Appeared at *Proc. of IEEE COMPSAC* 2007 Beijing, Previous Version

Fuxi: An Agile Development Environment for Embedded Systems

Abstract: Challenged by market and technical advance, the requirements of embedded products will keep changing throughout the whole process of development. How to introduce agility to the development process, to adapt to these changes? In this paper, we proposed an architecture-based, aspect-oriented methodology of agile software development, which takes expressivity and efficiency as two major concerns of embedded applications; expressivity yields agility, and efficiency meets the constraints of resources. Under the separation of concerns, we differentiate the system into functional aspect and several technical aspects which give supports to the functional one. Different concerns need different programming paradigms. At the functional aspect, we takes declarative programming, and at the technical aspects procedural programming. An abstract machine, as a pivotal of the architecture, weaves all these aspects dynamically at runtime, to achieve all the functionalities of the system.

Fuxi programming language and its environment are the embodiment of the above methodology; and Fuxi environment itself was developed through such an agile software process.

1. INTRODUCTION

Embedded system designers, in varied industry segments that include consumer electronics, automotive control, and medical equipment, are facing increased pressure of the development time, the changes of requirements during development processes entailed by market, and new more complexity by advances of circuit technology. Moreover, they must create product families that offer customers a wide-range of cost, function, and performance tradeoffs. These call for agility in the engineering processes; adapting to changes of requirements, keeping quality and complexity under control, while at the same time meeting the stringent cost and time-to-market constraints. Agility is dynamic, context-specific, aggressively change-embracing, and growth-oriented; it can help you to succeed in emerging competitive arenas, especially in the domain of consumer electronics, such as smart phone, PDA, etc.

Software is central to enable functionality in embedded systems. At the same time, software is susceptible to changes of requirements; and it is also a source of quality problems and constitutes a major part of the development cost. These problems are further accentuated by the increasing complexity and product integration. In recent years, agile software methodologies (also known as lightweight methodologies) [6, 11, 15] have been used increasingly in office and web projects. But because of the constraints of development environment and resources, and the diversity of technical backgrounds of individuals in the team of an embedded project, the core values proposed by agile alliance, such as interactions between individuals, customer

collaboration, cannot be realized easily as in the office environments; so that these agile software methodologies do not fit better in embedded domain. At the same time, the agile methodologies have also themselves some limitations [29, 30].

In this paper, we present a new agile methodology suitable for embedded systems, which is based on the orthogonalization of concerns (i.e., the separation of the various aspects of design to allow more effective exploration of alternative solutions) [8]; and we argued that different concerns need different programming paradigms.

The rest of the paper is organized as follows. Section 2 explains the core idea of our new agile design methodology. Section 3 discusses the design of Fuxi language, which is dedicated to specify the functionality of embedded system. Section 4 presents the AOP mechanism based on a meta-object model. Section 5 outlines the architecture of Fuxi Platform, a middleware, embodied our methodology. Section 6 discusses some related works. Section 6 concludes with a discussion of our findings and the roadmap to the future works.

2. AN AGILE METHODOLOGY FOR ES

The pillar of our new design methodology is to separate the system into two orthogonal dimensions: functionality and technicality; its core idea is that different dimensions need different programming paradigms [23].

In the dimension of functionality, it talks about the function, behavior and aesthetics of a system (what the system is supposed to do), and in the dimension of technicality, it is about the infrastructure (components, communication, and coordinators) to implement the functionality of a system (how the system does it). The components of functionality, usually application-specific, susceptible to changes of requirements, and of little reusability, need to be programmed expressively in order to introduce agility into development processes, adapting to changes; but the underlying techniques, such as concurrency, security, persistency and mobility, are usually efficiency-critical and can be reused in a variety of contexts, so need to be implemented efficiently.

In the context of our methodology, the declarative programming paradigm is taken in functionality and procedural programming in technicality.

Declarative programming languages are usually expressive; so that, declarative programming can facilitate to model the system, to specify the functions and behaviors of an embedded product, to adapt to the changes, and it can also quicken delivery of working software. The components of functionality run on the virtual machines, that, they can be verified, tested and evaluated on different platforms, such as prototyping platform, target device, or desktop environment.

