Extended Design-by-Contract Approach to Specification and Conformance Testing of Distributed Software*

Victor Kuliamin Nickolay Pakoulin Alexander Petrenko

Institute for System Programming (ISPRAS), Russian Academy of Sciences,

Moscow, Russia

ABSTRACT

Increasing effort in development of high quality distributed systems requires ground methodological base. Design by Contract approach looks very promising as a candidate since it helps to obtain component-wise specification and design, to separate concerns between developers, and makes development of high quality complex systems a manageable process. Unfortunately, in its classic form it can hardly be applied to distributed network applications because of lack of adequate means to describe asynchronous events. We extend Design by Contract with capabilities to describe callbacks and asynchronous communication between components and apply it to specify distributed software and to develop conformance test suites in automated manner. Specifications are developed in extensions of programming languages that makes them clear for industrial developers and decreases test construction effort. Practical results of numerous successful applications of the method are described.

Keywords: Design by Contract, asynchronous events specification, distributed software specification, model based testing, automated test construction, specification extension of programming language, automatic test oracle generation.

1. INTRODUCTION

Development technologies for high quality complex software is one of the main concerns in software engineering community nowadays. Current shift to software construction on the base of distributed components providing various services makes such technologies urgent. In its turn such a technology needs in an adequate methodological base, which should not compromise its scalability and practical orientation. Design by Contract (DbC) [1] approach is one of the most suitable methods for development of qualitative complex software system. The key points of this approach can be stated as follows.

- Software is considered as a system of components separated from each other and communicating with each other only through the specified interfaces.
- An interface of the component is a set of its operations. Each operation is described with its *precondition* and *postcondition*. Precondition states the obligations of environment – before a call of the operation a caller should ensure that the precondition holds. Postcondition states counter-obligations of the component. If the precondition holds just before an operation call, the component ensures that the postcondition holds just after that call. Preconditions and postconditions are usually formulated in terms of operation parameters and internal state of the component. Common parts of pre- and postconditions of all the component's operations can be stated as separate *invariants* representing integrity constraints on the component's state.

DbC provides effective separation of concerns between different components, separation of development activities between developers, and allows significant flexibility in components' implementation. It also ensures broad reuse of the components developed. The approach applies rather uniformly on different levels, since large subsystems can be considered as components with their own contracts. Contracts of a subsystem and its constituting components can ensure correctness of its decomposition. So, step by step, we can build rather complex systems on the same methodological base and obtain high quality results.

Unfortunately, the classic DbC can hardly be applied to develop complex distributed software. The following issues hinder this.

- Components in distributed software communicate in different ways. For example, they can provide asynchronous events and messages or can use callbacks to react to something. Classic DbC simply ignores asynchronous communications. Callbacks representing parameters of functional type also cannot be described in the DbC framework.
- DbC has no effective means to reason on several asynchronous communications performed in parallel.
- DbC leads developers to clear understanding of the system functionality and can help in debugging. But it lacks sound and full-scale quality control of the development results including automated test construction, test adequacy measurement, and regression testing.

This article proposes possible solution of these problems. We provide an extension of DbC approach that adds just several new entities to original method, but makes it applicable for specification of complex distributed applications and frameworks. In addition we demonstrate how this extended approach can be used to construct conformance tests based on DbC specifications in automated manner.

The methodological base of the suggested approach is formulated in the next section. Then we depict applications of the extended DbC to specification and testing complex distributed software. The fourth section briefly reviews approaches to model based test

* This work is partially supported by RFBR grant 04-07-90386, by grant of Russian Science Support Foundation, and by Program 4 of Mathematics Branch of RAS.

construction for distributed systems. The conclusion provides directions of possible future development.

2. EXTENDING DESIGN BY CONTRACT

The main point of the presented approach is the same as of the original DbC – software is considered as a system of components communicating with each other through the specified interfaces. Interfaces consist of operations and *event kinds* described by their pre- and postconditions.

Description of events

Observation of usual distributed system shows that component communications in it can be rather complex. They include the following.

