E-AoSAS++ and its Software Development Environment

Masami Noro
yoshie@it.nanzan-u.ac.jp

Atsushi Sawada
sawada@it.nanzan-u.ac.jp

Yoshinari Hachisu
hachisu@it.nanzan-u.ac.jp

Masahide Banno
Dept. of Info. & Telecomm. Eng., Nanzan University

Abstract

E-AoSAS++ is an aspect-oriented software architecture style for embedded software. It basically gives the style in which a set of state transition machines organizes a software. We have identified such concerns as state transition, concurrency, fault-tolerance, real-time, and error-handling. We categorize those concerns into two classes and found the way we call universal modularization pattern to package them in an orderly fashion. From the experience we have had through the construction of E-AoSAS++, we realized that we needed model for style construction. We defined XCC model which is construction model of architecture style. Based on E-AoSAS++, architecture centered software development environment is designed.

1. Introduction

Through the experience we have had in joint research projects with enterprises over a dozen of years, we came up with E-AoSAS++ which is an Aspect-Oriented Software Architecture Style for Embedded systems. Aspect-oriented technology, in these days, is assumed to be another better silver bullet in software engineering. The problem having been troublesome on modularization for almost half century appears to be, in these days, what is on crosscutting concerns. We found the way which we call XCC model (X-aspect-oriented software architecture construction model based on Component and Connector model) to constructing software architecture style on the universal modularization pattern (UMP henceforth) which we also found for modularizing crosscutting concern.

The objective of our research had been to construct an architecture style for embedded software. We realized that the only one single style was not enough to overcome the problem we faced. So-called embedded software is not a simple nor narrow application domain where the single style solves all problems in the domain. The domain needs more variety of the style and the style must be customizable, meta, and/or parameterized. We concluded that we need meta model which is a style for constructing an architecture style. Moreover, every science paradigm has a universal theorem which uniformly explains the world. We have come up with UMP which uniformly describes how components in the system having crosscutting concerns relate. Finally, we succeeded in defining E-AoSAS++ based on XCC model with UMP.

E-AoSAS++ is basically constructed by defining a component as a state transition machine (STM hereafter). Such concerns as concurrency, state-transition, real-time, fault-tolerance, and so on are handled in E-AoSAS++. In addition to the construction of E-AoSAS++, PLSE based on E-AoSAS++ has been discussed. Using PLSE as a methodology presenting meta-process, architecture centered software development environment has been designed.

In XCC model, concerns manipulated are sorted into three categories divided by two dimensions: architecture level and domain independency. Started from component-connector model, hierarchical definition of components is given, then aspect-oriented modularization in the categories is defined.

UMP resulted from that the nature for handling crosscutting problems was the configuration control. This approach can be conceived as Composition Filter[2] in architecture level. A set of meta description for presenting configuration change policy and configuration codes represents a package of aspect and core concern codes.

In E-AoSAS++ and its environment, UMP realizes aspect modularization at both base (architectural or code) and meta level. That is, in addition to that the pattern implements or represents the aspect, it handles different view of architecture in the development environment. Advantages on the usage of UMP are:

- It promotes simple implementation, that is, to give the implementation just for the pattern implements everything.
- Variety of the implementation will be much broader in an easy way since its implementation is simple.

In contrast to early aspect studies [6], E-AoSAS++ defines not just methodology on requirements study but gives the patterns. It also manipulates meta-concern for development support. Customizable development environment can be designed easily as a result. STM based model checking and code generation support such as Rapide[13], VERTAF[11], or so forth is for application engineering. E-AoSAS++ gives the basis for domain engineering from PLSE perspective as well as the application engineering. ArchMatE[5] could be another competitor of E-AoSAS++. It just shows the way to handle early aspect based on measurement. You can play with several indices it provides and automatic generation of software architecture. The research can be addressed as a quality assurance approach to architecture handling. Our research is much more towards how to describe the architecture. ArchMatE method can be used to the analysis and then E-AoSAS++ could be used to the design of software architecture. Hybrid System Method[3] also makes use of STMs to design architecture. It, however, clearly separates design and implementation. While state transition is a design issue in Hybrid System Method, E-AoSAS++ handles state transition uniformly through the development of architecture.

