PATTERN-DRIVEN REUSE OF EMBEDDED CONTROL DESIGN
Behavioral and Architectural Specifications in Embedded Control System Designs

Miroslav Sveda, Ondrej Rysavy
Faculty of Information Technology, Brno University of Technology, Bozetechova 2, 61266 Brno, Czech Republic
sveda@fit.vutbr.cz

Radimir Vrba
Faculty of Electrical Engineering & Communication, Brno University of Technology, Brno, Czech Republic
vrbar@feec.vutbr.cz

Keywords: Embedded systems, Formal specification, Finite automata, Timed automata, Case-based reasoning.

Abstract: This paper deals with reuse of architectural and behavioral specifications of embedded systems employing finite-state and timed automata. The contribution proposes not only how to represent a system's formal specification as an application pattern structure of specification fragments, but also how to measure similarity of formal specifications for retrieval with case-based reasoning support. The paper provides also an insight into case-based reasoning support as applied to formal specification reuse by application patterns built on finite-state and timed automata. Those application patterns create a base for a pattern language supporting reuse-oriented design process for a class of real-time embedded systems.

1 INTRODUCTION

Methods and approaches in systems engineering are often based on the results of empirical observations or on individual success stories. Every real-world embedded system design stems from decisions based on an application domain knowledge that includes facts about some previous design practice. Evidently, such decisions relate to system architecture components, called in this paper as application patterns, which determine not only a required system behavior but also some presupposed implementation principles. Application patterns should respect those particular solutions that were successful in previous relevant design cases. While focused on the system architecture range that covers more than software components, the application patterns look in many features like well-known software object-oriented design concepts such as reusable patterns (Coad and Yourdon, 1990), design patterns (Gamma et. al., 1995), and frameworks (Johnson, 1997). By the way, there are also other related concepts such as use cases (Jacobson, 1992), architectural styles (Shaw and Garlan, 1996), or templates (Turner, 1997), which could be utilized for the purpose of this paper instead of introducing a novel notion. Nevertheless, application pattern can structure behavioral specifications and, concurrently, they can support architectural components specification reuse.

Nowadays, industrial scale reusability frequently requires a knowledge-based support. Case-based reasoning (see e.g. Kolodner, 1993) can provide such a support. The method differs from other rather traditional procedures of Artificial Intelligence relying on case history: for a new problem, it strives for a similar old solution saved in a case library. Any case library serves as a knowledge base of a case-based reasoning system. The system acquires knowledge from old cases while learning can be achieved accumulating new cases. Solving a new case, the most similar old case is retrieved from the case library. The suggested solution of a new case is generated in conformity with the retrieved old case.

This paper proposes not only how to represent a system's formal specification as an application pattern structure of specification fragments, but also how to measure similarity of formal specifications for retrieval. In this paper, case-based reasoning support to reuse is focused on specifications by finite-state and timed automata, or by state and

timed-state sequences. The same principles can be applied for specifications by temporal and real-time logics.

The following sections of this paper introduce the principles of design reuse applied by the way of application patterns. Then, employing application patterns fitting a class of real-time embedded systems, the kernel of this contribution presents two design projects: petrol pumping station dispenser controller and multiple lift control system. Via identification of the identical or similar application patterns in both design cases, this contribution proves the possibility to reuse substantial parts of formal specifications in a relevant sub-domain of embedded systems. The last part of the paper deals with knowledge-based support for this reuse process applying case-based reasoning paradigm. The paper provides principles of case-based reasoning support to reuse in frame of formal specification-based system design aiming at industrial applications domain.

2 STATE OF THE ART

To reuse an application pattern, whose implementation usually consists both of software and hardware components, it means to reuse its formal specification, development of which is very expensive and, consequently, worthwhile for reuse. This paper is aimed at behavioral specifications employing state or timed-state sequences, which correspond to Kripke style semantics of linear, discrete time temporal or real-time logics, and at their closed-form descriptions by finite-state or timed automata (Alur and Henzinger, 1992). Geppert and Roessler (2001) present a reuse-driven SDL design methodology that appears closely related approach to the problem discussed in this contribution.

