Resource Properties Expression and Runtime assurance for embedded programs, using Qinna, a component-based software architecture

Laure Gonnord, Jean-Philippe Babau

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Resource Properties Expression and Runtime assurance for embedded programs, using Qinna, a component-based software architecture

Laure Gonnord - Jean-Philippe Babau

N° 6565

Juin 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6565--FR+ENG

INRIA
RHÔNE-ALPES

Resource Properties Expression and Runtime assurance for embedded programs, using Qinna, a component-based software architecture

Laure Gonnord* - Jean-Philippe Babau

Thème COM — Systèmes communicants
Projet Amazones

Rapport de recherche n° 6565 — Juin 2008 — 12 pages

Abstract: Designing embedded communicating systems such as PDAs, mobile phones, is getting more and more complex as the hardware performance grows. The component paradigm appears as promising mainly due to the reusability and flexibility of code, but new problems appear such as security, safety and quality of service managements, which have not been integrated in the component-based designs.

In this context, a component-based software architecture, Qinna, has been designed to manage quality of service from the resource point of view. In this research report, we propose a complete formalization of resource constraints expression and management using Qinna, and illustrate on a classic case study.

Key-words: Component Based Systems, Resource management at runtime, Automatic generation of monitors, Evaluation

* This work has been partially supported by the REVE project of the French National Agency for Research (ANR)

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Expression et Maintenance à l'exécution de propriétés de ressources avec Qinna, une architecture logicielle à composants

Résumé : La conception des systèmes embarqués communiquants (assistants personnels, téléphones portables) devient de plus en plus complexe à mesure que les performances matérielles grandissent. La programmation par composants apparaît intéressante notamment parce qu'elle permet une grande réutilisation du code et aussi une grande flexibilité au développement. Cependant de nouveaux problèmes apparaissent, car les paradigmes de programmation par composants n'intègrent pas les problématiques de sécurité, de sûreté et de qualité de service.

Dans ce contexte, une architecture logicielle à composants pour la gestion de la qualité de service au niveau ressources a été proposée. Dans ce rapport de recherche, nous proposons une formalisation complète de Qinna depuis l'expression des contraintes de ressources jusqu'à la garantie à l'exécution de ces contraintes. Nous illustrons sur un exemple.

Mots-clés : Développement par composants, management de ressources à l'exécution, génération automatique de moniteurs, évaluation.

1 Introduction

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

- easily and safely add/remove services at runtime
- adapt (degrade if necessary) component functionality to shared resources capacity
- 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 for the development of such kinds of systems. Indeed it offers an easier way to build complex systems by assembling basic components ([10]). The main advantages are the re-usability 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 these issues, Qinna ([14, 15]) provides components and algorithms, and also a design process for the development of resource aware applications. However, whereas the Qinna architecture contains the main concepts for resource management, the constraint specification part has not yet been formalized, and there is also a need for an automatic or computer-aided generation of some of the maintenance components.To address the resource issues, Qinna ([14]), a fractal-based framework, gives an explicit and dynamic resource management for resource-aware applications. The philosophy is to implement variability through discrete QoS levels and links between them. Qinna then provides algorithms to dynamically adapt these levels according to the resource availability *at runtime*. The objective then is both to ensure a safe usage of resource and also to provide a way to evaluate the threshold between resource usage and total resource.

The component-based Qinna framework has been designed to manage the adaptation of provided and required services *at runtime* through the notion of implementation levels, which are linked together. The adaptation is made dynamically by Qinna's components. The framework also provides a way to manage dynamic resource constraints at runtime. In this research paper, we propose a complete formalization of Qinna's resource properties management, from the formal specification(expression through formula expressed in a dedicated logic) to automatic management, and illustrate by a case study.

The paper is organized as follows : Section 2 sets the context of our study. Section 3 proposes a complete formalization of the framework. Section 4 focuses on the expression of quantity of resource constraints, and a way to implement the expression of these properties in Qinna. In Section 5, a case study illustrates the approach.

2 Background

2.1 Component Model

This section outlines the minimal component model concepts necessary to implement Qinna. Following CBSE philosophy, all the application modules are components, even resources like Memory, CPU, Network. 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.

