Practical Approach to Specification and Conformance Testing of Distributed Network Applications*

Victor V. Kuliamin, Nickolay V. Pakoulin, and Alexander K. Petrenko

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Kommunisticheskaya, 25, Moscow, Russia
{kuliamin, npak, petrenko}@ispras.ru
http://www.ispras.ru/groups/rv/rv.html

Abstract. Standardization of infrastructure and services in distributed applications and frameworks requires ground methodological base. Design by Contract approach looks very promising as a candidate. It helps to obtain component-wise design, to separate concerns between developers accurately, 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 nondeterministic asynchronous events. We extend Design by Contract with capabilities to describe callbacks and asynchronous communication between components. The resulting method was used to specify distributed applications and to develop conformance test suites for them in automated manner. Specifications are developed in an extension of C language that makes them clear and useful for industrial developers and decreases greatly test construction effort. Practical results of numerous successful applications of the method are described. More information on the applications of the method can be found at the site of RedVerst group of ISP RAS [1].

Keywords: Design by Contract, asynchronous events specification, distributed system specification, formalization of standards, model based testing, conformance testing, automated test construction, specification extension of programming language, test oracle generation, UniTesK.

1 Introduction

Standardization of infrastructure and base services of distributed systems builds up its strength as the important component of the movement to availability and dependability of such systems. This process needs adequate support from methods and technologies of software construction. One of the promising approaches to development of high-quality complex software systems is Design by Contract (DbC) [2]. The key points of this approach can be stated as follows.

* 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.

M. Malek, E. Nette, and N. Suri (Eds.): ISAS 2005, LNCS 3694, pp. 68–83, 2005.

- 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, which semantics is described with *preconditions* and *postconditions.* Precondition of an operation states the obligations of an environment – before the call of this operation a caller should ensure that the precondition holds. Postcondition states counter-obligations of the component. If the precondition holds just before the call of the operation, the component ensures that the postcondition holds just after the 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.

Design by Contract proposes a powerful and well-scalable software development method. It possesses the following advantages.

- Clear component boundaries and obligations make possible effective separation of concerns between different components, separation of development activities between their developers, and significant flexibility in their implementation.
- The approach ensures broad reuse. As long as we need some functionality stated as a postcondition, we can use any component providing this or more strict postcondition, if we in turn ensure the corresponding precondition. As long as developer can ensure some postcondition providing that the precondition holds, he or she may change the implementation of component without risk of introducing errors in the system.
- The approach applies rather uniformly to components of different scale. Subsystems consisting of many components can be also considered as components with their own contracts. With the help of contracts of a subsystem and constituent components we can ensure correctness of subsystem's decomposition, and so, step by step, can build rather complex systems on the same methodological base. The quality of the result can be predicted due to rigor of the approach combined with the simplicity of its application.

All this sounds great. Even more great it can be for modern service-oriented architectures, which are based on separate components providing services for each other. But Design by Contract in its classic form given in [2] can hardly be applied for modern complex software systems. We can formulate the following causes of this situation.

- Complex networking software uses many different kinds of communication activities between its components. For example, callbacks are rather common in distributed frameworks. Another widely used kind of communication between components of such systems is asynchronous events and messages. Consider these issues in more details.

Callback represents a parameter of functional type, constraints on which can be described only if we consider the properties of all functions that can be passed in this callback. So, we need to impose additional contract on callback, although it is only a parameter of some operation, not an operation itself. This kind of contracts and its use in system development is not concerned by the classic Design by Contract approach. Any time a callback parameter is used developers have to consider constraints on the corresponding operation outside of DbC framework or treat them rather informally.

DbC also has no special means to describe asynchronous communications, which is very important in modern software. Moreover, in DbC framework we can hardly find any means to reason about correctness of multiple asynchronous communications performed in parallel. This is really serious drawback of the approach, making it inapplicable to many modern systems.

– DbC approach was originally targeted for software design, and usually after coming to rather clear understanding of the system design designers and developers cannot get any more benefits from the contracts. So, the contracts, which require a lot of work to develop, become useless and are not supported after some phase of the project to minimize the total effort (sometimes they are also used for debugging). We think that to make contracts actually useful they need additional means to provide sound and full-scale quality control of the results of development performed on their base including automated test construction, test adequacy measurement, regression testing, and certification. The original approach says nothing about measurement of component's quality based on its contract – it provides only insights on possible usage of contracts to check runtime behavior of the components or to test them in a random fashion.

In this article we present possible solution of both problems. We provide an extension of DbC approach that adds just several new entities to original framework, but makes it applicable for specification of complex distributed applications and frameworks. In addition we present UniTesK test development technology, which used to construct conformance tests based on DbC specifications in automated manner.