Software design reuse belongs to highly published topics for almost 20 years, see namely Frakes and Kang (2005), but also Arora and Kulkarni (1998), Sutcliffe and Maiden (1998), Mili et al. (1997), Holzblatt et al. (1997), and Henninger (1997). Namely the state-dependent specification-based approach discussed by Zaremski et. al. (1997) and by van Lamsweerde and Wilmet (1998) inspired the application patterns handling presented in the current paper. To relate application patterns to the previously mentioned software oriented concepts more definitely, the inherited characteristics of the archetypal terminology, omitting namely their exclusive software orientation, can be restated as follows. A pattern describes a problem to be solved, a solution, and the context in which that solution works. Patterns are supposed to describe recurring solutions that have stood the test of time. Design patterns are the micro-architectural elements of frameworks. A framework -- which represents a generic application that allows creating different applications from an application sub-domain -- is an integrated set of patterns that can be reused. While each pattern describes a decision point in the development of an application, a pattern language is the organized collection of patterns for a particular application domain, and becomes an auxiliary method that guides the development process, see the pioneer work by Alexander (1977).

Application patterns correspond not only to design patterns but also to frameworks while respecting multi-layer hierarchical structures. Embodying domain knowledge, application patterns deal both with requirement and implementation specifications (Shaw and Garlan, 1996). In fact, a precise characterization of the way, in which implementation specifications and requirements differ, depends on the precise location of the interface between an embedded system, which is to be implemented, and its environment, which generates requirements on system's services. However, there are no strict boundaries in between: both implementation specifications and requirements rely on designer's view, i.e. also on application patterns employed.

A design reuse process involves several necessary reuse tasks that can be grouped into two categories: supply-side and demand-side reuse (Sen, 1997). Supply-side reuse tasks include identification, creation, and classification of reusable artifacts. Demand-side reuse tasks include namely retrieval, adaptation, and storage of reusable artifacts. For the purpose of this paper, the reusable artifacts are represented by application patterns.

The following two sections provide two case studies, based on implemented design projects, using application patterns that enable to discuss concrete examples of application patterns reusability.

3 PETROL DISPENSER CONTROL SYSTEM

The first case study pertains to a petrol pumping station dispenser with a distributed, multiple microcomputer counter/controller (for more details see Sveda, 1996). A dispenser controller is

interconnected with its environment through an interface with volume meter (input), pump motor (output), main and by-pass valves (outputs) that enable full or throttled flow, release signal (input) generated by cashier, unhooked nozzle detection (input), product's unit price (input), and volume and price displays (outputs).

3.1 Two-level Structure for Dispenser Control

The first employed application pattern stems from the two-level structure proposed by Xinyao et al. (1994): the higher level behaves as an event-driven component, and the lower level behaves as a set of real-time interconnected components. The behavior of the higher level component can be described by the following state sequences of a finite-state automaton with states "blocked-idle," "ready," "full fuel," "throttled" and "closed," and with inputs "release," (nozzle) "hung on/off," "close" (the preset or maximal displayable volume achieved), "throttle" (to slow down the flow to enable exact dosage) and "error":

blocked-idle $\xrightarrow{release}$ ready $\xrightarrow{hung\ off}$ full_fuel $\xrightarrow{hung\ on}$ blocked-idle
blocked-idle $\xrightarrow{release}$ ready $\xrightarrow{hung\ off}$ full_fuel $\xrightarrow{throttle}$ throttled $\xrightarrow{hung\ on}$ $\xrightarrow{hung\ on}$ blocked-idle
blocked-idle $\xrightarrow{release}$ ready $\xrightarrow{hung\ off}$ full_fuel $\xrightarrow{throttle}$ throttled $\xrightarrow{close}$ $\xrightarrow{close}$ closed $\xrightarrow{hung\ on}$ blocked-idle
blocked-idle $\xrightarrow{error}$ blocked-error
blocked-idle $\xrightarrow{release}$ ready $\xrightarrow{error}$ blocked-error
blocked-idle $\xrightarrow{release}$ ready $\xrightarrow{hung\ off}$ full_fuel $\xrightarrow{error}$ blocked-error
blocked-idle $\xrightarrow{release}$ ready $\xrightarrow{hung\ off}$ full_fuel $\xrightarrow{throttle}$ throttled $\xrightarrow{error}$ $\xrightarrow{error}$ blocked-error