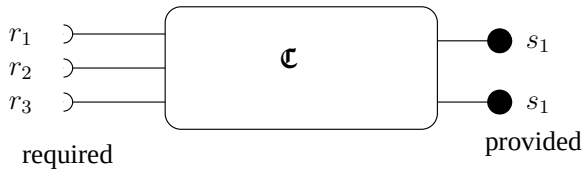

Figure 1: A component

A component $\mathfrak{C}$ (Figure 1) 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 2.

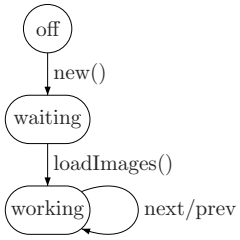

Figure 2: Behavior of a video component

Some components or group of components provide service which code may vary according to its "resource". In the rest of the paper, we will talk equally of resources issues as well of quality of service (QoS) issues, whereas the notion of quality of service also contains other non functional properties like delays.

2.2 Qinna

The Qinna architecture designed in [15] and [14] gives some development rules and algorithms in order to help the development of 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 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 service to provide and its required ones. The developer thus has the way to carry over for example that a video component provides an image with the highest quality when it has enough memory and sufficient bandwidth.
- 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 also be reserved at the corresponding implementation levels. If it is not the case, the negotiation fails.

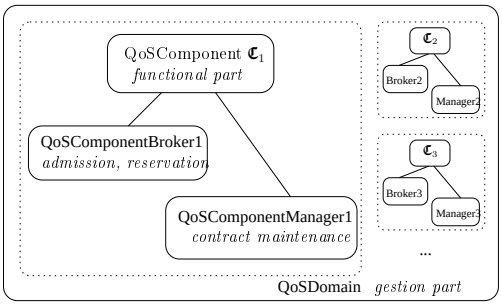

Figure 3: Architecture example

These characteristics are implemented through new components, which are illustrated in Figure 3 : to each application component (or group of components) which provide one or more variable service Qinna associates a *QoSComponent*. The variability of a variable service is made through the use of a corresponding variable

`implementation level`. 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 component. The Broker decides whether or not a new instance can be created.
- 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 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.

3 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. In particular, we distinguish resource constraints which will be ensured by the framework from linking constraints, which will have to be negotiated. Both managements are done at runtime.

3.1 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 :

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

  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 $CTC(s_i)$ (s_i being a provided service of the component) express constraints on the s_i arguments and its call occurrence sequence.
- The *Component Instance Constraints* (CIC) are of similar form :

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

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

Expression of quantity of resource constraints Qinna 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 $CTC_{compo}(C)$ 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 3.3.

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 4, 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 4 illustrates the difference between type and instance constraints.

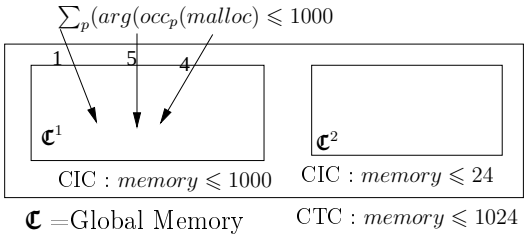

Figure 4: Type versus Instance constraints

- 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 $CTC(malloc) \equiv \sum_k arg(occ_k(malloc)) \leqslant 1024$ 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).

3.2 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. These implementation levels are totally ordered for a given service.

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 :

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

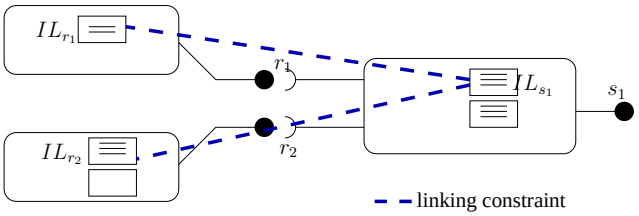

Figure 5: Implementation Levels

This function can easily be implemented in the QoSManager through a "mapping" table whose lines encode the tuples$(IL_{s_1}, IL_{r_1}, IL_{r_2})$.

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 5). This has a consequence not only on the code of all the involved services 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 behavioral 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.

