Statechart: A Visual Language for Workflow Specification

Wendy Wenhui Zhang, Theresa Beaubouef, and Huilin Ye

***Abstract*—The need for workflow management has become readily apparent in recent years, and in order to manage various business workflow features, a conceptual modeling or specification of workflow is paramount. Statechart is a visual language for software requirement specification that has been widely used due to its compactness, expressiveness, compositionality, and modularity. This paper presents a tutorial of the Statechart visual language and how it can be used for workflow specification.**

***Index Terms*—Statechart, workflow.**

I. Introduction

Today's business enterprises must deal with global competition, reduce the cost of doing business, and rapidly develop new services and products. To address these requirements enterprises must constantly reconsider and optimize the way they do business and change their information systems and applications to support evolving business processes. Osterweil and Sutton [1] showed that software technology has direct relevance and applicability to workflow.

The concept of workflow originated from the notion of process in manufacturing and office environments. Typical examples of workflow include such things as the processing of a credit request in a bank [2], the medical treatment of patients in a hospital, insurance claims, customer requests for telephone service, etc.. A workflow consists of a set of processing steps (tasks) together with some specification of the control and data-flow between these activities. Processes can be mapped into workflow descriptions that can be executed automatically by a workflow management system. Workflow technology facilitates this by providing methodologies and software [3].

Two distinct constructs, the transaction and the tasks, are used to model workflow. The transaction deals with the specification of the communication aspects of workflow, while the task deals with required adaptability of workflow systems. A workflow specification captures a process abstraction. There exist a multitude of languages for this purpose: process-programming language [4], rule-based [5], data-flow based, control-flow based [6], visual Petri Net based, etc. Regardless of language, the specification of workflow should be unambiguous and have a modular structure. It should be simple and clear, and it must contain only the information needed by the developers and analysts. The language for software requirement specification should be easy to use and result in more readable and revisable specifications. However, in current practice, the rule-based languages are most frequently used, yet they lack modularity, and the maintenance of rule-based programs composed of many thousands of rules is extremely difficult. A possible solution to this paradox is through the use of a visual language [7].

Statechart by Harel et al [8]-[11] is a visual language for software requirements specification that has been widely used [12]-[14]. In essence, it extends the conventional language of state-transition diagrams with three elements that accommodate the notions of hierarchy, concurrency, and communication. Additionally, it allows multilevel concurrency and the use of high- and low-level events. It is compact, expressive, compositional, and modular. Statechart is perceived by practitioners as intuitive and easy to learn, yet it has rigorous semantics [15].

The syntactic and semantic elements of Statechart use low level functional formalism, and the semantics appears to be novel in its treatment of shared variables, chain-reactions, and simultaneous multiple transitions. More recently, Statecharts are being used as UML (Universal Markup Language) state machine diagrams [16] and state machine notations for control abstraction [17].

This paper is a tutorial of Statechart used for workflow specification. Section 2 introduces the basic idea of Statechart and the reasons for using Statechart. Basic features of Statechart are discussed in Section 3. In Section 4, Statechart is applied to a workflow specification example. Finally, the merits, shortcomings, and future research in workflow specification will be discussed.

II. Statechart

A finite state machine (FSM) is a model of a system with discrete inputs and outputs. The system can be in any one of a finite number of internal states or configurations. The state of the system summarizes the information concerning past input that is needed to determine the behavior of the system on subsequent input. One state denoted by q_0 is the initial state. The system consists of a finite set of states and transitions from state to state that occur on input symbols. For each input symbol there can be exactly one transition out of each state or there can be more than one transition out of a state. A directed graph, called a *transition diagram* is associated with the FSM. The vertices of the graph correspond to the states. If there is a transition from state q to state p on input a, then there is an arrow labeled a from state q to state p. Fig. 1(a) shows a state transition diagram with four states and eight transitions. A Mealy machine is also a

Manuscript received July 23, 2012; revised September 25, 2012.

W. W. Zhang and T. Beaubouef are with the Computer Science and Industrial Technology Department, Southeastern Louisiana University, Hammond, USA (e-mail: wzhang@selu.edu, tbeaubouef@selu.edu).