Procedural programming languages (say C) are designated to build the infrastructure under some meta-object model, which serves the glue to integrate different components; the infrastructure gives supports to the virtual environment where the components of functionality run. Procedural programming can also yield efficient codes to meet the constraints of embedded systems.

The virtual machine in our context takes a pivotal role; it is the interpreter of virtual instructions and also serves a mechanism to weave dynamically the technicality into the functionality. Virtual machines make the components of functionality portable; and the portability in turn has many advantages, ranging from improved documentation and maintainability to more

rapid re-targeting of the implementation (also agility in embedded context).

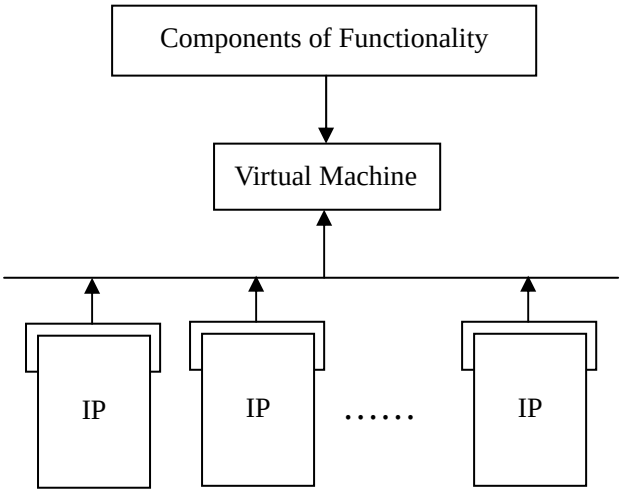

Fig. 1 The model of the system with a virtual machine

In real engineering processes, we usually do not build the components of technicality from scratch, for there are a diversity of off-the-shelf components, such as GPRS, GPS, TVB/NAVI modules (also called as IP cores), and even some middleware, ready to be integrated. Designers find using these modules to be advantageous because of their low cost and the way facilitating rapid realization of designs not only for prototyping but for production as well. In fact, many embedded products have declining lifetimes that make custom integrated circuits a less economically viable option.

In order to integrate these IP cores, we need to create some wrapper objects under the meta-object model mentioned above to interface them with functionality components. The components of technicality are also substitutable in different contexts. For example, in a packet movie project, we can build a wrapper for Microsoft MediaPlayer to simulate the functions of products in desktop environment, and wrap a BetaPlayer in a prototyping platform, and at last we re-implement the MPEG decoder with DSP on the target device.

We can summarize the development process as the following:

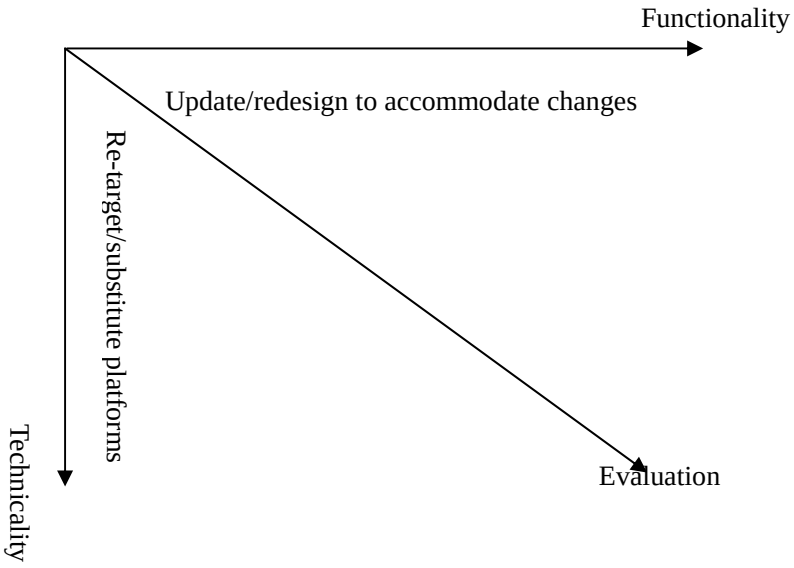

Fig. 2 The engineering process of our methodology

As an agile methodology, the declarative programming mentioned above has a tremendous difference with some model languages (such as UML) already widely used in design of embedded systems.

