Quantity of Resource Properties Expression and Runtime Assurance for Embedded Systems

Laure Gonnord, Jean-Philippe Babau

L.Gonnord and J.-P. Babau, accepted in ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-2009) .

Quantity of Resource Properties Expression and Runtime Assurance for Embedded Systems

Laure Gonnord
University of Lyon
LIP laboratory, ÉNS Lyon,
46 allée d'Italie
69364 Lyon Cedex 07, France
Email: laure.gonnord@ens-lyon.fr

Jean-Philippe Babau
Université Européenne de Bretagne
UBO, LISyC
Département Informatique, 20 avenue Le Gorgeu CS 93837
29238 Brest Cedex 3, France
Email: jean-philippe.babau@univ-brest.fr

***Abstract*—Recent work on component-based software design has proved the need of resource-accurate development of embedded software. In the more specific cases of mobile systems, the developer also needs tools to facilitate the adaptation of functionalities to resources (lack of memory or bandwidth, *etc.*), and also to evaluate the performance w.r.t. the resource issues ([1]).**

As we want to design and develop at the same time the application and its resource controllers, we chose to use Qinna, which was designed to manage resource issues (specification, contractualization, management) during the development process of such an application. We propose a complete formalization of the resource constraints specification, through the use of a variant of the event-based logics, MEDL and PEDL introduced in [2]. Qinna then automatically performs the runtime resource assurance. We illustrate this work in a case study.

***Index Terms*—Software Engineering, Embedded Systems, Sofware Achitecture, Quality of Service, Runtime Management, Evaluation, Monitoring.**

I. Introduction

The study takes place in the context of embedded handled systems (personal digital assistant, mobile phone), whose main characteristic is the use of limited and variable resources (CPU, memory).

While designing software for such handled systems, the developer is faced with the problem of offering a certain quality of service (QoS) in variable context : limited and variable resource capacity (network, battery) and variable set of running applications. And because handled system administration is limited to restart (or reset!) operation, safety is a sought-after property of such system. Thus the developer needs tools in order to :

- easily and safely add/remove services at runtime
- adapt (degrade if necessary) component functionalities to shared resources capacities
- evaluate the software performances : quality of provided services, consumption rate for some usual scenarios.

In this context, component-based software engineering appears as a promising solution. Indeed it offers an easier way to build complex systems by assembling basic components ([3]). The main advantages are the reusability of code and also the flexibility of such systems. However, while the functional part of the component models is well achieved, the resource usage part is considered by several approaches in a specific (*i.e.* CPU use and timing constraints) but not generic way ([1]).

To address the resource issues in a dynamic context, Qinna ([4]) provides a component-based framework to explicit dynamic resource policies. Qinna's philosophy is to integrate resource control facilities (a component is viewed as a resource) and to implement variability through discrete QoS levels for component's services Then, Qinna provides distributed algorithms to dynamically adapt QoS levels according to resource availability *at runtime*. The challenge is both to ensure a safe resource use and also to provide a way to evaluate the threshold between resource use and available resource. In this paper, we propose to formalize Qinna concepts. The idea is to propose a complete chain, from the formal specification of resource's constraints to automatic generation of executable control policies, embedded in the application code. The approach is illustrated by a case study.

The paper is organized as follows : Section II sets the context of our study. Section III proposes a complete formalization of the framework. Section IV focuses on the expression of quantity of resource constraints, and a way to implement the expression of these properties in Qinna. In Section V, a case study illustrates the approach.

This work was done at INSA Lyon, CITI laboratory, University of Lyon, and was partially supported by the REVE project of the French National Agency for Research (ANR)

L.Gonnord and J.-P. Babau, accepted in ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-2009) .

II. Background

A. Component Model

This section outlines the minimal component model concepts necessary to implement Qinna. Following Component-Based Software Engineering philosophy, all the application modules, even resource (Memory, CPU, Network) drivers are components. A component is a piece of code that provides services and eventually needs other ones. A component may have internal functions, or internal attributes, but we only focus on the external functions called *services*. The communication between components is made only by function calls.

