Scalable Computing: Practice and Experience
Volume 10, Number 3, pp. 253–264. http://www.scpe.org
ISSN 1895-1767

QINNA: A COMPONENT-BASED FRAMEWORK FOR RUNTIME SAFE RESOURCE ADAPTATION OF EMBEDDED SYSTEMS

LAURE GONNORD* AND JEAN-PHILIPPE BABAU†

Abstract. Even if hardware improvements have increased the performance of embedded systems in the last years, resource problems are still acute. The persisting problem is the constantly growing complexity of systems, which increase the need for reusable developement framework and pieces of code. In the case of PDAs and smartphones, in addition to classical needs (safety, security), developers must deal with quality of service (QoS) constraints, such as resource management.

Qinna was designed to face with these problems. In this paper, we propose a complete framework to express ressource constraints during the developpement process. We propose a component-based architecture, which generic components and algorithms, and a developpement methodology, to manage QoS issues while developing an embedded software. The obtained software is then able to automatically adapt its behaviour to the physical resources, thanks to "degraded modes". We illustrate the methodology and the use of Qinna within a case study.

Key words: component, software architecture, resource dynamic management, case study.

1. Introduction.

When faced to the problem of designing handled embedded systems, the developer must be aware of the management of limited physical resources (CPU, Memory).

In order to develop multimedia software on such systems where the quality of the resource (network, battery) can vary during use, the developer needs tools to:

- easily add/remove functionality (services) during compilation or at runtime;
- adapt component functionality to resources, namely propose "degraded" modes where resources are low;
- evaluate the software's performances: quality of provided services, consumption rate *for some scenarios*.

In this context, component-based software engineering appears as a promising solution for the development of such kinds of systems. Indeed it offers an easier way to build complex systems from base components ([9]), and the management of physical resource can be done by embedding the system calls in high level components. The main advantages thus appear to be the re-usability of code and also the flexibility of such systems.

The Qinna framework ([11, 12, 3]) was designed to handle the specification and management of resource constraints problems during the component-based system development. Variability is encoded into discrete implementation levels and links between them. Quantity of resource constraints can also be encoded. Qinna provides algorithms to ensure resource constraints and dynamically adapt the implementation levels according to resource availability *at runtime*. The main advantage of the method is then the reusability of the resource components and the generic adaptation algorithms.

In this journal paper, we propose a complete formalization of Qinna framework (algorithms and components), and as proof of concept, a case study consisting of the development of a remote viewer application with the help of Qinna's implementation in C++. In Section 2 we recall Qinna's main concepts, as introduced in [11] and formalized later in [3]. In Section 3, we give an overview of Qinna's C++ implementation, and then provide the general implementation steps to develop a resource-aware application with Qinna in Section 4. Finally we illustrate the whole framework on the viewer case study (Section 5).

2. Description of the Qinna framework.

2.1. Qinna's main concepts.

The framework designed in [11] and [12], and further formalized in [3] has the following characteristics:

- Both the application pieces of code and the resource are components. The resource services are enclosed in components like `Memory`, `CPU`, `Thread`.
- The variation of quality 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.
- The link between the implementation levels is made through an explicit relation between the implementation level of the provided service and the implementation levels of the services it requires. For instance, the developer can express that a video component provides an image with highest quality when it has enough memory and sufficient bandwidth.

*Université of Lille, LIFL `Laure.Gonnord@lifl.fr`

†UBO, LISyC, Université Européenne de Bretagne `Jean.Philippe.Babau@univ-brest.fr` This work has been partially supported by the REVE project of the French National Agency for Research (ANR)

- All the calls to a "variable function" are made through an existing contract that is negotiated. This negotiation is made automatically through the Qinna components. A *contract* for a service at some objective implementation level is made only if all its requirements can be reserved at the corresponding implementation levels and also satisfy some constraints called Quality of resource constraints (QoR). If it not the case, the negotiation fails.

