Computer Science Journal of Moldova, vol.**19**, no.1(55), 2011

Architecting software concurrency

Dumitru Ciorbă, Victor Beşliu

Abstract

Nowadays, the majority of software systems are inherently concurrent. Anyway, internal and external concurrent activities increase the complexity of systems' behavior. Adequate architecting can significantly decrease implementation errors. This work is motivated by the desire to understand how concurrency can constrain or influence software architecting. As a result, in the paper a methodological architecting framework applied for systems with "concurrency-intensive architecture" is described. This special term is defined to emphasize architectures, in which concurrent interactions are crucial. Also in the paper potential models for each phase of architecting framework are indicated.

Keywords: software architecture, concurrency, concurrency-intensive architecture, architecting framework, concurrency model, formalization, specification, CSP#

1 Key challenges in managing concurrency

1.1 Benefits and costs of concurrency

Software development undoubtedly passes a revolutionary period. Desired performance can be achieved not only by increasing processor frequencies. This outlines a future that is determined by multi-core/multiprocessor architectures and multi-threading programs [1].

The benefits of concurrency reside in the full exploitation of advantages offered by multi-core/multiprocessor architectures, primarily, through the possibility to represent naturally and separately concurrent activities. This leads directly or indirectly to the key characteristics of modern software such as: availability, protection, scalability, performance.

However, concurrency has its costs for organizing inter-thread synchronization and communication. Non-determinism, inherent in concurrent systems, requires taking into account a number of properties that must satisfy programs (mutual exclusion, absence of deadlock, livelock, starvation and race conditions, etc.) [2]. Thus the coordination of activities is complicated, and multithreading related errors are intermittent and difficult to reproduce. Moreover, synchronization and communication worsen effects of code scattering and code tangling [3]; involve reuse difficulties due to conflicts between basic functionalities and synchronization functionalities. These conflicts have been intensively studied and we find in the literature by the name *the inheritance anomaly.*

1.2 Inheritance anomalies

Several researchers [4,5,6] have attempted to classify and formalize the anomalies that make unreasonable or even impossible to reuse the base class by inheritance, since the redefinition involves an excessive number of methods. In [7] authors state that any popular modern programming languages do not exclude yet the occurrence of these anomalies. The authors propose three generic classes for anomalies where the role of inheritance as a form of reuse is greatly diminished: history-sensitive anomaly; partitioning of states; modification of acceptable states.

In [8] it is proposed unified treatment of phenomena in a general designation - *reuse anomalies,* arguing that in a concurrency context, the above mentioned effects can also occur in the case of aggregation and association relations. This fact is easily claimed in other works. An example is the paper [9] which states that adverse phenomena may occur in cases other than inheritance, referring to the composition anomaly.

In order to investigate this phenomenon, which appears in code reuse in the context of concurrency, formalization was used in [5]. The results of formal analysis presented in it have surprised the scientific community. It can be easily emphasized that the following statements stand out from a more general content of the research:

- Inheritance anomalies are common to other paradigms, not only to object-oriented programming: e.g. agent-oriented programming (based on the Actor model [10]);

- If anomaly is present in the implementation, it does not necessarily cause practical problems;
- Inheritance anomaly problem in one form or another still cannot be resolved, but rather may be reduced adverse effects induced by anomalies.

Making the concurrency explicit and isolating it into a concurrent component, seems to be reasonable solution [11].

1.3 Concurrency "isolation" techniques

Elimination of adverse consequences (by defining separate and explicit orthogonal functionalities) can be accomplished in many ways at all levels of abstraction of software development (Figure 1): mix-in classes [12] and aspects [3,13,14] can be used in order to compose functionalities at implementation phases; design patterns allow us localization and integration of orthogonal functionalities in analysis and design stages [15,16]; and programming frameworks can help us improving components reuse [17,18,19].

The technological possibility of locating the synchronization code is not the solution by itself. It happens because the problem of inheritance anomalies cannot be fully solved [5]. Moreover in [20] the authors state that *aspectization* (localization) of concurrency in some cases may even be dangerous. Thus we have to diminish bad effects of anomalies with appropriate methods of concurrency management in the early stages of software development.