A component $\mathfrak{C}$ has a *type* which is basically its name and it represents a component class. The *instances* are denoted by by $\mathfrak{C}^j$ ($j \in \mathbb{N}$). A component provides *services*, s_i ($i \in [\![1, p]\!]$) and has also required services r_k. Each call to a service of $\mathfrak{C}^j$ is called a *call occurrence* of the service, and the sequence of all call occurrences is $(occ_k(s_i^j))_{k\in\mathbb{N}}$. $(occ_k(s_i))_{k\in\mathbb{N}}$ denotes the sequence of all occurrences (whatever the instance) of the s_i service.

The component has a dynamic behavior described by an automaton encoding which services can be offered at each moment. One example of such automaton can be found in Figure 1.

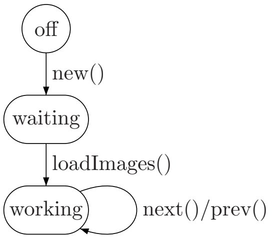


Fig. 1. Behavior of an image viewer component

Some components or groups of components provide services which code may vary according to its required services called “resources”.

B. Qinna

The Qinna architecture designed in [5] gives some development rules and algorithms in order to help the development of “resource aware” component based systems. The main purpose is to develop programs which can adapt themselves to the resource constraints.

The framework has the following characteristics :

- The quality variations of the provided services are encoded by the notion of *implementation level*. The code used to provide the service is thus different according to the current implementation level. Basically, the implementation level can be an integer attribute of the component which provides the service, and whose value is checked at the beginning of the service call in order to switch between different versions of the service code.
- As a variable service can call another variable services, Qinna proposes a static mapping between the implementation level of the service to provide and the respective implementation levels of the services it requires. Thus, an implementation level implicitely defines an implementation level graph call. The developer has the way to carry over for example that a video component provides a good image if precision is over X and the size more than Y pixels”.
- All the calls to a variable function are made through an existing contract that is negotiated. This negotiation is managed automatically through the Qinna components. A *contract* for a service at some objective implementation level is made only if all its requirements can also be reserved at the corresponding implementation levels. If it is not the case, the negotiation fails. In all other conditions, at each moment, the maximal objective implementation level is tried to be reached.

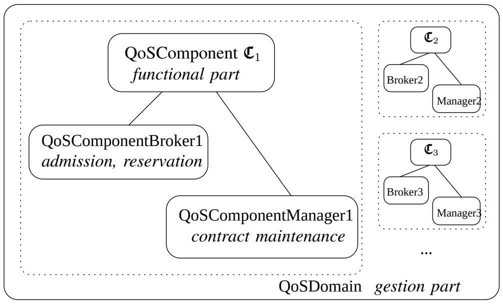


Fig. 2. A example of dynamic architecture

These characteristics are implemented through components, which are illustrated in Figure 2 : to each application component (or group of components) which provide one or more variable service Qinna associates a *QoSComponent* which is a wrapper around it. The variability of a variable service is made through the use of an `implementation level` inside the QoSComponent. Then, two new components are introduced by Qinna to manage the resource issues of the instances of this *QoSComponent* :

- a *QoSComponentBroker* which goal is to achieve the admission of a new component instance. The Broker decides whether or not a new instance can be created. At runtime, there is only one Broker for each type of QoSComponent, which is responsible for all its instances.

- a *QoSComponentManager* which manages the adaptation for the services provided by the component. It contains a mapping table which encodes the relationship between the implementation levels of each of these services and their requirements. At runtime, there is only one QoSComponentManager for all QoSComponent instances.

At last, Qinna provides a single component named *QoSDomain* for the whole architecture. It manages all the service demands inside and outside the application. The client of a service asks the Domain for reservation of some implementation level and is eventually returned a contract if all constraints are satisfied. Then all service asks are made inside this contract.