H. Ye is with the School of Electrical Engineering and Computer Science, The University of Newcastle, Callaghan, NSW 2308, Australia (e-mail: Huilin.Ye@newcastle.edu.au).

type of finite state machine, except it gives an output in response to input. Fig. 1(b) shows a Mealy machine, which takes input *0* or *1* and gives output *n*.

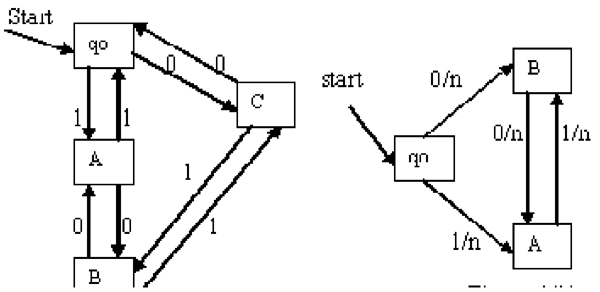

Fig. 1. (a) Transition diagram. (b) Mealy machine.

Harel [8], [9] constitutes an attempt to revive the classical formalism of finite state machines and state transition diagrams and make them fitting for use in large and complex applications. Statechart, a visual language for specification, is proposed to overcome the drawbacks of state diagrams while preserving and even enhancing the visual appeal of conventional state diagrams.

Statechart transforms a state transition diagram into a highly structured and economical description language. When coupled with the capabilities of computerized graphics, Statechart enables people to view the description at different levels of detail and makes very large process-control requirement specification manageable and comprehensible. The syntax and semantics of Statechart use low level functional formalism, and the semantics appears to be novel in its treatment of shared variables, chain-reactions and simultaneous multiple transitions. In particular, Statechart specifications are amenable to model checking [18], [19], so that critical workflow properties that are expressible in temporal logic can be formally verified.

In order to let application experts who know very little or nothing about computers or software to be able to understand and use Statechart, notations used in this paper are graphics, symbols, and plain English.

III. Basic Features of Statechart

Statechart is a finite state machine augmented with schemes for expressing hierarchy, parallelism, and communication. Rectangles are used to denote *states* at any level. A simple finite state machine (FSM) is composed of *states* connected by *transitions*. An arrow labeled with an event, and optionally with a parenthesized condition, denotes the *transition*. A small arrow marks *default* or *start* states. In the FSM of Fig. 2(a), for example, there are three states: A, B, and C. Event *c* occurring in state A transfers the system from state A to state C if and only if (iff) condition *p* holds at the instant of occurrence. State A here is the *default state*, which means that the system enters state A when the state machine is entered unless otherwise specified.

A. Composition of a Superstate

In a Statechart, states may be grouped into a *superstate*. The concept of a *superstate* has its origin in *higraph* [3], which combines the notions of Euler circles, Venn diagrams, and hypergraphs. A Statechart may contain states at any level, and encapsulation is used to express the hierarchical relation.

In Fig. 2(a), since event *b* takes the system to state B from either state A or state C, states A and C can be clustered into a new *superstate* D, and the two *b* arrows can be replaced by one as shown in Fig. 2(b). The semantics of D is the exclusive-or (XOR) of states A and C, i.e., being in state D is equivalent to being in either state A or state C, but not both. Superstate D is an abstraction of states A and C. Such groupings reduce the number of transitions need to be drawn on a Statechart. The *superstate* D and outgoing arrow *b* capture a common property of states A and C, viz., a transition from either of its substates A or C via arrow *b* to state B.

A *superstate* can be entered in two ways. First, the transition to the *superstate* may end at the border of the *superstate* as exemplified by arrow *a* in Fig. 2(b). In that case, the default state A is entered, i.e., it is equivalent to having arrow *a* drawn from state B to state A. Second, the transition may be made to a particular state inside a *superstate*, such as arrow *d* in Fig. 2(b) that leads from state B to state C.

A *superstate* may be exited in two ways. Analogous to transitions into the *superstate*, transitions out of the *superstate* may originate from the border of the *superstate* or from an inner state of the *superstate*. In Fig. 2(b), *b* arrow indicates a transition from the border of the *superstate*. In this case, the system leaves superstate D and all substates A and C. The *superstate* can be exited from a substate such as *f* arrow in Fig. 2(b) that leads from state A to state B.