1.4 Concurrency management

Concurrency management has to provide a proactive strategy that will allow concurrency organization and management throughout the software life cycle. A successful strategy will be determined by the following characteristics:

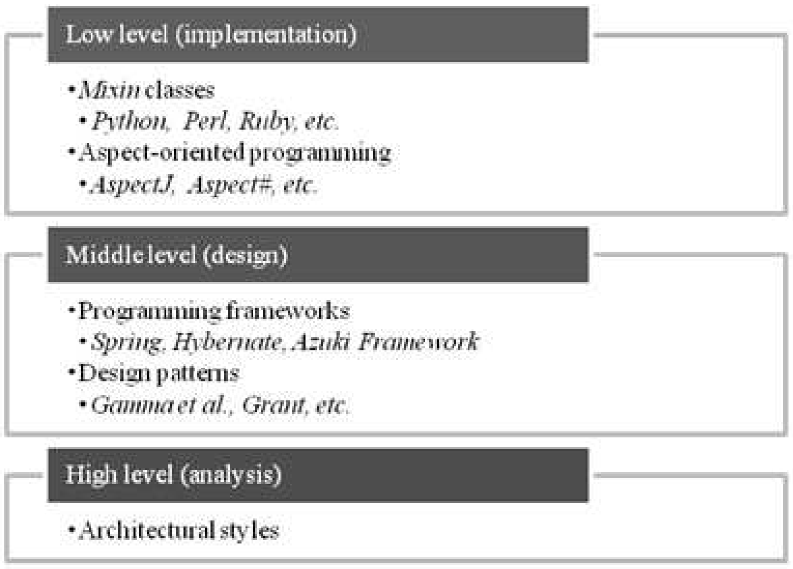

Low level (implementation)
•Mixin classes
•Python, Perl, Ruby, etc.
•Aspect-oriented programming
•AspectJ, Aspect#, etc.
Middle level (design)
•Programming frameworks
•Spring, Hybernate, Azuki Framework
•Design patterns
•Gamma et al., Grant, etc.
High level (analysis)
•Architectural styles

Figure 1. Abstraction levels of software development

- Process development centered: the strategy will require integration with existing processes and applications;
- Centrally managed: all development activities must comply with adopted policies;
- Heterogeneous: the strategy must take into account different organizational forms of concurrency.

One of the early activities of the strategy should include the classification of developed systems in accordance of the degree of concurrency. Some information systems have no concurrency whilst others become more concurrent, in order to maximize efficiency determined by a number of factors (e.g. cheap multiprocessor). In this context, there can be defined a special term: *concurrency-intensive architecture.* It is necessary for emphasizing architecture, where concurrency influences essentially the architecting. Concurrency-intensive architecture thus needs a special and distinct architecting approach.

2 Architecting and concurrency

Architecting is a process where the outcome is stakeholder's satisfaction towards architectural requirements. In the context of software architecting, *abstraction* is one of the main principles. It captures through *encapsulation* the essential properties of a system.

There are numerous definitions of architecture. An interesting idea is mentioned in [21] according to which "*a system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture*". Anyway the entire concept, according to some researchers, is ambiguous in relation to information systems research and practice [22]. Even though, most of them define architecture analogically: as an abstract concept, which provides a certain perspective on information system [23], it was still necessary to standardize the architecting process. Finally, IEEE in 2000 [24], and ISO in 2007 [25] have standardized this conception: "*the activities of the creation, analysis and sustainment of architectures of software-intensive systems, and the recording of such architectures in terms of architectural descriptions*". The main result of this architecting standardization is a clear and comprehensive documentation of the architecture representations of information systems in various views.

This approach can be traced from the earliest major research directions and frameworks for software development. According to [26], architecture can be described in five main views: *logical*, *process*, *physical*, *development* and *scenarios*; the last of them is essentially redundant, but it represents the interaction of the other four (Figure 2).

In the architectural model presented above, planned active processing entities, communication structure, integrity, and other architecturally significant concerns of the control flow management, synchronization and concurrency are described in the *process view.*

It is important to note that the view, according to definitions from [24,25,26], is not yet the localization of implemented functionalities;

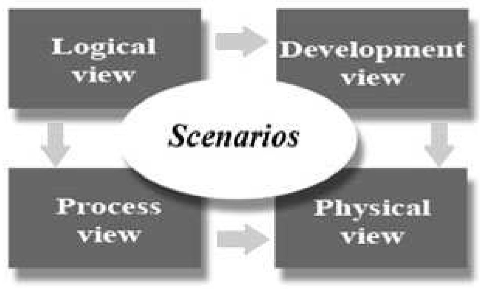


Figure 2. The "4+1" View Model of Software Architecture [26]

this is the representation of the entire system from some viewpoint of interrelated requirements (aspects).