2. The Design of E-AoSAS++ with XCC model

2.1. Overview of E-AoSAS++

E-AoSAS++ defines a system as a set of STMs which run in concurrent. As previous works for supporting embedded system development [11, 13] show, modeling the embedded software in the form of STM set is a promising approach. E-AoSAS++ also takes this approach and then it defines the way to describe embedded software as a set of STMs running in concurrent.

We have identified the following crosscutting concerns through over a dozen-year experience from joint research with enterprises:

- concurrency,
- state transition,
- fault tolerance,
- real time, and
- error handling.

To reflect those concerns on the architecture, E-AoSAS++ gives aspect-oriented approach to designing software architecture.

The concerns above could be put into one of two classes:

- global concerns crosscutting over a system, concurrency and state transition concerns are of this type, and
- local concerns which crosscut over several components of the system, fault tolerance, real time, and error handling concerns are classified into this type.

The first class prescribes the basis of the style. That is, all components in an architecture in E-AoSAS++ are STMs concurrently executed. For the concerns of the second class, E-AoSAS++ provides the way to package local concerns called UMP as Composition Filter does.

2.2. E-AoSAS++ with XCC model

History of E-AoSAS++ construction

The original idea at the starting point of E-AoSAS++ construction was that we could describe embedded software (mostly software for vending machine control) as a set of STMs. In the earlier trials, we attempted to implement such a system in an object-oriented fashion. We encountered a crosscutting problem, that is, an object-oriented design which seemed to be perfect was apt to be revised drastically, redesign in worse case, to take non functional requirements such as runtime efficiency, storage saving, and so forth into account. Finally, we realized that aspect-oriented approach could be solution.

Through the experience in the design of E-AoSAS++, we have been trying to define XCC model which is a construction model of software architecture style. We could play with XCC model to construct another architecture style replacing types of components and aspects.

XCC model

The generic architecture style is the atomic element of the model. To give component and connector category yields a general architecture style. Then, a concrete architecture style can be obtained with types of components, connectors and concerns.

Generic Architecture

We borrow the component-connector model[9] as the generic architecture style. A component is a unit for modularization. A connector represents interconnection of the components. The reason we selected the component-connector model is that the model is well accepted and the basis of architecture documentation [4]. Actually, what we have done during E-AoSAS++ construction is that we gave the patterns for description of architecture documents as well as architecture itself. Hence, it is natural for us to take the component-connector model as the core.

General Architecture

The purpose we introduce the general architecture is to draw a clear line between general structure and specialized structure. All of general things in designing architecture style are put into general architecture. Concepts such as

hierarchy or nesting should not mix together with domain dependent or fine grained structure.

We prepare two categories of components: primitive and composite. A primitive component is a leaf element in the hierarchical structure. A composite component consists of a set of components which, in turn, can be either primitive or composite.

A composite component is for hierarchical definition of components. It basically represents OR aggregation. At least one of its subcomponents is selected at run time and the subcomponent works as a representative for the function requested. In a special case, all of the subcomponents can be active. The composite component represent AND aggregation, in the case. The policy component controls the collaboration (see Figure 1 [1]).

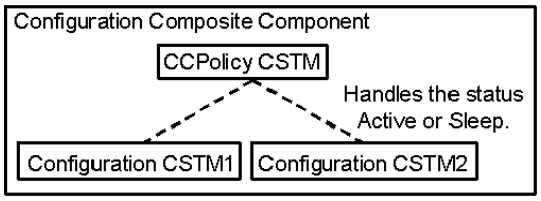


Figure 1. Composite component

Two categories of connecters: unicast and multicast are provided to bind components. The unicast connector is usually used one. The multicast connector in Figure 2 shows how it is used in a composite component. An event to a composite component is distributed to all of its subcomponents. The reason why the multicast connector is needed in addition to the unicast connector is that we need the way to naturally present potential concurrency.