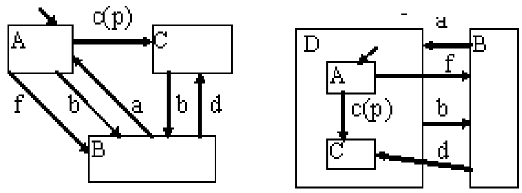

Fig. 2. (a) Finite state machine. (b) Superstate.

Grouping states into a *superstate* indeed reduces the number of transitions and makes the specification more readable.

B. Composition of a Parallel State

One of the most important innovations in Statechart is the *parallel state*, which is also referenced as orthogonal or product state. A *parallel state* contains two or more *parallel components (AND components)* separated by dashed lines.

In Fig. 3(a), *parallel state* H consists of two *parallel components*, state A and state D. The semantics of H is the product (AND) of states A and D, i.e., being in state H entails being in both state A and state D. When the parallel state H is entered, each of the parallel components, state A and state D within H, is entered too. In Fig. 3(a), when parallel state H is entered from the outside via arrow *a*, the substate B of A and substate F of D are entered by the default arrows. When any transition is taken out of the *parallel state* H, all states H, A, and D are exited.

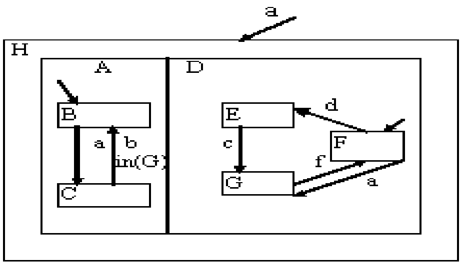

Fig. 3. (a) Parallel state consists of two parallel components.

The *parallel components* state A and D can be *superstate* or be *parallel states* themselves. In Fig. 3(a), *parallel components* state A and state D are *superstates* themselves. A *parallel state* in Statechart illustrates a certain kind of synchronization. In Fig. 3(a), if event *a* then occurs, the transition from state B to state C and from state F to G will take place simultaneously.

The use of a *parallel state* greatly reduces the size of the specification. Fig. 3(b) is the conventional AND-free equivalent "flat" version of Fig. 3(a). The usual product of a conventional state transition diagram is a disjoint product. Fig. 3(b) contains six states, the product of the two substates in A and three substates in D. Clearly, two components with one thousand states each would result in one million states in the product. This is the root of blow-up in number of states. *Parallel state* in Statechart introduces some dependence between components, i.e., in Fig. 3(a), the special condition "in(G)" attached to arrow *f* causes state A to depend on state D and indeed to "know" something about substates of D. If the parallel construct is used often, and on many levels, the state explosion problem can be overcome in a reasonable way.

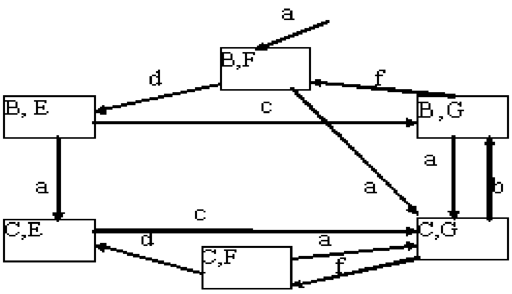

Fig. 3. (b) Disjoint product of parallel state.

A *parallel state* can be entered in four ways. First, the transition to the *parallel state* may end at the border of the parallel state as exemplified by arrow *a* in Fig. 3(c). (Fig. 3(c) adds a possible interface description of the *parallel state* H of Fig. 3(a). Internal transitions of Fig. 3(a) have been omitted for simplicity.) In this case, as mentioned before, state B and state F are entered by default. Second, the transition may be made to particular states inside the *parallel components*, such as split arrow *b* in Fig. 3(c) that leads transition from state J to state B and state E. Third, the transition may be made to one particular state, such as arrow *c* in Fig. 3(c) that leads the transition from state K to state C by arrow *c* and state F by default. Fourth, the transition may be made to one particular state and an H-entry such as split arrow *d* in Fig. 3(c) that leads the transition from state L to state C and the most recently visited state in state D.

