Available online at https://www.rsu.ac.th/rjas

Rangsit Journal of Arts and Sciences, January-June 2016 RJAS Vol. 6 No. 1, pp. 89-96
Copyright © 2011, Rangsit University ISSN 2229-063X (Print)/ISSN 2392-554X (Online)

Separation of concerns in designing mobile software

Paniti Netinant[1*] and Tzilla Elrad[2]

[1]College of Information and Communication Technology, Rangsit University, Patumthani 12000, Thailand
E-mail: paniti.n@rsu.ac.th
[2]Computer Science Department, Illinois Institute of Technology, Chicago, IL 60616-3793, USA

*Corresponding author

Submitted 30 November, 2015; accepted in final form 11 April 2016
Available online 21 June 2016

Abstract

Given initial decomposition system properties of a mobile software into processes or components we are faced with inter-concern and intra-concern properties that are tightly bounded in many processes or components. This paper concentrates on the challenges of expressing separation of concerns and imposing inter-concern system properties during an execution of mobile software development. The approach is based on two completely independent areas of research. The Communication Closed Layers (CCL) is a formal system for developing, maintaining, and verifying a distributed program. The Aspect-Oriented Software Development (AOSD) enables separations of concerns.

Keywords: *Aspect-Oriented Software Development (AOSD), separation of concerns, mobile software development*

1. Introduction

The recent expansion of smart devices has abundantly created a unique opportunity for researchers to use all their capabilities to provide new application software. Mobile application software development is rapidly changing the way we have commonly worked and interacted. Mobile software development has to comprehend how separation of concerns can be achieved and how individuals choose to properly develop mobile software to effectively utilize a separation of concerns. At present there are more than hundred thousand of application software available through the various stores, some of which are available for multilingual and multiple types of devices. Most of mobile application software are either native applications or web applications (Wasserman, 2010). Native applications run entirely on the mobile device. Web applications consist of a remote server and a small device-based client executing and interacting user's commands through communication networks. There are several comprehensive mobile application design and development available for the major mobile platforms. IPhone developers use Xcode package across all Apple products (Apple Developer, 2015). Android developers use the Android development tools (Android Developers, 2015) or eclipse programming tools (Eclipse website, 2016). Windows phone developers use Microsoft Studio for mobile development (Windows Phone Developer site, 2016). These dominant development gears and structures greatly simplify the task of design and implementation of a mobile application software. However, they are based on object-oriented design and implementation. The intra-concern system properties are associations and necessities over confined state of processes or components of states. The inter-concern system properties are associations and necessities over dissimilar confined processes or components of states that describe the reliabilities and collaboration among a collection of supportive processes or components. Both processes and components of system properties are critical for a system development and verification. The intra-concern properties are relatively easier to express and carry on through a system development life cycle.

The approach taken here is based on two completely independent areas of research. The first one is the Communication Closed Layers (CCL) which is a formal system for developing, maintaining, and verifying distributed programs (Elrad & Frances, 1982). The second approach is the Aspect-Oriented Software Development (AOSD) (Filman & Friedman, 2000; Elrad, Filman, & Bader, 2001) that enables explanation and mechanization of separation of concerns in the design and implementations. Aspect oriented approach delivers an ordinary framework to discourse inter-concern cooperation. The heart of aspect-oriented software development is the localization of crosscutting concerns. Under CCL limits cooperative global properties can be deliberated as a special case of crosscutting concerns.

Both CCL and AOSD are software development methodologies that can express a mobile software development. Their synergy for a mobile software development and runtime verification of virtual global system assertions is a promising match to handle the complexity of global system cooperation, consistencies, and correctness. CCL and AOSD are balanced on the notion of crosscutting concerns. CCL detentions the semantics whereas AOSD provides the sensitive tools. Together, it makes a design-by-contract discipline (Meyer, 1993) applicable to a wider range of mobile software.

2. The tyranny of process composition

Decomposition of a mobile software into processes and components provides only a syntactic representation of one dimension: the vertical dimension. No syntactic representation is available for the decomposition of the systems into its logical phases in terms of system's goals. For example, it is not syntactically observable when the system as a whole has accomplished its first sub-goal and is complete to launch into the second sub-goal. At any assumed time, different processes or components might be performing at different sub-goals. A designing software architectures of mobile applications using an aspect-oriented approach is an explicit and abstract way (Ali & Ramos, 2012). The tyranny decomposition into processes or components supervises the logical structure of the system as a whole.