Now the following question arises: is it possible to separate synchronization and concurrency concerns from the functional ones and localize them in a separate *structural unit*? The answer is simple: yes, it is possible and necessary [27,28]; but in order to integrate separate functionalities, developer has to either use non-traditional aspect-oriented programming [29], or create applications with loosely coupled architecture.

In loosely coupled architectures the separation, localization and composition of functionalities can be achieved by applying adequate design patterns [30], asynchronous messaging architectures [31], and/or event-driven architectures [32].

These days, most information systems are inherently concurrent, since they handle activities that can happen simultaneously in the world external to the system's world [33]. It is likely then, that concurrency is considered as a critical property of systems and that it must be considered in the early development stages – architecting stages.

3 Concurrency-intensive software architecting

According to IBM *Rational Unified Process*^{TM}, architecture is defined during the inception and elaboration phases [34]. This popular software development process is architecture centric. It means that the system's architecture is a primary artifact for system's development [35]. Thus the importance of an accurate architecting must be sustained by a distinct process.

In this context *The Visual Architecting Process*^{TM} (VAP) can be mentioned. It is promoted by Bredemeyer Consulting. According to it, the architecture specification phase consists of iterative five sub-phases: *Meta-Architecture*, *Conceptual Architecture*, *Logical Architecture*, *Execution Architecture* and *Architecture Guidelines* [36]. Concurrency issues are tightly related with *Execution Architecture*, where analysis focuses on *Process* and *Deployment* views.

Execution Architecture phase is the forth one. This allows us to confirm that concurrency, as a critical property of modern systems, is considered too late. This can be argued by means of structuralism [37], according to which "*a structure may be defined as a network of relationships between elements or elementary process... A structure thus manifests itself by means of relationships; a system manifests itself by means of communications of the relevant elements. A function within a system may be seen as a communicative relationship.*" Thus communication is necessary to "transport" data; but it is the means of communication by which *control information* is being transported as well. This particular communication form is well known as synchronization, which constraints event ordering and controls processing unit interference. It is usual then, that the concurrency influences systems via the communication style, hence concurrency must be an important factor for structural and functional analysis of architecture.

There are numerous architecting methodologies where concurrency is one of the key factors. Here it's case to mention *Nick Rozanski* and *Eoin Woods' work* [38], where concurrency concerns are described from the point of view of *Concurrency Viewpoint.*

A viewpoint is a way of looking at the system, and does not capture architecting focused on concurrency concerns. In this context a generic framework is proposed below. It will permit us to analyze concurrency-intensive architecture from the perspective of evolution (Figure 3).

The architecting process, represented in Figure 3, is a waterfall

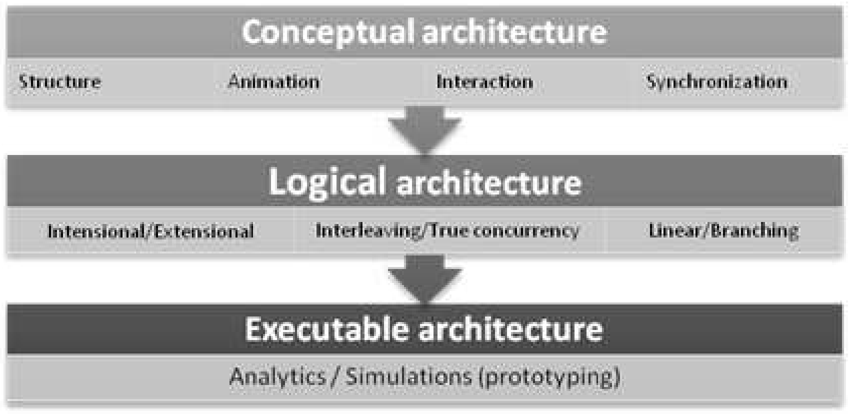


Figure 3. Concurrency-intensive software architecting

process with three phases. Each phase defines an architecture model, which may also include a number of views, where each view is related to a particular domain of the phase.

Conceptual architecture defines entities, their relationships and conceptual constraints. The structure view can show configurations in terms of components, which are units of runtime computation or data-storage, and connectors, which in their turn are the interaction mechanisms between components [39].