The states "full_fuel" and "throttled" appear to be hazardous from the viewpoint of unchecked flow because the motor is on and the liquid is under pressure -- the only nozzle valve controls an issue in this case. Also, the state "ready" tends to be hazardous: when the nozzle is unhooked, the system transfers to the state "full_fuel" with flow enabled. Hence, the accepted fail-stop conception necessitates the detected error management in the form of transition to the state "blocked-error." To initiate such a transition for flow blocking, the error detection in the hazardous states is necessary. On the other hand, the state "blocked-idle" is safe because the input signal "release" can be masked out by the system that, when some failure is detected, performs the internal transition from "blocked-idle" to "blocked-error."

3.2 Incremental Measurement for Flow Control

The volume measurement and flow control represent the main functions of the hazardous states. The next applied application pattern, incremental measurement, means the recognition and counting of elementary volumes represented by rectangular impulses, which are generated by a photoelectric pulse generator. The maximal frequency of impulses and a pattern for their recognition depend on electro-magnetic interference characteristics. The lower-level application patterns are in this case a noise-tolerant impulse detector and a checking reversible counter. The first one represents a clock-timed impulse-recognition automaton that implements the periodic sampling of its input with values 0 and 1. This automaton with b states recognizes an impulse after b/2 (b>=4) samples with the value 1 followed by b/2 samples with the value 0, possibly interleaved by induced error values, see an example timed-state sequence:

$(0, q_1) \xrightarrow{inp=0} \ldots \xrightarrow{inp=0} (i, q_1) \xrightarrow{inp=1} (i+1, q_2) \xrightarrow{inp=0} \ldots \xrightarrow{inp=0} (j, q_2) \ldots$
$\ldots \xrightarrow{inp=1} (k, q_{b/2+1}) \xrightarrow{inp=1} \ldots$
$\ldots \xrightarrow{inp=1} (m, q_{b-1}) \xrightarrow{inp=0} (m+1, q_b) \xrightarrow{inp=1} \ldots \xrightarrow{inp=1} (n, q_b) \xrightarrow{inp=0/IMP} (n+1, q_1)$
i, j, k, m, n are integers representing discrete time instances in increasing order.

For the sake of fault-detection requirements, the incremental detector and transfer path are doubled. Consequently, the second, identical noise-tolerant impulse detector appears necessary.

The subsequent lower-level application pattern used provides a checking reversible counter, which starts with the value (h + l)/2 and increments or decrements that value according to the "impulse detected" outputs from the first or the second recognition automaton. Overflow or underflow of the pre-set values of h or l indicates an error. Another counter that counts the recognized impulses from one of the recognition automata maintains the whole measured volume. The output of the letter automaton refines to two displays with local memories not only for the reason of robustness (they can be compared) but also for functional requirements (double-face stand). To guarantee the overall fault detection capability of the device, it is necessary also to consider checking the counter. This task can be maintained by an I/O watchdog application pattern that can compare input impulses from the photoelectric pulse generator and the changes of the total value; evidently, the appropriate automaton provides again reversible counting.

3.3 Fault Maintenance Concepts

The methods used accomplish the fault management in the form of (a) hazardous state reachability control and (b) hazardous state maintenance. In safe states, the lift cabins are fixed at any floors. The system is allowed to reach any hazardous state when all relevant processors successfully passed the start-up checks of inputs and monitored outputs and of appropriate communication status. The hazardous state maintenance includes operational checks and, for shaft controller, the fail-stop support by two watchdog processors performing consistency checking for both execution processors. To comply with safety-critical conception, all critical inputs and monitored outputs are doubled and compared; when the relevant signals differ, the respective lift is either forced (in case of need with the help of an substitute drive if the shaft controller is disconnected) to reach the nearest floor and to stay blocked, or (in the case of maintenance or fire brigade support) its services are partially restricted. The basic safety hard core includes mechanical, emergency brakes.

