UDC 004.423.4+004.415.5

# The formalism for semantics specification of software libraries

*V. Itsykson (Peter the Great St. Petersburg Polytechnic University)*

The paper is dedicated to the specification of the structure and the behavior of software libraries. It describes the existing problems of libraries specifications. A brief overview of the research field concerned with formalizing the specification of libraries and library functions is presented. The requirements imposed on the formalism designed are established; the formalism based on these requirements allows specifying all the properties of the libraries needed for automating several classes of problems: detection of defects in the software, migration of applications into a new environment, generation of software documentation. The conclusion defines potential directions for further research.

*Keywords:* formal specification, software library, behavioral description, software defect.

## 1. Introduction

Software libraries have become the de facto standard for implementing the component-oriented approach in which the software maker encapsulates specific functionality as a set of functions, data types and an application user interface. Modern libraries are extremely complex objects whose functionality is often considerably more sophisticated than that of the applications using them.

The key difference between libraries and standard applications is the manner in which they are used. Applications are used by users who follow instructions, operating manuals and built-in help systems, and have no need for formal specifications describing the applications. Libraries, on the other hand, are mainly used by other programmers who, in order to integrate the functionality of applications and libraries, need to clearly understand how a library works, how it can be used, how it affects the application, which changes are introduced to it from version to version, etc.

How does the library developer typically specify the library? One or several of the following methods are commonly used:

- headers with comments;
- verbal description of the library interface;
- verbal description of the behavior of individual functions;

- verbal description of some allowed sequences of function calls;
- examples provided by the developer.

However, none of these methods solves the problems of the formal specification of the library semantics. The library semantics consists of two components: the *semantics of individual functions* and of the allowed ways of joint use of library functions. The semantics of individual functions is determined by the function call conditions, the obtained results, the side effects, and the impact on the environment. Typically, the semantics of functions is described informally in the form of text descriptions. The *allowed ways* of joint use of library functions are at best described by the authors informally in the documentation accompanying the library,

In other words, the software engineering industry is currently lacking a set of tools for formalized description of the semantics of software libraries.

Since there is no formal specification for the libraries, it is, at present, impossible to satisfactorily solve several classes of problems:

- automatic verification of whether an application is correctly using a library. Here the term ‘correctly’ implies that the application accesses the library with satisfy a protocol specified by the designer[1]
- detection of programming errors in multi-file projects using third-party libraries when the source code is unavailable
- analysis of the compatibility between the applications and the new version of the library
- porting applications into a new library environment

Thus, the goal of this paper is to develop a formalism allowing to rigorously describe all the necessary aspects of libraries.

## 2. State of the Art

The specification of libraries and services has been long studied; a sufficient number of publications offer different approaches to describing the specifications. The first studies were related primarily to providing interoperability, with the main goal of the specifications created in designing the self-contained description of the interfaces of libraries and services that could be then used in different programming languages and operating systems. Examples of such specifications include the IDL language [1], as well as many of expansions, such as MIDL [2], and OMG IDL [3]. The main limitation of these languages for library specification is in the detailed API description

[1] Currently, this problem is typically solved dynamically at runtime by analyzing the return codes of library functions, or by exception handling.

without focusing on valid options for using the libraries. This means that the emphasis is on describing the signatures of functions and data types, while not enough attention is paid to the semantics specification of the entire library.

One of the first studies in the field of component interface specification is the work by Allen and Garlan [4], in which the authors reduce the problem of the interaction between the components of a software system to the specification of interaction protocols similar to computer networking protocols. The theory on communicating sequential processes (CSP), developed by Hoare [5], was taken as the basis for the formalism, and then altered in an appropriate manner. The introduction of special elements, such as ports, connectors and roles, into the formalism allowed separately specifying various aspects of the potential interaction between the components. Using the formalism can partially solve the problem of component compatibility with the help of the FDR model checker [6].

Alfaro and Henzinger describe in [7] their own version of the formalism for describing the interacting components, called the interface automata. The study uses an optimistic definition of component compatibility, based on the use of the environment model. The authors propose formal methods for verifying the optimistic compatibility of two interface automata.