In the next section the methodological base of the suggested approach is presented. Then we consider several practical applications of the extended Design by Contract to complex distributed systems, including both specification of system properties, formalization of the corresponding standards, and automated development of conformance test suites based on the stated specification. The fourth section presents a brief review of similar approaches to specification and test construction for distributed software. The last section of the article concludes the discussion and provides directions of possible future development.

2 Extending Design by Contract Approach

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 described by their pre- and postcondition. The differences begin when we deal with contract development for communication means of special kinds – callbacks and asynchronous events.

Callbacks. Callbacks are considered as parts of *inverse interface* – a kind of interface, which is used for calls from the system under consideration to its environment (cp. with usual direct interface used for calls from the environment to the system). So, a component implements some interface (its direct interface) and requires from the environment to support some inverse interface.

Operations in inverse interface are considered as ordinary operations and described by their pre- and postconditions. But when we define the behavior of an ordinary operation, which may make some calls to inverse interfaces (for example, it obtains callback as a parameter and its functionality requires to call this callback in certain situations), we should describe the constraints on these calls concerning their parameters and results.

To provide such a description we use *model trace* – each of components implementing inverse interfaces considered as storing a list of calls of its inverse operations. Each of those calls can be represented as a record with called operation identifier, values of its parameters, and value of the call result as fields. So, in postcondition of an operation using callback we can state that this callback was called with certain parameters. We also can state that the result of its call was used in a certain way to produce the result of the operation call.

This extension of DbC approach, although a minor one, provides powerful means to check systems interoperability or test whether the component can be used inside a framework. We should provide the system or the framework developed with description of contracts of both provided and *required* interfaces. To check that two systems can operate together we need to check that each one obeys the restrictions imposed by the other in preconditions of ordinary operations and postconditions of inverse operations. To check that a component can operate inside a framework we should test whether it ensures preconditions of operations it calls in the framework and postconditions of its own callbacks used by the framework.

Asynchronous events. More serious changes in usual DbC concepts are required to introduce asynchronous communications. Operations, whether they are performed synchronous or asynchronous calls, can be considered just in the same way. But asynchronous events are another kind of entities. We represent them as a special kind of operations without parameters, but having ordinary pre- and postcondition.

Precondition of an event describes situations when this event is valid. If the precondition does not hold, any occurrence of the event of this kind is incorrect. Postcondition of an event describes restrictions on data provided by the event. When precondition holds (so, events of this kind are possible) postcondition says whether this event provides correct data or not. Asynchronous messages can be also described in the same manner.

To define the correctness of a collection of events and calls occurring in the concurrent manner we use so called *interleaving* or *sequential semantics.* This semantics implies that the set of concurrent calls and 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 result 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.

Although the proposed extension of DbC approach is not complex, it 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 some methodology or tool based on formal notation in industrial practice. Widely used programming languages are commonly recognized means of communication between developers and 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 [23] based on the main concepts of our approach – pre- and postconditions, invariants, asynchronous events, and inverse interfaces – and some additional syntactic sugar useful in postconditions, when one needs 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 some global functions in C) can have `specification` modifier saying that they contain contracts of the corresponding operations in 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 (whether an object can be only read by the operation, only written, or both), *precondition* represented as additional block returning Boolean value, *postcondition* represented also as additional block also returning Boolean value. 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*.

In addition, specification operations may have `branch` constructs marking different behavior constraints and so defining specification-based coverage criteria for further testing.

- Operations marked with `reaction` modifier represent asynchronous reactions provided by the system. Such a reaction can also have access constraints, pre- and postcondition. But it has no functionality branches, since the behavior of the system in this situation is not determined by external input.
- Operations marked with `inverse` modifier represents inverse operations. They also can have access constraints, pre- and postcondition.
- 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.

The code example 2 in Appendix demonstrates some elements of specification extension of C. It presents specification of a component implementing banking account. The component may be implemented as a web-service, or EJB, or plain class – this does not matter for the description of its functionality.

An account has two operations and can produce events on change of balance. Each event stores the difference between the new value of the balance and the old one. The first operation, `deposit()`, is used to deposit money on the account. The second one, `withdraw()`, is used to withdraw money from the account. Both operations can give rise to an event on account change storing the actually deposited sum or withdrawn sum as a negative number, but the results of several operations can be summed by one such event with the total change of the balance.

Negative value of the balance means that a credit is given to the account owner. The credit is limited by fixed maximum possible credit value, which is state in the invariant. Postconditions of operations and event define their impact on the state of the account. `bank` variable stores map of account identifiers into account structures.

3 Practical Applications of the Method

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.

3.1 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) [3] 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 [3] 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 Conditional access solution from certain vendors, 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 [4] 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.

The formalization of semantics of IPMP-2 operations has the following facets.