The first attempt to break this tyranny of process decomposition is to enforce a complementary horizontal decomposition. Artificial barrier may be introduced to gather and hold processes or components together at a certain point before allowing them to continue accomplishment. Each process or component may have a set of halting points at which it suspends its execution. When all processes or components have reached a local halting point and the whole system halts, a global association among the different states of processes or components is verified for reliability. Only if the system is on a "the correct" milestone step, it allows to carry on. Also, this global checking point can be used for intelligent judgements concerning system future sub-goals. Now both decompositions are syntactically visible, global invariants could be implanted and the complementary structure of the system as a sequence of sub-goals is visible. The huge benefit from such capabilities can be primary by their use in sequential and non-distributed systems. Preconditions and postconditions use assertions and invariants during a sequential life cycle process.

Obviously, this approach is far from real-world. Over synchronization slows the system performance more than necessary. In many cases, just the process or component of detecting that separated processes have all reached a halting point is, in the best case, a hazard and in many cases just difficult.

Breaking the tyranny of composition of processes or components mandatory a more refined method. The knowledge behind the CCL is to novelty a reasonable set of boundaries under which processes or components do not have to "actually" halt at their limited halting points and yet the semantics of the program as a whole is as if they do. The horizontal decomposition of the concerns into its logical stages is visible and not yet impressive in terms of execution. The proof of such semantic correspondence is given in (Elrad & Frances, 1982).

In 1996, Elrad, Baoling, and Nastasic presented a synergy of object-oriented distributed programming and CCL "design by con-tract" for distributed applications where processes are tightly bounded to accomplish a unique common goal. The idea is that object orientation provides encapsulation of the communication among processes or components and hence enables a syntactic identification of a limited process or component that cooperates with parallel units in other processes or components. CCL enables the syntactic identification of such collaboration. The actual application of layer restrictions is inserted into each process or component code. This result with the well-known tangling code occurrence of crosscutting concerns and here is where aspect orientation can provide an expected solution.

The integration between CCL and aspect orientation for a mobile software development is simple; limited halting points for each process or component would be comprehended by aspect oriented approach called *joint points*. *A pointcut* designators will filter out a subset of all the joint points in different processes or components for the identification of a simulated global state that represent the reasoning behind the horizontal structure of the program. *Before advice*, *after advice*, *around advice*, and *aspects* would have their ordinary role. The complication results from a massy code tangling in the *chessboard* implementation are detached. The system obtains a higher degree of flexibility and adaptability associated with aspect-oriented design.

The Communication Closed Layer (CCL) is a language independent methodology that is characterized only in terms of semantic constraints among its components. Equally, this paper will present the CCL realization independent of any particular aspect oriented knowledge.

3. Communication closed layer

The notion of CCL is an independent language. A comprehensive formal definition and proof system can be found in (Elrad & Frances, 1982; Gerth & Shrira, 1986). More work on CCL can be found in (Elrad, Baoling, & Nastasic, 1996; Elrad, 1984; Elrad & Kumar, 1990; Elrad & Kumar, 1991; Elrad & Kumar. 1993; Fokkinga, Poel, & Zwiers, 1993; Gerth & Shrira, 1986; Janssen & Zwiers, 1992a; Janssen, Poel, Xu, & Zwiers, 1994). This paper will present only an overall, more instinctive description using CSP philological notation.Let [$P_1 \| P_2 \| \dots \| P_n$] be a program **P** composed of n processes or components P_1 to P_n.

Now assume each of these processes or components is decomposed into its rational segments. Each P_i is refined into:

begin $S_{i1}; S_{i2}; \dots ; S_{ik}$ end.

S_{ij} is the *j* part of processes or components i. The mobile software can be expressed as:

$$\begin{array}{c} \text{begin } S_{11};S_{12}; \dots ;S_{1k} \text{ end} \\ \| \\ \text{begin } S_{21};S_{22}; \dots ;S_{1k} \text{ end} \\ \| \\ \text{begin } S_{31};S_{32}; \dots ;S_{1k} \text{ end} \\ \| \\ \dots \\ \| \\ \text{begin } S_{n1};S_{n2}; \dots ;S_{nk} \text{ end} \end{array}$$

Note that this symbol reflects the tyranny of processes or components decomposition. Now assume that all the *j* segments are collaborating to accomplish the general system's sub-goal. We would like this fact to be syntactically denoted. [$S_{1j} \| S_{2j} \| \dots \| S_{nj}$] is called the *j* layer of the system L_j :: [$S_{1j} \| S_{2j} \| \dots \| S_{nj}$]. We would like to use layers symbol to combine the whole system back. There are two different composition rules: the Sequential Composition Rule (SCR) and the Distributed Composition Rule (DCR).

