Controlling the Performance Overhead of Component-Based Systems*

Olivier Lobry[1] and Juraj Polakovic[2,**]

[1] France Telecom R&D, Issy-les-Moulineaux, France
olivier.lobry@orange-ftgroup.com
[2] STMicroelectronic, Grenoble, France
juraj.polakovic@st.com

Abstract. Flexibility can significantly impact performance. Some component-based frameworks come with a near to zero overhead but provide only build-time configurability. Other solutions provide a high degree of flexibility but with an uncontrollable and a possibly unacceptable impact on performance. We believe that no flexible systems give programmers a means to control the inherent overhead introduced by flexibility. This prevents from reaching acceptable tradeoffs between performance and flexibility, according to the applications needs or hardware targets. This paper presents an ongoing work that aims to redesign the existing THINK component framework. Once revisited, the framework makes possible to finely adjust the flexibility to the actually desired needs and thus better control the induced performance overhead. A categorization of the dimensions of flexibility is also introduced in order to articulate our proposition.

1 Introduction

In the domain of embedded devices, Component-Based Software Engineering (CBSE) enables programmers to build operating systems tailored to specific platforms or application needs. Systems like OSKIT [8], ECOS [3] or TINYOS [9] provide tools, languages and compilers to construct, out of components, customized kernels of embedded operating systems. Programmers generate binary images of a system by specifying the desired modules through some configuration file or architecture description. Such solutions are able to produce efficient systems but with however no reconfiguration capabilities (that is, flexibility at runtime).

On the opposite, some other component-based solutions like MMLITE [13], SPIN [4], SYNTHETIX [11], CONTIKI [6] or lately K42 [12] implement various mechanisms that achieve run-time flexibility. In such systems, components are runtime entities that constitute as many *points of flexibility* in the architecture. They expose some kind of *control interfaces* in order to change its architecture or behavior. While they provide efficient mechanisms that bring flexibility, this

* This work has been partially supported by the ANR/RNTL project Flex-eWare.
** This work was done while the author was a PhD student at France Telecom R&D.

C. Pautasso and É. Tanter (Eds.): SC 2008, LNCS 4954, pp. 149–156, 2008.

provided flexibility is however tied to the architecture of the system to build. More precisely, such systems lack the possibility to finely choose where, when and how to pay for it: given a *same* architecture, it is not possible to produce different binaries having different number of flexible points, associated control interfaces or implementation directives, and thus, providing different tradeoffs between flexibility and performance.

This paper presents an ongoing work that intends to redesign the THINK component framework, that already provides flexibility [7,10] but in a rather fixed manner, in order to reach our goals. Section 2 proposes a categorization of the ability to tune the provided flexibility. Section 3 gives an overview of the flexibility provided by the THINK component while section 4 details the necessary design changes to be made and how we did them. Section 5 concludes the paper.

2 Dimensions of Flexibility

The ability of a component framework to adapt the injected flexibility to actual application needs can be characterized along the following dimensions[1].

Where. Component-based systems provide flexibility points at component boundaries. Unfortunately, the presence of a component often *imposes* a point of flexibility. This may lead to a prohibitive overhead, thus preventing from encapsulating small services into components. This unfortunately results in loosing other benefits of CBSE that go far beyond the ability to generate flexible systems [14].

What. The nature of flexibility is generally defined by the provided control interfaces. A system may simply provide introspection interfaces to query the architectural state of a component or it may provide more advanced interfaces to change a binding between two components, replace a component with another, add a stub before, etc. As all possible kinds of flexibility are not necessarily *always* wanted for *any* component, a component-based system should not impose a set of control interface.

How. Flexibility may be implemented in different ways with different impacts on performance. For example, one implementation of a control interface may optimize memory footprint whereas another may reduce CPU or power consumption. Also, implementations that take advantage of hardware specificities may not have the same overhead on all platforms. Therefore, programmers should be able to choose between different implementations so as to match the constraints imposed by the targeted platforms and application needs.