Because permanent blocking or too frequently repeated blocking is inappropriate, the final implementation must employ also fault avoidance techniques. The other reason for the fault avoidance application stems from the fact that only approximated fail-stop implementation is possible. Moreover, the above described configurations create only skeleton carrying common fault-tolerant techniques see e.g. (Maxion et al., 1987). In short, while auxiliary hardware components maintain supply-voltage levels, input signals filtering, and timing, the software techniques, namely time redundancy or skip-frame strategy, deal with non-critical inputs and outputs.

4 MULTIPLE LIFT CONTROL SYSTEM

The second case study deals with the multiple lift control system based on a dedicated multiprocessor architecture (for more details see Sveda, 1997). An incremental measurement device for position evaluation, and position and speed control of a lift cabin in a lift shaft can demonstrate reusability. The applied application pattern, incremental measurement, means in this case the recognition and counting of rectangular impulses that are generated by an electromagnetic or photoelectric sensor/impulse generator, which is fixed on the bottom of the lift cabin and which passes equidistant position marks while moving along the shaft. That device communicates with its environment through interfaces with impulse generator and drive controller. So, the first input, I, provides the values 0 or 1 that are altered with frequency equivalent to the cabin speed. The second input, D, provides the values "up," "down," or "idle." The output, P, provides the actual absolute position of the cabin in the shaft.

4.1 Two-level Structure for Lift Control

The next employed application pattern is the two-level structure: the higher level behaves as an event-driven component, which behavior is roughly described by the state sequence

initialization → position_indication → fault_indication

and the lower level, which behaves as a set of real-time interconnected components. The specification of the lower level can be developed by refining the higher level state "position_indication" into three communicating lower level automata: two noise-tolerant impulse detectors and one checking reversible counter.

4.2 Incremental Measurement for Position and Speed Control

The first automaton models the noise-tolerant impulse detector in the same manner as in previous case, see the following timed-state sequence:

$$(0, q_1) \overset{inp=0}{\rightarrow} \ldots \overset{inp=0}{\rightarrow} (i, q_1) \overset{inp=1}{\rightarrow} (i+1, q_2) \overset{inp=0}{\rightarrow} \ldots \overset{inp=0}{\rightarrow} (j, q_2) \ldots$$
$$\ldots \overset{inp=1}{\rightarrow} (k, q_{b/2+1}) \overset{inp=1}{\rightarrow} \ldots$$
$$\ldots \overset{inp=1}{\rightarrow} (m, q_{b-1}) \overset{inp=0}{\rightarrow} (m+1, q_b) \overset{inp=1}{\rightarrow} \ldots \overset{inp=1}{\rightarrow} (n, q_b) \overset{inp=0/IMP}{\rightarrow} (n+1, q_1)$$

i, j, k, m, n are integers representing discrete time instances in increasing order.

The information about a detected impulse is sent to the counting automaton that can also access the indication of the cabin movement direction through the input D. For the sake of fault-detection requirements, the impulse generator and the impulse transfer path are doubled. Consequently, a second, identical noise-tolerant impulse detector appears necessary. The subsequent application pattern is the checking reversible counter, which starts with the value (h + l)/2 and increments or decrements the value according to the “impulse detected” outputs

Table 1: Application patterns hierarchy.

fault management based on fail-stop behavior approximations
two-level (event-driven/real-time) structure
incremental measurement
noise-tolerant impulse detector checking reversible counter /O watchdog

from the first or second recognition automaton. Overflow or underflow of the preset values of h or l indicates an error. This detection process sends a message about a detected impulse and the current direction to the counting automaton, which maintains the actual position in the shaft. To check the counter, an I/O watchdog application pattern employs again a reversible counter that can compare the impulses from the sensor/impulse generator and the changes of the total value.

4.3 Lift Fault Management

The approach used accomplishes a consequent application pattern, fault management based on fail-stop behavior approximations, both in the form of (a) hazardous state reachability control and (b) hazardous state maintenance. In safe states, the lift cabins are fixed at any floors. The system is allowed to reach any hazardous state when all relevant processors have successfully passed the start-up checks of inputs and monitored outputs and of appropriate communication status. The hazardous state maintenance includes operational checks and consistency checking for execution processors. To comply with safety-critical conception, all critical inputs and monitored outputs are doubled and compared. When the relevant signals differ, the respective lift is either forced (with the help of a substitute drive if the shaft controller is disconnected) to reach the nearest floor and to stay blocked.