C. Motivations of this work

The Qinna framework has shown its relevance for developping a video application which adapts itself to its network environnement ([4]). However, the specification of resources' constraints has not been done yet, and we aim to state which type of properties we are able to deal with. We will in particular narrow the following notions :

- The distinction between quality of resource and linking constraints. Indeed, some resource properties (for instance the fact that a service is called a unique time), are bound to components, while some others ("this client has the right to do 64 calls to this service") are bound to the link between components.
- The distinction between constraints and contracts. The former are maintained by Qinna processes whereas the latter have to be dynamically negotiated.

The global aim of our work is to propose a dynamic management of these concepts by generic algorithms, so that the designer only has to provide the specific characteristics of its application :

- The resource and linking constraints will be expressed in a user-friendly way, and both managements would be performed at runtime, through the use of classical formula monitors.
- There will be an automatic management of contracts (degradation, end of service) made by Qinna components. The designer will only provide the relative importance of all the contrats he asks for.

Qinna's objective is not only to bring resource maintenance algorithms, but also to permit the evaluation of the influence of the parameters on the behavior of the application. An automatic discovery of the "best" value for the parameters vector *with respect to some criteria* is out of the scope of this paper, but will be later studied.

III. Formalization of the Qinna framework

In this section, we propose a formalization of the Qinna framework, the assumptions we make on the component design, and the constraints we aim to tackle.

A. Quantity of Resource Constraints

Quantity of resource constraints, or QRC in the sequel, are quantitative constraints on components and/or the service they propose. They will be used to express and guarantee constraints on the total number of instances of a given component (type), or some constraints on the call occurrence sequence of a given service. These constraints, such as memory limits, are induced by the limited characteristics of embedded components.

We separate these properties into two categories, depending on their purpose :

- *The Component Type Constraints* (CTC) are properties common to all components of the same type, or quantitative properties on all allocated components of a same type. They are defined by a formula of the form :

$$CTC(\mathfrak{C}) \overset{def}{=} \bigwedge_i CTC_{serv}(s_i) \wedge CTC_{compo}(\mathfrak{C}).$$

  The subformula $CTC_{compo}(\mathfrak{C})$ expresses global properties of the component, such as the maximal number of its instances, or a global limitation on some resource used by the instances. The subformulas $CTC_{serv}(s_i)$ (s_i being a provided service of the component) express constraints on the s_i arguments and its call occurrence sequence. A CTC is then a conjunction of such subformula.
- The *Component Instance Constraints* (CIC) are of similar form :

$$CIC(\mathfrak{C}^j) \overset{def}{=} \bigwedge_i CIC_{serv}(s^i_j) \wedge CIC_{instance}(\mathfrak{C}^j),$$

  but they are linked to a particular instance $\mathfrak{C}^j$ of the component $\mathfrak{C}$.

Expression of quantity of resource constraints

Qinna has the objective to maintain the resource constraints at runtime. For that purpose, Qinna requires the following decision procedures :

- In the `QoSComponent`, for each service, Qinna requires two functions : `testCIC` and `updateCIC`. The former decides whether or not the call to the service can be performed, and the latter updates variables after the function call. In addition, there must be an initialization of the CICs formulas at the creation of each instance.

- Similarly, in the `QoSComponentBroker`, for each provided service, Qinna requires the two functions `testCTC` and `updateCTC`.

Qinna's dynamic behavior Qinna maintains resource constraints through the following procedure :

- When the Broker for $\mathfrak{C}$ is created, the parameters used in `testCTC` are set.
- The creation of an instance of $\mathfrak{C}$ is made by the Broker iff $CTC_{compo}(\mathfrak{C})$ is true. During the creation, the CIC parameters are set.
- The $CIC(s_i)$ and $CTC(s_i)$ decision procedures are invoked at each function call. A negative answer to one of these decision procedures will cause the failure of the current *contract*. We will detail the notion of contract in Section III-C. Roughly speaking, if a resource service cannot be performed with high level of quality, then some (eventually other) service may be degraded or stopped.