3.1 The SCR – Sequential composition rule

Let L_1 and L_2 be two mobile program layers. The composition $\{L_1 + L_2\}$ is defined as the distributed mobile program [$S_{11} \| S_{12}$]; [$S_{11} \| S_{12}$]. This is basic, the semantics of our first attempt to make the rational configuration of the whole mobile software program syntactically observable. All processes or components must halt at layer limitations and only when everyone has reached this synchronization point, then the second segment may start.

3.2 The DCR – Distributed composition rule

Let L_1 and L_2 be two mobile program layers. The composition $\{L_1 * L_2\}$ is defined as the distributed mobile program [$S_{11} \| S_{12}$]; [$S_{11} \| S_{12}$]. This is, basically, the semantics result by discounting layer limitations at accomplishment time. DCR and SCR for more than two layers are defined inductively.

With respect to the example above, the Distributed Composition Rule (DCR); $\{L_1 * L_2 * \dots * L_k\}$ yields the distributed mobile software program in Figure 1. Whereas the Sequential Composition Rule (SCR); $\{L_1 + L_2 + \ + L_k\}$, yields the distributed in Figure 2.

Superlatively, we would like to use SCR through the mobile software life cycle but use the DCR at runtime. The mobile problem is that two compositions, in general, do not yield the same semantics. The SCR mobile program exhibits only a subset of all probable computation paths that can happen in the DCR mobile program. This means that, in general, the revolution from SCR to DCR at runtime does not necessarily preserve the program semantics.

begin $S_{11};S_{12}; \dots ;S_{1k}$ end
||
begin $S_{21};S_{22}; \dots ;S_{1k}$ end
||
begin $S_{31};S_{32}; \dots ;S_{1k}$ end
||
…
||
begin $S_{n1};S_{n2}; \dots ;S_{nk}$ end

Figure 1 The mobile distributed components composition rule

begin $S_{11};S_{12}; \dots ;S_{1k}$ end
;
begin $S_{21};S_{22}; \dots ;S_{1k}$ end
;
begin $S_{31};S_{32}; \dots ;S_{1k}$ end
;
…
;
begin $S_{n1};S_{n2}; \dots ;S_{nk}$ end

Figure 2 The Mobile sequential components composition rule

4. Aspect orientation for expressing

We use the familiar aspect oriented semantics (Kiczales, 2001) to provide a common frame or a reference that makes it possible to define the structure of the crosscutting concerns inherit in the CCL design.

The CCL Join Point: *Joint points* are certain well defined points in the execution flow of a program. The CCL join points are defined at the beginning and the end of each program segment S_{ij}.

The CCL Pointcut Designators: *Pointcut designators* identify particular joint points by filtering out a subset of the entire join points in the program flow. The CCL pointcut designators are filtering out the joint points at each layer boundaries. Pointcut layer j filters out all CCL join points S_{ij} for $i = 1 \dots n$.

The CCL Advice: *Advice* declarations are used to define an additional code that runs at join points. The CCL advices are used to communicate local process states; or just a relevant subset of it, to establish teamwork cooperation.

The CCL Aspect: An aspect is a modular unit of crosscutting implementation. The *CCL aspects* are assertions over virtual global states that are verified at runtime. Since a global state is a collection of local states its realization is a crosscutting concern.

Current research has already established the use of aspects in verifying and imposing preconditions and postconditions of "design by contract" (Meyer, 1993). Aspects make it possible to implement preconditions and postconditions in a modular form. Also a consistent behavior across a large number of operations could be implemented in a much simpler way because of the localization of crosscutting concerns. The contribution of this paper is the extension of the class of properties that can be verified and imposed using aspect orientation approach. The "virtual global states" as defined in (Elrad, Baoling, & Nastasic, 1996) is the distributed programming equivalence to the simple state in sequential programming over which assertions are defined.