When. The requirement for flexibility may evolves over the release timeline of the system software. Consider the case of semaphores. Conceptually, implementing semaphores with (very small) components makes sense since they participate to

[1] These dimensions characterize *control* over the specification and implementation of flexibility, not flexibility itself for which a classification can be found in [2].

the architecture of the system. Code for monitoring their state may help during debugging phases but may however be removed in production releases because of memory constraints. Programmers should then be able to specify *when* flexibility is actually required and not pay for it when it is not.

3 Flexibility in Think

THINK is an open-source implementation of the FRACTAL model, a hierarchical and reflective component model intended to implement, deploy and manage software systems [5]. A FRACTAL component is both a design and a runtime entity that constitutes a unit of encapsulation, composition and configuration. Components provide *server interfaces* as access points to the services that they implement while functional requirements are expressed by *client interfaces*. Components interact through *bindings* between client and server interfaces. Finally, a component may have attributes that represent primitive properties.

The model is hierarchical since components may contain functional code and/or sub-components. The FRACTAL model also defines standard control interfaces to observe and manipulate the internal structure of a component at runtime. In particular, a component may implement the `ComponentIdentity` interface to give access to its server interfaces, the `BindingController` interface to rebind its client interfaces, the `AttributeController` interface to query and change attribute values and/or the `ContentController` interface to list, add or remove subcomponents.

The THINK compiler (written in Java) takes as input the description of an architecture — written in a textual Architecture Description Language (ADL) — and a component repository. The *HelloWorld* example shown below gives an overview of the THINK ADL[2].

```
component helloworld {
  contains main = hwMain
  contains termConsole = terminalConsole
  binds main.console to termConsole.console
}
component hwMain {
  provides activity.api.Main as main
  requires video.api.Console as console
  attribute int position = 10
  content hwMain
}
```

The *helloworld* component contains two components *main* and *termConsole* bound to each other through their *console* interface. The *hwMain* component provides a server interface *main*, requires a client interface *console* and an attribute *position* of type *int* with an initial value of 10. Finally, the file *hwMain.c* that contains the functional code is listed bellow:

[2] To simplify, the component definition *terminalConsole* is not given here.

```
1 struct hwMaindata {
2   Rvideo_api_Console *console;
3   int    position;
4 };
5 static void mainentry(struct hwMaindata* self,
6                        int argc,
7                        char** argv) {
8   self->console->putcxys->proc(
9         self->console->selfdata,
10        0, self->position, "Hello World !");
11 }
12 struct Mactivity_api_Main hwMain_mainmeth={main: mainentry};
```

Beyond the difficulty of writing such a functional code, this simple example shows that the programming approach is strongly tied to the glue implementation choices. For example interface calls are implemented using as indirect function calls (lines 8), the *self* of component is systematically declared in method declarations (line 5) and passed when called (line 9), attributes are implemented as struct fields (line 3), etc. Should we change the structure of the produced meta-data, the the compiler would no longer accept the above code. This unfortunately prevents for doing any optimization or providing a way to choose among glue implementation alternatives.

As an implementation of the FRACTAL model, THINK allows developers to produce highly flexible systems. The framework however imposes, for a given architecture, what, where, when and how to implement this flexibility. Indeed, flexibility points are imposed by the architecture description (*where* dimension) since bindings are always dynamic, attributes are always variable, etc., Moreover, the control interfaces are imposed (*what* dimension) and there is no possibility to choose between alternative implementations (*how* dimension). Discussing the *when* dimension is pointless since there is no degree of freedom on the other ones.

Consequently, one can *never* adjust the performance vs. flexibility tradeoff of a given application according to its needs and the targeted platforms. Our proposition is then to redesign the framework so that it produces code and meta-data *only* where flexibility is desired (in a given release), with the ability to choose among several implementations. Once redesigned, THINK will still be able to produce highly flexible systems as before. But it will also better address the embedded world with the ability to produce different binary images of a same system with varying degrees of flexibility and varying implementation choices of flexibility, according to the constraints of targeted platforms.

4 Redesigning the Think Framework

This section first discusses the necessary design requirements to reach our objective before we show what we have been developing to satisfy these requirements.