1. we use a declarative language to program the functionality of system, not just to model or specify it;
2. the result of declarative programming is the components of working software, not just some documentation or visualized specifications;
3. a mechanism is used to weave dynamically the components of technicality into the ones of functionality at runtime; not to map statically the functionality onto some architecture at design time.

In order to adopt our new design methodology in engineering process of embedded system, a high level language, with common knowledge to developers with different technical backgrounds, must be designed to support declarative programming.

The mentioned language must possess the following features:

1. Provides a high-level abstraction, with little knowledge specific to computer science;
2. Can be used to program the working system, not just to specify functionality;
3. Has a good tradeoff between expressivity and efficiency;
4. Has a glue logic to integrate different off-the-shelf components;
5. Has a mechanism to weave different technical aspects together;
6. The code in such a language can be run at different platforms.

Efficiency and expressivity are the two major concerns of such a language that must be taken into account in the context of embedded domain. The design methodology proposed here pays special attention to the balance between expressivity and efficiency. Expressivity yields agility in engineering processes, and efficiency meets the constraints of resources.

Separation of concerns gives us an opportunity to modularize these concerns into separate components to allow more effective exploration of alternative solutions.

Fuxi, a declarative language, which provides a high-level abstraction of system functionality through the well-designed combination of functional, logic, and object-oriented programming paradigms, is dedicated to specify system functionality. Some procedural language, say C, is designated to implement the underlying techniques under the framework of Fuxi meta-object model. Fuxi source code is compiled into a so-called Fuxi Object Graph (FOG), a computational graph[15]; Fuxi Abstract Machine (FAM) evaluates the FOG through graph-rewriting, and weaves dynamically those technology aspects to achieve the whole system functionality.

3. DESIGN OF FUXI LANGUAGE

Fuxi is so conceived to provide a facility of high-level abstraction of system functionalities, to make the computer-specific technologies invisible, and to present some neat features of computation common to different disciplines. So, Fuxi can be easily used by the professionals of other disciplines, such as product planners, aesthetic designers, even the customers themselves.

Instead of invention of new language techniques, Fuxi takes a design philosophy: to combine the best ideas found in different languages within previous decades, through selecting some

compatible features among these languages and merging them into a single approach in uniformity and conventional form.

The detailed discussion of the syntax of Fuxi is out of the scope of this paper, but the syntax can be summarized as a pseudo-formula as follows:

<Fuxi>=<Java-like type frame> + <Pattern-matching methods>

The following is a simple Fuxi program:

```
import fuxi.*
public active class FibonaciApp : Application {
    Fib(0) = 1
    Fib(1) = 1
    Fib(int n) = Fib(n – 1) + Fib(n – 2)
    public OnError(ERROR_MEMORY_ALLOC) ->
            System.Console.Println("Out of Memory")
    public Activate() =
            let{ int n = System.Console.Readln().ToInteger() } in {
        System.Console.Println("Input Number:")
        System.Console.Println("Fib("+ n + ")"=Fib(n))
    }
}
```

Fig.3 A Simple Fuxi Program

From this example, we can summarize the following language features of Fuxi:

1) it takes a conventional syntactic system similar to Java, that makes the programmers with some experiences of C++, JAVA, use the language without any training;
2) it merges calculation, inference and reaction into a unique setting;
3) some style modifier guide the behavior of object.

Fuxi has no entry function as Java. When an active object is created, a thread will be assigned to it; the thread will first call automatically the function Activate().

3.1 Pattern Matching and Rules

A method is composed of a group of rules, with the same signature. The syntax of rule definition is as the following:

<functor-name> **(** *<pattern>* **)**
    <define-operator> <body-expression>

where *<define-operator>* is one of: **=** **<-** **->**

If <*define-operator*> is '=', the rule is called an equation; if <*define-operator*> is '**<-**', rule is called a clause, it stands for a Horn-clause; and if <*define-operator*> is '**->**', rule is called a trigger or production, its parameter usually passes an event to the <*body-expression*>. The <*body-expression*> of an equation can be any type, but <*body-expression*> for a clause or a production can only be Boolean.