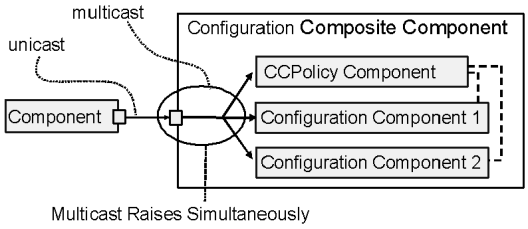


Figure 2. Components and connectors

To sum it up, a general architecture is obtained by giving component categories that are a primitive component consisting of aspects and a composite component.

E-AoSAS++

The concern we have identified can be categorized from two perspectives: architectural and of domain independency as in Figure 3. From the architectural perspective, concerns are divided into global and local. Local concerns are further categorized from the perspective of domain independency.

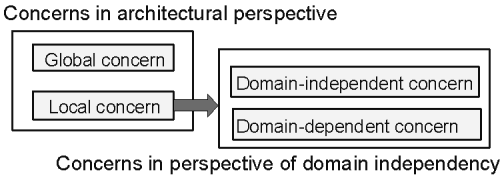


Figure 3. Categorized concerns

The global concern in architectural perspective crosscuts across all components over the system. Since the global concern crosses over the system, it is manipulated like Aspect-J[1] does. We believe that the manipulation is the most natural and popular way in the sense that you can find language processors here and there in implementation level. Aspect codes for the global concerns would be platform codes, libraries, or application frameworks in the implementation level.

The local concern in architectural perspective is, on the other hand, packaged into a component. A set of components in the component organizes a pattern to implement the concern. Real-time and fault tolerant concerns are domain dependent since the concerns are needed to present an architecture in a specific domain. They can be abstracted to the configuration control concern which we call domain independent one because it is general and universal. An exception handling concern is in-between of the two. Since the local concern is what cuts across not all components but just several components, it seems to be natural for us to handle it as Composition Filter does.

A component of E-AoSAS++ is an STM. To promote independency among components, the only connector type we adopt is inter-aspect description.

2.3. Handling Global Concerns

Global concerns we have identified are concurrency and state transition. At first, we tried to manipulate the concurrency as the secondary concern as usual. This approach is, however, not so attractive that aspect codes are apt to be specialized and are hard to peal from core concern codes. The fact that most of concurrent programming languages like Ada[14] employ syntax elements for concurrency as a first class agent tells that it is hard to separate from core concern. We put the paradigm upside down. To make concurrency core concern, it becomes easy to separate application logic from codes for concurrency. We have demonstrated, at the implementation level, that a simple mechanism of wait-and-signal is enough to implement the concurrency concern.

[1] We borrow a notation for a UML package to represent a component. We also borrow other UML diagrams such as sequence diagram for describing architecture.

2.4. Universal Pattern for Modularization

The composite component provides, in addition to representing hierarchy, UMP structuring components. A crosscutting problem on concerns, local concerns in particular, can be conceived as a problem on configuration control for handling multiple module structure caused by the cross cutting concerns. For the purpose of simplicity and uniformity of architecture style, we use the component-structuring pattern of configuration control to represent aspect (module which is defined by concern). As mentioning below, manipulating architectures from multiple views can be also defined as a problem on configuration control. Hence, we also use the pattern for integrating tools handling the architectures from the multiple views.

UMP for Representing Local Concerns

Local concerns are packaged into a component with UMP. Figure 4 examples real-time concern packed in a composite component. As in Figure 4(a), there are four subcomponents in real-time processing configuration composite CSTM (concurrent STM): CSTM for representing configuration control policy, for the timer, for normal processing configuration, and exception. All subcomponents, in turn, consists of three aspects: concurrency, state transition, and application logic.