- Constraints on data integrity.
- 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 [5,6] 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 [6]. 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 that sometimes contradict 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 [7]. 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 [8,9,10,11,12,13,14,15,16,17,18]. 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 under-specifications in IPv6 regulating documents. For example, the specification of IPv6 protocol [8] 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 [19]. 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).

3.2 Automated Conformance Test Construction

The 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 [20,21,22,23,24].

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 should manually write *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 to generate values of its parameters. This procedure can be written manually or taken from a library; 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 provide a powerful feature – they can be used to process possible nondeterminism of specifications very effectively. To do this, one can define scenario states on the base of classes of states described in specifications. This technique called *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 [20,24], see also [23]. 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 [19] and [22]. 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 [23].

The diagram of process including standard formalization and conformance test suite development on the base of the approach presented is shown on Figure 1.

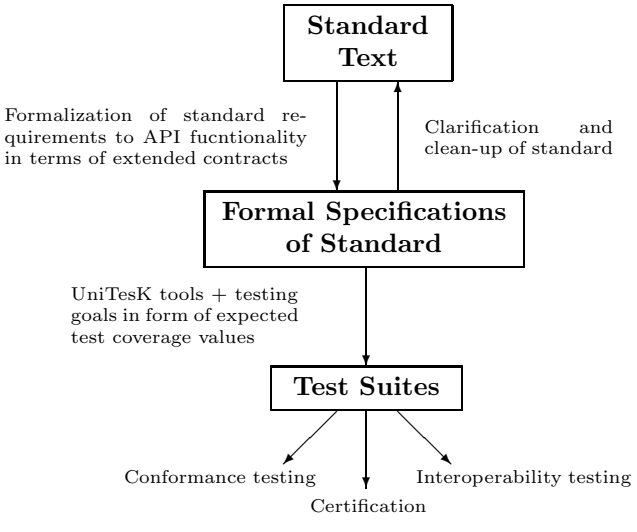

Fig. 1. Standard formalization and conformance test suite construction

4 Short Review of Similar Approaches

Here we present rather brief review of similar approaches taking into consideration only those that provide possibility to describe distributed systems formally 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 [26].

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 [27,28]. 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 on special *quiescent* states where the system could 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]. Some of those tools can take formal descriptions in such languages as SDL, LOTOS, or Estelle as input. In 2001-2003 years those tools were integrated into common environment developed in the AGEDIS project [31]. It includes 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. Contract based approach incorporates it naturally by stating the corresponding predicates in postcondition. Combination with factorization technique used in UniTesK, although not reducing concurrency-related nondeterminism to negligible level, makes it much more manageable.

5 Conclusion

The paper proposes an extension of Design by Contract approach for distributed network applications. The main extensions are constructs for specification of component-environment interaction through inverse interfaces and asynchronous events. Correctness of concurrent events is checked according to sequential semantics – a set of events is considered to be correct if and only if it can be ordered into a sequence conforming to all the contracts involved.

The extended DbC approach is used in practice-oriented UniTesK test development technology to construct conformance test suites in automated manner. UniTesK tools uses specifications in extension of programming languages (C, Java, and C# are supported now) to make them accessible and useful for ordinary industrial developers without background in formal methods. 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. Since UniTesK tools were already successfully used in several industrial projects, the authors consider the proposed approach quite mature to be used in practical development of standards, distributed applications, and corresponding test suites.

The focus point of the approach presented is integrated process of standard formalization and conformance test suite development for it. This provides the following advantages.

- Standards are intended to state not only the common syntax of interfaces, but the common understanding of the functionality of the services described. Formalization removes a lot of ambiguities and misunderstandings, makes this functionality clearly stated, and so prevents a lot of potential problems with interoperability, sustainability, and dependability of future applications based on this standard.
- Formally stated functionality opens the door to automated conformance test suite construction, which decrease the effort to produce conforming applications and also make them more qualitative. In addition it gives a natural measure of testing adequacy in terms of requirements – one can precisely say now what is tested and what is not, to what degree some application conforms the standard and to what degree it breaks it.

Standards and development of infrastructure for distributed network applications attract more and more attention now. Neglect of the modern specification and automated conformance testing techniques has negative influence on both the quality of approved standards and the dependability of the systems developed on their base. Maybe, the same causes inhibit advancement of component-based development as a whole and growth of independent software vendors in particular. At the same time, the main restrictions of possible development are imposed not by the lack of adequate methods and tools, but by the lack of engineering staff having corresponding skills and experience in their application in practice. Our experience shows that this problem can be solved successfully.

