SPECIFICATION-DRIVEN DESIGN OF EMBEDDED SYSTEMS
Design Support for Networked Embedded Software Applications

Miroslav Sveda
Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic

Radimir Vrba
Faculty of Electrical Engineering & Communication, Brno University of Technology, Brno, Czech Republic

Keywords: Embedded software, formal specification, structured design, object-oriented design.

Abstract: The paper presents an approach to formal specification, verification and prototyping of networked embedded software system applications ranging from large information systems down to small components embedded e.g. in mobile devices. Main attention focuses both on architectural and behavioral specifications of either reactive or real-time activities utilizing either structured or object-oriented approach depending on application requirements. The design approach fully respecting such requirements can eliminate not only behavioral and structural faults but also security flaws caused by design errors. Reflecting current trends in engineering software-intensive systems, this contribution discusses in more detail executable specifications and rapid prototyping for structured design, and structural specifications and verifications for object-oriented design. The paper presents Asynchronous Specification Language and Class Specification Language developed for that purpose.

1 INTRODUCTION

Current computer-based system applications are software, hardware, and communication intensive, and their functional, performance, reliability, and security requirements mandate tightly integrated information processing and physical platform behavior. Development of such complex systems necessarily stems from formal specifications and their verification and prototyping (Melhart and White, 2000). This paper discusses an approach to executable specifications and rapid prototyping for structured design, and to structural specifications and verifications for object-oriented design. The work presented in this paper focuses on a class of networked systems embedded in industrial applications and reflects current trends in engineering software-intensive systems as stated in (Broy, 2006) following the 'Verified Software Grand Challenge' initiated by Tony Hoare, see (Woodcock, 2006) and (Jackson, 2006).

The developed methods and tools cover front-end phases of design cycles, namely formal specification and rapid prototyping both of architecture and behavior of applications under design. The approach can be explained as an employment of complex reactive systems' universal development scheme, designed by Harel (2001), for the domain of industrial distributed computer-based systems. That scheme leads from a requirements capture method to full behavioral descriptions of system parts, and from there to final implementation.

2 STATE OF THE ART

Requirements on current embedded software system applications include both functional and non-functional constraints on real-time, safety and security properties. They should be formally specified and verified or, at least, properly explored before they are designed in detail and implemented (Lamport, 2002). Moreover, the specification approach should either conform or suitably complement anticipated design methods, namely structured or object-oriented techniques (Wieringa, 1998). While some applications demand to distinguish at the beginning of design structural and

Sveda M. and Vrba R. (2007).
SPECIFICATION-DRIVEN DESIGN OF EMBEDDED SYSTEMS - Design Support for Networked Embedded Software Applications.
In *Proceedings of the Second International Conference on e-Business*, pages 23-30
DOI: 10.5220/0002107100230030

behavioral specifications, later on, they request to integrate those two approaches to enable a complex viewpoint to study various application interdependencies. This paper discusses an approach to rapid prototyping for behavioral specifications, and to structural specifications and verifications for object-oriented design.

The design of well thought-out information system applications should consider namely functionality and dependability measures, see e.g. (Melhart and White, 2000) and (Hessami, 2004). Functionality means services delivery in the form and time fitting requirement specifications, where the service specification is an agreed description of the expected service. Functionality properties should be realized efficiently and cost-effectively, so reachable performance and simplicity of implementation belongs to the checked properties. Dependability is that property of a system that allows reliance to be justifiably placed on the service it delivers. Dependability measures consist of reliability, availability, security, safety and survivability, from which this project focuses on safety, which is the ability to deliver service under given conditions with no catastrophic affects, and security, which is the ability to deliver service under given conditions for a given time without unauthorized disclosure or alteration of sensitive information. Safety attributes add requirements to detect and avoid catastrophic failures. Security attributes add requirements to detect and avoid intentional faults.

Both safety and security deal directly with system's behavior that stems from a system's architecture. Therefore, structural and object oriented specifications of the system under design can contribute to the quality of its resultant implementation.

Specification is a written or graphical description (i) of what system is supposed to do, which is so called behavioral specification, or (ii) of system architecture, so called structural specification (Lamport, 2002). A formal specification asserts that a description has precise and unambiguous semantics. The language of specification should fit purposes of specification and be appropriate for a description of the system. The presented design approach employs both behavioral and structural specification styles through appropriate specification languages aiming at either structured or object-oriented developments including their rapid prototyping in frame of a design.

