure 1: on a meta level where contracts are specified in XML (illustrated in figure 2) and on a component level where contracts are specified as component attributes and message sequence charts [3]. Contracts on a component level are actually concrete implementations of the high level contracts specified in the meta layer. These low level contracts are used by the component system to determine

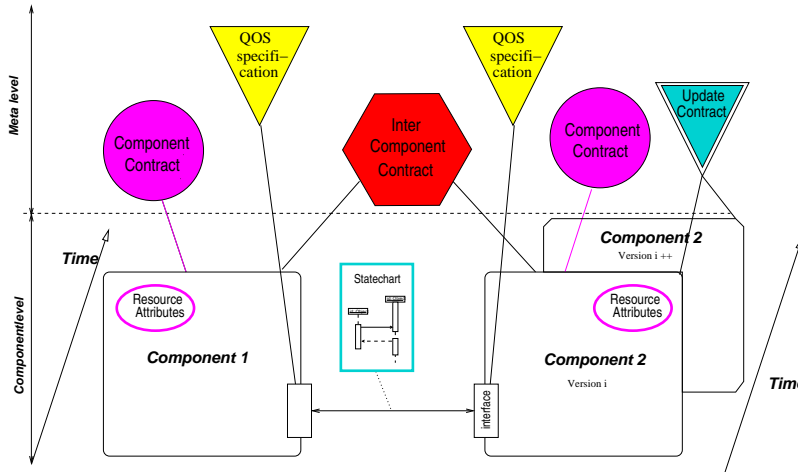


Figure 1: Contracts work in different dimensions and on different levels of abstraction.

scheduling priorities and allocate resources in order to guarantee the specified constraints. The meta-layer information will be used by our tool (see section 4 and [9]) to determine the impact of an update on the running program.

The reason for this split-up in a meta level and a component level is that we need high level information about the behavior of our components but we can't afford the performance hit induced by rerouting all transactions through a meta layer (which is common practice in *meta object protocols* [6, 8]). To reduce the performance penalty of using a meta layer we introduce concrete implementations of these high-level contracts in the component layer. These are the contracts that are used during normal execution. The meta-level contracts are only used at deployment time and when preparing an update.

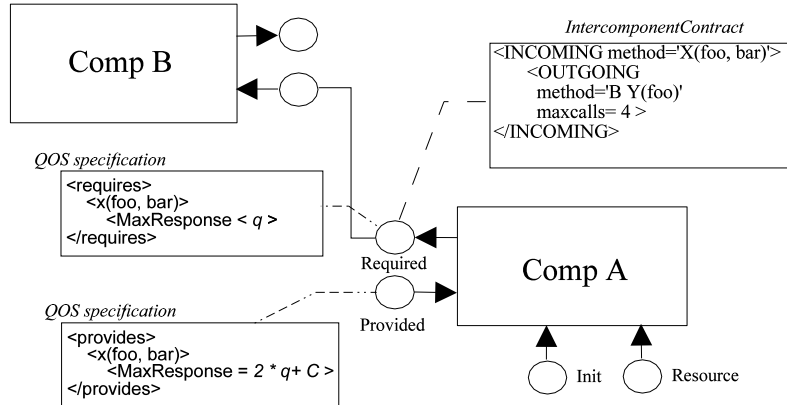


Figure 2: Two component instances with intercomponent contracts and QoS specifications associated to them.

3 System support

In this section we explain how we plan to realize contract support in our component system by introducing *resource awareness* into it.

Using and enforcing resource contracts at run-time requires three extensions of our component system:

Resource control: We need to have total control over the amount of resources that are used by each component. We will accomplish this by having each component negotiate over the maximum resources it will use with a *resource broker* which can approve leases on certain resources such as bandwidth, memory, etc. This resource broker manages access to the *abstract hardware layer* which encapsulates the interfaces to the actual hardware.

Monitoring: Next to controlling who can use what resources we need detailed information on the actual resource consumption at any given moment. Monitoring support in the component system will provide us with the information we need here. Next to *passive* monitoring as described above, we will also support *pro-active* monitoring where the monitor actively monitors components. The information provided by the monitor can be used to detect components who don't obey their contracts. These components can then have their contracts renegotiated or can even be completely stopped in case the contract violation threatens the correct functioning of the whole application.