Appendix

```
specification typedef struct account_model {
  int balance; int change; bool event;
} AccountModel = {};

invariant typedef AccountModel Account;

invariant int MaxCredit = 3;

invariant (MaxCredit) { return MaxCredit >= 0; }

invariant (Account * acc) { return acc->balance + acc->change >= -MaxCredit; }

typedef Integer AccountID;

Map * bank; // A map from Account ID to account

specification void deposit(AccountID *id, int sum) {
  Account * account = get_Map(bank, id);
  pre {
    return (sum > 0) && (account != NULL)
        && (account->balance + account->change < INT_MAX - sum)
        && (account->change < INT_MAX - sum);
  }
  post {
    return (account->balance == (account->balance))
        && (account->change == (account->change) + sum)
        && (account->event == true);
  }
}

specification void withdraw(AccountID *id, int sum) {
  Account * account = get_Map(bank, id);
  pre { return (sum > 0) && (account != NULL); }
  post {
    if(account->balance + account->change < sum - MaxCredit) {
      return (account->balance == (account->balance))
          && (account->change == (account->change));
    } else {
      return (account->balance == (account->balance))
          && (account->change == (account->change) - sum)
          && (account->event == true);
    }
  }
}

specification typedef struct account_notification {
  AccountID * id; int change;
} AccountNotification;

reaction AccountNotification * update() {
  Map * bank_saved = clone(bank); int i;
  pre {
    for (i = 0; i < size_Map(bank); i++) {
      if(((Account*)get_Map(bank, key_Map(bank, i)))->event) return true;
    }
    return false;
  }
  post {
    Account * account_saved = get_Map(bank_saved, update->id);
    Account * account = get_Map(bank, update->id);

    return (account_saved != NULL) && (account != NULL)
        && (update->change == account_saved->change)
        && (account->balance == account_saved->balance + account_saved->change)
        && (account->change == 0) && (account->event == false);
  }
}
```

Fig. 2. Example of specifications in C extension

References

1. http://www.ispras.ru/groups/rv/rv.html
2. Bertrand Meyer. Object-Oriented Software Construction, Second Edition. Prentice Hall, 1997.
3. ISO/IEC 13818-11:2004. Information technology – Generic coding of moving pictures and associated audio information – Part 11: IPMP on MPEG-2 systems. 2003.
4. ISO/IEC 14496-1:2001, Information technology – Coding of audio-visual objects – Part 1: Systems.
5. MPEG-2 IPMP Conformance Test Suite Development. AVS M1263: 2004/6.
6. Enhancing IPMP-2 for Conformance Testing. AVS M1487: 2004/12.
7. ISO/IEC 10731:1994. Information technology – Open Systems Interconnection – Basic Reference Model – Conventions for the definition of OSI services. 1994.
8. RFC 2460. S. Deering, R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. December 1998.
9. RFC 2461. T. Narten, E. Nordmark, W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). December 1998.
10. RFC 2462. S. Thomson, T. Narten. IPv6 Stateless Address Autoconfiguration. December 1998.
11. RFC 2463. A. Conta, S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. December 1998.
12. RFC 2464. M. Crawford. Transmission of IPv6 Packets over Ethernet Networks. December 1998.
13. RFC 3513. R. Hinden, S. Deering. Internet Protocol Version 6 (IPv6) Addressing Architecture. April 2003.
14. RFC 2373. R. Hinden, S. Deering. IP Version 6 Addressing Architecture. July 1998.
15. RFC 2292. W. Stevens, M. Thomas. Advanced Sockets API for IPv6. February 1998.
16. RFC 2553. R. Gilligan, S. Thomson, J. Bound, W. Stevens. Basic Socket Interface Extensions for IPv6. March 1999.
17. RFC 2675. D. Borman, S. Deering, R. Hinden. IPv6 Jumbograms. August 1999.
18. RFC 2710. S. Deering, W. Fenner, B. Haberman. Multicast Listener Discovery (MLD) for IPv6. October 1999.
19. http://www.unitesk.com/products/ctesk/
20. 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.
21. V. Kuliamin, A. Petrenko, A. Kossatchev, and I. Bourdonov. UniTesK: Model Based Testing in Industrial Practice. In proceedings of 1-st Europpean Conference on Model-Driven Software Engineering, December 2003.
22. 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.
23. http://www.unitesk.com
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. 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.
27. J. Tretmans. A Formal Approach to Conformance Testing. Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems, Pau, France, September 1993, pp. 257–276.
28. J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. Software – Concepts and Tools, 17(3):103–120, 1996.
29. J. -C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on the fly verification techniques for the generation of test suites. Proccedings of CAV'96, Conference on Computer Aided Verification, Rutgers University, New Brunswick, New Jersey, USA, July-August 1996.
30. J. Tretmans, A. Belinfante. Automatic testing with formal methods. In EuroSTAR'99: 7-th European Int. Conference on Software Testing, Analysis and Review, Barcelona, Spain, November 8-12, 1999. EuroStar Conferences, Galway, Ireland. Also: Technical Report TRCTIT-17, Centre for Telematics and Information Technology, University of Twente, The Netherlands.
31. http://www.agedis.de/