The basic safety hard core includes mechanical, emergency brakes. Again, more detailed specification should reflect not only safety but also functionality with fault-tolerance support: also blocked lift is safe but useless. Hence, the above described configurations create only skeleton carrying common fault-tolerant techniques.

5 APPLICATION PATTERNS REUSE

The two case studies presented above demonstrate the possibility to reuse effectively substantial parts of the design dealing with petrol pumping station technology for a lift control technology project. While both cases belong to embedded control systems, their application domains and their technology principles differ: volume measurement and dosage control seems not too close to position measurement and control. Evidently, the similarity is observable by employment of application patterns hierarchy, see Table 1.

The reused upper-layer application patterns presented include the automata-based descriptions of incremental measurement, two-level (event-driven/real-time) structure, and fault management stemming from fail-stop behavior approximations. The reused lower-layer application patterns are exemplified by the automata-based descriptions of noise-tolerant impulse detector, checking reversible counter, and I/O watchdog.

Clearly, while all introduced application patterns correspond to design patterns in the above-explained interpretation, the upper-layer application patterns can be related also to frameworks. Moreover, the presented collection of application patterns creates a base for a pattern language supporting reuse-oriented design process for real-time embedded systems.

6 KNOWLEDGE-BASED SUPPORT

Industrial scale reusability requires a knowledge-based support, e.g. by case-based reasoning (see Kolodner, 1993), which differs from other rather traditional methods of Artificial Intelligence relying on case history. For a new problem, the case-based

reasoning strives for a similar old solution. This old solution is chosen according to the correspondence of a new problem to some old problem that was successfully solved by this approach. Hence, previous significant cases are gathered and saved in a case library. Case-based reasoning stems from remembering a similar situation that worked in past. For software reuse, case-based reasoning utilization has been studied from several viewpoints as discussed e.g. by Henninger (1998), and by Soundarajan and Fridella (1998).

6.1 Case-based Reasoning

The case-based reasoning method contains (1) elicitation, which means collecting those cases, and (2) implementation, which represents identification of important features for the case description consisting of values of those features. A case-based reasoning system can only be as good as its case library: only successful and sensibly selected old cases should be stored in the case library. The description of a case should comprise the corresponding problem, solution of the problem, and any other information describing the context for which the solution can be reused. A feature-oriented approach is usually used for the case description.

Case library serves as the knowledge base of a case-based reasoning system. The system acquires knowledge from old cases while learning can be achieved accumulating new cases. While solving a new case, the most similar old case is retrieved from the case library. The suggested solution of the new case is generated in conformity with this retrieved old case. Search for the similar old case from the case library represents important operation of case-based reasoning paradigm.

6.2 Backing Techniques

Case-based reasoning relies on the idea that situations are mostly repeating during the life cycle of an applied system. Further, after some period, the most frequent situations can be identified and documented in the case library. So, the case library can usually cover common situations. However, it is impossible to start with case-based reasoning from the very beginning with an empty case library.

When relying on the case-based reasoning exclusively, also the opposite problem can be encountered: after some period the case library can become huge and very semi-redundant. Majority of registered cases represents clusters of very similar situations. Despite careful evaluation of cases before saving them in the case library, it is difficult to avoid this problem.

In an effort to solve these two problems, the case-base reasoning can be combined with some other paradigm to compensate these insufficiencies. Some level of rule-based support can partially cover these gaps with the help of rule-oriented knowledge; see Sveda, Babka and Freeburn (1997).

Rule-based reasoning should augment the case-based reasoning in the following situations:

- No suitable old solution can be found for a current situation in the case library and engineer hesitates about his own solution. So, rule-based module is activated. For a very restricted class of tasks, the rule-based module is capable to suggest its own solution. Once generated by this part of the framework, such a solution is then evaluated and tested more carefully. However, if the evaluation is positive, this case is later saved in the case library covering one of the gaps of the case-based module.
- Situations are similar but rarely identical. To fit closer the real situation, adaptation of the retrieved case is needed. The process of adaptation can be controlled by the rule-based paradigm, using adaptation procedures and heuristics in the form of implication. Sensibly chosen meta-rules can substantially improve the effectiveness of the system.