The approach is explained with the help of a case study derived from a real-world application reflecting current trends in application design.

3 FORMAL SPECIFICATIONS

This section discusses tools that enable to utilize behavioral and structural specifications of a class of computer-based systems that can be characterized as networked embedded systems in frame of industrial applications. The developed methods and tools, which can complement well-known and broadly available means, cover front-end phases of the related design cycles.

3.1 Formal Specification Tools

Formal specification concepts employed respect both structured and object-oriented design approach depending on the target implementation support or on the role of a tool in the development process. For structured behavioral specifications of reactive systems, process algebra CSP, temporal logic LTL and related transition systems in frame of the model checker SPIN (Holzmann, 1997) and the prover PVS (Owre et al., 1992) have been employed. Additionally for real-time systems, model checker UPPAAL (Kim et al., 1997) and related timed automata have been used. In addition to the above mentioned freely available and well-known means, the following tools have been developed in frame of the presented research: (i) Asynchronous Specification Language, ASL, with rapid prototyping technique for structured design (Sveda and Vrba, 2001), (Sveda and Vrba, 2003); and (ii) Class Specification Language, CSL, for object-oriented design (Rysavy and Sveda, 2003), (Rysavy, 2005). The next subsections introduce main concepts of those tools.

3.2 Behavioral Specifications

The Asynchronous Specification Language (ASL) employs distributed sequential processes with message passing. The real-time operational semantics of the language stems from the event-count model of local time, which represents a concept of physical timing stemming from some periodic physical oscillation whose frequency fits measurements of the duration of local process actions. Timing semantics can be derived from logical time, which is a partial ordering of events in the system, and from a physical generator of periodic events, which implements a real-time clock. An event-count, E, counts the number of a specific type of events that have occurred during execution. Each event occurrence invokes the implicit operation ADVANCE(E): E := E + 1. The explicitly callable operation AWAIT(E, s) suspends the calling process until the value of E is at least s. The call AWAIT(E, s) can reset the current value of E, enabling relative counting. An event-count monitors either a prescribed type of asynchronous external events or periodic internal events that an internal timer circuit implements as local-time clock ticks. The following primitives relate to process specification, timing, communication, and control.

```
process_name(is: list_of_s_inputs; os: list_of_s_outputs;
        ic: list_of_m_inputs; oc:list_of_m_outputs):
... endprocess;
wait(_, timeout); wait(event, _); wait(event, timeout, test);
send(message, destination);
loop ... [... when <cond> action ... exit;]* ... endloop;
```

Each of asynchronous processes can be equipped by its individually timed local clock, can receive messages through input buffer, and can send messages to other, directly or indirectly addressable processes. Process header contains in parentheses lists labeled by is, os, ic, and oc that act as the interface with the process' environment. The language distinguishes between signal inputs or outputs, which denote communication events signaling their occurrence, and message inputs or outputs as typed asynchronous channels between processes. Those signals and messages provide inter-process synchronization and communication, whose operations are driven by the statements wait(_, timeout), wait(event, _), wait(event, timeout, test), and send(message, destination).

The primitive wait(_, timeout) suspends a process for the interval defined by the value timeout. Operational semantics can be obtained through the event-count abstraction introduced above: in this case, an event is every tick of the local clock, so the related operation is AWAIT(local_ticks, timeout_value). For the primitive wait(event, _), which suspends a process until the specified event (external signal or message) appears, the model operation is AWAIT(event_type, 1). The semantics of the combined statement wait(event, timeout, test) requires two event-counts: the first anticipates the specified event and the second, with a lower priority, monitors the local clock. The reason of process activation can be checked through the value of the logical variable test: when the value is true, the event occurred within the interval timeout.

The primitive send(message, destination) implements asynchronous communication with non-blocking semantics. To respect different local clocks, a special clocking that is common for the source and the destination controls the information transfer. However, the nodes communicate asynchronously by message passing through an input buffer at the destination. The input of a message induces the event for the related operation AWAIT(message,1). If any synchronization is required, it must be described explicitly using wait statements.