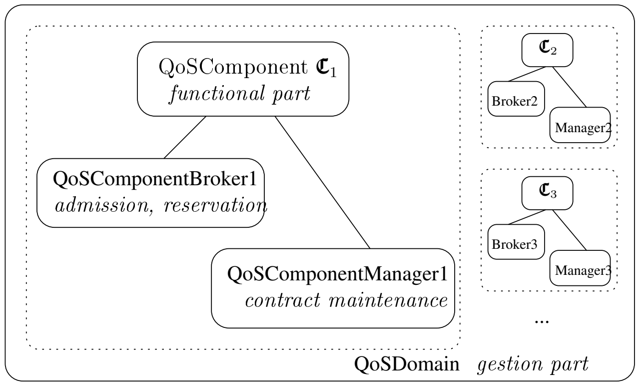

Fig. 2.1. *Architecture example*

These characteristics are implemented through new components which are illustrated in Figure 2.1: to each application component (or group of components) which provide one or more variable service Qinna associates a *QoSComponent* $\mathfrak{C}_i$. The variability of a variable service is made through the use of a corresponding `implementation level` variable. 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 realize the admission of a component. The Broker decides whether or not a new instance can be created, and if a service call can be performed w.r.t. the quantity of resource constraints (QoR).
- a *QoSComponentManager* which manages the adaptation for the services provided by the component. It contains a mapping table which encode the relationship between the implementation levels of each of these services and their requirements.

At last, Qinna provides a single component named *QoSDomain* for the whole architecture. It manages all the service requests 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, after each service request, the Domain makes an acknowledgment only of the corresponding contract is still valid.

2.2. Quantity of Resource constraints in Qinna. A Quantity of resource constraint (QRC) is a quantitative constraint on a component $\mathfrak{C}$ and the service (s_i) it proposes. QRCs are for instance formula on the total instance of a given component type, of the total amount of resource (memory, CPU) allocated to a given component. They are two types of constraints, depending on their purpose:

- Component type constraints (CTC) express properties of components of the same type and their provided services.
- Component instance constraints (CIC) express properties of a particular instance of a component.

The management of these constraints is automatically done at runtime, if the developer implements them in the following way:

- In the `QoSComponent`, for each service, implement the two functions: `testCIC` and `updateCIC`. The former decides whether or not the call to the service can be performed, and the later 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, implement the two functions `testCTC` and `updateCTC`.

Then, Qinna maintains resource constraints at runtime 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 2.4.

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 2.2 illustrates the difference between type and instance constraints.

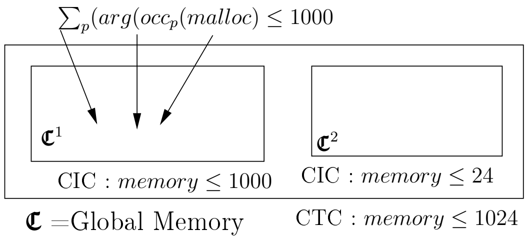

FIG. 2.2. *Type vs Instance constraints*

- CTC for $\mathfrak{C}$ = Memory: the formula $CTC_{compo}(\mathfrak{C}) \equiv \sum_j memory(\mathfrak{C}^j) \leq 1024$ 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 $CTC_{serv}(malloc) \equiv \sum_k arg(occ_k(malloc)) \leq 1024$ forces the calls to `malloc` stop when all the 1024 kilobytes have been allocated.
- 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).

Expression of resource constraints and code generation

Qinna also provides a way to describe the resource constraints into a higher-lever language called qMEDL, a variant of MEDL event logic described in [6], and whose precise syntax and semantics is described in [3]. Roughly speaking, the logic can express boolean formulae on occurences of *events*. Atoms are of the form $Q \bowtie K$, with K constant and $\bowtie \in \{\leqslant, =, <, \ldots\}$, and Q is a quantity. The quantity are obtained by the use of auxiliary variables and calls to `value` and `time` special functions: to each event e (or $new_{\mathfrak{C}}$), $time(e)$ and $value_k(e)$ give respectively the date of the last occurrence of the event and the k^{th} argument of the function call *when it occurs*.