- Simple calls of operations. Such a call is considered from DbC positions as an atomic event having pre-state and post-state. Relation between them is described in postcondition.
- *Synchronization calls.* They do not return the control until some set of events occurs. Synchronization calls cannot be considered as atomic since system state can be changed several times between call and return of control. So, we consider such a call itself and the corresponding return of control as different events with their own pre- and post-states and their own pre- and postconditions. Examples of synchronization calls are operations of synchronization primitives – locking a mutex, waiting a barrier or a condition.
- *Inverse calls.* Inverse call occurs when a component processes some external call and needs help from another component or environment to do its work. For example, callback parameter given to a component can be called to perform the operation. Another example of inverse call is call performed by template method [2] defined in a base class to one of methods, which should be overridden in inherited classes. Inverse calls performed during work of some operation also make impossible to consider this operation call as an atomic event in terms of its pre- and post-states.
- *Asynchronous events.* If the system includes several active threads or processes, it may react on some call not only with return of control to the caller, but also producing a set of asynchronous events in different channels of communication. They differ from previously mentioned event kinds by their occurrence in different threads of control. Example of such an event is e-mail report on unavailable recipient address of a previously sent e-mail.

All those event kinds are described by pre- and postconditions. Precondition imposes restrictions on system state, which an event can occur in, and corresponding event data (parameters). Postcondition describes relation between event pre-state, post-state, and event data (parameters and returned results). If this relation is broken, the system considered to behave incorrectly. Postcondition may also impose restrictions on the events that can occur after this one.

For example, template method (see [2] for description of this design pattern) call can require that a set of inverse call should occur before the control will be returned. Moreover, the returned result of the method may depend on the results of those inverse calls.

Concurrency semantics

Correctness of a collection of events occurring in concurrent manner is defined by so called *interleaving* or *sequential semantics.* It says that the set of concurrent events is performed in a correct manner if they can be performed in correct manner in some sequence. More precisely, a set $\{e_i, i \in [1..n]\}$ of calls of operations or occurrences of events performed on or provided by a component is considered to satisfy their contracts in a state s_1 of the component if there exists such a sequence $\{s_j, j \in [1..n+1]\}$ of component's states starting from s_1 and the corresponding ordering $\{i_j\}$ of those calls and events that each call or event e_{i_j} occurs in the state s_j, moves the component to the state s_{j+1}, and the contract of the corresponding operation or event holds for pre-state s_j, post-state s_{j+1}, provided values of operation parameters, and the data returned by the operation or by the event.

For example, if we have an operation printing "Hello, world!" on a printer and an event printing "Bye!", any result "Hello, world!Bye!" or "Bye!Hello, world!" is considered as correct result of concurrent call of the operation and occurrence of the event, but the result "Hello,Bye! world!" is invalid.

The proposed extension of DbC approach is not complex and has the uniform base – consideration of events provided to or by the system and the corresponding pre- and post-states of the system. This approach can be used successfully to describe distributed systems of practical significance, to obtain valuable results from more formal consideration of system properties, and to test the components of the system and a system as a whole. See the next section for examples of such applications.

Use of programming language extension

One more peculiarity of our approach is use of extensions of programming languages to specify software properties. This fact becomes important if one needs to apply a method or a tool based on formal notation in industrial practice. Widely used programming languages are commonly recognized means of communication between developers. So, specifications written in their extensions are comprehensible for average software engineers. Specialized formal notations often require advanced mathematical education, do not contain adequate counterparts for widely used programming concepts (such as pointers), and therefore are rarely used in practice.

We propose uniform extension of C, Java, and C# languages [3] based on the main concepts of our approach – pre- and postconditions, invariants, and events of different kinds – and some additional syntactic sugar useful in to describe events and to work with both pre-states and post-states of the same objects. The main elements of the extension are as follows.