A pattern has the form of expression made up of constants, variables, tuples, list constructors, and even wildcards. Pattern matching can be used in virtually all methods definitions where a case analysis is called for, and can eliminate most uses of if-expressions. Fig. 3 gives us an example for Fibonacci function definition.

A method invocation expression can invoke a function, which evaluates to some value; it can also invoke a predicate, which will start an inference process; or it can pass an event to some triggers. For Fuxi is a hybrid language, we can invoke some functions, predicates or triggers in a same expression, for example:

```
Likes("John","Apple")#
Likes("John","Orange")<-!sour("Orange")
Likes("Sophie", String X ) <- {
    X != "Orange"
    Likes("John", X )
}
let { String X } in
    if( Likes("Sophie", X ) ) Buy( X )
```

Here, '`Likes`' is a predicate, '`sour`' and '`Buy`' is functions. If a clause is trivial true, it is often called a fact. '#' defines a fact

3.2 Classes, Interfaces and Objects

We can consider a class as a union of states and rules, for that, method is a group of rules defined through pattern-matching.

Fuxi takes syntax of class declaration with some similarity to that of JAVA or C#.

[<*qualifier*>][<*styles*>] **`class`**<*class-name*> [**`:`**<*base-list*>] **`{`**<*member-declaration-list*>**`}`**

where <qualifier> is one of : `public internal`

<styles> is one or more of:

```
active mobile native persistent
@<identifier>
```

The optional <*qualifier*> is used to specify the encapsulation of classes. User-defined styles must be prefixed by '@'.

The syntax of interface definition is similar to that of class, as follows:

[<*qualifier*>][<*styles*>] **`interface`** <*interface-name*>
[**`:`**<*base-list*>]**`{`**<*member-declaration-list*>**`}`**

The <*member-declaration-list*> can only be some prototypes, without any rule implementations.
Fuxi classes/objects can bear some styles. The native style can only be applied to classes, and specifies those classes are implemented in native codes. Access to the instance of such a native class will invoke an external call to some binary module (a DLL in windows or a Shared Library in Unix/Linux).

3.3 Slots: Scripted Definition of Object

A public free field without initialization in constructors is considered as a slot. A class with slots gives a frame-like templates for objects; when an object declaration with such a class, we can attach a script-like text enclosed with {}, to fill the slots; and this way of object definition is called scripted definition of object. The scripted object definition is very useful in describing resources, and makes Fuxi more declarative. For example, we define a GUI object:

```
class Button : Control {
    public Button( String title ) = {…}
    public POINT at
    public SIZE  size
    …
}
    …
Button btnExit( “EXIT” ) {
    at :200, 200
    size: 80, 40
}
```

3.4 Specification of Native Classes

The class implemented in native code, is called native class. Before a native class could be used, we must specify its interface. The specification of native class is very similar to the definition of an interface, except keyword ‘class’ with a modifier ‘native’. In the body of definition, there is no any implementation code. For example:

```
public native class Window
{
    public Rect m_rcWnd      // Attributes of a Window
    public Window( Rect rcWnd, int style, Window parent ) = bool
    public OnMouseEnter() -> bool       // Triggers
    public OnMouseLeave() ->bool
    public OnClick() -> bool
    ……
}
```

After specified, a native class can be used freely as normal class.

4. Orthogonal Stylization of Objects

Orthogonal stylization of objects is the direct extension to the concept of orthogonal persistence by Atkinson[2,3]. Orthogonal stylization means that styles are orthogonal to the structural features of classes; a class can be instantiated with different styles, just as the colors are orthogonal to the spatial features of products. The built-in styles for objects in Fuxi include: active, mobile, persistent and remote. Stylization makes programmers concentrate on the logical design of classes, with little attention to technical implementation of these styles. Just as a product designer, concentrates on the structural design of a product, without attention to the colors, but when the products have been made, workers can paint them in different colors.