The Memory constraint for the whole application then can be encoded by $N \leq 1024$ where N counts the total amount of malloc's arguments: `malloc -> N:=N+value_1(malloc)`. The translation is then made by the qMEDL2 to C++ translator, and 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; }
```

2.3. QoS Linking constraints. Unlike quality of resource constraints, linking constraints 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. These implementation levels are totally ordered for a given service. As these implementation levels are finitely many, we can restrict ourselves to the case of positive integers and suppose that implementation level 0 is the "best" level, 1 gives a lesser quality of service, and so on.

We 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 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:

$$QLSC_s : \left| \begin{array}{ccc} \mathbb{N} & \longrightarrow & \mathbb{N}^2 \\ IL & \longmapsto & (IL_1, IL_2) \end{array} \right.$$

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.

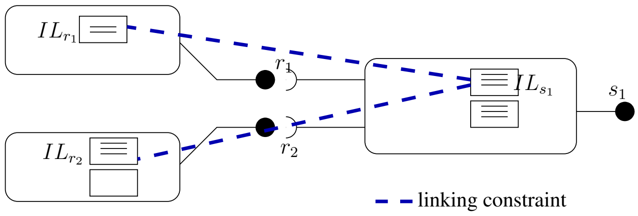

Fig. 2.3. *Implementation levels and linking constraints*

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 (Figure 2.3). 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.

Implementation of linking constraints in Qinna The links between the provided QoS and the QoS of the required services are made through a table whose lines encode the tuples of linked implementation levels: $(IL_s, IL_{r_1}, IL_{r_2})$. This "mapping" table is encoded in the QoSManager. The natural order of the lines of the table is used to determine which tuple to consider if the current negotiation fails.

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

2.4. Qinna's contracts. Qinna provides the notion of *contract* to ensure both behavioral constraints (Type Constraints and Intance Constraints of services, as described in Section 2.2) and linking constraints.

When a service call is made at some implementation level, all the subservices implementation level are fixed implicitly through the linking constraints. As all the implementation levels for a same service are ordered, the objective is to find the best implementation level that is feasible (w.r.t. the behavioral 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 behavioral constraints (CICs and CTCs of all the services involved in the call tree). If there is no intersection between feasible and satisfactory implementation levels, no contract is built. In the other case, a contract is made for the specific service. 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 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 behavioral constraints are updated. After that, 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.

Use of services Each call to a service at toplevel as consequences on the contract which has been negociated for him. We suppose that a contract is made before the first invocation of the desired service. The verification could automatically be done with Qinna, but is not not yet implemented. All the notifications of failures are logged for the developer.

3. Qinna's components implementation in C++. We implemented in C++ the Qinna components and algorithms. These components are provided through classes which we detail in this section.

3.1. Qinna's components for the management of services. QoSComponent The QoSComponent class provides generic constructors and destructors, and contains a private structure to save the current implementation levels of the component provided service. All QoS components will inherit from this class.

QoSBroker The QoSBroker class contains a private structure to save the references to all the corresponding components it is responsible for. It provides the two functions `Free(QoSComponent* refQc)` and `Reserve(...)`. As `testCIC` and `updateCIC` functions signature depends of each component/service, these functions will be provided in each instance of QoSBroker.

QoSManager The QoSManager class contains all information for the service provided by its associated component. It provide the following public functions:

- `bool SetServiceInfos(int idserv, QoSComponent *compo, int nbreq, int nbmap)` initializes the manager for the *idserv* service, provided by **compo*, with *nbreq* required services and *nbmap* different implementation levels. Return `true` if successful, `false` otherwise.
- `bool AddLevQoSReq(int idserv, int lv, int irq, int lrq)` adds the tuple (lv, irq, lrq) (the lv implementation level for *idserv* is linked to the lrq implementation level for irq service) in the mapping table for *idserv*.