- Some operations in class (or functions in C) can have `specification` modifier saying that they contain contracts of the corresponding operations of the system under consideration. Such an operation can have *access constraints* describing the set of objects the operation has access to and the kind of this access, *precondition* represented as additional block returning Boolean value, *postcondition* represented also as additional block marked with Boolean result. Postcondition has access to objects in the states preceding the call of the operation and the same objects in the states after the call (*pre-* and *post-states*). To refer a pre-value of a variable in a postcondition we can use *pre operator*.

- Operations marked with `reaction` modifier represent asynchronous events provided by the system. Such a reaction can also have access constraints, pre- and postcondition.
- Operations marked with `inverse` modifier represents inverse operations. They also can have access constraints, pre- and postconditions.
- Return of control for synchronization call is also considered as a separate event and represented as a special reaction with its own pre- and postcondition. They may refer the parameters of the corresponding call.
- Invariants are represented as special methods or functions marked with `invariant` keyword and returning Boolean result. The result says whether the invariant holds or not.
- *Branches of functionality.* Often a postcondition of a function describes several possible modes of operation corresponding to significantly different behaviors. For example, the function receiving a message from a mailbox can return different codes depending on whether it finds a message, cannot find any message during the specified timeout, or detects a broken mailbox. To mark such situations as different we use special `branch` operators. During testing we can measure the number of different branches covered and use it as a test quality measure. More detailed test coverage measure can be obtained if we consider DNFs of all branching conditions (except for cycles) in postcondition and correlate different disjuncts in these DNFs to different testing situations. Of course, all this work is performed automatically.

3. PRACTICAL APPLICATIONS

This section presents some results of practical application of the approach described above in two areas – clarification and formalization of standards and automated construction of conformance test suites for distributed software.

Formalizations of standards

This subsection concerns with two case studies in standard formalization related with distributed applications. The first example is standard clarification and conformance test suite development for ISO/IEC 13818-11, a standard on Intellectual Property Management and Protection in MPEG-2 domain. The second one is a part of specification-based test suite development for an implementation of IPv6 protocol suite – the next generation of the Internet protocol.

Formalization of IPMP. A standard for MPEG-2 Intellectual Property Management and Protection (IPMP-2) [4] is an attempt to create a flexible and interoperable solution for Digital Rights Management in MPEG-2 distribution chain from content provider to user. For the sake of readability we will refer to ISO/IEC 13818-11 [4] as "IPMP-2 specification" below in this section.

The original architecture for protecting MPEG-2 movies (called Conditional Access, CA) proved to be non-interoperable. Playing content from a particular producers required purchasing CA solution from certain vendor, and CA solutions from different vendors were incompatible.

IPMP-2 specification regulates IPMP operations on the side of a user. IPMP Device includes *a Terminal* and a number of *IPMP Tools*. IPMP Tools perform all operations needed to prepare data for playback such as user authorization, content deciphering, watermarks processing, etc. IPMP Tools are software or hardware modules that are plugged to specific *control points* in the MPEG-2 processing pipe. Terminal intercepts multimedia data and passes them to the corresponding instances of IPMP Tools for processing. Results of processing (e.g. deciphering) are returned to the Terminal for further processing. IPMP Tools interact with each other and the Terminal by means of message exchange. IPMP-2 specification provides a number of messages for several purposes such as authentication or notification.

Content providers add *control information* and *protection signaling* to their content. This information includes indications on which tools to use, how to initialize the tools, etc. The IPMP Device parses content and tries to acquire IPMP tools from the network if needed. Then the device instantiates tools with given parameters and starts playback.

IPMP-2 specification uses Syntax Definition Language [5] for defining syntax of messages and IPMP-related data in content. Still the semantics of messages and data is defined in plain text without any formal notation.

IPMP-2 operations semantics was presented in data integrity constraints and constraints on prerequisites and results of operations.

The work on IPMP-2 formalization was conducted for Audio Video coding Standard Working Group of China (AVS). Length of the studied specification is about 30 pages. The project resulted in two submissions [6, 7] to AVS DRM group and a prototype of conformance test suite for processing IPMP Control Information in bit streams.

Other results of the project include the following.