4.1 Requirements

Specification of Flexibility. THINK provides flexibility points to any component, binding or attribute. To make them optional the framework must permits programmers to selectively specify that a binding is static, an attribute is constant, etc. Also, programmers must be able to specify which control interfaces must be added and how to implement the different instances of the model entities.

Separation of concerns. As the flexibility requirements may change over the development cycle of a system, the above-mentioned specification must be separated from the description of the architecture itself. With such a separation, an architecture description will be reusable in different moments of the system release timeline, according to different flexibility requirements.

Glue-agnostic programming. Based on this specification, the compiler must be able to produce meta-data and code that add flexibility (and hence overhead) only where desired without having to change the functional code. A necessary condition is that the provided programming language does not make any assumption on the organization of the produced meta-data and code.

AST-based code generation. While independent in terms of decisions, builders need to exchange information concerning the produced code and the meta-data. They produce types and variables that may depend on types and variables produced by other builders. This implies the need for a tool to produce and represent an abstraction of produced code.

4.2 Implementation

Specification of Flexibility. We extended the THINK ADL so that entities can be tagged with textual properties that can be used to express flexibility as shown with the following annotated *HelloWorld* example:

```
component helloworld {
  contains main = hwMain [single=true]
  contains console = terminalConsole [single=true]
  binds main.console to console.console [static=true]
}
component hwMain {
  provides activity.api.Main as main
  requires video.api.Console as console
  attribute int position = 10 [const=true]
  content hwMain
}
```

Here, the components are specified as single instances of their respective component types, the binding is set as static and the attribute *position* is constant. We choose to extend the ADL instead of defining a new language to limit the number of languages to learn to program THINK components.

Separation of concerns. To specify flexibility separately from the description of the architecture, we adopt an approach close to aspect-oriented programming (AOP) applied to architectural description. This is done through the notion of *global extensions*. A mechanism is introduced in the compiler in order to automatically extend component definitions. For example, the following global extension makes static any binding of any component (*where dimension*):

```
component * {
  binds * to * [static=true]
}
```

In a similar manner, one can also specify to add an attribute controller to any component that has at least one attribute (*what dimension*):

```
component * {
  provides AttributeController as ac if hasAttribute
  content att-controller if hasAttribute
}
```

Global extensions can be seen as the equivalent of *pointcuts* in AOP[3], where the set of points is essentially expressed using regular expressions on the name of flexibility points (component, interfaces, attributes, ...) and predicates over component definitions (like the *hasAttribute* predicate in the above global extension).

Finally, programmers can use global extensions to specify the builders to be called during compilation:

```
component * [builder=MyCompBuilder] {
  attribute * *  [builder=MyAttributeBuilder]
  provides * as *  [builder=MyServerItfBuilder]
  requires * as *  [builder=MyClientItfBuilder]
  implementation * [builder=MyImplementationBuilder]
}
```

This time the mechanism is used to tweak the build process, and hence, to produce glue code and data that suit the platform constraints or application requirements (*how* dimension). For that reason, this approach provides some kind of compile-time meta-programming facility.

Using global extensions, it is now possible to specify what, where and how flexibility should be implemented. Besides, this specification can be completely separated from the architecture description, thus enabling different flexibility capabilities given a same architecture (*when* dimension).

[3] However quite simpler that the usual pointcut meaning since there is no notion of workflow here.

AST-based code generation. In order to help the collaboration of builders in the process of generating code and meta-data, we developed the CodeGen Java package. CodeGen aims to represent, manipulate and produce C code. The package provides a collection of classes to abstract semantic entities of the C language (types, variables, expressions, declarations, etc.) and functions to gradually produce C Abstract Syntax Trees (AST).

The CodeGen package includes a C parser that can transform a C translation unit into an AST. The parser notifies parsing events through a listener interface when it encounters interesting statements: new type and variable declarations, undefined symbols, etc. The parser also handles and notifies annotations found in comments.

Functional code is parsed by the CodeGen parser which notifies Think of annotations and code parsing events. For example, when parsing the *pos* symbol of the example bellow, Think is notified that this symbol is undefined. An attribute builder is called to return the right expression as a CodeGen AST: access to the corresponding variable if the attribute is modifiable or its initial value if it is constant. Other optimizations like removing the *self* parameter in calls to single components, produce direct calls for static bindings can be perform in a similar way.