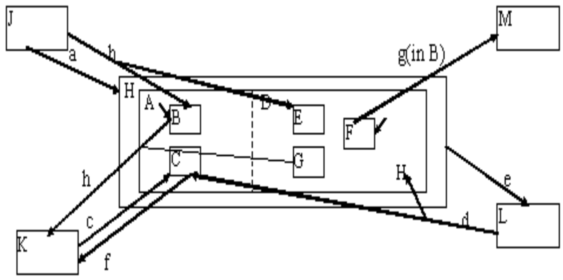

Fig. 3. (c) Multiple exits of parallel state.

A *parallel state* may be exited in three ways. First, the transition exits a *parallel state* from the border of the *parallel state* as exemplified by arrow *e* in Fig. 3(c). In this case, the *parallel state* H and all *parallel components,* state A and state D are exited unconditionally. Second, an "exiting independently" transition exits a *parallel state* from an inner state such as arrow *f* in Fig. 3(c) that leaves state H, state A, and state D, and enters state K. Third, an "exiting dependently" transition exits a *parallel state* from a certain combination of states as exemplified by arrow *h* in Fig. 3(c). In that case, the event *h* occurred in state B, and state G causes transferring from *parallel state* H to state K. An alternative to the third case is to replace one of the outgoing branches of the merging arrows by a condition as shown in arrow *g* from state F in Fig. 3(c). In this case, transition exits *parallel state* H to state M only from state F and state B.

Using *parallel states* reduces the state explosion problem in conventional state machines. The *parallel state* components can be carried out on any level of states and is therefore more convenient than allowing only single level sets of communicating in a FSM. The use of a *parallel state* enables Statechart to describe independent and concurrent state components and eliminates the need for multiple control activities within a single activity.

C. Actions

In Section 3.2, the reaction part is expressed only by the system changing its internal state configuration to incoming or sensed events and conditions. None of the transitions contain any outputs. *Parallel components* can synchronize only through common events and can affect each other only through *in(s)* special conditions. The real subtlety of the way Statechart models concurrence is in their output events. Statechart can be viewed as an extension of a *Mealy machine,* in that it has the ability to generate events and change the values of conditions. These output events denoted by /s are called *actions* to be attached optionally to the label of a transition. The enriched transition labeling is the form *e[p]/s* where *e* is the event triggering the transition, *p*, the condition that guides the transition, and *s*, the action to be carried out upon the transition.

In contrast to conventional *Mealy machines*, however, an *action* appearing along a transition in a Statechart is not merely sent to the "outside world" as an output. The action typically will affect the behavior of the Statechart itself in its *parallel components*. This is achieved by a simple *broadcast* mechanism in the same way as the occurrence of an external event that causes transitions in all *parallel components*.

D. Timeout

The ability to limit the system's delay in a state and putting a time constraint on a state is an important property of real-time system requirements specification. Statechart uses implicit timers to respond to time restrictions. Formally, this is done using the event expression *timeout*(*event, number*). This expression represents that *timeout* event occurs precisely when the specified number of time units have elapsed from the occurrence of the specified *event*.

In Fig. 4, the system will exit from state A to state B when 120ms have elapsed from the occurrence of event *f*.

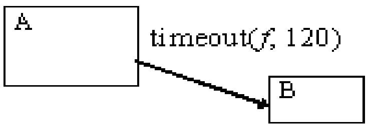

Fig. 4. Timeout.

IV. Workflow Specification

In this section, Statechart is applied to a workflow example, credit request processing in a bank. A more detailed application study in the area of credit processing is currently conducted within the Mentor (Middleware for Enterprise Wide Workflow Management) project [19]. The purpose of this example is to demonstrate the principal suitability of Statechart for the workflow specification.

When a company makes a credit request, the bank will check the company's current credit balance with the corresponding checking of the company's credit rating and risk evaluation. In addition, the bank will determine the ownership and control relations, which the company has with other national and international companies. The decision is eventually made on the credit request either to approve it or to turn it down.

In this simplified workflow specification, there are six activities that reflect the functional decomposition of a system and correspond directly to the activities of the workflow.

INIT: Initializes the system;

ENCR: Enters the credit request into a credit database;

CCW: Checks credit balance and credit with other financially related companies;

RISK: Evaluates the potential risk that is associated with the requested credit;

DEC: Records the decision about credit request;

ERROR: Handles errors.