- We identified significant inconsistencies in syntax specification of IPMP data in bit streams. For example, it allowed inserting up to 65 536 bytes of data (16-bit length field) in a descriptor which length is limited to 256 bytes.
- Under-specifications were found in the semantics of the Mutual Authentication – a security protocol for establishing trust between two tool instances. We demonstrated that current specification of Mutual Authentication does not ensure interoperability between implementations from different vendors.
- Correctness criteria of data in IPMP-2 specification are poorly defined. Discussion with IPMP developers showed that there are many implicit rules of what is correct and what is not. For example, IPMP-2 specification defines IPMP Tool List structure as a container for IPMP Control Info classes, but it is intended to carry information about tools only. We put this implicit constraint into explicit form: each element of IPMP Tool List is of IPMP Tool Info type. The list of constraints educed during the formalization for IPMP Control Information classes is presented in [7]. The constraints are not written in formal notation yet.

Taking into account numerous misspellings in code parts of IPMP-2 specifications the exact number of fixes we proposed is hard to count. The standard study showed that IPMP-2 specification consists of several loosely related pieces sometimes contradicting to each other. Certain requirements are under-specified or contain errors.

Contract formalization of IPv6. IPv6 is a group

of protocols located at the Network Layer of the OSI Reference Model [8]. IPv6 provides services to protocols of transport layer, such as UDP and TCP.

IPv6 features a much greater address space compared to IPv4, the current version of the Internet Protocol. Large address space enables true point-to-point connectivity within global scope. Besides extended address space IPv6 includes improved routing architecture and integrated suite of protocols for autoconfiguration and discovering the state of the communication.

Implementations of IPv6 provide three classes of interfaces: procedural (API), binary (ABI), and message-based.

Procedural interfaces include generic sockets API and several IPv6-specific extensions. Binary interfaces are non-standard, implementation-specific ways to access the kernel part of an implementation. Examples of such interfaces are request code for `ioctl` call on Unixens or control code for `DeviceIOControl` routine in Windows accompanied with memory layouts for inputs and outputs. Message-based interface is an abstraction for sending and receiving IPv6 datagrams to or from Data Link Layer.

IPv6 messages and part of procedural interface are standardized by Internet Engineering Task Force in IPv6-related *Requests for Comments* (RFCs). Binary interface and some part of procedural interface are not standardized and are implementation-specific. Since the component functionality should be understood unambiguously to apply Design by Contract fruitfully, it is natural to limit formalization to the scope of messages and standard API of IPv6.

The scope of our projects on IPv6 conformance testing was formalization and testing of the following basic features of IPv6.

- Sending datagrams from the transport layer to the network and processing of incoming IPv6 packets.
- Neighbor Discovery on hosts. Neighbor Discovery is a suite of service protocols for identifying router and neighbor nodes attached to a link and detecting their reachability status.
- Multicast Listener Discovery on hosts. Multicast Listener Discovery is a protocol to obtain information about multicast listeners attached to a link.
- UDP over IPv6.

The contract formalization is based upon requirements presented in regulating RFCs. We studied the requirements of many RFCs, most notably [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. More than 400 separate functional requirements were elicited. RFCs define protocol semantics in plain text mostly. Syntax is defined in tabular format with textual definition of bit-wise message layout.

We identified a number of inconsistencies and underspecifications in IPv6 regulating documents. For example, the specification of IPv6 protocol [9] enumerates a number of cases that should be considered as errors in incoming fragmented IPv6 packets, and a number of cases that are not errors. Unfortunately this enumeration misses several important cases, such as fragments overlap.

Despite the defects found we can state that IPv6 regulating requirements are well-defined as a rule. They are detailed enough to ensure interoperability between implementations and at the same time leave much flexibility to implementers.

The formal model of the IPv6 subset described above is about 8500 lines of code in the specification extension of C language [20]. The model was used to build a test suite that was applied to several open and commercial implementations of IPv6 protocol stack (see the next subsection).