- `int Reserve(int idserv, int lv)` is used for the reservation of the *idserv* service at level il. It returns the local number of (sub) contract of the Manager or 0 if the reservation has failed (due to resource constraints).

QoSDomain The QoSDomain class provides functions for managing contracts at toplevel:

- `bool AddService(int service, int nbRq, int nbMp, QoSManager *qm)` adds the service *service* with *nbRq* required services and *nbMp* implementation levels, with associated manager $*qm$.
- `int Reserve(QoSComponent *compo,int ns , int lv, int imp)` is used for reservation of the service ns provided by the component $*compo$ at level lv and importance imp. it returns the number of contract (in domain) if successful, 0 otherwise.
- `bool Free(int id)` frees the contract number id (of domain).

ManagerContract This class provides a generic structure for a subcontract which encodes a tuple of the form $< id, lv, *rq, v >$ where id is the contract number, lv the current level, rq is the component that provides the service and v is a C++-vector that encode the levels of the required services. This class provides access functions to these variables and a function to change the implementation level.

DomainContract This class provides a structure for contracts at toplevel. A Domain contract is a tuple of the form $< di, i, lv, *rq >$ where di is the global identifier of the contract, $*rq$ is the manager associated to the component that provides the service, i is the local number of subcontract for the manager, and lv is the current level of the service.

Remark 1 *All services and contracts have global identifiers used in toplevel. However, it is important to notice that service and (sub) contracts have local identifiers in their respective managers.*

3.2. Basic resource components. In the call graph of one service, leaves are physical resources (Memory, CPU, Network). As all resources must be encapsulated inside components, we need to encapsulate the base functions into QoSComponents. For instance, the `Memory` component must be encoded as a wrapper around the `malloc` function, and the associated broker basically implements the CIC functions which decide if the global amount of allocated memory is reached or not.

Sometimes, the basic functions are encapsulated in higher level components. For instance, a high level library might provide a `DisplayImage` function which makes an explicit call to `malloc`, but this call is hidden by the use of the library. In this particular case, the management of basic resource functions can be done in two different but equivalent ways:

- the creation of a "phantom" Memory component which provides the two services `amalloc` (for abstract malloc) and `afree`. Each time the developer makes a call to an "implicit" resource function (*i. e.* when the called function needs a significant amount of memory, like `DisplayImage`), he has to call `Memory.amallloc`. The Qinna's C++ implementation provides some basic components like Memory, Network and CPU and their associated brokers.
- the creation of QoSComponent around the library function `DisplayImage` which is responsible (through its broker) for the global amount of "quantity of resource" used for the `DisplayImage` function.

Both solutions need a precise knowledge of the libraries functions w.r.t the resource consumption. We assume that the developer has this knowledge since he designs a resource-aware application. In our case study we used the first solution.

4. Methodology to use Qinna. We suppose that in the application all resources, including hardware resources (Memory, CPU) or software ones (viewer, buffer), are encoded by components. Here are the main steps for integrating Qinna into an existing application designed in C++:

1. **Identify the variable services** which are functions whose call may fail due to some resource reasons. They are of two types:
   - simple functions like `Memory.malloc` whose code does not vary. They have a unique implementation level.
   - "adaptive" functions whose code can vary according to implementation levels.

   The first step is thus to identify the services whose quality vary and associate to each of this services a *unique* key, and if the code vary, clearly identify the variant code through a code of the form:
   ```
   switch(implLevel)
      {
       case 0 :
          ...
      }
   ```
   where implLevel is the associated (variable) attribute of the host component for this service. We must identify which variable services are required for each provided service, and the relationship between the different implementation levels.
