

The Method for Verifying Software Architecture with FSP Model

Kim, Jungho

SKC&C Inc., SK u-tower 25-1, Jeongja-dong, Bundang-gu, Seongnam-si, Gyeonggi-do 463-844, Korea

kimjh@skcc.com

Abstract— C&C view of Software architecture is one of the modelling languages to describe and analyze runtime architecture of a software system. However, it is not sufficient to describe runtime architectural behaviours with only C&C view because C&C view architecture does not provide a sequence and state-chart of a software system. On the other side, we would use Finite State Process (FSP) model for describing behaviour and property of component in a software system. FSP model can be simple to show behavioural sequence and state of component. Moreover, we can check a property of a component with FSP model, once a FSP model of a software system is given.

In this thesis, we propose an approach to model and verify C&C view architecture for a software system. Additionally, in using FSP model, the requirements represented using the FSP properties can be automatically checked against the C&C view architecture with a Labeled Transition Systems Analyzer (LTSA). This approach can support early detection of system's behaviour errors automatically.

Keywords— C&C viewtype, Software Architecture, FSP Model

I. INTRODUCTION

C&C view architecture is one of the modelling languages that are used to describe and analyse runtime architecture of a software system. However, even if C&C view is for runtime architecture, we cannot see behaviours and state of a software system only in C&C view. We would use Finite State Process (FSP) for describing behaviour and property component of a software system. FSP would make it simple to show sequence and state of component. Moreover, we can check the property of a component using FSP property, if we have a FSP model for the component.

In this thesis, we suggest a way of translating C&C view architecture and requirements (or use case scenario) into a FSP model for describing behaviour of a software system. Additionally, once FSP model is created, we can check behaviour properties of a software system using FSP property, such as concurrency problem, before implementing a software system.

The first contribution of this thesis is to make FSP model for components and connectors using C&C view architecture and requirements in a software system. If we have system requirements in the form of scenario fragments like use case

scenario or sequence diagram and C&C view architecture of a software system, we can get Message Sequence Chart (MSC), and MSC can be converted into state-charts by synthesized algorithm [4]. These state-charts could be converted into FSP model straightforward. This means that behaviour of connectors and components can be Modeled with FSP.

The second contribution of this thesis is to check property of a software system using Labeled Transition Systems Analyzer (LTSA). If we have FSP model for components and connectors, we can get a property of a software system from requirements. This property can be modeled with FSP property. Thus, we can check a property of components and connectors using FSP model for a software system in FSP tool, such as Labeled Transition Systems Analyzer (LTSA). It means that if we can check a system property using LTSA before implementing a software system.

The organization of the remainder of this dissertation is as follows. Chapter 2 briefly considers architectural research related this thesis, such as ADL (e.g. Wright and Darwin), DiscoTect. Another part is about translating to statechart from MSC. Moreover, we define what FSP is and C&C view and give some general information of C&C view and FSP. In chapter 3, we describe the theory of this thesis, and give a simple example for verifying this theory. In Chapter 4 we can see an application example for showing the describing FSP model for components and connectors and checks the properties of a software system by using FSP property in LTSA. Chapter 5 is a conclusion part that describes the contributions of this thesis.

II. BACK GROUND

In this section, we consider general information of architectural view and FSP model are introduced for supporting this theory. Additionally, architectural research, which is related this thesis, are described in this chapter. We introduce ADL like Wright and Darwin, which describes behaviour of architecture. Integration theory is introduced for coming up with differences between architecture model and a developed software system, such as DiscoTect and ArchJava. Then we give an introduction of synthesized algorithm for translating to statechart from MSC.

