ON FORMAL TOOLS IN THE SOFTWARE ENGINEERING

Arslan Enikeev, PhD
Mahfoodh Bilal Ahmed Mohammed, MSc
Elina Stepanova, MSc
Kazan Federal University, Russian Federation

Abstract

This paper presents an overview of different approaches to a creation of the technique of software application development based on the integrated development environment which contains a model and tools for its implementation. Our results in this field are also presented. We study a formal model specification and analysis tools which may have potential for the software application development.

Keywords: Formal specifications, model, CSP, UML

Introduction

Software systems have significantly increased in complexity and diversity in recent years, requiring the use of new, more efficient technological tools for their development. Traditional tools based only on programmer‘s intuition cannot guaranty high reliability in software products and do not allow their complete analysis. These problems can be solved using formal mathematical models which provide a rigorous approach to the software development. However, the experience of software development shows that the use of formal methods in the design of software systems often leads to cumbersome constructions which cause a serious obstacle to the development process. It follows that we need to create an appropriate conceptual apparatus which will make these formal methods more applicable to software development in practice. In this paper we study different formal tools and methods which could be useful to apply to software application development.

The formal tools for the software models

The strongly held common philosophy of the current state of software engineering in 1970s is admirably expressed in the following quotation from professor Christopher Strachey, the founder of the Programming Research

Group which represents a part of the Computing Laboratory at Oxford University ("History and Structure", 2007, "Undergraduate handbook", 2007, Hoare, 1982): "It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. One of the central aims of the Programming Research Group as a teaching and research group has been to set up an atmosphere in which this separation cannot happen." If we consider this statement from today's perspective we can see that the gap between theory and practice in the field of software engineering has not only become larger but has in fact increased significantly. However what reasons would have caused this? There are many reasons; among them the following seem to be significant:

- most software developers rely on their intuition and experience, ignoring formal methods and tools in the software development process;
- despite many theoretical works having appeared recently in the software engineering, the appropriate formal tools continue to be inadequate for their application to the practice of software engineering.

Next, we will consider some of the formal tools and methods which could be useful in software engineering.

Hoare logic is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs (Hoare,1982, Hoare,1969). It was proposed in 1969 by the Oxford professor C.A.R. Hoare. The central feature of Hoare logic is the Hoare triple. A triple describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form {S} P {S'} where S and S' are assertions and P is a command. S is named the precondition and S' the postcondition: when the precondition is met, executing the command establishes the postcondition. Assertions are formulae in predicate logic. Based on this logic Hoare formally defined basic programming constructions. For example: if b then P else Q = $\neg(D_b) \vee (b\&P) \vee (\neg b\&Q)$, where D_b is a definition domain of logical expression b, P and Q are the predicates defining specifications of the corresponding programming constructions. This formal system can be useful in the transformational programming technique (Georgiev , Enikeev, 1992). Program transformation is the process of converting one program to another using the appropriate transformational rules which can be deduced and proved on the base of Hoare logic. Transformational programming technique used to be applied to code optimization and program

generators. Code optimization is a transformation of source code into a simpler and more efficient code using the appropriate transformational rules. The greatest effect is achieved by automating the process of transformation. Here is an example of the transformational rule: while i<n do i:=i+1 = if i<n then i:=n. Program generators provide automated source code creation from generic frames, classes, prototypes and templates to improve the productivity of software development. This technique is often related to code-reuse topics such as component-based software engineering and product family engineering. In this case the transformational rules are represented by substitution rules. One of the most interesting fields of the transformational programming technique is partial evaluations (mixed computations) concerning automatic compiler generation from an interpretive definition of a programming language (Ershov, 1982). The technique also has important applications in scientific computing, logic programming, metaprogramming, and expert systems.

The theory of communicating sequential processes (CSP) (Hoare, 1985) is a formal system, which, by using the conceptualization of sequential processes, enables the specification and analysis of various patterns of communication between processes (including parallelism). In the next part of the paper we present a study of CSP tools based on the example of a menu - select interactive system model specification (Enikeev, Hoare, Teruel, 1987). The behavior of a menu –select interactive system can be modeled as a set of sequences of possible responses. This allows the description of the menu-select interactive system model using CSP theory. In CSP notation these sequences are called traces. A trace is a finite sequence of symbols recording the actual or potential behavior of a process from its beginning up to some moment of time. Each symbol denotes a class of events in which a process can participate. The set of symbols denoting events in which a process can participate defines the alphabet of a process. A process is defined by the set of all traces of its possible behavior. From the definition of a trace, it follows that; process P with alphabet A:

P0. $P \subseteq A^*$, where A*denotes the set of all traces with symbols from a pre-defined alphabet A;

P1. $<> \in P$, where <> denotes an empty trace;

P2. $st \in P \Rightarrow s \in P$, for all $st \in A^*$,where st is the concatenation of s with t.