2. **Create Qinna components**. First, cut the source code into QoSComponents that can provide one or more QoSservices. As the QoS negotiation will only be made between QoSComponents of different types, this split will have many consequences on the QoS management. For each QoSComponentC (which inherits from the `QoSComponent` class), the designer must encode two classes: `QoSBrokerC` and `QoSManagerC` which respectively inherit from the `QoSBroker` and `QoSManager` generic classes. For the whole application, the designer will directly use the `QoSDomain` generic class.
3. **Implement Quality of Resource constraints**. These constraints are set in two different ways:
   - The type constraints (CTC) for component C implementation is composed of additional functions in *QoSBrokerC* : `initCTC` which is executed at the creation of the Broker, and which sets the decision procedures parameters ; a `testCTC` function to determine whether a new instance can be created or not ; an `updateCTC` to save modifications of the resources after the creation. For each provided QoS service s_i, we add to new functions: `testCTC(idsi)` which is executed before the call of a service and tells if the service can be done, and `updateCTC(idsi)` to be executed after the call.
   - The instance constraints (CIC) for C are also composed of three functions to encode in the *QoSComponentC*: `setCIC` to set the resources constants, `testCTC(idsi)` which is used to de-

cide if a service of identifiant `ids` can be called, and `updateCTC(idsi)` to update the resource constraints after a call to the s_i function.

4. **Implement the linking constraints**. The links between required services and provided service via implementation levels are set by the invocation of the `SetService` and `AddLevQoSReq` functions of the managers. These functions will be invoked at toplevel.
5. **Modify the main file to initialize Qinna components at toplevel**. Here are the main steps:
   - For each base resource (CPU, Memory, ...)
     (a) Invoke the constructor for the associated Broker. The constructor's arguments must contain the initialization of internal variables for type constraints (the total amount of memory for example).
     (b) Create the associated Manager with the Broker as argument.
     (c) Register the QoS services inside the Manager through the use of `SetServiceInfos` function.
     (d) Create QoSComponents instances via the `Broker.reserve(...)` function. The arguments can be a certain amount of resource used by the component.
   - For all the other QoSComponents, the required components first:
     (a) Create the associated Broker and Manager.
     (b) Set the services information.
     (c) If a service requires another service of another component, use the function `Manager.AddReq` to link the required manager. Then use `Manager.AddLevQoSReq` to set the linking constraints.
     (d) Create QoSComponent instances by invoking the corresponding reservation function (`Broker.Reserve`).
   - Create the QoSDomain and add the services that are used at toplevel (`Domain.AddService`)
   - Reserve services via the QoSDomain and save the contracts' numbers.

5. Viewer Implementation using Qinna.

5.1. Specification. Our case study is a remote viewer application whose high level specification follows:

- The system is composed of a mobile phone and a remote server. The application allows the downloading and the visualization of remote images via a wireless link.
- The remote directory is reached via a ftp connection. After connection, two buttons "Next" and "Previous" are used to display images one by one. Locally, some images are stored in a buffer. To provide a better quality of service, some images are downloaded in advance, while the oldest ones are removed from the photo memory.
- The application must manage different qualities of services for the resources: shortage of bandwidth and memory, or disconnections of the ftp server. When needed it can download images in lower quality (in size or image compression rate).
- Different storage policies are possible, and there are many parameters which can be modified; like the size of the buffer, or the number of images that are downloaded each time. We want to evaluate which policy is the best *according to a given scenario.*

We want to use Qinna for two objectives:

- the maintenance of the application with respect to the different qualities of service,
- the evaluation of the influence of the parameters, on the non-functional behavior (timing performance and resource usage).