Glue-agnostic programming. Using CodeGen we can propose a new component programming language, called NuptC, to program functional code, exclusively based on annotations:

```
1 // @@ Attribute(position, pos) @@
2 // @@ ClientMethod(console, putxycs, putxycs) @@
3 // @@ ServerMethod(main, main, mainentry) @@
4 void mainentry(int argc, char** argv) {
5   putxycs(0, pos, "Hello World !");
6 }
```

Annotations are here used to express the mapping between architectural entities and C symbols: attribute *position* is represented by symbol *pos* (line 1), *putxycs* method of interface *console* is represented by symbol *putxycs* (line 2) and server method *main* of interface *main* is represented by symbol *mainentry* (line 3). These symbols can then be used in the functional code to define server methods (lines 4-6), call client methods (line 5) or access the attributes (line 5).

This new programming approach does not make assumption on the produced meta-data. For instance, it does not impose to pass a *self* parameter when calling interfaces, implement interface calls as indirect function calls, or implement an attribute as a variable.

5 Conclusion

This paper showed a work-in-progress that consists in redesigning the Think framework to make it able to generate, given a same architecture description,

different system images having different flexibility versus performance tradeoffs. The new version, called NUPTSE, exploits the possibility provided by the CODEGEN Java package designed to transform and produce C code in a collaborative way[4]. We also propose NUPTC, a component programming language based on annotations that does not make assumption on the generated meta-data, hence enabling optimizations and implementation alternatives. The ADL has been extended to enable the specification of flexibility properties and a global extension mechanism based on pattern-matching is introduced in order to separate, if desired, this specification from the architectural description.

As a work-in-progress, our proposition needs better evaluation. At the conceptual level, we need to better formalize the approach, identify its limitations and compare it to aspect-oriented programming and compile-time meta-programming [1]. At the technical level, we need to evaluate the performance gain that we can actually obtain on a real case-study.

References

1. Assmann, U.: Invasive Software Composition. Springer, New York (2003)
2. Denys, G., Piessens, F., Matthijs, F.: A survey of customizability in operating systems research. ACM Comput. Surv. 34(4), 450–468 (2002)
3. eCos, `http://sources.redhat.com/ecos`
4. Bershad, et al.: Extensibility safety and performance in the SPIN operating system. In: Proc. of the 15th ACM Symposium on Operating Systems Principles (1995)
5. Bruneton, et al.: The Fractal Component Model and its Support in Java. Software - Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems (2006)
6. Dunkels, et al.: Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In: LCN 2004: Proc. of the 29th Annual IEEE Intl. Conf. on Local Computer Networks (LCN 2004) (2004)
7. Fassino, et al.: Think: a software framework for component-based operating system kernels. In: Proc. of the 2002 USENIX Annual Technical Conference (June 2002)
8. Ford, et al.: The Flux OS Toolkit: Reusable Components for OS Implementation. In: Proc. of the 6th Workshop on Hot Topics in Operating Systems (1997)
9. Hill, et al.: System architecture directions for networked sensors. In: Proc. of the ninth Intl. Conf. on Architectural support for programming languages and operating systems (ASPLOS) (2000)
10. Polakovic, et al.: Building reconfigurable component-based OS with THINK. In: 32nd Euromicro Conf. on Software Engeneering and Advanced Applications (2006)
11. Pu, C., et al.: Optimistic incremental specialization: Streamlining a commercial operating system. In: Proc. of the 15th Symp. on Operating System Principles (1995)
12. Soules, et al.: System support for online reconfiguration. In: Proc. of the 2003 USENIX Annual Technical Conference (June 2003)
13. Helander, J., Forin, A.: MMLite: a highly componentized system architecture. In: EW 8: Proc. of the 8th ACM SIGOPS European workshop on Support for composing distributed applications (1998)
14. Szyperski, C.: Component Software, 2nd edn. Addison-Wesley, Reading (2002)

[4] THINK and CODEGEN are freely available at `http://think.objectweb.org`