3.3 Qinna's contracts

Qinna provides the notion of *contract* to ensure both behavioral constraints 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 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).

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. Therefore, a contract may not be valid any more. 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.

4 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 [7] and compile it into sequential code for use in Qinna.

4.1 The MaC framework and the Event Definition Language logic

In [7], the author introduce a framework called MaC (for Monitoring and Checking) in order to ensure *at runtime* that a system is running correctly according to a set of formal requirement specifications. The major phase of the method, which can be applied at any time of the development process (from the highest level model to the implementation) are the following :

- System requirements are formalized and a monitoring specification is made. This specification is used to automatically instrument the code and establish a mapping from low level information into high-level events. These events are monitored at runtime by an *event recognizer*.
- *A runtime checker* which is automatically generated by a set of formal specification checks at runtime whether or not the specification are met. It not, it throws an alarm.

To express the specifications, the authors introduce two logics which deal with the notion of event, and which can be easily compiled. The first one, called PEDL, is used to express the transformation. The second one (MEDL) express the requirements that will be checked at runtime. They have the same above logic called EDL (Event Definition Language) which is introduced in the two next paragraphs.

We recall here a simplified version of the EDL logic introduced in [7]. Here we suppose that all functions are totally defined, for sake of simplicity.

Syntax Let $\mathcal{E} = \{e_1, e_2, \ldots\}$ be a set of *primitive events*. Each event occur instantaneously during the system execution. These events can be updates of monitored variables, and calls and returns of monitored functions.

$\mathcal{C} = \{c_1, c_2, \ldots\}$ is a set of primitive conditions. Basically, these are boolean conditions on monitored variables, whose truth value can be instantaneously computed. However, the conditions hold for a duration of time (until a monitored variable change its value, for instance).

Here is the syntax for conditions ($c \in \mathcal{C}$) :

$$C ::= [E, E) \mid C \&\& C \mid C || C \mid \ldots \mid c$$

And the syntax for events ($e \in \mathcal{E}$) :

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

In addition, MEDL formulas may contain the definition of auxiliary variables, because the previous logic has limited expressive power. Auxiliary variables must be one of the basic types in java, and their updates are triggered by events : `e1-> count_e1 := count_e1 + 1` Then, the primitive conditions c_i can also be boolean formulas on the auxiliary variables.

Semantics Models are of the form $M = (\Sigma, \tau, L_C, L_E)$ where :

- $\Sigma = \{\sigma_1, \sigma_2, \ldots\}$ a set of states corresponding to the observation times of the monitored system;
- $\tau : \Sigma \to \mathbb{N}$ is the time at each state;
- $L_C : \Sigma \times \mathcal{C} \to \mathbb{B}$ is an evaluator for primitive conditions at each state.
- $L_E : \Sigma \times \mathcal{E} \to \perp \uplus D$ where $D = \prod_{e_i \in \mathcal{E}} D_{e_i}$ gives information about the event e at state σ_k. If $L_E(\sigma_k, e) = \perp$, it means that e does not occur. In the converse case, $L_E(\sigma_k, e)$ gives the "values" associated with the event e, value taken in the domain D_e. It can be the values of arguments for functions calls, for instance.

Then, $M, t \models C$ and $M, t \models E$ are recursively defined :
Base cases

- $M, t \models c_k$ if and only if $L_C(\sigma_i, c_k) =$ `true` at some state σ_i verifying $\tau(\sigma_i) \leqslant t$ and $\forall$ (in other words c_k condition is true at the previous observation time).

- $M, t \models e_j$ if and only if there is some σ_i such that $\tau(\sigma_i) = t$ and $L_E(s_i, e_j) \neq \perp$ (in other words the signal e_j is present at time t).

Recurrence cases

- &&, || have respectively and and or logical classic semantics.
- $start(C)$ and $end(C)$ are events that occur respectively when condition C changes from `false` to `true`.
- event 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.

Tools in the MaC Framework The EDL logic can be compiled into an event recognizer and a monitor, through a procedure described in [12]. This procedure is implemented inside the M^{2}IST toolsuite ([8]), the tools `p2c` and `m2c` respectively compile PEDL and MEDL into CHARON hybrid automata. These automata can then be compiled into C++ code.