To best explain the use of aspect orientation in breaking the tyranny of process composition in distributed programs, we use the well-known two-phase commit protocol. A complete formal CCL development of this protocol is given in (Elrad & Kumar, 1991; Elrad & Kumar.1993; Janssen & Zwiers, 1992a and b; Poel & Zwiers, 1992)

5. Aspect orientation for expressing CCL

The two-phase commit protocol is an example used in distributed database to guarantee consistency of the database. A coordinator process receives a request to initiate a voting, it should return, "*COMMIT*," if all processes participating in the voting process vote either "*yes*" or "*fail*." The voting process is a distributed program called the "coordination." The coordination can be farther decomposed into four layers that reflect the logical structure of the program into its sequential sub-goals. The REQUEST is a vote requesting message passed between the coordinator and each of the participants. The VOTE is a voting process, based on its local state deliberates yes or no reply. The DECIDE is a process that the coordinator collects the votes and computes the collective consensus. The EFFECTUATE is a process that the final decision is passed back to all participants by acting accordingly. Note that this decom-position is orthogonal to the processes decom-position. The process decomposition is assigned every voting participant process(i) a roll in each of these layers.

```
$\forall i \in (1..n)\, Process(i) ::$
begin
    reguest-$i$;
    vote-$i$;
    decide-$i$;
    effectuate-$i$;
end
```

The two-phase commit protocol decomposition into layers is given in Figure 3. The communication closed layers safety theorem applied to this example states that if each of the four layers in closed communications are allowed only between request segments, between vote segments, between decide segments, and between effectuate segments but never between a request segment and a vote segment – then the two compositions are semantically equivalent. This means that we can use the SCR during software life cycle development and the DCR for the actual execution

REQUEST layer is [request-*1* || request-*2* || …|| request-*n*]

VOTE layer is [vote-*1* || vote-*2* || …|| vote-*n*]

DECIDE layer is [decide-*1* || decide-*2* || …|| decide-*n*]

EFFECTUATE layer is [effectuate-*1* || effectuate -*2* || …|| effectuate -*n*]

Figure 3 The SCR of two-phase commit protocol

Figure 4 illustrates a formal specification in terms of preconditions and postconditions that reflects the "design by contract" of the two-phase commit protocol using the SCR. What we like to emphasize here is not so much detail of the specifications, but rather the nature of the global assertions that reflects distributed cooperation. These assertions are called virtual global assertions because there might not be any real time at which any one of them holds. Each process reaches its own layer boundaries at a different time

6. A formal aspect oriented CCL

We can use join points and pointcut designators to define virtual global time and virtual global state.

Let [P_1 || P_2 || … || P_n] be a distributed program **P** composed of n processes P_1 to P_n. and let BEGIN layer-1; layer-2; … ; layer-k END be the complementary program composition into k layers.

Regarding the two-phase commit protocol, the executable program is the one yieled by the DCR, so first, we need to wrap each segment with an aspect.

```
$\forall i \in (1..n), Process(i) ::$
BEGIN
    request-$i$;
    vote-$i$;
    decide-$i$;
    effectuate-$i$;
END
```

There are four CCL join points for every process(i): around request-*i*, around vote-*i*, around decide-*i* and around effectuate-*i*. There are four CCL pointcut designators: $\forall i \in (1..n)$ requiest-*i*, $\forall i \in (1..n)$ vote-*i*, $\forall i \in (1..n)$ decide-*i*, $\forall i \in (1..n)$ effectuate-*i*.

Virtual global times that we would like to express are: the virtual time when the system achieved its *REQUEST* sub-goal, the virtual time when the system achieved its *VOTE* sub-goal, the virtual time when the system achieved the *DECIDE* sub-goal, the virtual time when the system achieved the *EFFECTUATE* sub-goal. At each virtual time we have the associated virtual global state and the virtual global assertions as given in Figure 4.

The advice code that runs at join points takes a snapshot of the process local state (or just an appropriate subset of all variables that appear in a global invariant) and copies it into a pool of all such snapshots. When a pool is full; all processes have passed their appropriate layer boundaries, the verification of the global assertion can be evaluated. An intelligent decision could be made based on this evaluation.

The roles played by states, assertions, and invariants in sequential programming using design by contract discipline – can be played by virtual global states, virtual assertions, and virtual invariants in distributed programming. The effectiveness of this approach increases with the degree of logical cooperation and the degree of communication between the processes.