5.2. The functional part. The functional part of the viewer is developed with Qt[1] (a C++ library which provides graphical components and implementations of the ftp protocol). Figure 5.2 describes the main parts of the standalone application. We chose to make the downloading part via the ftp protocol. The wireless part is not encoded.

- The `FtpClient` class makes a connection to an existing ftp server and has a list of all distant images. It provides a `getSome` function to enable the downloading of many files at once.
- The `ImageBuffer` class is responsible for the management of downloaded files in a local directory. As the specification says, this buffer has a limited size and different policy for downloading images. The class provides the two functions `donext` and `doprevious` which are asynchronous functions. A signal

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

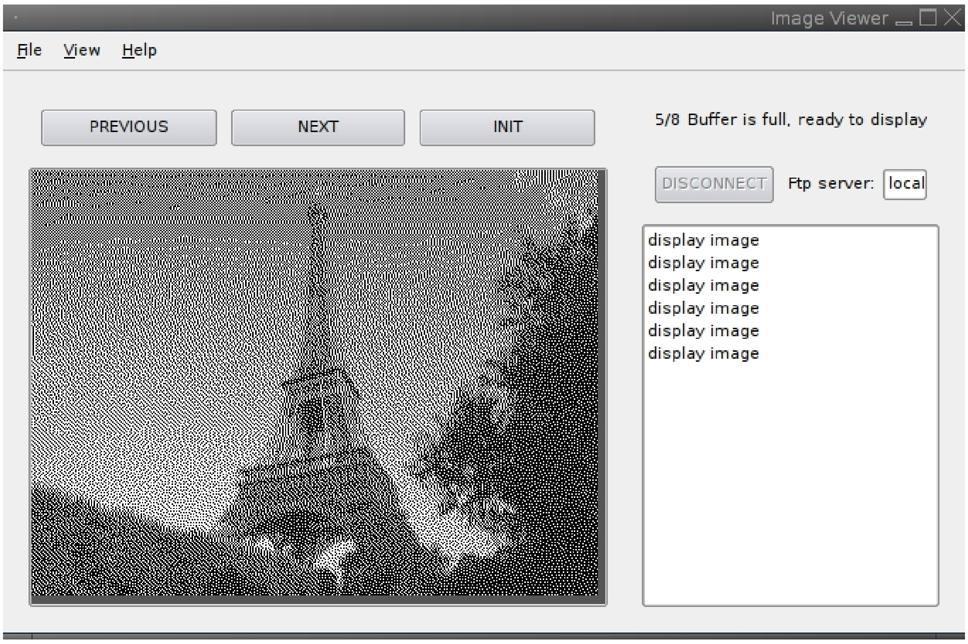

FIG. 5.1. *Screenshot of the viewer application*

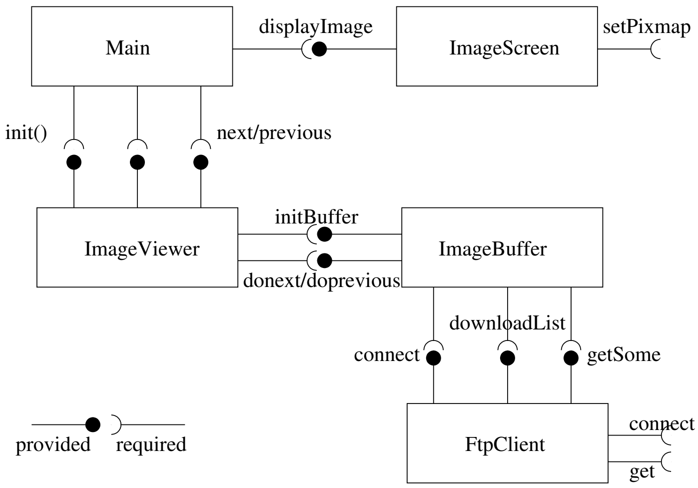

FIG. 5.2. *Functional view of the application*

is thrown if/when the desired image is ready to be displayed. It eventually downloads future images in current directory.

- The `ImageViewer` class is a high level component to make the interface between the ftp and buffer classes to the graphics components.
- The `ImageScreen` class is responsible for the display of the image in a graphic component named `QPixmap`.
- The `main` class provides all the graphics components for the Graphical User Interface.

5.3. Integration of Qinna. Now that we have the functional part of the application, we add the following resource components: Memory, and Network which are QoSComponents that provide variable services. We only focus on these two basic resources. The Network component is only linked to the FtpClient, whereas Memory will be shared between all components. For Memory, the only variable service is `amalloc` which can fail if the global amount of dedicated memory is reached ; this function has only one implementation level. For Network, the provided function `get` can fail if there is too much activity on network (notion of bandwidth).

Then we follow the above methodology in the particular case of our remote viewer.

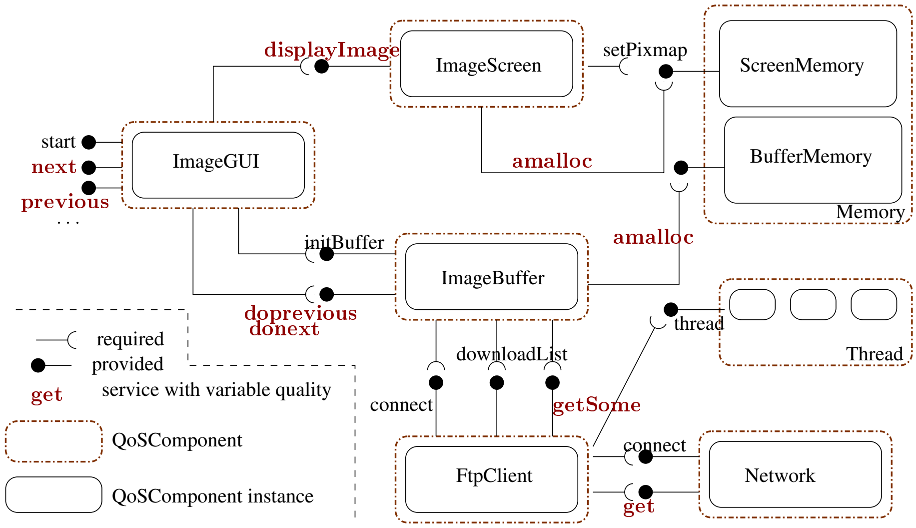

Fig. 5.3. *Architecture example*

Identification of the variable services (step 1)

Now as the variable services for low level components have been identified, we list the following adaptive services for the functional part:

- `ImageScreen.displayImage` varies among memory, it has three implementation levels which correspond to the quality of the displayed image. We add calls to `Memory.amalloc` function to simulate the use of Memory.
- `Ftpclient.getsome`'s implementation varies among available memory and the current bandwidth of network. If there is not enough memory or network, it adapts the policy of the downloads. It has three implementation levels. We add calls to `Network.bandwidth` to simulate the network resources that are needed to download files.
- `ImageBuffer.donext/previous` varies among available memory: if there is not enough memory the image is saved with high compression.

Creation of the QoSComponents (step 2)

The resource components are QoSComponents. Then, the three components `ImageScreen`, `FtpClient` and `ImageBuffer` are QoSComponents which provide each one variable service. `Imageviewer` and `Main` are QoSComponents as well. Figure 5.3 represents now the structure of the application at this step.

For the sake of simplicity, we only share Memory into two parts, a part for `ImageBuffer` and the other part for `imageBuffer`. That means that each of these components have their own amount of memory.

Resource constraints (steps 3 and 4)

The quantity of resource constraints we have fixed are classical ones (bounds for the memory instances, unique instantiation for the `imageScreen` component, no more than 80 percent of bandwidth for the ftpClient, etc). The QLSC are very similar to those described in [11] for a videogame application. Here we show how we have implemented some of these constraints in our application.

- *Quantity of resource constraints* The `imageScreen` component is responsible for the unique service `display_image` (display the image on the graphic video widget). Here are some behavioral constraints we implemented for this component:
  - There is only one instance of the component once.
  - The display function can only display images with size lesser or equal to $1200 * 800$.
  - There is only one call to the display function once.