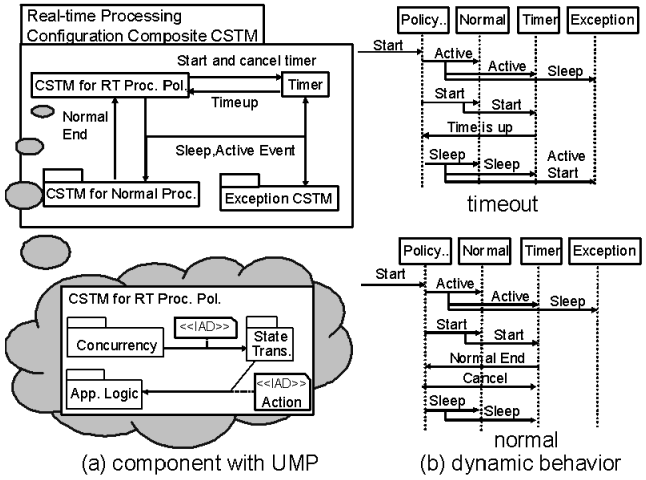


Figure 4. Real-time concern in composite component

Dynamic behavior of the subcomponents is given in Figure 4(b). It cartoons the case of "time-out". All but the exception CSTM are active at first. Then, the timer and normal configuration CSTM are simultaneously started. When a certain amount of time is expired before the required processing is achieved, configuration changes to the exception. Figure 4(b) also presents the normal processing.

The way we packed real-time processing is not just a usual modularization but aspect-oriented because the following statements can be held.

1. UMP could be common to all real-time processing.
2. Normal configuration processing CSTM would need no change.
3. Since the composite component intercept the event a client raises, no modification is needed for the client code.
4. The timer can be reused without any modification.
5. Moreover, a sleep or active event is handled by concurrency aspect being a part of CSTM.

In this example, the timer and exception handling organize real-time aspect. The policy CSTM can be regarded as a meta and inter-aspect description.

UMP for Handling Multiple Views of Software Architecture

As Kruchten claimed[12], multiple views are necessary to describe software architecture. Nowadays, software architecture documentation research group in SEI also claims that three viewtypes are needed for the documentation[4]. Those views can be abstracted and converged into two types of architecture as Kang demonstrated [8]. UMP can be used to capture these multiple views of architecture in a packaged component. For the simplicity and as the first step, we select Kang's views to represent software architecture: conceptual and implementation.

With UMP, multiple views of software architecture can be packaged into one composite component (see Figure 5). Assume that there are a conceptual architecture and an implementation architecture and each of them is modularized into a composite component. The composite components are configuration components. The policy component, in this case, would be meta description which tells how the conceptual and implementation architecture are handled. In case of machine support for the handling, there may be tools for manipulating each architecture. The meta description in the policy may describe how tools are collaborated. Or, the way to manipulate intermediate code for the tools may be described. Thus, the policy is used for tool integration in software development environment based on E-AoSAS++.

3. Case Study

In order to demonstrate that E-AoSAS++ is a sound and useful architecture style, an example of describing an architecture for vending machine control software is given. This case study includes the followings.

1. How crosscutting concerns are manipulated in an architecture based on E-AoSAS++.
2. How the conceptual and implementation architecture are handled in UMP.

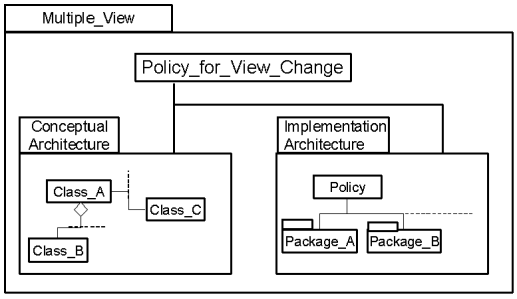


Figure 5. Multiple views of software architecture

3.1. Requirements

Assume that it is required to design software for controlling vending machine for selling canned beverage. The vending machine displays beverage cans on its surface. It accepts only coins but bills. To buy a can of beverage, you have to press the button for your a coke. Then, a can of the beverage you chose will come out of the outlet of the vending machine. You can get change by turning the change lever. The change will come out of the change pool. The amount you put is shown in the display. Buttons for beverages you can buy will be on if enough amount is put, off otherwise.