The problem of adaptation is quite serious when a cluster of similar cases is replaced by one representative only - to avoid a high level of redundancy of the case library. The level of similarity can be low for marginal cases of the cluster. So, adaptation is more important here.

Three main categories of rules can be found in the rule-based module:

- Several *general heuristics* can contribute to the optimal solution search of a very wide class of tasks.
- However, the dominant part of the knowledge support is based on a *domain-specific rule*.
- For a higher efficiency, *metarules* are also attached to the module. This “knowledge about knowledge” can considerably contribute to a smooth reasoning process.

While involvement of an expert is relatively low for case-based reasoning module, the rules are mainly based on expert’s knowledge. However, some pieces of knowledge can also be obtained by data mining.

6.3 Similarity Measurement of State-based Specifications

Retrieval schemes proposed in the literature can be classed based upon the technique used to index cases during the search process (Atkinson, 1998): (a) classification-based schemes, which include keyword or feature-based controlled vocabularies; (b) structural schemes, which include signature or structural characteristics matching; and (c) behavioral schemes; which seek relevant cases by comparing input and output spaces of components.

The problem to be solved arises how to measure the similarity of state-based specifications for retrieval. Incidentally, similarity measurements for relational specifications have been resolved by Jilani, et al. (2001). The primary approach to the current application includes some equivalents of abstract data type signatures, belonging to structural schemes, and keywords, belonging to classification schemes. While the first alternative means for this purpose to quantify the similarity by the topological characteristics of associated finite automata state-transition graphs, such as number and placement of loops, the second one is based on a properly selected set of keywords with subsets identifying individual patterns. The current research task of our group focuses on experiments enabling to compare those alternatives.

7 CONCLUSIONS

The original contribution of this paper consists in proposal how to represent a system's formal specification as an application pattern structure of specification fragments. Next contribution deals with the approach how to measure similarity of formal specifications for retrieval in frame of case-based reasoning support. The above-presented case studies, which demonstrate the possibility to effectively reuse concrete application pattern structures, have been excerpted from two realized design cases.

The application patterns, originally introduced as "configurations" in the design project of petrol pumping station control technology based on multiple microcontrollers (Sveda, 1996), were effectively -- but without any dedicated development support -- reused for the project of lift control technology (Sveda, 1997). The notion of application pattern appeared for the first time in (Sveda, 2000) and developed in (Sveda, 2006). By the way, the first experience of the authors with case-based reasoning support to knowledge preserving development of an industrial application was published in (Sveda, Babka and Freeburn, 1997).

ACKNOWLEDGEMENTS

The research has been supported by the Czech Ministry of Education in the frame of Research Intention MSM 0021630528: Security-Oriented Research in Information Technology, and by the Grant Agency of the Czech Republic through the grants GACR 102/05/0723: A Framework for Formal Specifications and Prototyping of Information System's Network Applications and GACR 102/05/0467: Architectures of Embedded Systems Networks.

REFERENCES

Alexander, C. (1977) A Pattern Language: Towns / Buildings / Construction, Oxford University Press.

Alur, R. and T.A. Henzinger (1992) Logics and Models of Real Time: A Survey. In: (de Bakker, J.W., et al.) Real-Time: Theory in Practice. Springer-Verlag, LNCS 600, 74-106.

Arora, A. and S.S. Kulkarni (1998) Component Based Design of Multitolerant Systems. IEEE Transactions on Software Engineering, 24(1), 63-78.

Atkinson, S. (1998) Modeling Formal Integrated Component Retrieval. Proceedings of the Fifth International Conference on Software Reuse, IEEE Computer Society, Los Alamitos, California, 337-346.

Coad, P. and E.E. Yourdon (1990) Object-Oriented Analysis, Yourdon Press, New York.

Frakes, W.B. and K. Kang (2005) Software Reuse Research: Status and Future. IEEE Transactions on Software Engineering, 31(7), 529-536.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995) Design Patterns -- Elements of Reusable Object-Oriented Software, Addison-Wesley.