Resource enforcing: The above two extensions allow us to *monitor* contracts, but don't enforce them. To accomplish this we introduce a feedback controlled scheduler [7, 13] in the component system which uses the information provided by the monitoring system and the component contracts to determine scheduling priorities for component threads.

With the extensions described above we will be able to guarantee that an update of a certain component will not hinder components that are not directly depending on that component. For a component instance Bar which should be upgraded to Bar' using update technique U this is accomplished as follows (illustrated in figure 3):

1. First we determine the maximal resource consumption for the update using our tool (see section 4).
2. Second, we will use our tool to examine min and max resource consumption of all component instances in the system in order to verify if the update is possible given the possibly available resources. Also, intercomponent contracts will be examined to determine if other components will be influenced by the update of Bar to Bar' (for instance when the rate of messages that are sent by Bar is increased or decreased in version Bar').
3. Third, we query the component system on current resource usage and maximally available resources. If the resource consumption of the update is smaller than the maximally available resources (including the minimalization of all QoS contracts for the running components), we queue the update for execution.
4. Fourth, when the necessary resources become available, the component system will execute the update.

Of course, in practice things aren't always that simple: updates can turn out to be corrupt for instance. Currently, our system doesn't support a rollback mechanism to deal with corrupt updates. Rollback support implies recording all actions that are performed on a certain component and saving old versions of a component, which is time – and memory consuming and therefore not straightforward on an embedded system. However, when we are working in a distributed environment where rollback information can be stored on another host, it can still prove to be a viable approach which will be investigated in future work.

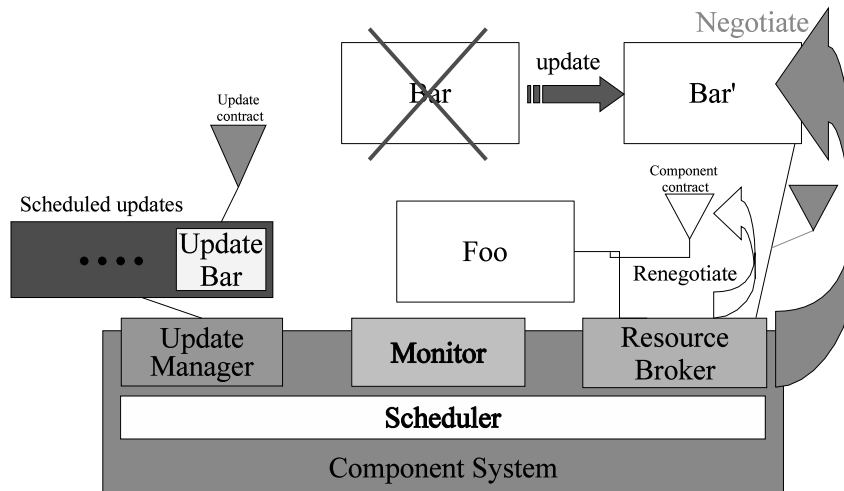


Figure 3: Schematic representation of the component system with its most important building blocks.

4 Tool support

This section provides a brief overview of how we plan to realize tool support in order to be able to determine the actual *impact* of an update on a running program.

The ability to predict the magnitude of impact of a certain update on a running program is the missing link between the behavioral models in the form of contracts of section 2 and the supporting component system as described in section 3. Using the information provided by the component contracts and relying on our resource aware system we can determine how much influence a given update will have on the other components in the system being updated.

In particular we will be looking at how much resources a certain update will consume. Using this information we can then either reject the update as invalid (for instance because it would require more resources than the system will ever be able to provide) or we can inform the component system that it should schedule the update and perform it when the necessary resources are available. Being able to reject or schedule an update provides us with a practical means to effectively minimize the potential interruption in the service of our embedded system which an update is likely to introduce.