Below we present some important definitions from CSP that will be used subsequently. If s is a nonempty trace, we define s_0 as its first symbol, and s' as the result of removing the first symbol from s. Let √ be a symbol denoting successful termination of the process. As a result, this symbol can appear only at the end of a trace. Let t be a trace recording a sequence of events

which start when s has been successfully terminated. The composition of s and t is denoted (s; t). If √ does not occur in s, then t cannot start. If s is a copy of an initial subsequence of t, it is possible to find some extension u of s such that su = t . We therefore define an ordering relation $s \leq t =_{df} \exists u\ (su=t)$ and say that s is a prefix of t . For example, <x,y> ≤ <x, y, z> , <> < <x, y> . The ≤ relation is a partial ordering, and its smallest element is <>. The expression (t ⌈A) denotes the trace t when restricted to symbols in the set A; it is formed from t simply by omitting all symbols outside A. For example, <a, d , c, d> ⌈ {a,c} =<a, c> . The expression P^0 denotes the set of first symbols of all traces in process P (initial state of process P). To put it formally: $P^0 =_{df} \{ c \mid < c > \in P \}$. Process FAIL $=_{df} \{<>\}$, which does nothing, process SKIP $=_{df} \{<>, <\surd>\}$, which also does nothing, but unlike FAIL it always terminates successfully. Let x be an event and let P be a process. Then (c → P) (called 'c then P') describes an object which first engages in the event c and then behaves exactly as described by P. The process (c → P) is defined to have the same alphabet as P; more formally, $(c \rightarrow P) =_{df} \{c \rightarrow s \mid c \in \alpha P \ \&\ s \in P\}$, where $c \rightarrow s =_{df} < c > s$, αP denotes an alphabet of process P. Let P be a process and s ∈ P then P / s (P after s) is a process which behaves the same as P behaves from the time after P has engaged in all the actions recorded in the trace s. If s ∉ P, P / s is not defined; more formally, $P / s =_{df} \{t \mid st \in P\}$. Let P and Q be processes. The operation P | Q is defined as following: $P \mid Q =_{df} P \cup Q$, where $\alpha (P \mid Q) = \alpha P \cup \alpha Q$ (the choice between P and Q). The choice depends on which event from $(P \mid Q)^0$ occurs. For example, if R= (a → P) | (b →Q), R/<a> =P and R/< b > = Q. Let P and Q be processes. Sequential composition P; Q is defined as a process which first behaves like P; but when P terminates successfully, (P; Q) continues by behaving as Q. If P never terminates successfully, neither does (P;Q). More formally, $P;Q =_{df} \{ s;t \mid s \in P \ \&\ t \in Q \}$. Let P and Q be processes. The operation of parallel composition P || Q is defined as following: $P \parallel Q = \{s \mid s \in (\alpha P \cup \alpha Q)^* \ \&\ s \lceil \alpha P \in P \ \&\ s \lceil \alpha Q \in Q\}$, where $(\alpha P \cup \alpha Q)^*$ is a set of all possible traces from the alphabet $(\alpha P \cup \alpha Q)$.

In CSP a menu select interaction can be specified as communicating process P. The initial menu, with a set of events, is displayed on the screen, represented as P^0. After the user has selected one of these events, say x ($x \in P^0$), the subsequent interaction is defined by P/< x.> (P after x), i.e. $(P/< x.>)^0 \ldots$.

Example1.

P^0={ a, b}, $P/< a > = P_a$, $P/< b > = P_b$, P= a-> P_a | b-> P_b

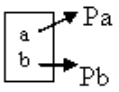

Figure 1.

A set of menus and interactive prompt representations are provided for a set of functions logically used together. Each symbol in the menu denotes a function, invoked after the user's selection. We will make a distinction between the commonly used functions, controlling the interaction process, and the problem dependent functions, the choice of which can be defined depending on the particular sort of problems. The most typical commonly used functions of the menu-select interaction are the following:

(1) functions for terminating or quitting a process;

(2) functions allowing the return to any of the previous steps.