The control structure primitives loop ... endloop delimit an indefinite cycle, which is exited upon a true result of testing the condition following the primitive when. Consequently, the statements, which occur between the action and exit primitives and which follow the endloop primitive, are executed. This structured statement enables to extend the language with additional control structures by simple macro-like text replacements such as

```
if <cond> then <s1> else <s2> fi;
        ~ loop when <cond> action <s1> exit;
           <s2> when true exit; endloop;
timeloop(timeinterval) ... endloop;
        ~ loop ... wait(_, interval); endloop;
```

Actually, the control structure timeloop(timeinterval) ... endloop specifies an isochronous loop, which is periodically initiated whenever the timeinterval expires and which can be exited like the indefinite cycle. The operation AWAIT(local_ticks, timeinterval_value) defines the exact semantics of timing these initiations.

The associated rapid prototyping, which makes ASL specifications executable, arises from attribute grammar and Prolog deployment. Any Prolog interpreter can drive expansion of an ASL specification into the related executable code. This expansion is based on an attribute grammar specifying both syntax and static semantics by a definite clause grammar and Prolog rules. It provides a simple language translator prototype, which tackles the ASL as the input language, and a target executable language as the output language.

The resulted prototyping technique uses interconnected node prototype boards with microprocessors equipped with a simple real-time operating system kernel. While the timing and communication primitives are mapped onto relevant real-time executive services and communication services of the operating system kernel, the rest of ASL specification is prototyped by the executable code generated with the help of the Prolog translator prototype introduced above.

3.3 Structural Specifications

The Class Specification Language, CSL, relates to language constructs for description of definitions and assumptions on specification in the form of logical formulas. The specifications and assumptions provide for the proof system that verifies whether a specification is valid under the given assumptions. The specification means consists, from the structural point of view, of the language of predicate logic and the language of object calculus. Their synthesis provides a base language with expressive power of higher-order logics. In terms of logic, this language contains standard predicate logical symbols, i.e. quantifiers and propositional connectives, and constants as objects defined by terms of object calculus. Those terms are interpretable in the object calculus; concurrently, the language allows quantification over the set of constants. The Gentzen deduction system may be used as a formal proof system for this language.

A specification consists of a set of classes that forms a model of the specified system. The reasoning about specification involves the use of the above-mentioned sequent proof system. Because the specification language in this case is object-based, the classes are represented as special objects. A class is a basic structure of specifications that covers implementations of objects and logical judgments on properties of objects.

The logic represents a higher-order theory based on typed object calculus. It consists of a small set of primitive syntactic forms. An object is defined as a collection of attributes. The following two operations only can operate on objects: (i) attribute selection $a.l$ that results in the term obtained as an evaluation of the attribute body, and (ii) attribute update that has the form $a.l \leftarrow b$. The letters a, b represent terms of the language and l is a label. Computational semantics of the calculus for both operations arise from the rules for reduction relation. A select operation provides reduction to a term that arises from the body of a selected attribute in which all occurrences of the bind variable are replaced by the object supplying the selected attribute. The result of update operation defined by reduction relation provides a new object identical with the target object up to the update attribute, which body is that of the updating term.

The logic includes a type theory constraining the set of well-formed terms. Typing rules of the calculus permit subtyping while providing for special treatment with bool type. Although the language does not contain functions, they can be easily inferred as simple objects. A function abstraction $\lambda(x : A).a$ of type $A \rightarrow B$ denotes the structure $[x = \zeta(s : T)s.x, val = \zeta(s : T)\ a\ \{x \leftarrow s\}]$ provided that $T \equiv [x : A, val : B]$. Then a function application MN is directly given as $(M{:}x \leftarrow N).val$. Instances of bool type represent, from the computational viewpoint, conditional expressions and serve as constants of propositional types considering theirs logical meanings. The propositional connectives are introduced as a set of constants with usual meanings. Moreover, quantifiers and predicates are introduced inside the object language. To model classes, predicates allow writing constraints on types that delimit sets of objects satisfying intended conditions. Subtype creation uses operator + for denoting that a new type is obtained by adding new attributes to the old type.

The logic calculus of objects provides a suitable formal environment for specifying and logical reasoning with properties of objects. However, writing specifications directly in this calculus is tedious. More practical notation, the Class Specification Language (CSL), enables to write compact specifications, but preserves possibility to transform any specification straightforwardly to the object calculus whenever required for reasoning.