The amount you put will be automatically returned when you do not perform any action for thirty seconds since you put the amount.

3.2. Constructing Software Architecture

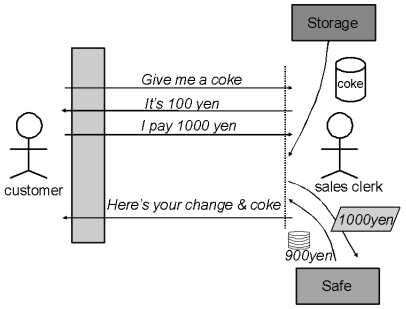
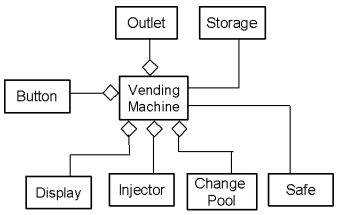


Figure 6. Scenario to buy a coke

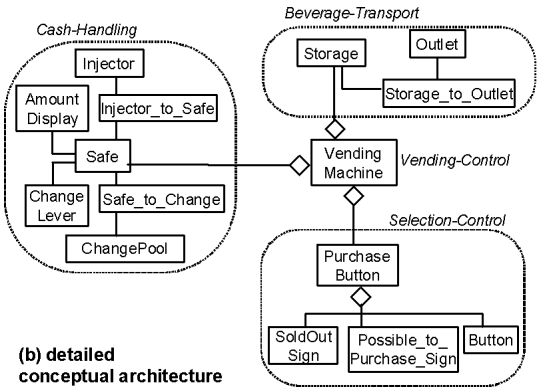
We start from writing a scenario for beverage purchase. We, first, assume the situation in Disneyland, that is, you ask a salesclerk to give you a coke. Figure 6 shows a simple scenario to buy a coke. The second steps are not required to implement since you can see the price and the can of the beverage you want on the vending machine. When the vending machine is used, the first step becomes third, that is, after you put the amount. Pressing the corresponding button means that you tell the machine to give a coke. We can recognize objects such as the storage and the safe.

Conceptual Architecture

From what we can get by reading the requirements and the scenario, we could construct a conceptual architecture for vending machine as in Figure 7(a). The architecture is object-oriented. The conceptual architecture recognizes only hardware components that can be identified naturally from the requirements and the scenario. It does not care anything about any electric hardware inside the vending machine (such as processors packaged into the machine).



(a) rough conceptual architecture



(b) detailed conceptual architecture

Figure 7. Conceptual architecture of vending machine

The detailed conceptual architecture in Figure 7(b) can be obtained when the inside of the machine is examined in the deeper level. Paths through which coins or cans go was added to the architecture in Figure 7(a). The structure of the button is also analyzed and a new component, the purchase button having a button, a sold-out sign, and a possible-to-purchase sign.

Implementation Architecture

The implementation architecture is a surgeon view of the architecture if we call the conceptual architecture a physician view of the architecture. As a matter of fact, there are four processors inside the vending machine. One of the processors is for handling cash, other for beverage transportation, one more for selection control, and yet another

for coordination of the four. Figure 8 is the implementation architecture where the functionality of processors are took into consideration. As in the figure, UMP is used.