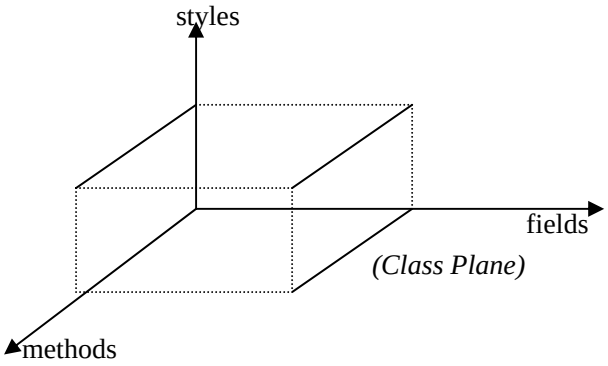

Fig.4 Three dimensions of instance space

4.1 User-Defined Styles

The user-defined styles manifest the aspect-oriented characteristics of Fuxi. It is very similar to Bryant's explicit programming [9].

User-defined style includes two parts: style specification and style implementation. Style specification is written in Fuxi, and style implementation is usually built through some procedural programming. The style specification is a class derived from class Style, which is defined in Fuxi Fundamental Library, and implemented in native code. The style implementation is derived from StyleImpl.

4.2 Specification of User-defined Styles

The style specification is used to declare when, where and which procedure affiliated to the style is to be invoked. The dynamics of Fuxi is the pattern-matching and unification. So, the style specification is just to declare a group of triggers; when execution goes to some point, and certain context pattern is matching with the parameters of a trigger, the routine defined in style implementation will be invoked.

The names of these triggers are predefined as follows:

before(GET_ATTR, <attr-name>) -> { … }
after(GET_ATTR, <attr_name>, <attr_value>) -> { … }
before(SET_ATTR, <attr_name>, <value>) -> { … }
after(SET_ATTR, <attr_name>, <value>) -> { … }
before(INVOKE, <method_name>, <param-list>) -> { … }
after(INVOKE, <method_name>, <param-list>) -> { … }

```
after( BACKTRACK, <method_name>, <param_list> ) ->
                { … }
```

'_' can be used as a wildcard pattern, and <param-list> can contains variables or certain values. For example,

```
before( SET_ATTR, _ , _ ) -> { … } // Any setter matched
before( SET_ATTR, _, 10 ) -> { … } // Setters with
                                   // value 10 matched.
before( INVOKE, _, int x, float y ) -> { … }
 // Any function with signature (int, float) matched.
```

The members of style specification include:

1) a static field of type of the StyleImpl, at the class initialization time, the field will be initialized, and a StyleImpl object will be created.
2) A group of triggers, which are designated as pointcuts as AspectJ.

The following is an example of user-defined style:

```
public class WebSecurity : Style {
    static WebSecurityImpl    security()
    before(GET_ATTR, _, _ ) -> security.Check()
    before(INVOKE, _) ->security.Check()
    …
}
```

5. FUXI PLATFORM

The minimum environment where a Fuxi program could run is called Fuxi Platform, it includes Fuxi Abstract Machine (FAM), Fuxi Fundamental Libraries. FAM is a WAM extension, which can be used as an inference and graph-reduction engine. Fuxi Fundamental Library specifies a meta-object, called GOBJ (Generic Object), which is mapped to Fuxi language as fuxi.Object.

Each object is derived from a meta-object GOBJ, which is implemented by the native code, or machine code. And through this native implementation, FAM can interact with its environment.

The native objects/classes are those implemented completely in native code and data formats. But normally, a class is programmed in Fuxi based on some native class, such as Control, Window.

5.1 Meta-Object Model

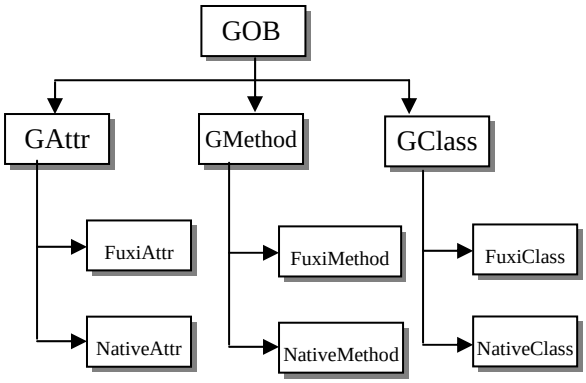

Fig.5 Meta-Object of Fuxi

The definition of GOBJ is only a virtual function table(VTable), its entries include a series of reflection functions (such as GetAttr(), GetMethod(), …). Besides GOBJ, the platform also defined several meta-objects derived from GOBJ, as shown in Fig. 5.
Normally, a Fuxi object include two VTable, as follows.