Remarks As we deal with embedded systems and these formulas are evaluated at runtime, we restrict ourselves to formulas that can be evaluated with limited memory. In particular, $CTC_{serv}(s_i)$ must only depend on the current call arguments and some finite fixed memory (one integer that computes the total of the resources used until this call, for instance). We must also be aware of the global complexity of the decision procedure. In addition, it is important to notice that some instance constraints and type constraints may be redundant. Coherence property of CIC and CTC resource constraints can be checked during pre-run time analysis.

In section IV, we will provide a way to express the resource properties in a specific logic in order to automatically generate the decision procedures.

Example The `Memory` component provides only one service `malloc`, which has only one parameter, the number of blocks to allocate. It has an integer attribute, `memory`, which denotes the global memory size and is set at the creation of each instance. We also suppose that we have no garbage collector, so the blocks are allocated only once. Figure 3 illustrates the difference between type and instance constraints.

- CTC for $\mathfrak{C}$ = Memory : the formula $CTC_{compo}(\mathfrak{C}) \equiv \sum_j memory(\mathfrak{C}^j) \leqslant 1024$ expresses that the global memory quantity for the whole application is 1024 kilobytes. A new instance will not be created if its `memory` constant is set to a too big number. Then $CTC_{serv}(malloc) \equiv \sum_k arg(occ_k(malloc)) \leqslant 1024$ forces the calls to `malloc` stop when all the 1024 kilobytes have been allocated.

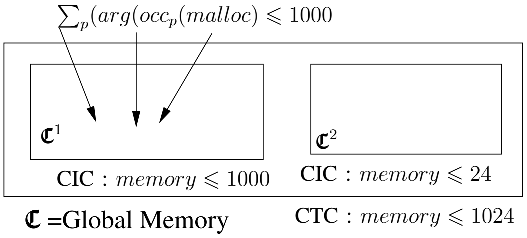

$\sum_p(arg(occ_p(malloc) \leqslant 1000$
$\mathfrak{C}^1$
$\mathfrak{C}^2$
CIC : $memory \leqslant 1000$
CIC : $memory \leqslant 24$
$\mathfrak{C}$ =Global Memory
CTC : $memory \leqslant 1024$

Fig. 3. Type versus Instance constraints

- CIC for Memory : if we want to allocate some Memory for a particular (group of) component(s), we can express similar properties in one particular instance (see $\mathfrak{C}^1$ on the Figure).

B. QoS Linking constraints (QLSC)

As for linking constraints, they express the relationship between components, in terms of quality of service. For instance, the following property is a linking constraint : " to provide the `getImages` at a "good" level of quality, the `ImageBuffer` component requires a "big" amount of memory and a "fast" network". This relationship between the different QoS of client and server services are called QoS Linking Service Constraints (QLSC).

Implementation level

To all provided services that can vary according to the desired QoS we associate an *implementation level*. This implementation level (IL) encodes which part of implementation to choose when supplying the service. For a given service, all its implementation levels are totally ordered.

As there is a finite number of implementation levels, we can restrict ourselves to the case of positive integers and suppose that implementation level 0 is the "best" level, 1 gives lesser quality of service, *etc.*

We also assume that required services for a given service doesn't change according to the implementation level, that is, the call graph of a given service is always the same. However, the arguments of the required services calls may change.

Linking constraints expression

Let us consider a component $\mathfrak{C}$ which provides a service s_1 that requires r_1 and r_2 services. Qinna permits to link the different implementation levels between callers and callees. The relationship between the different implementation levels can be viewed as a function which associates to each implementation level of s an implementation level for r_1 and for r_2 :

$$QLSC_{s_1} : \left| \begin{array}{ccc} \mathbb{N} & \longrightarrow & \mathbb{N}^2 \\ IL & \longmapsto & (IL_{r_1}, IL_{r_2}) \end{array} \right.$$

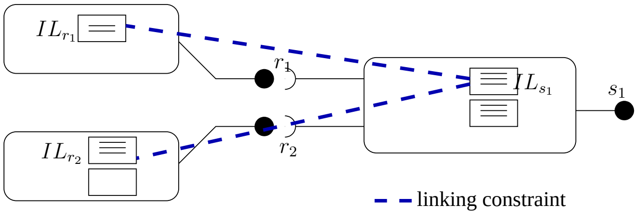


Fig. 4. Implementation Levels and Linking Constraints

This function is *statically encoded* by the developer within the application. For instance, it can easily be implemented in the QoSManager through a "mapping" table whose lines encode the tuples of linked implementation levels : $(IL_{s_1}, IL_{r_1}, IL_{r_2})$. The natural order of the lines of the table is used to determine which tuple to consider if the current negotiation fails.