A. Architectural View

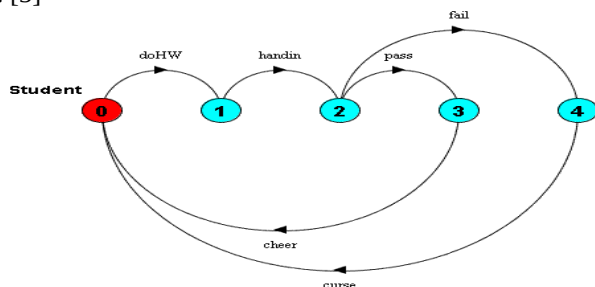
Modern software architecture practice embraces the concept of architectural views. A view is a representation of a set of system elements and relations associated with them. Views are representations of the many system structures that are present simultaneously in a software system. Modern systems are more than complex enough to make it difficult to grasp them all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures, which we represent as views. Different views often carry different burdens with respect to quality attributes of the system.

For instance, a view that shows a system structured as layers can be used to engineer certain kinds of modifiability (such as platform portability) into the system, and can also be used to communicate that aspect of the system's design to others. By contrast, a view showing how the same system is structured as a set of communicating processes can be used to engineer certain kinds of performance into the system, and also be used to communicate that aspect of the system's design to others.

Some authors prescribe a fixed set of views with which to engineer and communicate architecture. Rational's Unified Process, for example, is based on Kruchten's 4+1 view approach to software. The Four Views model is another example. A recent trend, however, is to recognize that architects should produce whatever views are useful for the system at hand. IEEE 1471 exemplifies this philosophy; it holds that an architecture description consists of a set of views, each of which conforms to a viewpoint, which in turn is a realization of the concerns of one or more stakeholders. This philosophy about views leads to the fundamental principle of the views-and-beyond approach: Documenting architecture is a matter of documenting the relevant views, and then adding documentation that applies to more than one view. [1, 2]

B. FSP model and LTSA

The Finite State Process (FSP) is a specification language which provides a concise way of describing Labeled Transition Systems (LTS). FSP model is to describe the sequence of a software system, and to describe property of a software system. Figure 1 is a simple example of FSP model, which model behaviour of a student with FSP model. The brief explanation of student behaviour is that he hands in his homework after a student does homework. If homework is passed, student is cheer up. Otherwise, he curses because he is fail. [3]



Student = (doHW -> handin -> (fail -> curse -> Student || pass -> cheer -> Student)).

Figure 1. An FSP model and its state diagram

The purpose of FSP can be to model a software system as sequence and state of procedure. In general, when we want to make FSP model, we used to draw a structure diagram, which is composed of component of a software system for FSP model [3]. If we make FSP model, we can check the safety and progress check with FSP model tool like Labeled Transition Systems Analyzer (LTSA). This may be a useful and easy check of a system property.

LTSA is a model checker developed by Jeff Kramer and Jeff Magee at Imperial College. The input to LTSA is a set of interacting finite state machines written in the process algebra FSP. Each FSP state machine definition is translated into a Labeled Transition System for analysis by LTSA. A state machine is defined in terms of states consisting of a list of transition choices. Each transition may have a condition associated with it, followed by a list of sequential events, and ending with the name of another state. When two or more state machines use the same event, the event is assumed to happen synchronously in all state machines.

CLOCK = (tick -> tock -> CLOCK).
CHIME(MaxValue=10) = COUNT_SECONDS[0],
COUNT_SECONDS[seconds:0..MaxValue-1] =
(when seconds < MaxValue-2
tock -> COUNT_SECONDS[seconds+2]
| when seconds >= MaxValue-2
tock -> chime -> COUNT_SECONDS[0]).

III. APPROACH STEPS

In this thesis, we use FSP model for describing architectural behaviour and check the system's behavior property with LTSA. Figure 2 is the steps for showing of this thesis' approach.