Aspect-oriented software development principles support the CCL distributed software development. CCL practical implementation relies on an effective handling crosscutting concerns.

$$\left\{\forall i\in(1..n)(vote-i=NONE\wedge decision-i=NONE)\right\}$$

REQUEST

$$\left\{\forall i\in(1..n)(vote-i=NONE\wedge decision-i=NONE)\wedge message=VOTE\_REQ\right\}$$

VOTE

$$\left\{\forall i\in(1..n)(vote-i=yes)\Leftrightarrow stable-i\wedge(vote-i=no)\Leftrightarrow stable-i\right\}$$

DECIDE

$$\left\{\begin{array}{l}\forall i\in(1..n)(vote-i=yes)\Leftrightarrow stable-i\wedge(vote-i=no)\Leftrightarrow stable-i\wedge\\((decision=COMMIT\wedge\forall i\in(1..n)(vote-i=yes))\vee\\(decision=FAIL\wedge\forall i\in(1..n)(vote-i=no))\end{array}\right\}$$

EFFECTUATE

$$\left\{\begin{array}{l}((decision=COMMIT\wedge\forall i\in(1..n)(vote-i=yes)\vee\\(decision=FAIL\wedge\forall i\in(1..n)(vote-i=no))\wedge\\\forall i\in(1..n)(decision-i=decision))\end{array}\right\}$$

Figure 4 Virtual global assertions for two-phase commit protocol

One of the principles in software development is the visibility rule: a significant concern should be syntactically visible. Aspect orientation strength is mainly due to elevating crosscutting concerns to be syntactically visible. The CCL strength is mainly due to elevating the cooperative structure of distributed software to be syntactically visible. In the past, we had mostly application where processes, for the most part, did not interfere with each other. Resources management enforced sharing. Now, we see more applications where there is a higher degree of processes cooperation, the processes do not merely share resources, but actually have common goals. Such a mobile program common goals are significant concerns, yet these concerns are not syntactically visible. Given a mobile software program, it is impossible to decompose it back to its logical structure in terms of common sub-goals. These types of applications can benefit from an aspect orientation realization of CCL development.

The following are examples of the benefits:

Testing- Virtual global assertions and global invariants of the program could be tested during run example, if one process is executing at layer-1 and all the rest are already at layer-2, any cooperation with the legging process concerning the second sub-goal needs to be put on suspension. A smart scheduling can prevent this by always preferring a process that is executing in a lower layer over one that is executing in a higher one.

Real-time application – CCL provides the virtual global time vector. The vector components with the highest value can be considered as the "real-time" at which a sub-goal has been achieved. Different applications might need to eliminate execution of a non-crucial layer in case of time constraints. When a real-time computation cannot be completed, at least we get an approximation by concerning the latest assertion evaluated.

A partial list of more application of CCL can be found in (Elrad, Baoling, & Nastasic, 1996; Elrad, 1984; Elrad & Kumar, 1990; Elrad & Kumar, 1991; Elrad & Kumar.1993; Fokkinga, Poel, & Zwiers, 1993; Gerth & Shrira, 1986; Janssen & Zwiers, 1992a; Janssen & Zwiers, 1992b; Janssen & Zwiers, 1993; Janssen, Poel, Xu, & Zwiers, 1994; Kiczales, 2001; Stomp & Roever, 1987)

7. Conclusion

Two of the aspect orientation characteristics defined by Filman, and Friedman (2000) and Elrad et al. (2001) are enlightening here: the quantification and the understood invocation. Without these, the implementation of an aspect-oriented mobile software using CCL is problematic, rich in code tangling and hence not attractive from practical point. Aspect orientation approach separates concerns from the rest of the software. It enables clean integration between mobile processes or components composition and the mobile software layer composition. The tyranny of mobile processes or components of software composition is mildly substituted with synchronicity of both process or component composition and layer composition. Design and code implementation using this approach composition is not tangle with the design and code implementing using other compositions.

The roles played by states, assertions, and invariants in sequential programming using design by contract discipline. The effectiveness of this approach increases with the degree of rational collaboration and the degree of communication between the processes or components. The CCL practical implementation relies on an effective handling of separation of concerns for the mobile software design and development.

8. References

Ali, N., & Ramos, I. (2012). Designing mobile aspect-oriented software architectures with ambient. In P. Alencar & D. Cowan (Eds.), Handbook of Research on Mobile Software Engineering: Design, Implementation, and Emergent Applications, Volume II (Chapter. 29, 526-543). PA, USA: Engineering Science Reference (an imprint of IGI Global). DOI: 10.4018/978-1-61520-655-1.ch029

Apple Developer. (2016). https://developer.apple.com Accessed on March 15, 2016.