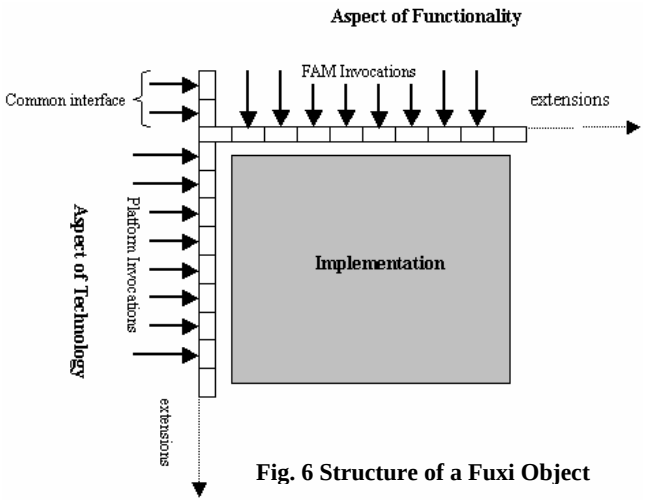

Fig. 6 Structure of a Fuxi Object

The VTables can be extended to contain extra functions, and its entry can be overridden. The vertical VTable, called t_VTable, is to store the pointers to subroutines in native codes as C++; the horizontal one, called f_VTable, is for the pointers to meta-object GMethod. GMethod can be FuxiMethod, where the implementation is a FOG; or it can be a NativeMethod, which contains only native code. FAM can only invoke the methods listed in f_VTable and functions defined in GOBJ (common interface as shown in Fig. 6). So, the technology is invisible to FAM. Through these meta-objects, FAM can dynamically insert some invocation to the methods defined in aspects according to the style the object bears.

5.2 Native Implementation

Usually, native classes are coded in C. A group of macros have been defined to facilitate implementing a native class. The following example code segment is to implement a native class 'Control':

```
// Declare a native class Control
FUXI_DECLARE_CLASS( Control, GOBJ )
    G_CSTR    m_lpszName;
    LPGOBJ    m_pParent;
    G_UINT    m_iStyle;
    G_UINT    m_iState;
    Point*m_pLoc;
    Size*        m_pSize;
FUXI_END_CLASS
// Definition of native class Control
FUXI_IMPLEMENT_CLASS_EX( Control, GOBJ, 15 )
     OVERRIDE(Control, SetParent,    __SetParent )
    OVERRIDE(Control,GetDisplayText, __GetDisplayText )
    EXTEND( Control,    0, __InitControl        )
```

```
    EXTEND( Control,    1, __ShowControl    )
    ……
    EXTEND( Control, 14, __Refresh      )
FUXI_END_IMPLEMENTATION()
```

5.3 USING STYLES

Style can be applied to both classes and fields. If the class bears some style, means that all the instance bears the style intrinsically. If a field bears some style, means that only the instance created there with the style. For example,

public @WebSecurity Voucher m_voucher = PurchaseVoucher()

the instance created by PurchaseVoucher() bears the user-defined style @WebSecurity. Any access to the field m_voucher will invoke some procedure defined in WebSecurityImpl which implements WebSecurity.

At the initialization stage, when WebSecurity style class is initialized, a WebSecurityImpl instance 'security' is created (for it is a static member of WebSecurity).

At the execution time, when an access to field 'm_voucher' occurs, the pointer to the object that m_voucher denotes will be loaded into FAM's __this__ register (if __this__ is not pointing to that object), and a new frame created, and a style list for m_voucher created at the bottom of the frame.

If this style list is not empty, when the node GET_FIELD, SET_FIELD, or INVOKE in FOG is to rewrite, or has been rewritten, or a backtrack occurs, some events will be raised, and sent to the style objects in the list to check if some triggers matching with it.

In our example, security.Check() will be invoked before GET_FIELD, SET_FIELD, INVOKE to rewrite.