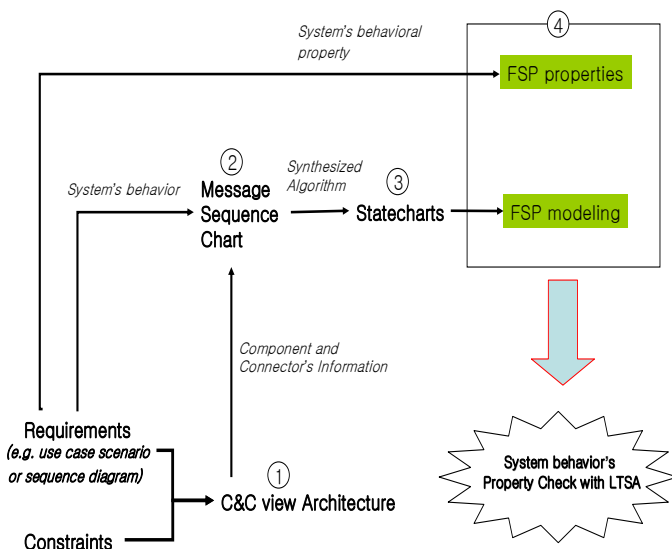


Figure 2. The approached step for checking system's behavior

As Figure 2 is shown, the detail explanation of steps is described in below order.

- ① C&C view architecture is created with requirements and constraints. [1, 2]
- ② C&C view architecture describes components and connector, and gives information of runtime behaviour of a software system. Requirements can give information of behaviour in a software system. Thus, Message Sequence Chart (MSC) is created with requirements (e.g. use case scenarios or sequence diagram) and C&C view architecture. [4, 5]
- ③ After making the MSC, synthesis algorithm is applied to translate MSC to statecharts of components and connectors. These statecharts of components and connectors must be described with FSP model straightforward. [3, 4, 5]
- ④ The system's property can be described in FSP property from requirements. Moreover, if we have FSP model of components and connectors and FSP property of a software system's behaviour, the system's property can be checked with FSP modeling tool like LTSA before implementing a software system.

The above steps are described and proved with a simple example from chapter 4.

IV. APPLICATION EXAMPLE

An application example is a translator that generates C code from TTCN –MP specification. This system is a real system that can be used for protocol testing. TTCN has been used to specify tests for many kinds of applications, including mobile communications, wireless LANs, cordless phones, broadband technologies, CORBA-based platforms, and Internet protocols. Accordingly, this system is to create

translator for translating TTCN-MP specification to Abstract Test Suite (ATS) written in ANSI C code, and to provide an adaptation library for targeting from Abstract Test Suite (ATS) to Executable Test Suite (ETS). The rationale for extensibility of the system is that the system itself is merely a basic model for translating test cases from TTCN-MP form to C language. Figure 3 is the business context diagram of this translator system. A test engineer prepares test cases written in TTCN-MP form before running this translator system and adds the test cases to the translator system as an input file. Then, the translator system generates abstract test suites (ATS) written in C. The user may modify the test cases to run in the target environment, compile the test suites (ETS), and link them with the pre-provided adaptation library by using an ANSI C compiler. Finally, the user can test a target system with the test suites designed for the target system.

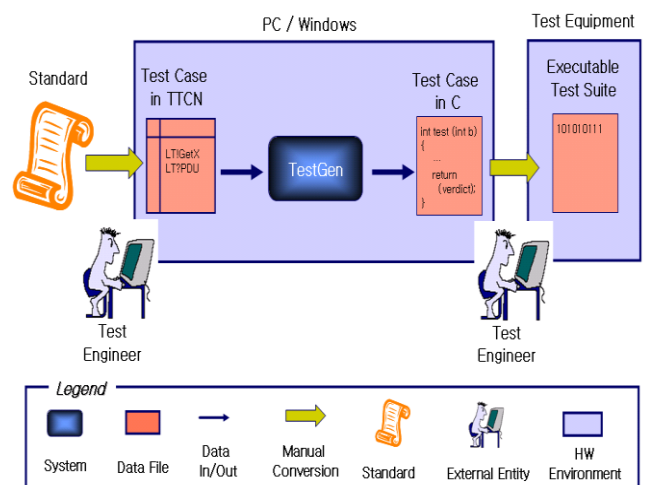


Figure 3. Business Context Diagram of TestGen

This case uses simple synchronous connector (e.g. call-return, data flow style), so connector does need to be modeled with FSP model in application example system.