Thus, as soon as an implementation level is set for the s_1 service, the implementation levels of all required services (and all the implementation levels in the call tree) are set. This has a consequence not only on the executed code of all the involved services (and also internal functions) but also on the arguments of the service calls.

Therefore, if a user asks for the service s_1 at some implementation level, the demand may fail due to some resource constraint. That's why every demand for a service must be negotiated and the notion of contract will be accurate to implement a set of a satisfactory implementation levels for (a set of) future calls.

Now we have all the elements to define the notion of contract.

C. Qinna's contracts

Qinna provides the notion of *contract* to ensure both resource constraints and linking constraints.

When a service call is made at some implementation level, all the subservices implementation levels are fixed implicitly through the linking constraints. As all the implementation levels for a same service are ordered, the aim is to find the best implementation level that is feasible (w.r.t. the resource constraints of all the components and service involved in the call tree).

Contract Negotiation

All service calls in Qinna are made after negotiation. The user (at toplevel) of the service asks for the service at some interval of "satisfactory" implementation levels. Qinna then is able to find the best implementation level in this interval that respects all the quantity of resource constraints (the resource constraints of all the services involved in the call tree). If there is no intersection between feasible and satisfactory implementation levels, no contract is built. A contract is thus a tuple $(id, s_i, IL, [IL_{min}, IL_{max}], imp)$ denoting respectively its identifiant number, the referred service, the current implementation level, the interval of satisfactory implementation levels, and also the *importance* of the contract. This last variable is used to sort the list of all current contracts and is used for degradation (see next paragraph). The importance value is statically set by the developer each time he asks for a new contract.

After contract initialization, all the service calls must respect the terms of the contract. In the other case, there will be some renegotiation.

Contract Maintenance and Degradation

After each service call the decision procedure for resource constraints are updated. Therefore, a contract may not be valid anymore. As all service calls are made through the Brokers by the Domain, the Domain is automatically notified of a contract failure. In this case, the domain tries to degrade the contract of least importance (which may be not the same as the current one). This degradation has consequences on the resource and thus can permit other service calls inside the first contract.

Basically, degrading a contract consists in setting a lesser implementation level among the satisfactory ones, but which is still feasible. If it is not possible, the contract is stopped.

It is important to notice that contract degradation is effective only at toplevel, and thus is performed by the Domain. It means that there is no degradation of implementation level outside toplevel. That is why we only speak of contract for service at toplevel.

IV. EDL IMPLEMENTATION OF QUALITY OF RESOURCE CONSTRAINTS

In this section, we focus on the expression and management of the resource constraints of components in Qinna. We use a a fragment of the event-based logic EDL (event definition language) introduced in [2] and compile it into sequential code for use in Qinna.

qMEDL syntax

Like in EDL, our variant is based on *events*, which are used to notify changes in the systems. Here we consider as event set $\mathcal{E}$ the set of all service calls (e_i denotes the call to the s_i function) and fabric calls ($new_{\mathfrak{C}}$ for the $\mathfrak{C}$ component). To each event e (or $new_{\mathfrak{C}}$) we associate the attributes $time(e)$ and $value_k(e)$ which give respectively the date of the last occurrence of the event and the k^{th} argument of the function call *when it occurs*.