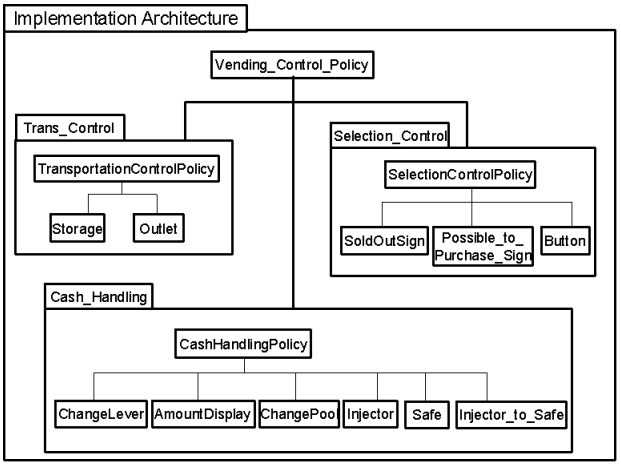


Figure 8. Implementation architecture of vending machine

Architecture for non Functional Requirements

There is one non functional requirement: real-time. Here, we consider how real-time concern is modularized. There are two ways to answer this question. One way is to package it on the conceptual architecture and another is to do it on the implementation architecture.

Components relating to real-time concern in the conceptual architecture are Injector, Safe, and Change Pool in Figure 7(b). If we try to localize the modification as much as possible, to implement the "Safe with time-out" is the way to go. Figure 9(a) shows the structure of the "Safe with time-out". Dynamic behavior is given in Figure 9(b).

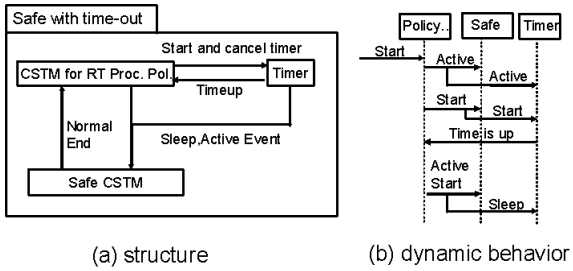


Figure 9. Safe with time-out

In the implementation architecture, it is also possible to implement the "Safe with time-out". Here, we describe another answer using a more rough-grained component. The STM named CashHandling in Figure 8 would be packaged with UMP. Figure 10 shows the answer.

Bridging over Conceptual and Implementation Architecture

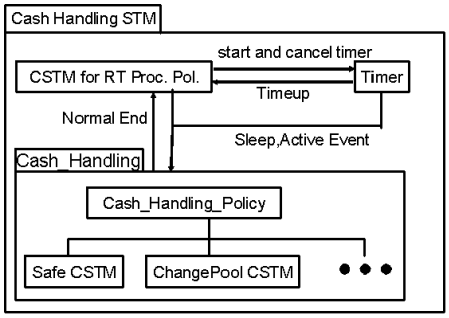


Figure 10. Cash Handling STM with UMP

The conceptual and implementation architecture are packaged into one component with UMP as well as that a set of components can be packaged into one composite component. Figure 11 shows how we can describe it in E-AoSAS++. In addition to having the root component, all primitive components at leaf level are shared by both conceptual and implementation architecture. In other words, intermediate composite components representing conceptual and implementation architecture show interpretation of components gathering. The root component bridging over the two would tell how they are converted back and forth.

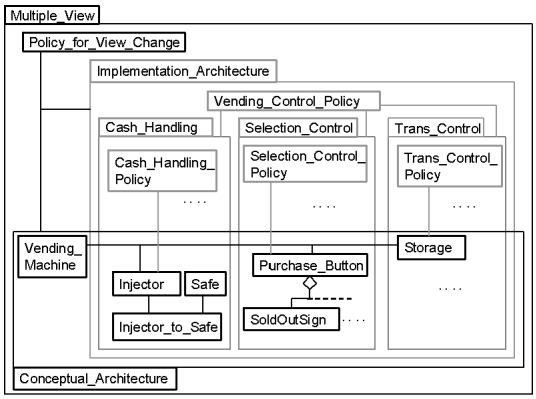


Figure 11. Conceptual and implementation architecture

4. Discussion

E-AoSAS++ gives UMP which could be conceived as a tool for handling early aspect. Like that design patterns [7] gives a guide-map for designing software in an object-oriented way, in some sense the way for separation of concerns, E-AoSAS++ guides a software architect to better design of software architecture. Moreover, E-AoSAS++ defines hierarchy of concerns that the architect has to take into account and gives the way for describing aspects for concerns with UMP. UMP is actually a meta pattern in two

senses. First, UMP defines the pattern for writing an architecture witch is, in turn, a pattern of software. Second, UMP can be utilized for handling views of architecture in a development environment. As mentioned in Section 3, the conceptual and implementation architecture can be packed into one composite component and its poly component may describe the view transformation from conceptual to implementation and vice versa. This section gives discussions to demonstrate E-AoSAS++'s contribution to software development support in the context of UMP.