A class is defined by specifying all of its visible properties. The term property means in this case a field that represents the state of an object, or an observer that serves for the read-only access to an object, or a modifier whose execution can change the state of an object. A field declaration includes the field name and class. Specification of a field may be refined using invariant statement.

```
Field fieldName : fieldClass
Inv fieldInv = formula
```

Modifier and observer methods include definitions consisting of method's name, arguments, and a pair of constraints. Declarations differ for modifiers that disable a user to specify a result of the method. Due to modifiers, declarations always evaluate to the object reflecting performed changes. Constraints may involve variables referencing actual objects and variables denoting specified arguments of the method.

```
observer methodName(, arg : argClass, …) : retClass
pre methodPre = formulaPre
post methodPost = formulaPost
```

The language CSL enables to define a new class by application of simple inheritance. An inherited class automatically receives all fields and methods of its parent class. To handle inheritance properly, a schema for definition of invariants and conditions of inherited fields and methods is needed. Considering that class B inherits class A, then the specification of those classes has to preserve inheritance constraints assuming each inherited field and method in the schema as follows:

$$\mathrm{fieldInv}^B \Rightarrow \mathrm{fieldInv}^A$$
$$\mathrm{methodPre}^A \Rightarrow$$
$$\mathrm{methodPre}^B \wedge \mathrm{methodPost}^B \Rightarrow \mathrm{methodPost}^A$$

The definition of inheritance constraints in this manner enables method overriding. The precondition of the overridden method relaxes constraints of method execution, contrary to post-conditions that involve additional constraints.

The logic calculus defines directly certain common classes as they depend on particular aspects of the calculus. The class of Bool is defined simply as a predicate on propositional type. The logic evaluates all possible instances of this class to T and F objects declared previously as abbreviations. The class of Natural numbers exploits recursive object type. It consists of three attributes; two of them serve as links to predecessor and successor objects. The iszero attribute marks the numeral zero. Usual notations 0, 1, 2 ... explicitly denote related numerals.

4 CASE STUDY

This section demonstrates the above-introduced concepts and tools applied to development of a gas-pipes pressure analyzer consisting of pressure sensors interconnected by Internet (Sveda and Vrba, 2006). The application is based on the IEEE 1451 family of standards, which is introduced in subsection 4.1. Subsection 4.2 explains a subset of the application functions selected for formal specifications in the rest of this section. To provide examples of specification styles using developed tools, the next two subsections present selected facets of the pressure analyzer specification. While subsection 4.3 demonstrates structured specifications using ASL, subsection 4.4 exemplifies object-oriented specifications using CSL.

4.1 IEEE 1451.1 Architecture

The IEEE 1451 consists of the family of standards for a networked smart transducer interface that include namely (i) a smart transducer software architecture, 1451.1 (IEEE 1451.1 Standard for a Smart Transducer Interface for Sensors and Actuators -- Network Capable Application Processor Information Model, IEEE, New York, April 2000), targeting software-based, network independent, transducer applications, and (ii) a standard digital interface and communication protocol, 1451.2, for accessing the transducer or a group of transducers via a microprocessor modeled by the 1451.1. The next three standards extend the original hard-wired parallel interface 1451.2 to serial multidrop 1451.3, mixed-mode (i.e. both digital and analog) 1451.4, and wireless 1451.5 interfaces.

The 1451.1 software architecture provides three models of the transducer device environment: (i) an object model of a network capable application processor (NCAP), which is the object-oriented embodiment of a smart networked device; (ii) a data model, which specifies information encoding rules for transmitting information across both local and remote object interfaces; and (iii) network communication model, which supports client/server and publishers/subscribers paradigms for communicating information among NCAPs. The standard defines a network and transducer hardware neutral environment in which a concrete sensor/actuator application can be developed.

The object model definition encompasses a set of object classes, attributes, methods, and behaviors that specify a transducer and a network environment to which it may connect. This model uses block and base classes offering patterns for one Physical Block, one or more Transducer Blocks, Function Blocks, and Network Blocks. Each block class may include specific base classes from the model. The base classes include Parameters, Actions, Events, and Files, and provide component classes.