Automated Conformance Test Construction

Historically first application of the extended DbC approach was automated test development. The specifications written in the described manner can be used to construct conformance test suite with the help of UniTesK technology. Here we provide a short introduction into UniTesK. The interested reader can find more details on it in [21, 22, 23, 3, 24].

The main principles of UniTesK test development may be summarized as follows.

- UniTesK is intended to develop conformance test suites automatically on the base of the specifications to be tested. The main approach to testing is black-box, testing adequacy is measured as the achieved during testing *coverage of specifications* according to some criterion. *Test oracles* – programs automatically checking the correctness of the behavior of the system under test – are generated automatically from contracts specified.
- User manually writes *test scenarios* providing very brief descriptions of the automaton model of the component under test, including structure of its state and the list of operations to be called in an arbitrary state. Each operation is supplemented with some procedure (manually written or taken from a library) to generate values of its parameters. Its main goal is to provide a large set of different arrays of operation parameters values. The development of test scenario can be facilitated with the help of the template, taking several choices of the user as its input and generating all the other parts of the scenario. The main goal of a scenario is to ensure high level of test coverage in certain specification-based coverage metric.
  Test scenarios can be used to process possible nondeterminism of specifications effectively. To do this, one can define scenario states on the base of classes of states described in specifications. *State factorization* allows creation of rather efficient and compact tests for complex subsystems. Details of the technique can be found in [25].
- Similar template technique is used to create test adapters providing binding between specifications and implementation under test.
- The UniTesK tool used translates specifications, adapters, and scenarios into the base language of the tool (C, Java, or C#) and executes the resulting test. During test execution the sequence of test calls is generated on-the-fly using the data presented in the scenario and the actual behavior of the system under test. The generation algorithm tries to call each operation in each state achieved, but do not perform calls that add nothing to already achieved test coverage in term of specifications (`branch` statements are an example of construct that can be used to define coverage of specifications).

UniTesK technology was used to develop conformance tests in the following projects.

- Development of regression test suite for switch operating system kernel for Nortel Networks. Results of this project was

already presented in [21, 24], see also [3]. Total size of the system under test is about 250 KLOC, the size of resulting suite of specifications and scenarios is about 140 KLOC. To our knowledge, this is the largest piece of formally specified software and the largest system tested in such a formal way. The total effort of the project is about 10 man-years, total duration – about one year and a half. 372 test scenarios were developed for about 500 procedures of the operating system kernel, 304 of those scenarios tested single procedure, 68 – a group of interoperating procedures. With different parameters of execution the resulting test suite can perform from dozens of thousands to several millions of test cases. Several hundreds of defects were detected in critical telecommunication software already working in the field for about 10 years. Several of bugs found could cause cold restart of the system.

- Development of test suite and testing several IPv6 implementations. The detailed results those projects can be found in [20] and [23]. The projects also demonstrated the approach's capability to clarify ambiguous parts of informal telecommunication standards. The first project was conducted to test open IPv6 implementation of Microsoft Research. The results showed that the test suite provides good error detection – it found more errors that the counterparts we could compare with at that time (Microsoft Research organized an international contest in testing of this IPv6 implementation). 4 serious bugs were found in the system under test, one of them leads to operating system crash and can be used to shut down any remote node in IPv6 network. The second project is conducted in the Russian telecommunication software development company Octet by its own developers trained in our technology. It also resulted in several serious bugs found in another proprietary implementation of IPv6.
- Test development for a part of bank CRM system based on J2EE technology. This project demonstrated that UniTesK technology and tools can be applied to test distributed software constructed with the help of modern component-based technologies for multi-tier applications development. The duration of the project was about 2 months, and its results include about a dozen of bugs detected. The details of this and several other projects can be found on [3].

4. STATE OF THE ART

This section gives brief review of similar approaches taking into consideration only those ones that provide formal descriptions of distributed systems and support test development automation for conformance testing, so a lot of interesting solutions stay out of scope of this section. More detailed and systematic review of various model-based testing techniques can be found in [27].