The main objective of this paper is the specification of an abstract menu-select interaction on the basis of CSP, concentrating our attention on these three commonly used function types. In greater detail, these functions are;

(1) 'stop' – to terminate or quit a process;

(2) 2.1. 'reset' – to start again from the beginning of a process;

2.2. 'back' - to undo the most recent action in a process;

The model of a menu-select interactive system is based on the specification of these functions, which can be described in CSP. But CSP facilities are not enough to describe a menu-select interaction model completely. Therefore we need to extend CSP facilities with new processes which define the above mentioned functions. For a definition of the appropriate processes we will use a derivative definition, defining a process P by two objects: a set I, defining the initial state of process P , i.e. P^0 and a function mapping each member 'c' of I into a process, defining the subsequent behavior of P, i.e. P/<c>. If P is a process, let's define the following processes:

Stoppable(P).

Let 'stop' be a symbol not in the alphabet αP. Then the process stoppable (P) can be defined as a process which behaves like P, except that
(1) 'stop' is in its alphabet;
(2) 'stop' is in every menu of stoppable (P);
(3) when 'stop' occurs, stoppable (P) terminates successfully.
For example: <a, b, c, stop, $\surd$ > $\in$ stoppable (P) $\Leftrightarrow$ <a, b, c> $\in$ P, where symbol $\surd$denotes the event of a successful termination of the process.
Definition 1.
stop$\notin \alpha P$& $(\text{stoppable}(P))^0 = P^0 \cup \{\text{stop}\}$ &

SKIP, if x=stop

(stoppable(P))/<x>= {

stoppable(P/x), if x ≠ stop

Example 2.

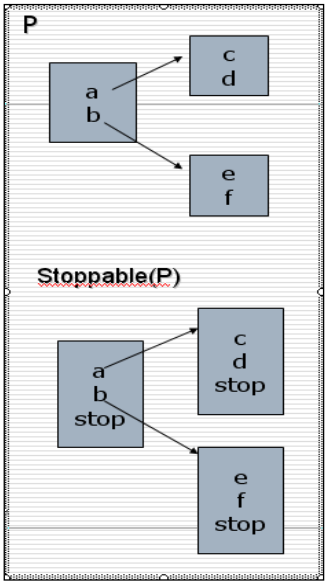

Figure 2.

Resettable (P).

Let 'reset' be a symbol not in the alphabet αP. Define 'resettable (P)' as a process that behaves like P, except that (1) 'reset' in its alphabet; (2) 'reset ' is in every menu of resettable (P); (3) when 'reset' occurs, resettable (P) starts again from the beginning. For example: <a, b, reset, c, d>∈ resettable(P) ⇔ <a, b>∈ P & <c, d> ∈ P.

Definition 2. reset∉αP& resettable(P) =start(P,P), where

$(start(P,Q))^0 = P^0 \cup \{reset\}$ &

start(Q, Q), if x=reset

(start(P,Q))/<x>= {

start(P/x, Q), if x ≠ reset

Example 3.

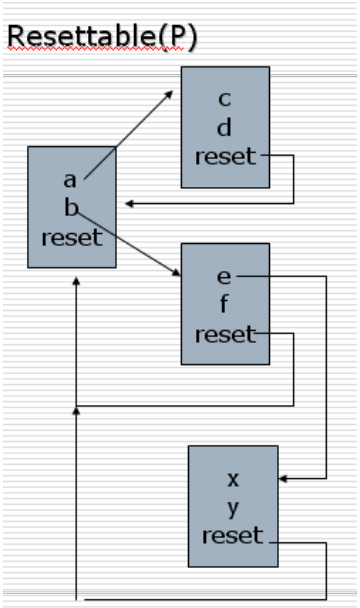

Figure 3.

Backtrackable (P).

Let 'back' be a symbol not in the alphabet αP. Define backtrackable(P) as a process that behaves like P, except that (1) 'back' in its alphabet; (2) 'back'is in every menu of backtrackable (P); (3) backtrackable (P)/s<x,back> = backtrackable (P)/s provided x ≠ 'back'.

The intention is that 'back' will cancel the effect of the most recent action which has not already been cancelled (other than 'back' itself). For example: <a, b, back, d>∈ backtrackable (P) ⇔ < a, d >∈ P, <a, b, c, back, back, d>∈ backtrackable (P) ⇔ < a, d >∈ P

Definition 3.

back∉ αP& backtrackable(P) =recover(P,P), where

$(recover(P,Q))^0 = P^0 \cup \{back\}$ &

$(recover(P,Q))/\langle x\rangle = \begin{cases} Q, & \text{if } x=back \\ recover(P/x, recover(P,Q)), & \text{if } x \neq back \end{cases}$

Example 4

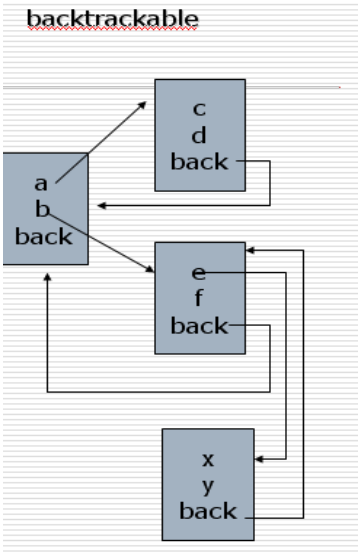

Figure 4.