4.1. Assumed Development Process

Software development based on E-AoSAS++ assumes a software process of PLSE[10]. Core assets are:

- software architecture in E-AoSAS++,
- components (STMs and their actions),
- platform codes, and
- software tools including generators.

In the domain engineering part (core assets development), the main activity is refinement of architecture. Every time the application development is achieved, a revised or new architecture will come out. In core assets development, the new architecture would be abstracted and stored.

In the application engineering part, a software architecture for the development is selected from core assets and customized for the development. Based on the architecture, the following steps are usually performed.

0. There assumed to be platform codes for global concerns and pattern or application framework implementing local concerns. The application architecture is also assumed to exist.
1. For the conceptual architecture, the following steps are performed.
   - 1.1. To design a set of CSTMs is to design an application.
   - 1.2. Generate CSTMs' specification in CSP from a set of CSTMs.
   - 1.3. Describe a specification of inter-aspect description.
   - 1.4. Pre-execution check.
2. For the implementation architecture, the steps 1.1 through 1.4 followed by the steps below are enacted.
   - 2.1. If a CSTM meeting the specification is in the repository then customize the CSTM. Generate the CSTM in platform codes, otherwise.
   - 2.2. If an inter-aspect description corresponding to the specification is in the repository then customize the inter-aspect description. Otherwise, describe inter-aspect description as the specification denotes.
3. Register the CSTM and inter-aspect description in the repository.

4.2. Assumed Development Team

Development team is assumed to consist of three kinds of members. One group has domain engineers and another is a group of architects. Yet another group is composed of application engineers.

A domain engineer develops core assets. The maintenance of the core assets is also the task assigned to the domain engineer. Before the application development starts, the engineer selects and tailors the specification which meets the requirements. The engineer abstracts assets developed in the application development and stores them into core assets.

The first stage of each application development, an architect prepares the software architecture for the application in E-AoSAS++. The architect eats the requirement specification and selects and fine tunes software architecture in the core assets.

The application engineer develops software based on the architecture in E-AoSAS++. Components, libraries, patterns, etc. in the core assets are used to write the software.

4.3. How E-AoSAS++ Contributes to the Development

In E-AoSAS++, concerns are categorized into global and local. The global concerns contributes to enhance the portability of platform codes. Since the local concerns are described by UMP, it is not only highly reusable but also easy to implement. UMP is also used to present view association. To use the pattern in this way enhances the customizability of development environment.

The portability of platform codes is realized by handling environmental factors as global concerns in E-AoSAS++. Concurrency is the typical instance of this. E-AoSAS++ assumes the simplest signal-wait type concurrency control as a primary concern. Only the runtime library for the concurrency control is environment dependent. All we have to do to port the platform are design and implement the library for the environmental factors such as differentiation of language, operating systems, etc.

UMP is an only and single pattern for packaging local concerns. In order to implement a local concern aspect, all we need to do is to implement UMP and customize it. As a matter of fact, we have designed a generator of application framework for the pattern. The framework is generated from the component in the architecture and an application engineer would customize it.

To package, with UMP, multiple views of architecture in a component enables describing a tool integration policy. We can write policy for view association and the policy could be used for tool collaboration. That is, the pattern

gives facility to customize tool integration policy, in other words, environment itself.

4.4. Design of Software Development Environment based on E-AoSAS++

An instance of software development environment based on E-AoSAS++ is shown in Figure 12. The environment is architecture-centered and is based on above development process on E-AoSAS++. Actually, aspect categorization in E-AoSAS++ gives process centered tool integration. From the process view, E-AoSAS++ shows the guide-map to handle early aspect. The development process mentioned above integrates methods to realize codes from early aspect while certain level of reliability is kept.