We also use some auxiliary variables $v \in \mathcal{V}$, which are updated each time an event occurs. These events are defined in a separate way using events and their attributes : for instance, we can define the total number N of the arguments of the `malloc` function since the beginning (of the current contract) by : `malloc -> N:=N+value_1(malloc).`

Our formulas are the following constraints :

$$C ::= \ [E, E) \mid C\&\&C \mid C||C \ |Q \bowtie K$$

with K constant and $\bowtie \in \{\leqslant, =, <, \ldots\}$, and where E denotes events of the form :

$$\begin{aligned} E :: \ &= \ e \mid start(C) \mid end(C) \mid E \text{ when } C \\ &\mid \ E\&\&E \mid E||E \end{aligned}$$

and $Q ::= \ v \ |Q \ \square \ Q,$ with $\square \in \{+, -, *, /\}$.

Semantics To evaluate these constraints, we observe the logical states induced by the occurrence of the (monitored) events. At each of these states, we are able to evaluate boolean conditions on auxiliary variables ($Q \leqslant K$ is true when the numerical evaluation of Q is lesser or equal to K) ; similarly we have access to $value_k(e)$ and $time(e)$ each time a basic event occurs.

Then, $P, t \models C$ and $P, t \models E$ (P denotes the program) are recursively defined like in MEDL :

Base cases

- $M, t \models c_k$ if and only if c_k condition is true at the previous state/observation time.
- $M, t \models e_j$ if and only if the signal e_j occurs at time t.

Recurrence cases

- &&, $||$ have respectively and/or logical classic semantics.
- $start(C)$ is an event that occurs when condition C changes from `false` to `true`.
- E when C occurs when both C and E occur.
- condition $[E_1, E_2)$ is true at t when there exists a previous time t_0 where event E_1 occurs and for all $t_0 \leqslant t' \leqslant t$, E_2 does not occur.

Let us also point out the fact that some logical/timing properties are expressible in this logic, which is interesting for future work on this non-functional properties.

The Memory example The Memory constraints of Section III-A are easily expressible with the proposed logic : the constraint for the whole application is $N \leqslant 1024$ where N counts the total amount of malloc's arguments : `malloc -> N:=N+value_1(malloc).`

Translation into sequential code

We wrote in OCaml[1] a translator from qMEDL to C++. The translator (2000 LoC) takes as parameter a file which describes the QMEDL constraints for a component (respectively a component type) and for each variable service s (or call to `new(component)`) provides two C++ functions, `testCIC_s` and `updateCIC_s` (resp. `test_CTC` and `updateCTC`) to include in the QoSComponent (resp. the QoSBroker) code. The translation is straightforward, so we do not detail the procedure in this paper.

We could also have used the M^{2}IST toolsuite [2] in order to generate the C++ code, but the Charon2C++ translator is not yet available.

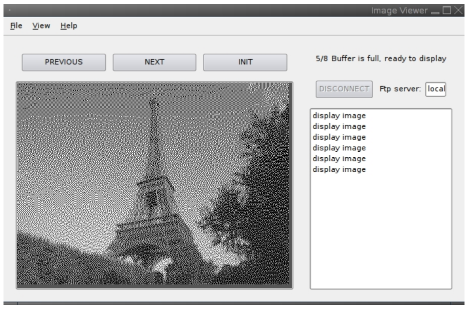
The translation of the basic above formula for memory gives the following procedures (the identifiers have been changed for lisibility, `usedmem` is a local variable to count the global amount of memory used yet) :

```
bool testCIC_malloc(int nbblocks){
 return (usedmem + nbblocks <= 1024)}
bool updateCIC_malloc(int nbblocks){
    usedmem = usedmem + nbblocks; }
```

V. Case study : image viewer

To show the feasability of the approach, we have applied the framework on a remote viewer application prototype (Figure 6). This integration of Qinna and the precise specifications of this case study are precisely described in [6] and [7]. Basically, the viewer uses a ftp connection to download files in a local buffer, and then the user can visualize them. We implemented this viewer using Qt[3], a C++ library which provides graphical components and used Qinna's C++ implementation for resource's management (CPU, Memory, Network). The architecture of the code is illustrated in figure 5.