The most widely used practical approach to conformance test suite construction for distributed applications is based on informally determined *test purposes* and test cases manually developed on their base. In comparison with methods based on some formal description of application functions, it lacks strict and measurable definition of testing adequacy based on functional requirements and forces test developers to provide correct results only on the base of their understanding of the functions under test. Both disadvantages can be overcome by diligence and cross-checking, but not for large-scale systems.

The usual approach to formal specification and further testing of distributed software are based on some kind of transition systems – it may be labeled transition systems, input-output automata, and systems of communicating (extended) finite automata. Theoretical background for most part of those works was laid by J. Tretmans [28]. He proposed a formal definition of conformance relation between specifications and system under test and a method for test case generation based not only on possible inputs and outputs of the system under test, but also using special *quiescent* states where the system can not produce any output without some input from the environment. A series of tools based on those ideas were developed in the academic community, the most prominent from them are TGV [29] and TorX [30]. These tools can take formal descriptions in SDL, LOTOS, or Estelle as input. In 2001-2003 years developers of most of the tools mentioned took part in the AGEDIS project [31] introduced uniform testing tool architecture and UML-based statecharts as standard input for such tools.

Transition systems used for automatic test generation proved to be very useful instrument, but they have the following disadvantages.

- State explosion problem. When one tries to model a real system on a detailed level, he obtains an unmanageable model with huge numbers of states and transitions. This is a demonstration of more serious drawback – transition systems can hardly be decomposed to separate different concerns and functions, they usually require considering the system as a whole to get valuable results. Design by Contract looks much more promising in this view since it provides a method to consider components of a complex system separately. In UniTesK state explosion problem can be overcome with the help of state factorization technique.
- Inefficient processing of nondeterminism. It is rather hard to introduce nondeterminism natural to distributed applications in transition systems and keep them useful. Most of them become inoperative after such a procedure. So, some special actions are always needed to introduce necessary nondeterminism in such a model.

  Nondeterminism of a distributed system includes two aspects. First, the exact reaction of the system cannot be predicted on high levels of abstraction. One can impose only some restriction on the results. To reflect this in a transition system the corresponding collection of transitions should be defined, thus making state explosion more probable. Contract based approach incorporates this kind of nondeterminism naturally by stating the corresponding predicate in postcondition.

  Second source of nondeterminism is concurrency of distributed systems, which produces a lot of possible combinations of observable events. So, it became very hard to check whether the system behaves correctly in complex cases. The influence of this aspect is so much that now one can find very few tests targeted at checking system behavior in response to two or more concurrent actions. Extended DbC approach proposes sequential semantics, which makes possible automatic checking of system correctness. Combination with factorization technique used in UniTesK, although not reducing this kind of nondeterminism to negligible level, makes it much more manageable.

5. CONCLUSION

The paper proposes an extension of Design by Contract approach for distributed software. The key points of the new approach are the following.

- Components' interaction in a distributed system is considered as a series of events occurring in different states of the system. Each event is identified by its event type and data (parameters of operation call or data returned by an event).
- Event's semantics is described in the contract of the corresponding event kind. Each event contract consists of precondition and postcondition. The first puts restrictions on the pre-state and data of event occurrence (if precondition does not hold, occurrence of such an event is incorrect). The second provides a relation between pre-state and post-state of event occurrence and event data.
- Correctness of concurrent events is defined according to sequential semantics – a set of such events is considered to be correct if and only if it can be ordered into a sequence conforming to all the contracts involved.
- Contracts are represented in extensions of widely use programming languages. That makes them comprehensible and useful for average software engineer.

The extended DbC approach has been used in formalization and clean-up of several telecom standards. Also it used in practice-oriented UniTesK test development technology to construct conformance test suites in automated manner. Since UniTesK tools were successfully used in several industrial projects, the authors consider the proposed extended DbC approach quite mature to be used in practical development of distributed applications.

Although the approach and the test development technology based on it seems to be quite general, there are a lot of technical issues concerning their use in testing applications through GUI or Web interfaces, or through interfaces including timing events. Those issues should be resolved in future development.