Some studies are focused on the mechanisms of automated construction of specifications of interfaces and libraries based on analyzing the existing software. For example, the authors of [8] propose an approach to library specification inference based on static predicate mining. The authors use data flow and control flow analyzes for collecting predicates characterizing the interface functions. Another approach is described in [9], where the authors offer using dynamic output of library specifications based on unit testing. For this purpose, library functions, interfaces, data types and transactions are defined in terms of the Datalog formalism. Valid sequences of function calls are specified through special predicates. Specifications inference is based on analyzing and generalizing the results of random unit testing of the library's functions.

One of the most interesting approaches to describing library APIs and their application rules is the SLAM approach proposed by Microsoft Research for driver verification. SLAM uses the SLIC language [10] for specifying the libraries and the rules of interaction between the programs and the API. The SLIC specification is used for the instrumentation of the program and/or the library for further dynamic or static compliance control. The lack of semantic descriptions for the library back-ends prevents SLAM from being used for automated migration.

In his paper 'The future of library specification' [11], Leavens describes several indirect approaches in addition to the known ones associated with informal documentation and formal specification; these are specification through example uses, specification through library source

codes and specification through unit tests. The main conclusion reached by the author is that library specification must combine all of these approaches.

In our previous studies [12, 13], we also proposed a formalism for library specification and a language supporting the description of such specifications. However, the options for using the libraries (i.e., the behavior) are described implicitly within this approach, and the language does not allow defining function contracts and the influence of the functions on the environment to the full extent.

Thus, at present, there is no universal approach to library specification that would allow to:

- describe the external interface of the library in detail;
- define the potential protocols for using the library;
- specify the side effects of the library, i.e., its influence on the environment;
- explicitly introduce semantic descriptions of library behavior.

## 3. Library Organization Specifics

The specifics of using libraries is that a library is not just a purely functional object; it can possess an internal state and various side effects that significantly affect the opportunities for calling individual functions.

Let us introduce a classification of function libraries in terms of their internal state.

1. Libraries without an internal state
2. Libraries with the internal state of the library
3. Library with the internal state of the object created
4. Combined libraries

The first class comprises libraries containing pure functions without side effects. These include, for example, libraries of the mathematical functions of the standard C language library (math.h).

The second class includes libraries that preserve their state, that is to say, the behavior of individual functions depends on the state of the library. An example of such a library is the part of the stdlib library providing random number generation. Calling srand() sets the initial value of the generator, while rand() returns the next random number in the sequence constructed on the basis of the initial value.

The third class consists of libraries that preserve context within an object created by the library's functions. Such an object may be, for example, a newly created socket or a file descriptor. In the first case, the context contains the parameters of the socket (IP-addresses, port numbers, state),

while in the second case, the context contains the file parameters, the opening mode and the next read data pointer.

The fourth class is the most general, containing libraries that combine the features of the second and the third classes.

# 4. Formal Specification of Libraries

A full formal specification of libraries should describe:

- a signature of all functions making up the library;
- a contract for each library function (preconditions, postconditions, the influence on the environment, etc);
- a behavioral model of the library taking into account all possible options for using the library's functions and specifying, in particular, the behavior of the library in case of invalid use;

Based on the above, let us define the full specification of libraries as <F, L>, where

- $F = \{F_i\}$ is the set of library functions;
- L is the behavioral description of the library.

An individual library function, Fi, is defined as <Name, Arg, Res, Pre, Post, A, CondA, D, CondD>, where

- Name is the name of the function;
- Arg is the set of the formal arguments of the function;
- Res is the result of the function;
- Pre are the preconditions of the function expressed by the formula in the first-order logic of the arguments Arg and Res;
- Post are the postconditions of the function expressed by the formula in the first-order logic of the arguments Arg and Res;
- A is the set of semantic actions[2] performed by the function;
- CondA is the set of conditions for semantic actions to be performed. An action Ai is performed during the execution of a function if the expression $CondA_i$ is true.
- D is the set of launched child state machines;
- CondD is the set of launch conditions for child state machines. A machine Di is launched during the execution of a function if the expression $CondD_i$ is true.