A. Describing C&C view architecture

Figure 4 gives an overview of the runtime architecture of Translator system using the C & C view. The components are Parser, Semantic Analyzer (Semantic), Code Generator (Code_Gen), Symbol_Table, Tree, and Main Controller (Main_Controller). Main_Controller starts the translation process by calling three components sequentially. Parser reads TTCN-MP specification and collates source characters into tokens. Symbol Table manages a table mapping these words to symbol descriptions which are passed around during the rest of the compilation process. Parser checks the symbols in the Symbol Table against a grammar and usually constructs a tree representing the source. Semantic Analyzer checks the tree for semantic correctness and modifies it to account for implicit operations. Finally, Code Generator produces C code by traversing the tree.

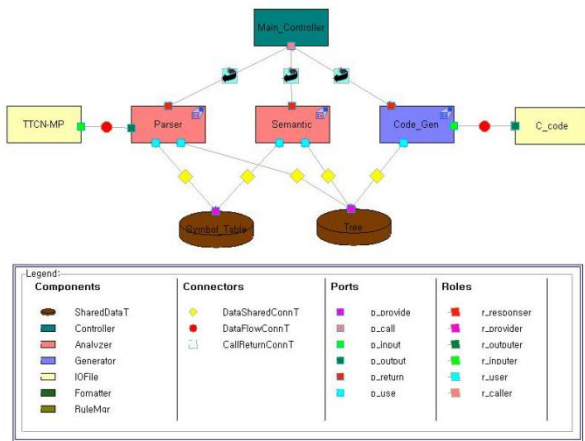


Figure 4. C&C view of translator

B. Creating MSC

We can make the MSC(Message Sequence Chart) of application with requirements and C&C view architecture in Figure 5.

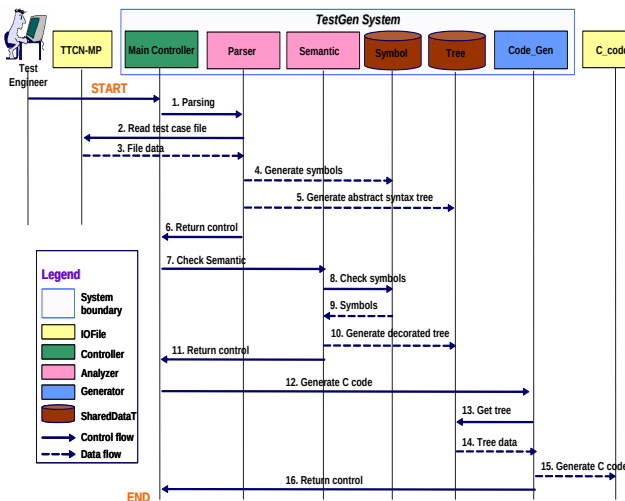


Figure 5. A MSC for the translator system.

Components of MSC are from C&C view architecture, and a traditional compiler provides a procedure of compiling within Parser, Semantic, Code generator, Symbol and Tree.

C. Describing Statechart and FSP Model

FSP model can be described from MSC by synthesized algorithm [4]. The connector is too simple to describe FSP model, so we make only component of system. At first, we can make FSP model of 'Main_Controller' component. In sequence chart, 'Main_Controller' component call 'Parser' first, and then it call 'Semantic'. Finally this component call 'Code_Gen' component. The statechart of 'Main_Controller' is like Figure 6.

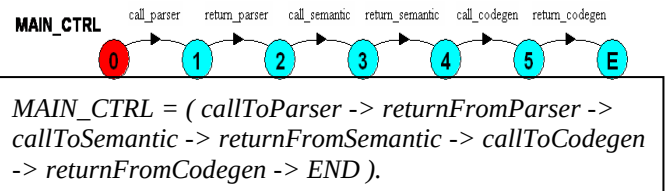


Figure 6. The state diagram of Main_Controller

Next, we can generate 'Parser'. In Figure 7, when 'Parser' component is called by 'Main_Controller', this component calls 'TTCN-MP' first, then it calls 'Symbol_Table' and 'Tree'. After completely reading TTCN-MP, this component is ended.