6. RELATED WORKS

Our design methodology is much different from the ones proposed by Agile Alliance [11]. The pillar of our methodology here is the separation of concerns [19, 24, 28], and we argue that different concerns need different programming paradigms; Agile Alliance proposed that different projects need different methodologies.

AGILE [1] is software architecture for mobility, with much similarity with Fuxi Platform, and kernel language KLAIM is used to build mobile codes. But Fuxi is more general, not limited to some specific domain.

Maté [22] is a tiny virtual machine that runs on TinyOS, a specialized operating system for sensor networks. Maté helps sensor network programmers to build expressive and concise applications, and it protects the system from being crashed by applications. Fuxi has not been applied in sensor networks yet, but we believe that Fuxi can manifest some power in this domain.

Many AOP mechanisms exist, but most of them are the extensions to some major OO languages (such as Java, C++, C#) [17, 18, 27] through introduction of some embedded domain-specific languages to specify the weaving strategies. These embedded weaving specification language (WSL), are dedicated to guide the weaving of aspects built in the host languages. The weaving can take place at compiling, loading, or runtime.

Among these works, Handi-Wrap[4] is based on a meta-object model to implement dynamic weaving. Wrapper has some similarity to meta-object GMethod. But Fuxi uses meta-object in a

transparent way.

The monitor-based AOP[12] shares some similarity with Fuxi. Compared with it, Fuxi's event monitor is a dynamic one, changing with the object activated. But Fuxi can not recognize the event sequence.

The style of object owes much to Bryant's explicit programming [9], which gives user the chance to introduce new modifiers. User-defined styles in this work are just such modifiers, but the context and mechanism are different. User-defined style also shares some similarity with meta-object annotation [7].

Weave.NET[21] is a multi-paradigm AOP mechanism independent of languages, implemented on .NET platform. Weave.NET uses XML to specify weaving strategy.

Bossa[5] and CATAPULTS[25] are two domain-specific languages dedicated to specify scheduling policy, which give a good balance between efficiency and expressivity. Compared with these works, Fuxi is more general.

7. CONCLUSION

With respect to the values proposed by Agile Alliance, we present a new agile methodology suitable for the embedded systems. Our new methodology is based orthogonal separation of concerns, that allows more effective exploration of alternative solutions; the core value of our methodology is that different concerns need different programming paradigms. In the context of embedded system, we propose the separation between functionality and technicality of system; and declarative programming is used to program the components of functionality, and procedural programming to implement the components of technicality.

Taken this methodology in mind, we designed and implemented a declarative language, Fuxi, to build the components of functionality; and C is used to implement the components of technicality under the Fuxi meta-object model. The components of functionality are running in virtual environments presented by Fuxi Abstract Machine; FAM also serves weaver of the components in the two dimensions.

Though our design methodology is proposed in the context of embedded systems, it can also be used in other domains, such as office and web projects.

But experience of development also shows that our design methodology has some pitfalls. The major problem is that Fuxi language can describe the functionality well, but has no meanings yet to specify the constraints, especially the real-time constraints. This may limit our methodology in some time-critical systems.

REFERENCES

[1] Andrada, L., Baldan, P. et al, "AGILE: Software Architecture for Mobility", *Recent Trends in Algebraic Development Techniques – 16th Intl. Workshop, WADT 2002*, Frauenchiemsee, Germany. Vol. 2755 of LNCS, 2003

[2] Atkinson, MP et al, "An approach to persistent programming", *Computer Journal*, 26(3), 1983, pp. 360-365

[3] Atkinson, MP et al [1995] "Orthogonally persistent object systems", *VLDB Journal*; 4(3), 1995, pp. 319-401

[4] Baker, J. and Hsieh, W., "Runtime Aspect Weaving through Metaprogramming". *Proc. of AOSD 2002*, ACM Press. 20002, 86-95

[5] Barreto, L. and Muller, G., "Bossa: A language-based approach for the design of real time schedulers". In 10th International Conference on Real-Time Systems (RTS), 2002.

[6] K. Beck, *Extreme Programming Explained: Embrace Change*: Addison Wesley Longman, Inc., 2000.

[7] Bloch, J. "A Metadata Facility for the Java Programming Language", 2004.