There are several conditions, events, and actions defined in this specification:

1) Event *en(s)* occurs upon entering state *s*
2) *ENCR_OK* and *ENCR_NOK* are the corresponding conditions for ENCR;
3) *CCW_OK* and *CCW_NOK* are the corresponding conditions for CCW;
4) *RISK_OK* and *RISK_NOK* are the corresponding conditions for RISK;
5) *REQUEST_REJECTED* is the corresponding condition for state CR_S;
6) *DEC_OK* and *DEC_NOK* are the corresponding conditions for DEC;
7) Action *st!(activity)* starts the *activity;*
8) Action *sp!(activity)* stops the *activity;*
9) Event *PANIC* occurs upon system failure.

Statechart reflects the behavior of a system. The Statechart depicted in Figure 5 shows the control flow between the six activities. One state is entered exactly when the corresponding activity is started. The system enters into the INIT and ERR_INIT states simultaneously. A TIMEOUT is triggered whenever a state is not left within specified DELAY time after it has been entered; the ERROR activity is started and ERROR_S state is entered. A NOK_activity condition is generated if anything is wrong in that activity. This NOK_activity condition causes the ERROR activity to start.CCW_S and RISK_S are parallel components of CR_S state, meaning that the corresponding activities will be executed concurrently.

V. Discussion and Future Studies

Cichochi, *et al*.[7] view the role of Statechart in practical business environments as that of a canonical representation for an underlying executing engine with rigorously defined semantics. Other specifications can be converted into Statechart, and Statechart may serve as an exchange format across different workflow engines.

Verification techniques can be used to check the correctness of Statechart based workflow specifications. In particular, Wodtke and Weikum [14] validate Statechart properties by means of reach-ability tests [10] and symbolic model checking [18]. The property that every credit request in our example will eventually be granted or rejected, and these two results exclude each other, can be easily expressed in the temporal logic CTL[20] and efficiently verified by model checking.

However there are several drawbacks of the original Statechart. The order of the transitions taking place is important. The Statechart shows structure non-determinism caused by the freedom of selecting subsets in micro-steps and the uncertainty of selecting concurrent events. One of the most important properties of any real-time system is the time constraint that should be clearly indicated in the requirement specification. Although Statechart provides *timeout* feature and *time bound*, these features are not well defined and are not sufficient to represent the critical time requirements.

From the perspectives of practitioners, Statechart is still too formal and not appropriate for wide use in business environments. The capabilities for invoking external software and some form of callback facility are important features. For example, the need for flexible decision making while minimizing the risks may require dynamic modifications to the specified control flow by introducing additional "ad-hoc" activities while the workflow is being executed. Another challenge is the problem of ensuring the consistency of the underlying information when several workflows of this type are to be executed concurrently.

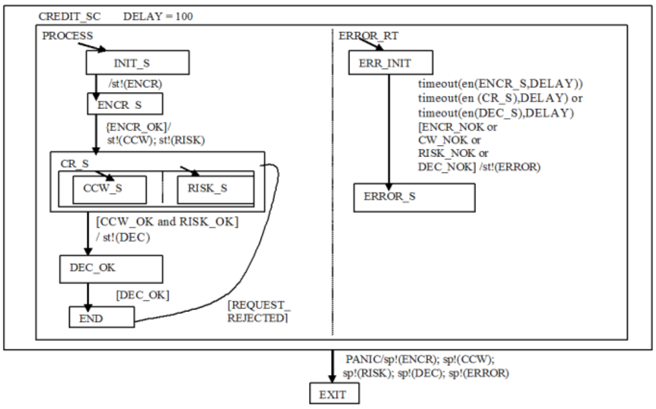

Fig. 5. Workflow of credit request processing in a bank.

REFERENCES

[1] L. J. Osterweil and S. M. Sutton, "Using software technology to define workflow processes," *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, Athens, GA, May 8-10, 1996.

[2] D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz Dittrich, "The mentor project: steps toward enterprise-wide workflow management," *Proceedings of 11th International Conference on Data Engineering*, New Orleans, 1996, pp. 556-565.

[3] D. Georgakopoulos, M. Hornick, and A. Sheth, "An overview of workflow management: from process modeling to workflow automation infrastructure," *Distributed and Parallel Database*, vol. 3, pp. 119-153, 1995.