The above sited definitions can be adequately implemented in the form of recursively defined functions or procedures. CSP theory permits the creation of models for a variety of software applications and is especially appropriate for those that are based on an event-driven programming paradigm. This paradigm is widely used in the majority of object –oriented programming systems. The evolution of the object-oriented programming technique led to the appearance of the CSP-OZ theory, which is based on a combination of CSP and the object-oriented specification language Object-Z (Fischer, 1997, Duke, 1995). This theory provides a specification of the behavior of communicating processes and, in addition to CSP it permits the description of object-oriented models. Materials concerning the application of CSP-OZ theory to the development of information systems have been published in a monograph (Enikeev, Benduma, 2011).

OCCAM is a concurrent programming language that builds on Communicating Sequential Processes (CSP) and shares many of its features. It was developed as the programming language for the transputer microprocessors, although implementations for other platforms are available (Roscoe, 1986, Inmos Limited Prentice-Hall,1984). The example of the OCCAM program is:

An analog volume control of a digital radio

```
DEF max=10, min=2

VAR volume:

SEQ

  volume:=0

  WHILE true

   ALT

    (volume<max ) & ( louder ? ANY )

       SEQ

        volume:=volume +1

        amplifier ! volume

    (volume>min ) & (softer ? ANY)

       SEQ

        volume:=volume -1

        amplifier ! Volume
```

As an example, a digital radio replaces an analog volume control with two buttons, one marked “louder”, the other marked “softer”. These buttons are connected to two channels, “louder” and “softer” respectively, and whenever either button is pressed it causes a message to be sent along the corresponding channel. By pressing the buttons we may increase or decrease the volume, the value of which is transmitted to the amplifier. Here SEQ(P,Q) denotes P;Q, chan?var is an input of a value from the channel chan into the variable var, chan!expr is an output of the value of the expression expr to the channel chan, alter(P,Q) is P | Q (alternative processes).

Model-driven engineering (MDE) is a software development methodology which focuses on creating and exploiting domain models (that is, abstract representations of the knowledge and activities that govern a particular domain of application) [Meyer,1997, Swithinbank, Chessell, Gardner, Griffin, Man, Wylie, Yusuf, 2005, E. Gamma, R. Helm, R. Johnson, and J. Vlissides, 1994). The first project to support MDE were the Computer-Aided Software Engineering (CASE) tools developed in the 1980s (Rational Software Corporation, Rational Rose,2001, Hubert, Johnson, Wilkinson, 2003). Companies like Integrated Development Environments (IDE - StP), Higher Order Software (now Hamilton Technologies, Inc., HTI), Cadre Technologies, Bachman Information Systems, and Logic Works (BP-Win and ER-Win) were pioneers in the

field. Several variations of the modeling definitions (see Booch, Rumbaugh, Jacobson, Gane and Sarson, Harel, Shlaer and Mellor, and others) were eventually combined to create the Unified Modeling Language (UML). Rational Rose, a product for UML implementation, was made by Rational Corporation (Larman, 2004). UML language representing a high level of model-driven engineering approach provides the possibility of software application development using special diagrams (see figure 5 below).

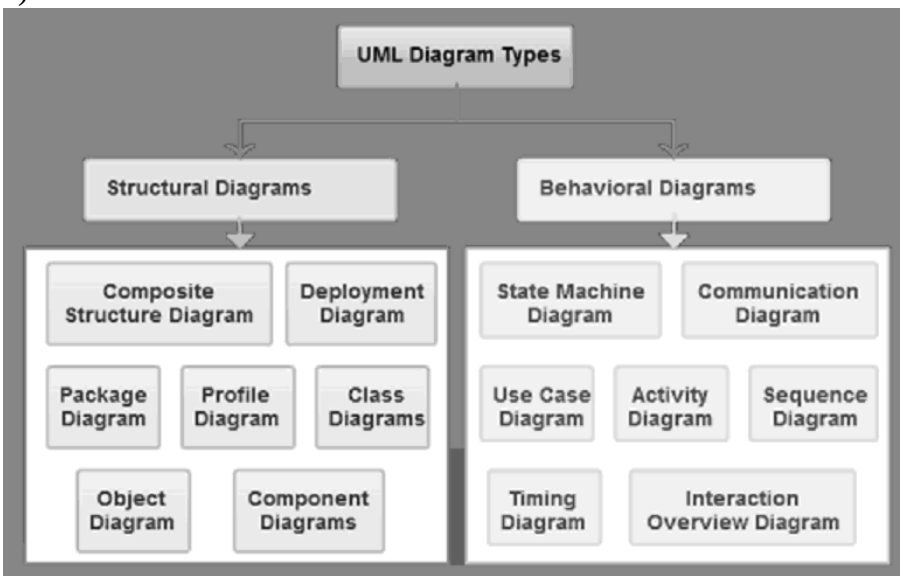

Figure 5.