Realizing an *impact estimation tool* as introduced above is far from trivial and consists of three steps:

1. First of all we will augment our component system with support for update scheduling. This includes adding self-monitoring capabilities in order to allow the component system to have detailed information on resource consumption at any given moment.
2. Second, a tool will be built to examine the contracts of all components in the system and of the new version of the component that is to be updated. Using this information this tool should be able to detect *ripple effects* (one update forcing several other components to update). When this is the case the whole group of components influenced by the update should be updated atomically and the resource consumption of the group update will be considered and eventually rejected if the update proves to be impossible in the current configuration. Also this tool will provide a safe estimation of the resource consumption of the pending update by investigating the byte-size of the component instances, the update method to be used, the complexity of state transfer, etc.
3. Third, to ease the task of scheduling an update for the component system we will make

our components dynamically reconfigurable. The added flexibility obtained this way will allow the component system to perform *graceful degradation* of components: it can temporarily reduce the resources available to some components in order to have some more room to squeeze in an update.

5 Related work

The Comquad project [1] is currently working on an architecture and a development methodology to support the composition of adaptive software from components with assurable non-functional characteristics. They accomplish this by using *system contracts* between component containers and the underlying run-time environment which reserve virtual resources. One of the main differences with our approach is that they don't have the equivalent of our *update contracts*.

Another project which uses an approach similar to ours is QCCS [12]. QCCS is a European IST project that aims at developing a methodology and supporting tools for the creation of components with contracts using on aspect oriented programming. Their approach is similar in the sense that they use contracts to specify component behavior and QoS properties. However, just like Comquad they don't have an equivalent of our update contracts.

Andreas Rausch [5] combines components and contracts with assertions to arrive at *signed contracts*. Signed contracts specify not only what a supplier provides to its environment, but also what a client needs from its environment. Signed contracts guarantee that client needs are satisfied by corresponding properties provided by suppliers. These contracts also allow a precise specification of the composition of component-based systems and a formal verification of the correctness of these systems. This way, software system defects can already be detected and prevented at the specification level.

6 Conclusions

We have laid out a base to support update verification in terms of resource consumption based on component, intercomponent and update contracts and a resource aware component system. However, a significant amount of work still separates us from a system which can be used in practice. Future work will include research on how to determine the resource consumption of an update, we will extend our component system and design and implement a tool to verify updates.

References

- [1] The comquad project, 2002. <http://www.comquad.org/>.
- [2] J. K. Filipe. A logic-based formalisation for component specification. *Journal of Object Technology*, 1(3):231–248, 2002. Special issue: TOOLS USA 2002 proceedings.
- [3] S. L. H. Ben-Abdallah. Timing constraints in message sequence chart specifications. In *IFIP*. Chapman & Hall, 1997.
- [4] B. Meyer. Applying design by contract. In *Computer IEEE*, volume 25 No. 10, pages 40–51. October 1992.
- [5] A. Rausch. Design by contract + componentware = design by signed contract. In *Journal of Object Technology special issue: TOOLS USA 2002 proceedings*, volume 1, pages 19–36, 2002.
- [6] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, pages 205–230, June 2002.
- [7] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao. The case for feedback control real-time scheduling. In *Proceedings of 11th EuroMicro Conference on Real-Time Systems*, June 1999.
- [8] E. Truyen, B. N. Joergensen, and W. Joosen. Concepts and experiments in computational reflection: Customization of object request brokers through dynamic reconfiguration. In *Proceedings of Tools Europe 2000*, June 2000.

- [9] D. Urting, S. V. Baelen, T. Holvoet, P. Rigole, Y. Vandewoude, and Y. Berbers. A tool for component based design of embedded software. In *Proceedings of Tools Pacific 2002*, pages 159–168, Februari 2002.
- [10] Y. Vandewoude and Y. Berbers. A meta-model driven methodology for state transfer in component-oriented systems. Technical Report CW 349, Department of Computer Science, K.U.Leuven, October 2002.
- [11] Y. Vandewoude and Y. Berbers. An overview and assessment of dynamic update methods for component-oriented embedded systems. In H. R. Arabnia and Y. Mun, editors, *Proceedings of The International Conference on Software Engineering Research and Practice*, pages 521–527, Las Vegas, Nevada, USA, June 2002. CSREA Press.
- [12] T. Weis and N. Plouzeau. Qccs: Quality controlled component based systems. presented at SIVOES-MONA: workshop on Component-Based Software Engineering and Modeling Non-Functional Aspects, October 2002.
- [13] R. West. Adaptive real-time management of communication and computation resources, 2000.