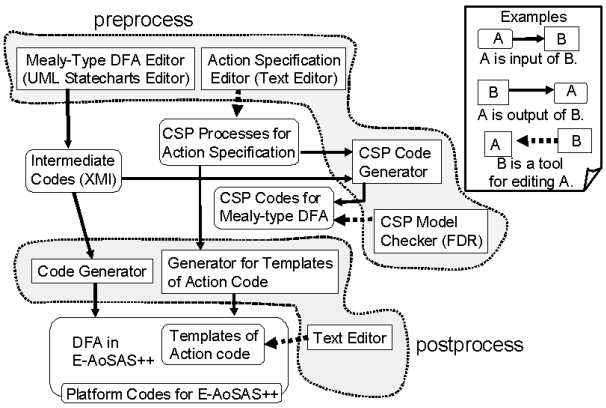


Figure 12. Software development environment based on E-AoSAS++

5. Conclusion

We have described the design of E-AoSAS++ which is an aspect-oriented software architecture style for embedded software and its environment. We came up with E-AoSAS++ through joint research experience over a dozen of years. We abstract the way we defined E-AoSAS++ and ended up with XCC model which is a construction model for architecture style.

In XCC model, we started from the component-connector model as the generic architecture, then define the general architecture where UMP is used to make hierarchy of the components. Concerns sorted in global and local, STM as a component, and inter-aspect description as a connector are given, then E-AoSAS++ is obtained.

To handle environmental factors as global concerns, platform codes that has to do with portability are separated as an aspect. The approach enhances the portability of the platform code.

UMP is used for packaging not only local concerns but also multiple views of software architecture. Modularizing the local concerns with the pattern supports easy implementation of local aspects. To pack multiple views of software architecture enables customization of development environment.

The design of software development environment was given. E-AoSAS++ implicitly defines software development process mentioned earlier. Tool integration is based on the process.

One possible future research topic is to incorporate requirements specification handling. We are planning to integrate requirements specification model in feature diagram into core assets. If we could find out the mapping rule or procedure to associate requirements specification to architecture, PLSE on E-AoSAS++ would be completed.

Acknowledgment
This research is supported by KAKENHI, Grant-in-Aid for Scientific Research (C) (18500030, 19500028) , and Nanzan University Pache Research Subsidy I-A-1 2007.

References

[1] *AspectJ*. http://eclipse.org/aspectj/.
[2] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *CACM*, 44(10):51–57, 2001.
[3] C.Angelov, N.Marian, K.Sierszecki, and J.Ma. Model-based design and verification of embedded software. In *Proc. of the 5th European Workshop on Research and Education in Mechatronics*, 2004.
[4] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.
[5] J. A. Diaz-Pace and M. R. Campo. ArchMatE: from architectural styles to object-oriented models through exploratory tool support. In *OOPSLA '05*, pages 117–132, 2005.
[6] *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*. http://early-aspects.net/.
[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.
[8] K. C. Kang, J. Lee, and P. Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, pages 58–65, 2002.
[9] M.Shaw and D.Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice Hall, 1996.
[10] L. M. Northrop. SEI's Software Product Line Tenets. *IEEE Software*, pages 32–40, July 2002.
[11] P.A.Hsiung and C.Y.Lin. VERTAF: An Application Framework for the Design and Verification of Embedded Real-Time Software. *IEEE Trans. on SE*, pages 656–674, 2004.
[12] P.B.Kruchten. Architectural blueprints - the "4+1" view model of software architecture. *IEEE Software*, pages 42–50, 1995.
[13] *The Rapide Project*. http://pavg.stanford.edu/rapide/.
[14] S. T. Taft and R. A. Duff, editors. *Ada 95 Reference Manual: Language and Standard Librraries*. Springer-Verlag, 1997.