REFERENCES

[1] Bertrand Meyer. Applying 'Design by Contract'. IEEE Computer, vol. 25, No. 10, October 1992, pp. 40–51.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

[3] http://www.unitesk.com

[4] ISO/IEC 13818-11:2004. Information technology – Generic coding of moving pictures and associated audio information – Part 11: IPMP on MPEG-2 systems. 2003.

[5] ISO/IEC 14496-1:2001, Information technology – Coding of audio-visual objects – Part 1: Systems.

[6] MPEG-2 IPMP Conformance Test Suite Development. AVS M1263: 2004/6.

[7] Enhancing IPMP-2 for Conformance Testing. AVS M1487: 2004/12.

[8] ISO/IEC 10731:1994. Information technology – Open Systems Interconnection – Basic Reference Model – Conventions for the definition of OSI services. 1994.

[9] RFC 2460. S. Deering, R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. December 1998.

[10] RFC 2461. T. Narten, E. Nordmark, W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). December 1998.

[11] RFC 2462. S. Thomson, T. Narten. IPv6 Stateless Address Autoconfiguration. December 1998.

[12] RFC 2463. A. Conta, S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. December 1998.

[13] RFC 2464. M. Crawford. Transmission of IPv6 Packets over Ethernet Networks. December 1998.

[14] RFC 3513. R. Hinden, S. Deering. Internet Protocol Version 6 (IPv6) Addressing Architecture. April 2003.

[15] RFC 2373. R. Hinden, S. Deering. IP Version 6 Addressing Architecture. July 1998.

[16] RFC 2292. W. Stevens, M. Thomas. Advanced Sockets API for IPv6. February 1998.

[17] RFC 2553. R. Gilligan, S. Thomson, J. Bound, W. Stevens. Basic Socket Interface Extensions for IPv6. March 1999.

[18] RFC 2675. D. Borman, S. Deering, R. Hinden. IPv6 Jumbograms. August 1999.

[19] RFC 2710. S. Deering, W. Fenner, B. Haberman. Multicast Listener Discovery (MLD) for IPv6. October 1999.

[20] http://www.unitesk.com/products/ctesk/

[21] V. Kuliamin, A. Petrenko, I. Bourdonov, and A. Kossatchev. UniTesK Test Suite Architecture. Proc. of FME 2002, LNCS 2391, pp. 77–88, Springer-Verlag, 2002.

[22] V. Kuliamin, A. Petrenko, A. Kossatchev, and I. Bourdonov. UniTesK: Model Based Testing in Industrial Practice. Proc. of 1-st Europpean Conference on Model-Driven Software Engineering, December 2003.

[23] V. Kuliamin, A. Petrenko, N. Pakoulin, I. Bourdonov, and A. Kossatchev. Integration of Functional and Timed Testing of Real-time and Concurrent Systems. Proc. of PSI 2003, LNCS 2890, pp. 450–461, Springer-Verlag, 2003.

[24] I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. FM'99: Formal Methods. LNCS 1708, Springer-Verlag, 1999, pp. 608–621.

[25] I. B. Burdonov, A. S. Kossatchev, and V. V. Kulyamin. Application of finite automatons for program testing. Programming and Computer Software, 26(2):61–73, 2000.

[26] http://www.ispras.ru/groups/rv/rv.html

[27] V. Kuliamin. Multi-paradigm Models as Source for Automated Test Construction. Proc. of Workshop on Model Based Testing, Barcelona, Spain, March 2004. Also available in Electronic Notes in Theoretical Computer Science 111:137–160, 2005, Elseveir.

[28] J. Tretmans. A Formal Approach to Conformance Testing. Proc. of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems, Pau, France, September 1993, pp. 257–276.

[29] J. -C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on the fly verification techniques for the generation of test suites. Proc. of CAV'96, Rutgers University, New Brunswick, New Jersey, USA, July-August 1996.

[30] J. Tretmans, A. Belinfante. Automatic testing with formal methods. Proc. of EuroSTAR'99, Barcelona, Spain, November 8-12, 1999.

[31] http://www.agedis.de/