All classes in the model have an abstract or root class from which they are derived. This abstract class includes several attributes and methods that are common to all classes in the model and provide a definition facility for instantiation and deletion of concrete classes including attributes. Block classes form the major blocks of functionality that can be plugged into an abstract card-cage to create various types of devices. One Physical Block is mandatory as it defines the card-cage and abstracts the hardware and software resources that are used by the device. All other block and base classes can be referenced from the Physical Block.

The Transducer Block abstracts all the capabilities of each transducer that is physically connected to the NCAP I/O system. During the device configuration phase, the description is read from hardware device what kind of sensors and actuators are connected to the system. The Transducer Block includes an I/O device driver style interface for communication with the hardware. The I/O interface includes methods for reading and writing to the transducer from the application-based Function Block using a standardized interface. The I/O device driver provides both plug-and-play capability and hot-swap feature for transducers.

The Function Block provides a skeletal area in which to place application-specific code. The interface does not specify any restrictions on how an application is developed. In addition to the variable State, which all block classes maintain, the Function Block contains several lists of parameters that are typically used to access network-visible data or to make internal data available remotely. The Network Block abstracts all access to a network employing network-neutral, object-based programming interface supporting both client-server and publisher-subscriber paradigms for configuration and data distribution.

4.2 Application

The case study, based on a real-world application, which was introduced in more detail but from distinct perspectives by (Sveda and Vrba, 2003) and (Sveda and Vrba, 2006), is used in this paper to demonstrate basic features of deployment of the specification languages ASL and CSL discussed in subsections 3.2 and 3.3.

The application architecture comprises several groups of wireless pressure and temperature sensors with safety valve controllers as base stations connected to wired intranets that dedicated clients can access effectively through Internet. The web server supports each sensor group by an active web page with Java applets that, after downloading, provide clients with transparent and efficient access to pressure and temperature measurement services through controllers. Controllers provide clients not only with secure access to measurement services over systems of gas pipes, but also communicate to each other and cooperate so that the system can resolve safety and security-critical situations by shutting off some of the valves.

Each wireless sensor group is supported by its controller providing Internet-based clients with secure and efficient access to application-related services over the associated part of gas pipes. In this case, clients communicate to controllers using a messaging protocol based on client-server and subscriber-publisher patterns employing 1451.1 Network Block functions. A typical configuration includes a set of sensors generating pressure and temperature values for the related controller that computes profiles and checks limits for users of those or derived values. When a limit is reached, the safety procedure closes valves in charge depending on safety service specifications.

In the transducer's 1451.1 object model, basic Network Block functions initialize and cover communication between a client and the transducer. The client-server communication style, which in this application covers configurations of transducers, is provided by two basic Network Block functions: *execute* and *perform*. The standard defines a unique ID for every function and data item of each class. If the client requests to call any of the functions on server side, it uses command execute with the following parameters: ID of requested function, enumerated arguments, and requested variables. On server side, this request is decoded and used by the function perform. That function evaluates the requested function with the given arguments and, in addition, it returns the resulting values to the client. Those data are delivered by requested variables in execute arguments.

4.3 Behavioral Specifications by ASL

The following example demonstrates the ASL specification of a client accessing the transducer. This specification includes in form of comments the most important references to sections of the IEEE 1451.1 standard.

```
process CLIENT(oc: data_out, request; ic: response):
const number_of_channels = 10;
const interval_of_reading = 100;
const ServerDispatchAddress = NCAP;
const port_timeout = 200;
type buffer = array[1..number_of_channels] of Float32;
{IEEE 1451.1-6.1.1}
    var data_out:buffer;
    var i: integer;
    var request, response: ArgumentArray;
{IEEE 1451.1-6.2.14}
    var server_inputsarguments: ArgumentArray;
    var server_outputsarguments: ArgumentArray;
    var success:boolean;
    var execute_mode: UInteger8;
{IEEE 1451.1-6.1.1}
    var sever_operation_id: UInteger16;
    var data:Float32;
    i = 1;
timeloop(interval_of_reading)
{IEEE 1451.1-14.2.1}
Encode_inputsarguments(server_inputsarguments);
    execute_mode = EM_RETURN_VALUE;
    server_operation_id = READ_VALUE;
{IEEE 1451.1-8.2.3.5 - Ethernet}
```

```
    MarshalArguments(server_operation_id,
                    server_inputsarguments, request );
    send(request, ServerDispatchAddress);
    wait(response,port_timeout,success);
    if success and execute_mode = EM_RETURN_VALUE
{IEEE 1451.1-8.2.3.5 - Ethernet}
        then DemarshalArguments(response,
                    server_output_arguments);
Decode_outputsarguments(data,server_outputsarguments);
        else data = 0; fi;
    data_out[i] = data;
    i = i + 1; when i > number_of_channels action exit;
endloop;
endprocess;
```

The above exemplified behavioral specification was prototyped using the technique mentioned at the end of subsection 3.2 that resulted in executable model heavily utilized for experiments during not only early design phases, but also later on when investigating variants for reuse and re-design of the application.