In order to facilitate structuring, architectural patterns can be used. A coherent set of related patterns makes up a pattern language. An interesting pattern language is presented in [40].

The last three views have been inspired by a survey of concurrency issues presented in [6]. The survey is organized of taxonomy of the features of concurrent object-oriented languages. In spite of the generalization of the described models, they allow us to use them in our architecting process as views.

Animation view shows the relationship between objects and active entities (process/thread/task). The treatment of threads and objects as independent or dependent concepts, defines two alternatives of activity organization: unrelated and related models (Figure 4).

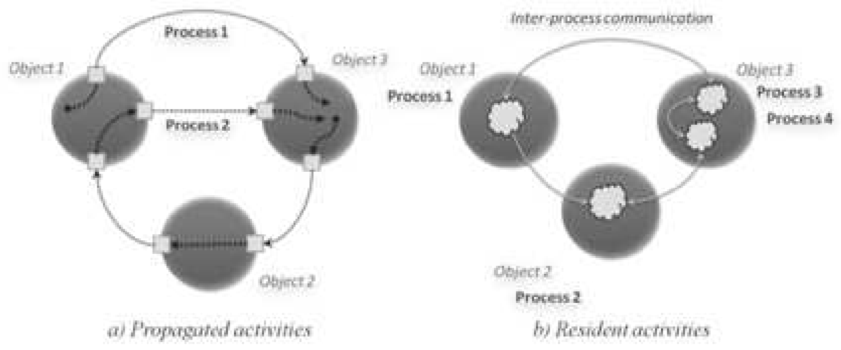


Figure 4. Unrelated (a) and related (b) models of Animation view

The interaction view depicts interactions between objects initiated by the client's invocation, which may be either synchronous or asynchronous. Semantics of returns is defined in this view as well.

Concepts, related to Synchronization view, specify concurrent invocation management. It is important to mention that *Conceptual architecture* phase defines rules to synchronize, select and accept operations, and these rules define control constraints used at the next Logical architecture phase.

Logical architecture conforms to the principles and rules of the conceptual architecture. This phase involves a variety of structures, which have the nature of mathematical formalisms. Logical architecture thus is represented by a generalized formal structure that determines *logic of specification,* which helps describing and reasoning about behavior of concurrent architecture.

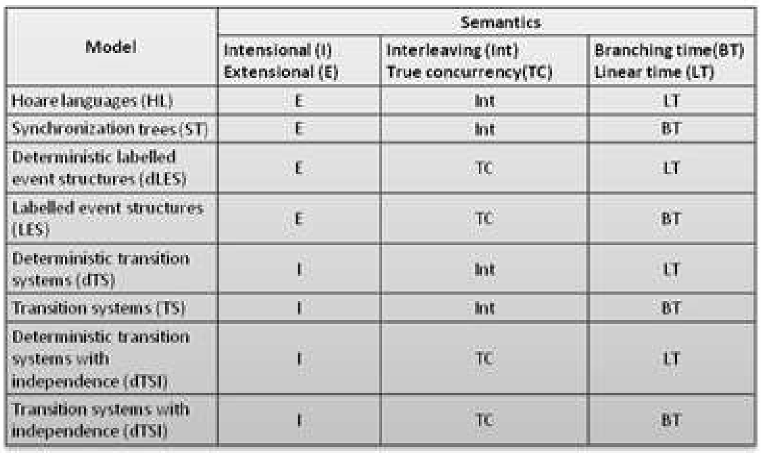
Numerous formal models have been studied over the past 20-25 years. Formal semantics, provided by these models, can be classified by partitioning criteria of the following dichotomies [41,42]:

- *Intensional* and *Extensional* semantics,
- *Interleaving* and *True concurrency* semantics,

- *Branching time* and *Linear time* semantics.

Specifying systems as "machines", determined by states (and possible state changes), obtains *intensional* models. *Extensional* (*also known as* behavior) models focus on occurrence patterns of actions over time. Concurrency is an implicit property in "true concurrency" models. Yet interleaving models reduce it to nondeterministic interleaving representations. Last dichotomy splits models into nondeterministic branching models and linear time setting models. In the former case, models describe concurrency in terms of the sets of their possible (partial) executions.