[8] Borriello, G., Chou, P., Ortega, R., "Embedded System Co-Design: Towards Portability and Rapid Integration", Hardware/Software Co-Design, Sami, M.G. and Micheli,. G. Eds. Kluwer Academic Publishers, 1995.

[9] Bryant, A., Catton, A., Volder, K.D., Murphy, G.C., "Explicit programming". Aspect-Oriented Software Development, 2002, ACM Press, 10-18.

[10] Coady, Y., Kiczales, G., Feeley, M. and Smolyn, G., "Using AspectC to improve the modularity of path-specific customization in operating system code". Foundations of Software Engineering (FSE), 2001, ACM Press, 88 - 98.

[11] Cockburn. A., *Agile Software Development*. Boston: Addison-Wesley, 2002.

[12] Douence, R., Motelet, O. and Südholt, M., "A formal definition of crosscuts". In *Proc. of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, vol. 2192 *LNCS*. Springer Verlag, Sep. 2001.

[13] Elrad, T., Filman, R.E. and Bader, A. "Aspect-oriented Programming". *Communications of the ACM*, *44* (10), October 2001, pp.29-32.

[14] Filman, R.E. and Friedman, D.P., Aspect-Oriented Programming is Quantification and Obliviousness. In *OOPSLA2000 Workshop on Advanced Separation of Concerns*, (Minneapolis, USA, 2000).

[15] Highsmith, J., Cockburn, A., "Agile Software Development: Business of Innovation", *IEEE Computer*, Sep. 2001, pp.120-122

[16] Jones, P. S, *The Implementation of Functional Programming Languages*, Prentice-Hall 1987

[17] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. "Getting Started with AspectJ". *Communications of the ACM*, *44* (10), October 2001, pp.59-65.

[18] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., An Overview of AspectJ. In *ECOOP 2001*, (Budapest, Hungary, 2001), Springer-Verlag, pp.327-355.

[19] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M. and Irwin, J., Aspect-Oriented Programming. In *The 1997 European Conference on Object-Oriented Programming (ECOOP'97)*, (Finland,1997), Springer-Verlag, pp.220-242.

[20] Kiczales, G., Mezini, M., "Separation of Concerns with Procedures, annotations, Advice and Pointcuts", ECOOP 2005, LNCS 3586, pp195–213

[21] Lafferty, D. Cahill, V., "Language-Independent Aspect-Oriented Programming", OOPSLA'03 Anaheim, USA, pp1-12.

[22] Levis, P., Culler, D., "Maté: A Tiny Virtual Machine for Sensor Networks", *Proc. of 10th Intl. Conf. On Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct. 2002

[23] Mitchell, J. C., *Concepts in Programming Languages*, Cambridge University Press, 2003.

[24] Ossher, H. and Tarr, P. "Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software". *Communications of the ACM*, *44* (10) 2001, pp.43-50.

[25] Roper, M. D., Olsson, R. A., "Developing Embedded Multithreaded Applications with CATAPULTS, a Domain-specific Language for Generating Thread Schedulers", CASES'05, Sep. San Francisco, CA. USA, 2005, pp.295-233

[26] Sgroi, M., Lavagno, L., Sangiovanni-Vincentelli, A., "Formal Models for Embedded System Design", *IEEE Design & Test of Computers*, 2000 April-June, pp.2-15

[27] Spinczyk, O., Gal, A. and Schröder-Preikschat, W., "AspectC++: an aspect-oriented extension to the C++ programming language". 4th International Confernece on Tools Pacific: Objects for internet, mobile and embedded applications, 2002, Australian Computer Society, 53 - 60.

[28] Tarr, P., Ossher, H., Harrison, W. and Stanley M. Sutton, J., "N Degrees of Separation: Multi-Dimensional Separation of Concerns". In *Proceedings of the 21st International Conference on Software Engineering*, (Los Angeles, USA, 1999), IEEE Computer Society Press, pp.107-119.

[29] Theunissen, W.M., Kourie, D.G., Watson, B. W., "Standards and Agile Software development", Proc. of of SAICSIT 2003, pp.178–188

[30] Turk, D., France, R. and Rumpe, B. "Limitations of Agile Software Processes," *Proc. of Third International Conference on extreme Programming and Agile Processes in Software Engineering*, Italy, May 2002.