  These type constraints are easily implemented in the associated Broker (`imageScreenBroker`) in the following way: the constraint "maximum of instance" requires two private attributes `nbinstance` and `nbinstancemax` which are declared and initialized at the creation of the Broker with values 0 and 1. Then the reservation of a new `imageScreen` by the Broker is done after checking whether or not $nbinstance + 1 \leq nbinstancemax$. If all checks are true, it reserves the instance and increments `nbinstance`.

The checking of memory is done by setting the global amount of memory for `ImageBuffer` and `imageBuffer` in local variables which are set to 0 at the beginning of each contract, and updated each time the function `amalloc` is called.
These constraints are rather simple but we can imagine more complex ones, provided they can be checked with bounded complexity (this is a constraint coming from the fact the Qinna components will also be embedded).

- *QoS Linking constraints*
  To illustrate the difference between quality of resource constraints and linking constraints, we show here the constraints for the `FtpClient.getSome`:
  - The implementation level 0 corresponds to 3 successive downloads with the `Network.get` function. The function has a unique implementation level but each call to it is made with 60 as argument, to model the fact it requires 60% of the total bandwidth. These three calls are made through the use of the `Thread.thread` with implementation level 0 (quick thread, no active wait).
  - The implementation level 1 corresponds to 2 calls to the `get` function with 40% of bandwidth each time. These two calls are made through the use of the `Thread.thread` with implementation level 1 (middle thread, few active wait).
  - The implementation level 2 corresponds to 1 call to the `get` function with 20% bandwidth. This call is made through the use of the `Thread.thread` with implementation level 2 (more active wait).

Thus if the available bandwidth is too low, a negotiation or an existing contract will fail because of the resource constraints. The creation of the contract may fail because a thread cannot be provided at the desired implementation level.

Modification of toplevel (step 5) This part is straightforward. The only choices we have to make are the relative amount of resource (Memory, Network) which are allocated to each QoSComponents. The test scenario is detailed in section 5.5.

5.4. Some statistics. 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.

Moreover, Qinna's implementation can be viewed as a prototype to evaluate the resource use and the quality of service management. After a preliminary phase with the whole implementation used to find the best linking constraints, we can imagine an optimized compilation through glue code which neither includes brokers nor managers.

5.5. Results. We realized a scenario with a new component whose only objective is to use the basic resources Memory and Network. This `TestC` component provides only the `foobar` function at toplevel. This function has two implementation levels, and requires two functions: `ScreenMemory.amalloc` and `Network.get`. The whole application provides four functions at toplevel: `TestC.foobar`, `ImageViewer.donext` (and `doprevious`) and `ImageScreen.displayimage`. Three contracts are negotiated, in the following importance order: `foobar` first, then `donext` and `doprevious`, then `displayimage`. We made the three contracts and download and visualize images at the highest qualities, but at some point the foobar function causes the degradation of the contract for displayimage, and the images are then shown in a degraded version, like the Eiffel tower on Figure 5.1.

The gap between the characteristics of the contract and the effective resource usage can be make through the use of log functions provided by the Qinna implementation. Figure 5.4 shows for instance the memory usage for another played scenario.

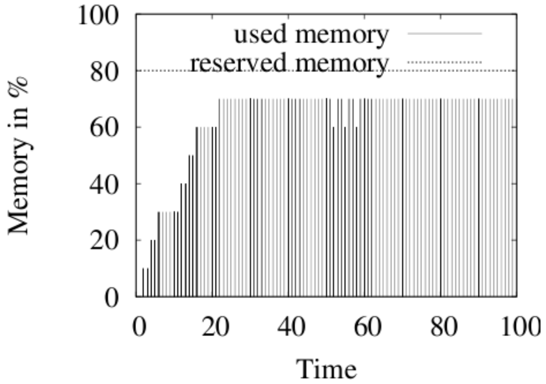

FIG. 5.4. *Memory use*

6. Related works. Other works also propose to use a development framework to handle resource variability. In [10] and [6], the author propose a model-based framework for developping self-adaptative programs. This approach uses high-level specifications based on temporal logic formula to generate program monitors. At runtime, these monitors catch the system events and activates the reconfiguration. This approach is similar to us except that it mainly deals with hybrid automata and there is no notion of contract degradation nor generic algorithm for negociation.