The advantages of UML

- UML is an object-oriented language, therefore the methods of describing the results of the analysis and design are semantically similar to the methods of programming in modern object-oriented languages;
- UML permits the description of the system from any possible point of view and with different aspects of system behavior;
- UML diagrams are relatively easy to learn;
- UML permits the definition of new text and image stereotypes;
- UML is widely used and is currently being intensively developed.

The disadvantages of UML

- superfluity of language;
- ambiguous semantics;
- Trying to be everything to everyone;

Conclusion

This paper presents an overview of formal tools for a creation of mathematical software models which may be useful in the software

engineering. Despite many theoretical works having recently appeared in software engineering, the problem of their application to software engineering continues to be crucial. One of the most promising ways of solving the problem is to train professionals who combine practical experience in software engineering with an appropriate mathematical background. In addition to this we need to create new technological tools based on an optimal combination of developer intuition and a formal, rigorous approach to the software development process.

References

"History and Structure". Oxford University Computing Laboratory, Internet Archive. 2007. Retrieved 3 May 2013. 2. "Undergraduate handbook 2006–07". Oxford University Computing Laboratory, Internet Archive. 2007. Retrieved 3 May 2013.3.
3. C.A.R. Hoare. Programming is an engineering profession. //Oxford University Computing Laboratory, PRG, 1982.
C.A.R. Hoare. Specifications, Programs and Implementations. //Oxford University Computing Laboratory, PRG, 1982.
5. Hoare, C. A. R. (October 1969). "An axiomatic basis for computer programming". Communications of the ACM 12 (10): 576–580. doi:10.1145/363235.363259,1969.
6. Georgiev V.O., Enikeev A.I. Transformation Approach in Dialogue Systems Engineering. SOFTWARE & SYSTEMS. № 1, 1992, 9 – 17.
7. Ershov, A. P. 1982. Mixed computation: Potential applications and problems for study. Theor. Comput. Sci., 18, 41-67.
8. C. A. R. Hoare, Communicating sequential processes, Prentice-Hall, Inc., Upper Saddle River, NJ, 1985.
9. A.I. Enikeev, C.A.R. Hoare and A. Teruel, A model of the theory of communicating processes for a menu-select interactive system, Mathematica. Vol. 3. 1987, pp.28-36, in Russian).
10.C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J.Derrick, editors, Formal Methods for Open Object-Based Distributed Systems, volume 2, pages 423–438. Chapman & Hall, 1997.
11. R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. Computer Standards and Interfaces, 17:511– 533, 1995.
12.Arslan Enikeev, Tahar Benduma, SPECIALIZED MODELS FOR THE DEVELOPMENT OF INFORMATION SYSTEMS, LAP, LAMBERT Academic Publishing, ISBN: 978-3-8454-4045-3, 2011, 97 p.
13. Roscoe, A.W. Laws of Occam Programming, Oxford University Computing Laboratory, Programming Research Group 1986 ISBN 0 902928 34 1.

14. Inmos Limited Prentice-Hall 1984 ISBN 0 13 629296 8 Japanese edition: ISBN 4-7665-0133-0.
15.Meyer, B.: "Object-Oriented Software Construction, 2nd ed"; Upper Saddle River, NJ, USA: Prentice Hall PTR (1997).
16.P. Swithinbank, M. Chessell, T. Gardner, C. Griffin, J. Man, H. Wylie, and L.Yusuf, "Patterns: Model-Driven Development Using IBM Rational Software Architect", IBM, Redbooks, December 2005, http://www.redbooks.ibm.com/redbooks/pdfs/sg247105.pdf
17.E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.
18.Rational Software Corporation, Rational Rose, http://www.rational.com, 2001.
19. Hubert A. Johnson, Laura Wilkinson, Case tools in object-oriented analysis and design, Journal of Computing Sciences in Colleges , Volume 19 Issue 2, 2003.
20. Craig Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), Prentice Hall; 3 edition, 2004 - 736 p.