Formal relationships between models have been analyzed by many researchers. Here should be mentioned the work [43], where translation between models have been studied in terms of category theory. Eight models, more precisely model classes, have been obtained by varying, in all possible ways, the aforementioned criteria (Figure 5).



Model	Semantics		
	Intensional (I) Extensional (E)	Interleaving (Int) True concurrency(TC)	Branching time(BT) Linear time (LT)
Hoare languages (HL)	E	Int	LT
Synchronization trees (ST)	E	Int	BT
Deterministic labelled event structures (dLES)	E	TC	LT
Labelled event structures (LES)	E	TC	BT
Deterministic transition systems (dTS)	I	Int	LT
Transition systems (TS)	I	Int	BT
Deterministic transition systems with independence (dTSI)	I	TC	LT
Transition systems with independence (dTSI)	I	TC	BT

Figure 5. Concurrency models

Positioning of models from Figure 5 in a three-dimensional space relative to dichotomies is represented in Figure 6.

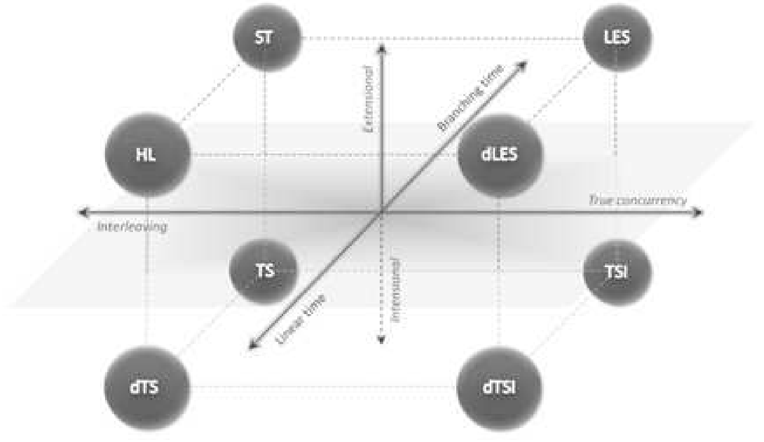


Figure 6. Semantic models for concurrency specification

Executable architecture is a result "product" of the last architecting phase. In the first instance, this term expresses the description of the system's architecture in a formal notation, the semantic of which is being determined by the logic architecture phase. In the second instance, after using automatic or semi-automatic generation tools, this term may signify a partial implementation of the system – a prototype, which must validate all architecturally significant requirements. A closely related term is the term of *evolutionary prototype*, which is not a work product, but it is stable enough to be considered as a first approximation of a system. According to the article [44], "*producing an evolutionary prototype means that you design, implement, and test a skeleton structure, or architecture, of the system...*"

4 Conclusion

Systems become more concurrent and a new term is needed to define an architecture influenced essentially by concurrency. In this work a new term *concurrency-intensive architecture* is proposed. Also it is being shown that concurrency, as a key factor, can determine a generic architecting framework by providing an architectural prototype. It perfectly fits in the modern architecture-centric development methodologies, such as Rational Unified Process. However concurrency generates difficulties. Still the right methods and tools can decrease the architecting effort. Mature theories and models of concurrency exist; thus the key target of researches is: developing of methods and tools of specification and verification. So an immediate and important objective is to develop and integrate a graphical language of concurrency specification in one of the popular development environments. Language must expressively specify concurrency, thus must use denotational semantic. Such specification language based on events will be presented in the future article. Also, in this article it will be shown how models are verified with operational semantic of CSP# language.

References

[1] S. Herb. *The Free Lunch Is Over. A Fundamental Turn Toward Concurrency in Software.* Dr. Dobb's Journal. [Online] March 2005, Vol. 30, 3. [Cited: April 08 2011] http://www.gotw.ca/publications/concurrency-ddj.htm.

[2] S. Owicki, L. Lamport. *Proving Liveness Properties of Concurrent Programs.* ACM Transactions on Programming Languages and Systems. July 1982, Vol. 4, 3, pp. 455–495.

[3] G. Kiczales, et al. *Aspect-Oriented Programming.* Proceedings of ECOOP'97. s.l.: Springer-Verlag, 1997.

[4] S. Matsuoka, A. Yonezawa. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages.* Research Directions in Concurrent Object-Oriented Programming. s.l.: MIT Press, 1993.