4.4 Structural Specifications by CSL

The following example demonstrates the CSL specification of the NCAP Block class in frame of the class hierarchy of Block objects. The Block abstract class provides the root for the class hierarchy of all Block objects. The BlockMajorState type is enumeration of possible block states. The state blUninitalize is reserved for local activities related to creating that Block object and performing any related local preparations. The object in the state blInactive is able to configure its network properties, initialize itself, and diagnose and maintain the BlockMajorState. The working state blActive is reached after all initialization and start-up procedures and represents the state in which the object remains for the time of its normal activity.

```
BlockMajorState :: {blActive, blInactive, blUninitialized}
IEEE1451Block :: [GetBlockMajorState : BlockMajorState,
GoActive : IEEE1451Block, GoInactive : IEEE1451Block,
Reset : IEEE1451Block, Initialize : IEEE1451Block,
Owns : IEEE1451Entity → bool] + IEEE1451Entity
```

A behavior of the object may implicitly change the state of this object usually in response to the environment stimulus, and a set of defined operations drives the object to change its state explicitly. The meanings of those operations are defined by sets of constraints. The behavior of the object influenced by those operations is specified in form of IEEE1451BlockBehavior invariant.

```
INV IEEE1451BlockBehavior(x : IEEE1451Block) ≡
x.GetBlockMajorState ↔ blInactive
⊃ x.GoActive.GetBlockMajorState ↔ blActive
∧ x.GetBlockMajorState ↔ blActive
⊃ x.GoInactive.GetBlockMajorState ↔ blInactive
∧ x.Reset.GetBlockMajorState ↔ blUninitialized
∧ ¬x.GetBlockMajorState ↔ blActive
⊃ x.Initialize.GetBlockMajorState ↔ blInactive
```

The NCAP Block class provides resources and operations within an NCAP process to support Block, Service, and Component management.

```
NCAPBlockState :: {nblInitialized, blUninitialized, nbErro}
IEEE14 51Block :: [GetNCAPBlockState : NCAPBlockState,
RegisterObject : IEEE1451Block,
DeregisterObject: IEEE1451Block,
registers : IEEE1451Entity → bool] + IEEE1451Block
```

The state space derived from Block object is divided into more specialized substates that reflect purpose of NCAP Block objects. The state of object, which is stored in GetNCAPBlockState item, may obtain values of NCAPBlockState type.

For the exemplified static structural specification some of its properties were proved using the PVS system, see a next paper currently under preparation.

5 CONCLUSIONS

This paper discusses in more detail executable specifications and rapid prototyping for structured design, and structural specifications and verifications for object-oriented design. Main attention focuses both on architectural and behavioral specifications of either reactive or real-time activities utilizing either structured or object-oriented approach depending on application requirements. The paper presents Asynchronous Specification Language and Class Specification Language developed for that purpose. A case study respecting real world constraints demonstrates utilization of the developed approach.

The presented paper introduces some relevant facets of a currently launched project – for complementary information see (Sveda and Vrba, 2006) and (Sveda, et al., 2005) -- that aims at front-end parts of networked, distributed system application designs. The project targets creation of a formal specification, verification and prototyping framework for network applications ranging from large information systems down to small components embedded e.g. in mobile devices. The design approach, fully respecting dependability requirements of real-world applications, can eliminate not only behavioral and structural faults but also security flaws caused by design errors.