These tools are then used to validate systems at runtime. For instance, in [13], the author generate the hybrid event recognizer and monitor and use them to detect bad behaviors at runtime, but the program ends when there is an alarm. Some test generation is also made. In [11], the author use a variant of edl called rEDL to specify how an hybrid system can reconfigure itself at runtime.

4.2 Our Variant

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 occur. 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 :

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

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

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

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

Semantics The semantic is very similar to MEDL, except for the base events whose semantic are straightforward.

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 3.1 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)`.

4.3 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.

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

`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.

For instance, the translation of the above formula for memory gives the following procedures (where 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;
 }
```

5 Case study : image viewer

We have applied the framework on a remote viewer application (Figure 6). This integration of Qinna and the precise specifications of this case study are precisely described in [5]. 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).

We use Qinna for two main objectives : 1) the maintenance of the application with respect to the different resource constraints 2) the evaluation of the influence of the parameters on the resource usage of the application.

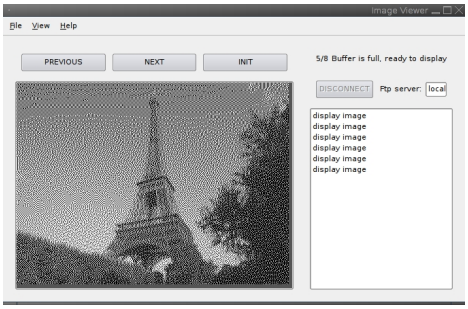

Figure 6: Screenshot of the viewer

Memory (resource and linking) constraints As we have implemented the viewer application in a high-level language, we have no call to the real `malloc` function. The management of memory is thus simulated via a call to the `abstractmalloc` 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 `ImageDisplay` 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 [5].

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.

As a first result, we are then able to log the current memory use (for the video) component) during the execution of the application. Then we can evaluate the gap between the reservation and the use for memory, as Figure 7 shows.

[2] http://www.tricity.wsu.edu/ litan/tools/mist.html
[3] http://trolltech.com/products/qt/

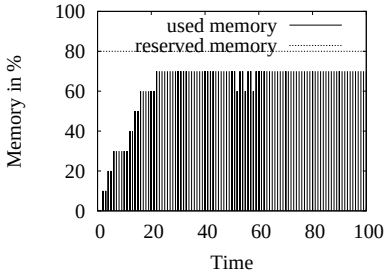

Figure 7: Memory use

6 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 ([4]) 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 work in the domain of verification try to prove conformance of one program to some specification : in [9], the author use synchronous observers to encode and verify logical time contracts. In [13], 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.

Our approach is is rather similar to the controller synthesis domain ([6]). In these last techniques, the (automatically generated) controller (here the Broker), catches some inputs of the systems and constraints the program (automata) to ensure safety properties. There is however two main difference : we deal with non functional and numeric properties, while the controller synthesis provide solution for boolean control ; we only use little knowledge of the system while it uses the global model to generate the controller.

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 ([2, 3]) component-based design of embedded software.
- automatic discovery of the "best" linking constraints (w.r.t. some criteria).

References

[1] Steffen Becker, Lars Grunske, Raffaela Mirandola, and Sven Overhage. Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*. Springer, 2006.

[2] 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.

[3] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall, and Gilles Muller. THINK: A software framework for component-based operating system kernels. In *Proceedings of Usenix Annual Technical Conference*, Monterey (USA), June 2002.

[4] Svend Frølund and Jari 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, 1998. USENIX Association.

[5] Laure Gonnord and Jean-Philippe Babau. Resource management with Qinna framework : the remote viewer case study. Research Report 6562, INRIA, June 2008.

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

[7] 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.

[8] M2IST Toolsuite. http://www.tricity.wsu.edu/~litan/tools/mist.html.

[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, August 2004.

[10] Michael Sparling. Lessons learned through six years of component-based development. *Commun. ACM*, 43(10), 2000.

[11] 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.

[12] L. Tan, J. Kim, and I. Lee. Testing and monitoring model-based generated program. In *Proceedings of the third international workshop on Runtime Verification (RV'03)*, 2003.

[13] 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.

[14] 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.

[15] 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.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399