Here we illustrate how we use Qinna for the maintenance of the application with respect to the different resource constraints.



5/8 Buffer is full, ready to display

Ftp server: local

display image
display image
display image
display image
display image
display image

Fig. 6. Screenshot of the viewer

Memory (resource and linking) constraints As we have implemented the viewer application in a high-level

[1] http://caml.inria.fr/index.en.html

[2] http://www.tricity.wsu.edu/ litan/tools/mist.html

[3] http://trolltech.com/products/qt/

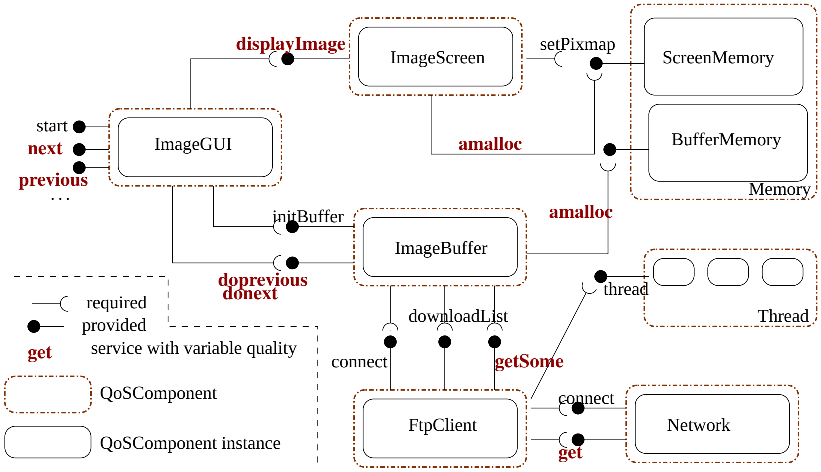


Fig. 5. Architecture of the application

language, we have no call to the real `malloc` function. The management of memory is thus simulated via a call to the `amalloc` function (provided by the Memory component) each time we encode a function that significantly needs memory. The implemented resource constraints for Memory mainly deal with local and total amount of memory for a (group of) component(s). The `ImageScreen` component is linked to Memory via its Manager, so that a shortage of memory influences the quality of the displayed image (see Figure 6). More details can be found in [6].

Results

As soon as all the decision functions and linking constraints are encoded, Qinna's generic components and algorithms are able to maintain the resource, provided all functions calls are made through an existing contract. For instance, Figure 7 shows that the current choice of implementation levels suits well the global amount of available memory.

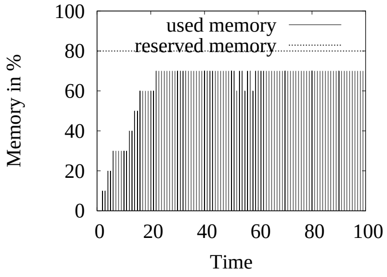


Fig. 7. Memory use

Thus the implementation provides a first formal tool to control resource consumption. ”End-User QoS” control policies could then use the results of such approach to build efficient architectures : find a correct (from resource usage) implementation level mapping, for a certain scenario, which gives a correct end-user Quality of Service level.

Technical overhead

The viewer is written in 4350 lines of code, the functional part taking roughly 1800 lines. The other lines are Qinna's generic components (1650 loc.), 600 lines of code for the new components (imagescreenBroker, imageScreenManager *etc.*) and 300 lines of code for the test scenarios. The binary is also much bigger 4.7Mbytes versus 2Mbytes without Qinna.

Thus Qinna is costly, but all the supplementary lines of code do not need to be rewritten, because:

- Generic Qinna components, algorithms, and the basic resource components are provided with Qinna.
- The decision functions for Quality of service constraints could be automatically generated or be provided as a “library of common constraints”.
- The initialization at toplevel could be computed-aided through user-friendly tables.

We think that the cost of Qinna in terms of binary code can be strongly reduced by avoiding the existing redundancy in our current implementation.