Reflecting current trends in engineering software-intensive systems, main attention focuses both on architectural and behavioral formal specifications of either reactive or real-time system actions, utilizing either structured or object-oriented approach depending on application requirements. Formal specification tools considered include temporal logics, real-time logics, object calculi, process algebras and transition systems. The implementation and integration phases of the project provide pilot versions of techniques and tools for conceptual design, for behavioral and structural specifications, and for rapid prototyping. Moreover, formal verification support will include dedicated tools both for model checking and for proving

ACKNOWLEDGEMENTS

The research has been supported by the Czech Ministry of Education in the frame of Research Intentions MSM 0021630528: *Security-Oriented Research in Information Technology* and MSM 0021630503 *MIKROSYN: New Trends in Microelectronic Systems and Nanotechnologies*; and in part 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*. The current contribution, which stems from the previous paper (Sveda and Vrba, 2005), delivers not only an overview of the work done after the preceding publication, but also a brief information about the contents of a research intention, *Security-Oriented Research in Information Technology,* which focuses also on relationships between safety and security.

The authors appreciate contributions of their colleagues from the Department of Information Systems and the Department of Microelectronics - namely Ondrej Rysavy, Roman Trchalik, Pavel Ocenasek, Petr Matousek, Jarek Rab, Rudolf Cejka and Frantisek Scuglik -- to this work.

REFERENCES

Broy, M., 2006. The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems. *IEEE Computer,* Vol.39, No.10, pp.72-80.

Harel, D., 2001. From Play-In Scenarios to Code: An Achievable Dream. *IEEE Computer*, Vol.34, No.1, pp.53-60.

Hessami, A.G., 2004. A Systems Framework for Safety and Security: The Holistic Paradigm. *Systems Engineering*, Vol.7, No.2, pp.99-112.

Holzmann, G.J., 1997. The Model Checker Spin. *IEEE Transactions on Software Engineering*, Vol.23, No.5, pp.279-295.

Jackson, M., 2006. What Can We Expect from Program Verification? *IEEE Computer,* Vol.39, No.10, pp.65-71.

Kim, G., L. et al., 1997. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, Vol.1, No.1-2, pp.134-152.

Lamport, L., 2002. *Specifying Systems*, Addison-Wesley, Boston, USA.

Melhart, B. and S. White, 2000. Issues in Defining, Analyzing, Refining, and Specifying System Dependability Requirements. *Proceedings of the IEEE Conference and Workshop ECBS'2000*, IEEE Computer Society Press, Edinburgh, Scotland, pp.334-340.

Owre, J.M., et al., 1992. PVS: A Prototype Verification System. *Automated Deduction,* (D. Kapur, Ed.), Lecture Notes in Artificial Intelligence, Vol.607, pp.748-752, Springer, New York, USA.

Rysavy, O. and M. Sveda, 2003. A Minimal Formal Language for Object-Oriented Specifications. *Proceedings of the IEEE TC-ECBS and IFIP WG10.1 Joint Workshop on Formal Specifications of Computer-Based Systems.* University of Stirling, UK, pp.35-40.

Rysavy, O., 2005. *Specifying and Reasoning in the Calculus of Objects*, PhD Dissertation, Brno University of Technology, Brno, Czech Republic.

Sveda, M. and R. Vrba, 2001. Executable Specifications for Distributed Embedded Systems. *IEEE Computer*, Vol.34, No.1, pp.138-140.

Sveda, M. and R. Vrba, 2003. An Integrated Framework for Internet-Based Applications of Smart Sensors. *IEEE Sensors Journal,* Vol.3, No. 5, pp.579-586.

Sveda, M. and R. Vrba, 2005. An Approach to the Design of Networked Embedded Systems. *Proceedings of the 16th IFAC WORLD CONGRESS*, Praha, pp.2223-2228.

Sveda, M., et al., 2005. Introduction to Industrial Sensor Networking. A book chapter in: Ilyas, M., and I. Mahgoub, (Eds.). Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems, CRC Press LLC, Boca Raton, FL, USA, pp.10.1-10.24.

Sveda, M. and R. Vrba, 2006. Internet-Based Embedded System Architectures -- End-User Development Support for Embedded System Applications. *Proceedings of the International Joint Conference on e-Business and Telecommunications (ICETE 2006, ICE-B)*, INSTICC and IEEE, Setúbal, Portugal, 2006, pp.63-68.

Wieringa, R., 1998. A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. *ACM Computing Surveys*, Vol.30, No.4, pp.459-527.

Woodcock, J., 2006. First Steps in the Verified Software Grand Challenge. *IEEE Computer,* Vol.39, No.10, pp.57-64.