[2] Semantic actions are an abstraction for describing significant behavioral elements [16]

Let us represent the behavioral description of the library by a of set of parameterized extended finite-state machines (EFSM): L = {L, S1(q,P), …, Sn(q,P) ()}, where

- L is the main extended finite-state machine describing the behavior of the entire library;
- Si is an $i^{th}$ child EFSM launched if certain conditions are fulfilled;
- the parameter q is the initial state of the child finite-state machine;
- P is the optional parameter of the child finite-state machine

The state of the main state machine corresponds to the state of the library, and the state of the child ones corresponds to the state of the objects created. The stimuli forcing the machine to pass from one state to another are the calls of library's API functions.

An individual machine is defined as a modified EFSM <Q, $Q_0$, X, V, C, T>, where

- Q is the set of control states of the machine (the states of the library objects);
- $Q_0$ is the non-empty set of initial states of the machine. Several initial states can exist for child state machines, since initial conditions may be different when an instance of the machine is created;
- X is the set of finish states. Child machines are destroyed after reaching these states;
- V is the set of internal variables of the machine;
- C is the set of function calls acting as stimuli, $C_i$ is the call of an $i^{th}$ function; $C_i \in F$ ;
- $C_i^A$ is the set of semantic actions initiated by the function launch when $C_i^{CondA}$ is true;
- $C_i^D$ is the set of child state machines launched by the function when that $C_i^{CondD}$ is true;
- T is the transition relation.

Due to limitations of space, the formalism is presented without going into too much detail. Such issues as the specification of invalid behavior, the default actions, the data types, etc., have been left outside the scope of our investigation. These issues will be discussed in more depth in other studies.

Actually, from a developer's perspective, the behavioral description of libraries is better represented in graphical form rather than from the standpoint of set theory.

Fig. 1 shows an example of graphically describing the client side of the TCP-socket library. The solid line indicates the transitions of the machine, and the dashed line indicates the launches of the child machines; the finish states are highlighted in red. The machine "L" describes the overall behavior of the bsd-socket library, which, in contrast to WinSock, does not require initialization. A side effect of calling socket() is the creation of a new machine "P", corresponding to the newly created socket, with its own life cycle. It should be noted that several machines can be created, differing only in the launch parameter of the child machine (an element corresponding to calling socket()).

Fig. 2 presents a more complex example, showing a graphic model of the server side of the TCP protocol of the bsd-socket library. In addition to the top-level machine corresponding to the library ("L"), the figure shows two families of machines: the first ("P") encapsulates the properties of the listening sockets, and the second one ("S") those of the server sockets created.

Both examples demonstrate only the behavioral description of libraries, without specifying a set of functions.