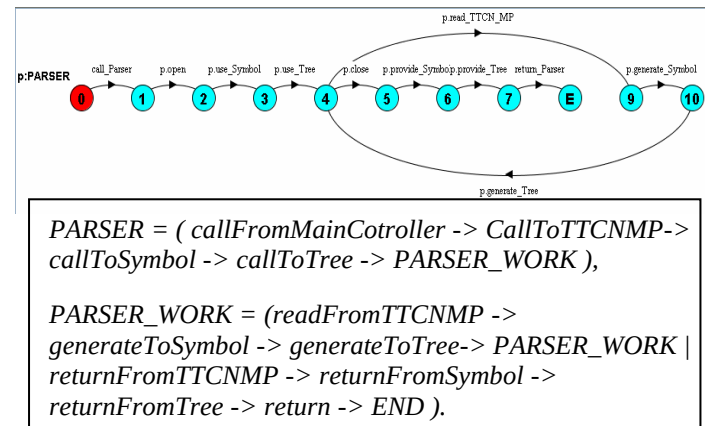


Figure 7. The state diagram of Parser

We can describe 'Code_Gen' component as same as 'Parser' component. In Figure 8, when 'Code_Gen' component is called by 'Main_Controller', this component use 'Tree'. This component opens the C code file and writes C codes until 'Tree' component is empty. After completely writing C code, 'Code_Gen' component is finished.

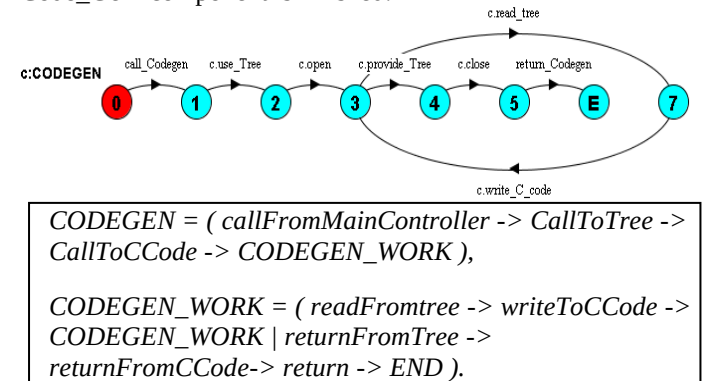
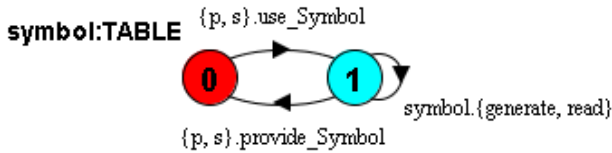


Figure 8. The state diagram of Code_Gen

We can describe FSP of 'Symbol_Table' component in sequence diagram. Symbol table can be called 'Parser' and 'Code_Gen'. 'Symbol' asserts to generate and read symbol. The statechart is described in Figure 9.



```
TABLE = (call-> TABLE_WORK),
TABLE_WORK = ( read -> TABLE_WORK | generate ->
TABLE_WORK | return -> TABLE).
```

Figure 9. The state diagram of Symbol_Table

. Finally, if we composite all component, we can make FSP model of all system. FSP model is like below box;

```
||SYSTEM = (ttcnmp:FILE || ccode:FILE || symbol:TABLE
|| tree:TABLE || p:PARSER || s:SEMANTIC ||
c:CODEGEN || MAIN_CTRL) /{callToParser/p.call,
returnFromParser/p.return,callToSemantic/s.call,
returnFromSemantic/s.return,callToCodegen/c.call,
returnFromCodegen/c.return,p.callToSymbol/symbol.call,
p.returnFromSymbol/symbol.return,p.callToTree/tree.call,
p.returnFromTree/tree.return,s.callToSymbol/symbol.call,
s.returnFromSymbol/symbol.return,s.callToTree/tree.call,
s.returnFromTree/tree.return,c.callToTree/tree.call,
c.returnFromTree/tree.return}.
```

D. Check system behavioural property with LTSA

We can also make an architectural behaviour property from requirement and architectural decision. The below FSP property describes the way of generating C code from TTCN spec, which is most important safety property in translator system.

```
property P = ( p.read_TTCN_MP -> p.generate_Symbol ->
p.generate_Tree -> s.check_Symbol -> c.read_Symbol ->
c.write_C_code -> P).
```

We can check the above property in generating FSP model of translator system with LTSA in Figure 10.