[5] L. Crnogorac, A.S. Rao, K. Ramamohanarao. *Classifying Inheritance Mechanisms in Concurrent Object-Oriented Programming.* Proceedings of ECOOP'98. s.l.: Springer-Verlag, 1998. LNCS 1445.

[6] D.G. Kafura, G. Lavender. *Concurrent Object-Oriented Languages and the Inheritance.* [ed.] T. L. Cassavant. Parallel Computers: Theory and Practice. s.l.: IEEE Press, 1994. pp. 165–198.

[7] G. Milicia, Vl. Sassone. *The inheritance anomaly: ten years after.* Proceedings of Symposium on Applied Computing (SAC). Nicosia, Cyprus : ACM, 2004.

[8] D.M. Suciu. *Techniques for Implementing Concurrency in Object-Oriented Analysis and Design.* Ph. D. thesis. [http://cs.ubbcluj.ro/∼tzutzu/Research/Teza.pdf]. Cluj-Napoca: Babeş-Boleay University, 2001.

[9] L. Bergmans, M. Aksit. *Composing Software from Multiple Concerns: A Model and Composition Anomalies.* Proceedings of ICSE 2000 (2nd) Workshop on Multidimensional Separation of Concerns. 2000.

[10] C. Hewitt. *Actor Model of Computation.* arXiv.org. [Online] Noiembrie 08, 2010. [Cited: Noiembrie 26, 2010.] http://arxiv.org/abs/1008.1459v8. arXiv:1008.1459v8.

[11] B. Goetz. *Concurrency: Past and Present.* InfoQ. [Online] August 26, 2006. [Cited: April 08, 2011.] http://www.infoq.com/presentations/goetz-concurrency-past-present.

[12] G. Bracha, W. Cook. *Mixin-based inheritance.* Proceedings of OOPSLA/ECOOP '90. s.l.: ACM Press, 1990.

[13] V. Pavlov. *Aspect-oriented programming.* IT-Archiv. [Online] June 2003. [Cited: November 21, 2010.] http://www.javable.com/columns/aop/workshop/01/index.pdf.

[14] R. Pawlak, L. Seinturier, Ph. Retaillé. *Foundations of AOP for J2EE development.* [http://books.google.com] s.l.: Apress, 2005. ISBN 1-59059-507-6.

[15] E. Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l.: Addison-Wesley, 1994. ISBN 0-201-63361-2.

[16] D. Lea. *Concurrent programming in Java: design principles and patterns.* [http://books.google.com] s.l.: Addison-Wesley, 2000. ISBN 0-201-31009-0.

[17] M. E. Fayad, D. C. Schmidt. *Object-Oriented Application Frameworks.* Communications of the ACM. 1997, Vol. 40, 10.

[18] R.E. Johnson. *Frameworks = (components + patterns).* Communications of the ACM. 1997, Vol. 40, 10.

[19] D.C. Schmidt, M. E. Fayad. *Building Reusable OO Frameworks for Distributed Software.* Communications of the ACM. 1997, Vol. 40, 10.

[20] J. Kienzle, R. Guerraoui. *AOP: Does it Make Sense? The Case of Concurrency and Failures.* École Polytechnique Fédérale De Lausanne (EPFL). [Online] 2002. [Cited: April 08, 2011.] http://ic2.epfl.ch/publications/documents/IC_TECH_REPORT_200216.pdf.

[21] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD dissertation. Irvine: University of California, 2000.

[22] M. S. Corneliussen. *IT Architecturing – Reconceptualizing Current Notions of Architecture in IS Research.* European Conference on Information Systems. Mart 04, 2008.

[23] D. Garlan. *Software Architecture: a Roadmap.* [ed.] A. Finkekstein. The Future of Software Engineering. s.l.: ACM Press, 2000.

[24] M. W. Maier, D. Emery, R. Hilliard. *Software Architecture: Introducing IEEE Standard 1471.* Computer. April 2001, Vol. 34, 4, pp. 107–109.

[25] International Organization for Standardization. *Recommended Practice for Architectural Description of Software-Intensive Systems.* [Online] 2007. [Cited: Decembre 2, 2010.] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45991. ISO/IEC 42010:2007.

[26] Ph. Kruchten. *Architectural Blueprints — The "4+1" View Model of Software Architecture.* IEEE Software. 1995. Vol. 12, 6.

[27] W. Hürsch, Cr. Lopes. *Separation of Concerns.* College of Computer Science, Northeastern University. Boston, USA: s.n., 1995. Technical report.