Android Developers. (2016). http://developer.android.com Accessed on March 15, 2016.

Eclipse website. (2016). http://www.eclipse.org Accessed on March 15, 2016.

Elrad, T., & Frances, N. (1982). Decomposition of distributed programs into Communication Closed Layers. *Science of Computer Programming*, *2*(1), 155-173. North-Holland.

Elrad, T. (1984). A practical software development for dynamic testing of distributed programs. *IEEE Proceedings on the International Conference on Parallel Processing,* Bellaire, Michigan, USA, 388-392.

Elrad, T., & Kumar, K. (1990). State space abstraction of concurrent systems: a means to computation progressive scheduling. *Proceedings of the 19th International Conference on Parallel Processing***,** Bellaire, Michigan, USA, pp. 482-483.

Elrad, T., & Kumar, K. (1991). The use of communication closed layers to support imprecise scheduling for distributed real-time programs. *Proceedings of the 10th Annual International Conference on Computer and Communications***,** Scottsdale, AZ, USA, pp. 226-231. DOI: 10.1109/PCCC.1991.113815

Elrad, T., & Kumar, K. (1993). Scheduling cooperative work: viewing distributed system as both CSP and SCL. *Proceedings of the 13th International Conference on Distributed Computing Systems,* Pittsburgh, PA, USA, pp. 532-539. DOI: 10.1109/ICDCS.1993.287670

Elrad, T., Baoling, S., & Nastasic, N. (1996). A synergy of object-oriented concurrent programming and program layering. *Concurrency and Parallelism, Programming, Networking, and Security,* *Lecture Note in Computer Science No. 1179*, pp. 223-233, Springer-Verlag Press. DOI: 10.1007/BFb0027795

Elrad, T., Filman, B., & Bader, A. (2001). Aspect-oriented programming: Introduction. *Communications of ACM*, *44*(10), 29-32. DOI: 10.1145/383845.383853

Filman, R. E., & Friedman, D. P. (2000). Aspect-oriented programming is quantification and obliviousness. *Workshop on Advanced Separation of Concerns. Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA 2000)*, Minneapolis, MN, USA,

Fokkinga, M., Poel, M., & Zwiers, J. (1993). Modular completeness for communication closed layers. *Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems*, pp. 50-65, Springer-Verlag Press. DOI: 10.1007/3-540-57208-2_5

Gerth, R. T., & Shrira, L. (1986). On proving communication closeness of distributed layers. *Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science*, New Delhi, India. DOI: 10.1007/3-540-17179-7

Janssen, W., & Zwiers, J. (1992a). From sequential layers to distributed processes: deriving a distributed minimum weight spanning tree algorithm. *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pp. 215-227. DOI: 10.1145/135419.135461

Janssen, W., & Zwiers, J. (1992b). Protocol design by layered decomposition: A composition approach. *Proceedings of the second International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 307-326. London, UK: Springer-Verlag Press.

Janssen, W., & Zwiers, J. (1993). Specifying and proving communication closeness in protocol. *Proceedings of the 13th IFIP Symposium on Protocol Specification, Testing and Verification*.

Janssen, W., Poel, M., Xu, Q., Zwiers, J. (1994). Layering of real-time distributed processes. *Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems*, Springer-Verlag Press. DOI: 10.1007/3-540-58468-4_175

Kiczales, G. (2001). Getting started with AspectJ. *Communications of ACM*, (44)10, 59-65. DOI: 10.1145/383845.383858

Meyer, B. (1993). Systematic concurrent object-oriented programming. *Communications of ACM*, *36*(9), 56-80. DOI: 10.1145/162685.162705

Poel, M., & Zwiers, J. (1992). Layering techniques for development of parallel systems, Proceedings of Computer Aided Verification. *Lecture Note in Computer Science No. 663*, pp. 16-29, Springer-Verlag Press. DOI: 10.1007/3-540-56496-9_3

Stomp, F. A., & Roever, W. P. (1987). A correctness proof of distributed minimum weight spanning tree algorithm. *Proceedings of the 7th International Conference on Distributed Computer Systems*, Berlin, West Germany.

Wasserman, I. A. (2010). Software engineering issues for mobile application development. *Proceedings of 2010 Future of software engineering research*, Santa Fe, New Mexico, USA, pp. 397-400. DOI: 10.1145/1882362.1882443

Windows Dev Center (2016). http://dev.windows.com/ Accessed on March 15, 2016.