VI. Related works and conclusion

Related works

The expression of resource constraints is considered as a fundamental issue, so much work deal with this topic. For instance the QML language ([8]) is now well used to express quality of service properties. We also could have used this syntax for our resource properties, but QML mainly focuses on the probabilistic point of view of the

QoS variables. We chose to use a lesser logic w.r.t. the expression power, mainly to simplify the over cost for the developer.

Some other works in the domain of verification try to prove conformance of one program to some specifications : in [9], the authors use synchronous observers to encode and verify logical time contracts. In [10], the author generates an event recognizer and a monitor and uses them to detect bad behaviors at runtime. Our approach while using similar techniques is slightly different from the notion of *observers*. While observers techniques aim to emit warnings when some (safety) property is violated, our method catches some internal variables of the systems and constraints the program (automata) to ensure properties.

In the area of component-based software engineering, some other works develop the idea of using software architecture to adress automatic reservation and automatic control for variable resource. In [11], the author use a distributed architecture in order to dynamically manage network resource. Like Qinna, the tool uses decision procedures written by the developer to manage resource at runtime ; however there is no notion of order between varying pieces of code.

Conclusion

In this paper, we have presented a formalization of a QoS Component-Based architecture, and illustrated how to use this framework to integrate implementation variability like limited resources. The first experiments show that the implementation of the resource constraints and contracts is very effective, and that Qinna effectively manages them at runtime.

Future work involves both theoretical and implementation issues :

- development of generic Qinna components for use within the Fractal/Think ([12], [13]) component-based design of embedded software.
- automatic discovery of the “best” linking constraints (w.r.t. some criteria).
- efficient implementation of Qinna framework

References

[1] S. Becker, L. Grunske, R. Mirandola, and S. Overhage, “Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective,” in *Architecting Systems with Trustworthy Components*, ser. LNCS. Springer, 2006, vol. 3938.

[2] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, “Runtime assurance based on formal specifications,” in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (IPDPS’99)*, 1999.

[3] M. Sparling, “Lessons learned through six years of component-based development,” *Commun. ACM*, vol. 43, no. 10, 2000.

[4] J.-C. Tournier, “Qinna: une architecture à base de composants pour la gestion de la qualité de service dans les systèmes embarqués mobiles,” Ph.D. dissertation, INSA-Lyon, 2005.

[5] J.-C. Tournier, J.-P. Babau, and V. Olive, “Qinna, a component-based qos architecture,” in *Component-Based Software Engineering, 8th International Symposium*, St. Louis, USA, May 2005.

[6] L. Gonnord and J.-P. Babau, “Resource management with Qinna framework : the remote viewer case study,” INRIA, Research Report 6562, Jun. 2008. [Online]. Available: https://hal.inria.fr/inria-00288593

[7] ——, “Runtime resource assurance and adaptation with Qinna framework : a case study,” in *Real Time Software, RTS’08*, Wisla, Poland, Aug. 2008.

[8] S. Frølund and J. Koistinen, “Quality of services specification in distributed object systems design,” in *Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. Berkeley, CA, USA: USENIX Association, 1998.

[9] F. Maraninchi and L. Morel, “Logical-time contracts for reactive embedded components,” in *30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE’04*, Rennes, France, Aug. 2004.

[10] L. Tan, J. Kim, I. Lee, and O. Sokolsky, “Model-based testing and monitoring for hybrid embedded systems,” in *Proceedings of IEEE International Conference on Information Reuse and Integration (IRI’04)*, 2004.

[11] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy, “A distributed resource management architecture that supports advance reservations and co-allocation,” in *In Proceedings of the International Workshop on Quality of Service*, 1999, pp. 27–36.

[12] E. Bruneton, T. Coupaye, and J. Stefani, “Recursive and dynamic software composition with sharing,” in *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP’02)*, 2002.

[13] J.-Ph. Fassino, J.-B. Stefani, J. Lawall, and G. Muller, “THINK: A software framework for component-based operating system kernels,” in *Proceedings of Usenix Annual Technical Conference*, Monterey (USA), Jun. 2002.