[4] S. M. Sutton, H. Ziv, D. Heimbigner, H. E. Yassayam, M. Maybee, L. J. Osterweil, and X. Song, "Programming a software requirements-specification process," *Proceedings of First International Conference on the Software Process: Manufacturing Complex Systems*, Redondo Beach, USA, October, 1991, pp. 68-89.

[5] M. Scott, "A rule-based approach for thdesign and implementation of information systems," *Advanced in Database Technology, 4th International Conference on Extending Database Technology*, March 1994

[6] D. G. Kopoulos and M. F. Hornick, "A framework for enforceable specification of extended transaction models and transactional workflows," *International Journal of Intelligent and Cooperative Information Systems*, vol. 3, no. 3, 1994.

[7] A. Cichochi, A. Helal, M. Rusinkiewsicz, and D. L. Woelk, *Work Flow and Process Automation Concepts and Technology*, Kluwer Academic Publishers, 1998.

[8] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, 1987.

[9] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman, "On the formal semantics of statecharts (extended abstract)," *Proceedings of the Second Symposium, Logic in Computer science*, Ithaca, N.Y. 1987.

[10] D. Harel, H. Lachover, A. Naamad, A Pnueli, M. Politi, R. Sherman, A. S. Trauring, and M. Trakhtenbrot, "Statemate: A working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, April 1990.

[11] D. Harel and A. Naamad, "The statemate semantics of statecharts," *Technical Report*, I-Logix Inc. October 95.

[12] M. P. E. Heimdahl and N. Leveson, "Completeness and consistency in hierarchical state-based requirements," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, June 1996.

[13] F. Jahanian and A. Mok, "Safety analysis of timing properties in real time systems," *IEEE Transactions on Software Engineering*, vol. SE-12, September 1986.

[14] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 9, September 1994.

[15] D. Wodtke and G. Weikum, "A formal foundation for distributed workflow execution based on state charts," *Data Theory - proceedings of 6th International Conference*, Delphi, Greece, January 1997, pp. 230-246.

[16] S. Ambler. (2010). UML 2 State Machine Diagram. [Online]. Available: http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm.

[17] Working draft. (May 13, 2010). State chart XML (SCXML): State Machine Notation for Control Abstraction. [Online]. Available: http://www.w3.org/TR/2010/WD-scxml-20100513/.

[18] J. Helbig and P. Kelb, "An OBDD-representation of statecharts," *Proceedings of European Design and Testing Conference*, 1994.

[19] K. L. McMillian, *Symbolic model checking*, Kluwer Academic Publishers, 1993.

[20] E. A. Emerson, "Temporal and modal logic," *Handbook of Theoretical Computer Science*, Elsevier, 1990.

Wendy Wenhui Zhang was born in Shanghai, China. She got her M.S. and Ph.D. in computer science from University of Houston, Texas, United States. She is full professor of Computer Science & IT Department in Southeastern Louisiana University, Louisiana, United State. Her research interests include spatial database, high performance computing, and hyperspectral remote sensing. Dr. Wendy Zhang is a member of ACM and IEEE. She was awarded summer research fellowships from NASA and Naval Research Lab during 2008-2009 and research grant from National Science Foundation (NSF) from 2009-2012.

Theresa Beaubouef earned the B.S. in Computer Science from Louisiana State University and the M. S. and Ph.D. degrees from Tulane University in New Orleans, Louisiana in 1992. Dr. Beaubouef has worked as a computer scientist for the U.S. Navy and its contractors, and as Assistant Professor at Xavier University in New Orleans. She is currently a professor at Southeastern Louisiana University.

Her research interests include uncertainty in databases, data mining, spatial databases, artificial intelligence, and computer science education. Dr. Beaubouef is also interested in scientific computing and mathematical applications and formal modeling of processes.

Dr. Huilin Ye is an Associate Professor at School of Electrical Engineering and Computer Science, University of Newcastle, Australia. Her main research interest is in the area of software engineering, including software product line engineering, object-oriented software development, software reusability, and software library systems etc. She is the leader of Software Engineering Research Group at University of Newcastle and currently leading an Australian Research Council funded project in feature model based software product line engineering. Prior to her academic career she had been a senior software engineer and system analyst in software engineering industry for more than 10 years.