The expression and maintenance of resource constraints is also considered as a fundamental issue, so much work deals with this subject. In [5], the author use a probabilistic approach to evaluate the resource consumed by the program paths. Some other works in the domain of verification try to prove conformance of one program to some specification: in [7], for instance, the authors use synchronous observers to encode and verify logical time contracts. At last, the QML language ([2], [1]) is now well used to express QoS properties. This last approach is complementary to our one since it provides a language which could be compiled into source code for QoSComponents or Brokers.

7. Conclusion and future work. In this paper, we have presented a case study using the software architecture Qinna which was designed to handle resource constraints during the development and the execution of embedded programs. We focused mainly on the development part, by giving a general development scheme to use Qinna, and illustrating it on a case study. The resulting application is a resource-aware application, whose resources constraints are guaranteed at runtime, and whose adaptation to variability of service is automatically done by the Qinna components, through the notion of contracts. At last, we are able to evaluate at runtime the threshold between contractualised resource and the real amount of resource effectively used.

This work has shown the effectivity of Qinna with respect to the programming effort, and the performance of the modified application.

Future work include some improvements of Qinna's C++ components, mainly on data structures, in order to decrease the global cost of Qinna in terms of binary size, and more specific and detailed resource components, in order to better fit to the platform specifications. Integrating Qinna into a model driven development tools, such as Gaspard ([8]), can be a way to improve this efficiency.

From the theoretical point of view, there is also a need for a way to manage the linking constraints. The developer has still to link the implementation levels of required and provided services, and the order between all implementations levels is fixed by him as well. The tuning of all these links can only be done though simulation yet. We think that some methods like controller synthesis ([4]) could be used to discover the/a optimal order and linking relations w.r.t. some constraints such as "minimal variability", "best reactivity" *etc.*.

Finally, some theoretical work would be necessary in order to use Qinna as a prediction tool, and provide an efficient compilation into "glue code".

REFERENCES

[1] S. FRØLUND AND J. KOISTINEN, *Qml : A language for quality of service specification*, tech. rep., HPL-98-10, 1998.
[2] ———, *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, 1998, USENIX Association.

[3] L. Gonnord and J.-P. Babau, *Quantity of Resource Properties Expression and Runtime Assurance for Embedded Systems*, in ACS/IEEE International Conference on Computer Systems and Applications, AICCSA'09, Rabbat, Morocco, May 2009, pp. 428–435.

[4] F. M. K. Altisen, A. Clodic and E. Rutten, *Using controller synthesis to build property-enforcing layers*, in European Symposium on Programming (ESOP), April 2003.

[5] H. Koziolek and V. Firus, *Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation*, in Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA), Electronical Notes in Computer Science, Vienna, Austria, 2006.

[6] 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.

[7] 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.

[8] I.-R. Quadri, S. Meftali, and J.-L. Dekeyser, *An mde approach for implementing partial dynamic reconfiguration in fpgas*, in Proceedings of the 16th International Conference on IP-Based System-on-chip, Grenoble, France, 2007.

[9] M. Sparling, *Lessons learned through six years of component-based development*, Commun. ACM, 43 (2000).

[10] L. Tan, *Model-based self-monitoring embedded systems with temporal logic specifications*, in Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05), 2005.

[11] J.-C. Tournier, *Qinna: une architecture à base de composants pour la gestion de la qualité de service dans les systèmes embarqués mobiles*, PhD thesis, INSA-Lyon, 2005.

[12] J.-C. Tournier, V. Olive, and J.-P. Babau, *Towards a dynamic management of QoS constraints in embedded systems*, in Workshop QoSCBSE, in conjunction with ADA'03, Toulouse, France, June 2003.

Edited by: Janusz Zalewski
Received: September 30, 2009
Accepted: October 19, 2009