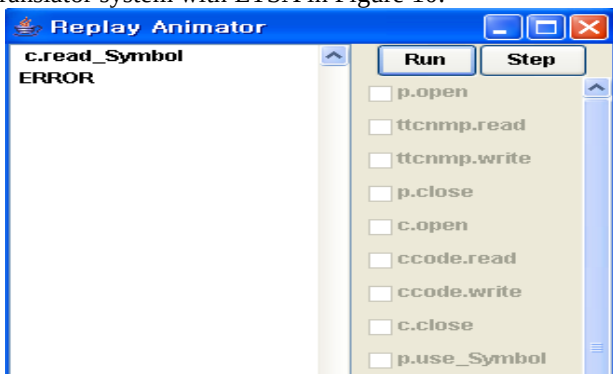


Figure 10. . FSP property check in application system

There must be dead lock in FSP model. It is because 'Symbol_Table' may update by 'Parser' component until the

TTCN-MP file is empty. That is to say, while 'Parser' component update symbol in 'Symbol_Table', 'Code_Gen' component should wait to read symbol in 'Symbol_Table'. Thus, 'Main_Controller' should exist and have an ability of controlling concurrency problem in symbol table in translator system. In conclusion, if we only have sequence chart without FSP model, we cannot check the concurrency problem in a translator system. However, because we can describe FSP model in translator, we are able to check the safety property of translator system with FSP model with LTSA before implementing.

V. CONCLUSIONS

C&C view architecture is one of the modeling languages to describe and analyze runtime architecture of a software system. However, it is not sufficient to describe runtime architectural behaviours with only C&C view because C&C view architecture does not provide a sequence and statechart of a software system [2], so we would make Message Sequence Chart (MSC) of sequence diagram in UML for describing system behaviour [5]. MSC has also some limitation of describing behaviour of component and connector, because MSC is focus on describing system behaviour itself. FSP model can be simple to show behavioural sequence and state of component of a system. Moreover, we can check a property of a component with FSP model, once a FSP model of a software system is given.

In this thesis, we proposed an approach to model a component and a connector using FSP model from C&C view architecture. If we have system requirements in the form of scenario fragments like use case scenario or sequence diagram and C&C view architecture of a software system, we can get Message Sequence Chart (MSC), and MSC can be translated into statecharts by the synthesized algorithm [4]. These statecharts could be converted into FSP model straightforward. This means that behaviours of a connector and a component can be modeled with FSP.

Additionally, in using FSP model, the requirements represented using the FSP properties can be automatically checked against the C&C view architecture with a Labeled Transition Systems Analyzer (LTSA). This approach can support early detection of system's behaviour errors automatically, so this thesis has a contribution of checking system' behaviour before implementation.

VI. REFERENCES

- [1] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik. "Abstractions for Software Architecture and Tools to Support Them." In IEEE Trans. on Soft. Eng. 21(4), 1995.
- [2] Mary Shaw, D. Garlan, "Software Architecture; Perspectives on an Emerging Discipline" Prentice Hall, pp. 19 – 32, 1996.
- [3] P. Clements, F. Bachmann, Len Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. "Documenting Software Architecture; Views and Beyond" Carnegie Mellon Univ. Software Engineering Institute. Addison-Wesley, pp. 18 – 22, 135 – 138, May 2003.
- [4] J. Magee, J. Kramer. "Concurrency; State Models & Java Programs" on Wiley, pp. 122 – 122 – 133, November, 2000.
- [5] D. Harel, H. Kugler, A. Pnueli. "Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements" on Springer-Verlag Berlin Heidelberg, LNCS 3393, pp. 309-324, 2005.