[28] C. A. Constantinides, T. Elrad. *On the Requirements for Concurrent Software Architectures to Support Advanced Separation of Concerns.* OOPSLA'2000, Workshop on Advanced Separation of Concerns in Object-Oriented Systems. 2000.

[29] G. Kiczales, et al. *Aspect-Oriented Programming.* Proceedings of ECOOP'97. s.l.: Springer-Verlag, 1997.

[30] D. Ciorba, et al. *Arch-pattern based design and aspect-oriented implementation of Readers-Writers concurrent problem.* The Computer Science Journal of Moldova. 2007, Vol. 15, 3, pp. 338–353.

[31] G. Hohpe. *Integration Patterns Overview. Enterprise intergration pstterns.* [Online] 2010. [Cited: Dec 06, 2010.] http://www.eaipatterns.com/eaipatterns.html.

[32] M. Fowler. *Event Collaboration.* Development of Further Patterns of Enterprise Application Architecture. [Online] Jun 19, 2006. [Cited: Dec 05, 2010.] http://martinfowler.com/eaaDev/EventCollaboration.html.

[33] J. Bacon, T. Harris. *Operating systems: concurrent and distributed software design.* s.l.: Pearson Education, 2003. ISBN 0-321-11789-1.

[34] IBM Rational Software. *Rational Unified Process. Best Practices for Software Development Teams.* developerWorks®:

IBM's resource for developers and IT professionals. [Online] Jul 23, 2005. [Cited: May 11, 2011.] http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf.

[35] H. Yim, et al. *Architecture-Centric Object-Oriented Design Method for Multi-Agent Systems.* Fourth International Conference on Multi-Agent Systems (ICMAS'00). Los Alamitos, CA, USA: IEEE Computer Society, 2000. ISBN: 0-7695-0625-9.

[36] R. Malan, D. Bredemeyer. *The Visual Architecting Process.* [http://www.bredemeyer.com/papers.htm] s.l.: Bredemeyer Consulting, January 2005.

[37] J.M. Broekman, *Structuralism.* Synthese library. Monographs on epistemology, logic, methodology, philosophy of science, sociology of science and of knowledge. s.l.: Springer, 1974. Vol. 67. ISBN 9-027-70478-3.

[38] N. Rozanski, E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives.* s.l.: Addison-Wesley Professional, 2005. ISBN 0-321-11229-6.

[39] P. Clements. *Documenting software architectures: views and beyond.* s.l.: Addison-Wesley, 2003. ISBN 0-201-70372-6.

[40] P. Avgeriou, U. Zdun. *Architectural Patterns Revisited – A Pattern Language.* In Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005). Irsee, Germany: s.n., July 2005. pp. 1–39.

[41] R. Cleaveland, S. A. Smolka. *Strategic Directions in Computing Research Concurrency Working Group Report.* Stony Brook University – Department of Computer Science. [Online] September 30, 1996. [Cited: January 31, 2011.] http://www.cs.sunysb.edu/∼sas/sdcr/report/final/final.html.

[42] T. T. Hildebrandt. *Categorical Models for Concurrency: Independence, Fairness and Dataflow.* [http://www.brics.dk/DS/00/1/BRICS-DS-00-1.pdf] Aarhus, Denmark: University of Aarhus, February 2000. BRICS Dissertation Series. ISSN 1396-7002.

[43] Vl. Sassone, M. Nielsen, G. Winskel. *Models for Concurrency: Towards a Classification.* Theoretical Computer Science. 1996. Vol. 170, 1-2, pp. 297–348.

[44] P. Krol. *Transitioning from waterfall to iterative development.* developerWorks®: IBM's resource for developers and IT professionals. [Online] April 16, 2004. [Cited: December 18, 2010.] http://www.ibm.com/developerworks/rational/library/4243.html.

Dumitru Ciorbă, Victor Beşliu
Received May 28, 2011

Technical University of Moldova
Automation and Information Technology Department
Str. Studenţilor, 7/3, corp 3, 504 Chişinău, MD-2068
Phone: (+373 22) 509908
E–mail: *victor.besliu@ati.utm.md*, *besliu@mail.utm.md*, *vbesliu@yahoo.com*
E–mail: *diciorba@yahoo.com*, *dumitru.ciorba@ati.utm.md*