Geppert, B. and F. Roessler (2001) The SDL Pattern Approach – A Reuse-driven SDL Design Methodology. Computer Networks, 35(6), Elsevier, 627-645.

Henninger, S. (1997) An Evolutionary Approach to Constructing Effective Software Reuse Repositories. Transactions on Software Engineering and Methodology, 6(2), 111-140.

Henninger, S. (1998) An Environment for Reusing Software Processes. Proceedings of the Fifth International Conference on Software Reuse, IEEE Computer Society, Los Alamitos, California, 103-112.

Holtzblatt, L.J., R.L. Piazza, H.B. Reubenstein, S.N. Roberts and D.R. Harris (1997) Design Recovery for

Distributed Systems. IEEE Transactions on Software Engineering, 23(7), 461-472.
Jacobson, L. (1992) Object-Oriented Software Engineering: A User Case-Driven Approach, ACM Press.
Jilani L.L., J. Deshamais and A. Mili (2001) Defining and Applying Measures of Distance Between Specifications. IEEE Transactions on Software Engineering, 27(8), 673-703.
Johnson, R.E. (1997) Frameworks = (Components + Patterns), Communications of the ACM, 40(10), 39-42.
Kolodner, J. (1993) Case-based Reasoning, Morgan Kaufmann, San Mateo, CA, USA.
Mili, R., Mili, A. and Mittermeir, R.T. (1997) Storing and Retrieving Software Components: A Refinement Based System. IEEE Transactions on Software Engineering, 23(7), 445-460.
Sen, A. (1997) The Role of Opportunity in the Software Reuse Process. IEEE Transactions on Software Engineering, 23(7), 418-436.
Shaw, M. and D. Garlan (1996) Software Architecture, Prentice Hall.
Soundarajan, N. and S. Fridella (1998) Inheritance: From Code Reuse to Reasoning Reuse. Proceedings of the Fifth International Conference on Software Reuse, IEEE Computer Society, Los Alamitos, California, 206-215.
Sutcliffe, A. and N. Maiden (1998) The Domain Theory for Requirements Engineering. IEEE Transactions on Software Engineering, 24(3), 174-196.
Sveda, M. (1996) Embedded System Design: A Case Study. IEEE Proc. of International Conference and Workshop ECBS'96, IEEE Computer Society, Los Alamitos, California, 260-267.
Sveda, M., O. Babka and J. Freeburn (1997) Knowledge Preserving Development: A Case Study. IEEE Proc. of International Conference and Workshop ECBS'97, Monterey, California, IEEE Computer Society, Los Alamitos, California, 347-352.
Sveda, M. (1997) An Approach to Safety-Critical Systems Design. In: (Pichler, F., Moreno-Diaz, R.) Computer Aided Systems Theory, Springer-Verlag, LNCS 1333, 34-49.
Sveda, M. (2000) Patterns for Embedded Systems Design. In: (Pichler, F., Moreno-Diaz, R., Kopacek, P.) Computer Aided Systems Theory--EUROCAST'99, Springer-Verlag, LNCS 1798, 80-89.
Sveda, M. and R. Vrba (2006) Fault Maintenance in Embedded Systems Applications. Proceedings of the Engineering of Computer-Based Systems. Proceedings of the Third International Conference on Informatics in Control, Automation and Robotics (ICINCO 2006), INSTICC, Setúbal, Portugal, 183-186.
Turner, K.J. (1997) Relating Architecture and Specification. Computer Networks and ISDN Systems, 29(4), 437-456.
van Lamsweerde, A. and L. Willemet (1998) Inferring Declarative Requirements Specifications from Operational Scenarios. IEEE Transactions on Software Engineering, 24(12), 1089-1114.
Xinyao, Y., W. Ji, Z. Chaochen and P.K. Pandya (1994) Formal Design of Hybrid Systems. In: (Langmaack, H., W.P. de Roever and J. Vytopil) Formal Techniques in Real-Time and Fault-Tolerant Systems, Springer-Verlag, LNCS 863, 738-755.
Zaremski, A.M. and J.M. Wing (1997) Specification Matching of Software Components. ACM Trans. on Software Engineering and Methodology, 6(4), 333-369.