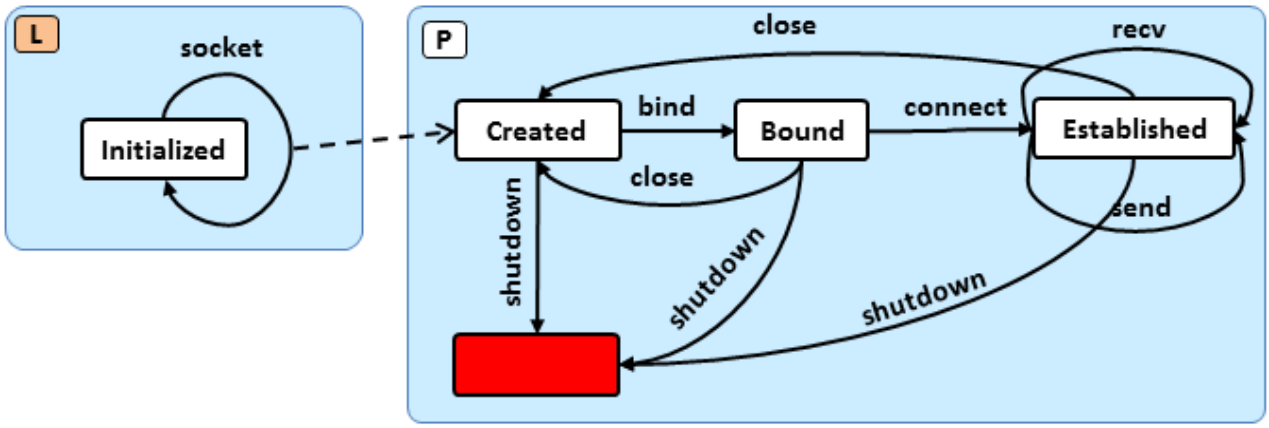


Рис. 1. An example of a simple machine corresponding to the client side of the TCP protocol of the bsd-socket library

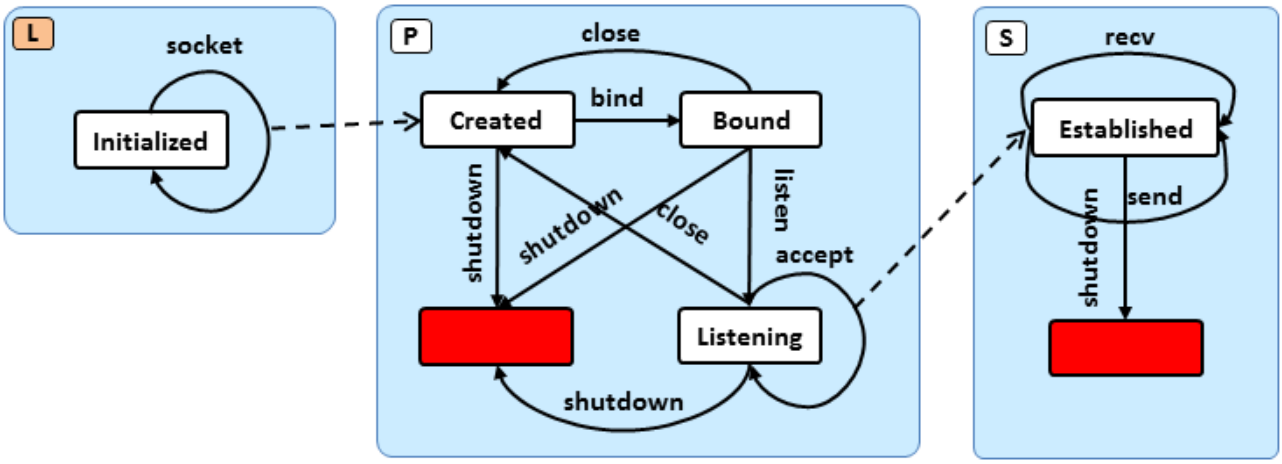


Рис. 2. 2. Example of a machine describing the server side of the TCP protocol of the bsd-socket library

Obviously, a library developer requires convenient tools for defining the formalisms introduced. We propose using special language for describing the sets of library functions, and an object-oriented graphical editor for the behavioral description of the library.

## 5. Prospects of Using the Developed Formalism

## 5.1 Constructing Specifications

Formal specifications could be constructed in one of two ways: with the help of library designers and of developer communities.

In the first case, the library specification is created by its developer. A language for describing library specifications (similar to the previously developed PanLang language [13]) is formed for this purpose; all the properties of the library expressed by the formalism developed can be defined by means of that language.

In the second case, the extraction methods will be based on exploiting the international programming experience (Empirical Software Engineering), with the structural and the behavioral components of the specifications stemming from the analysis of software repositories (Mining Software Repositories). In this case, only a skeleton of the specification is formed, with the remaining part to be refined manually.

The language for describing specifications corresponding to the formalism presented, and the methods for analyzing software repositories with the purpose of obtaining specification skeletons are currently being developed by the author's research team; describing them is beyond the scope of this paper.

## 5.2 Using Formal Library Specifications

The formalism developed and described in this paper can be used in the future for solving a wide range of research and engineering problems, including automated defect detection in complex multi-component software projects, automated porting of applications to new libraries and automated generation of software documentation.

Library specifications are used as part of the solution for the problem of detecting software defects with the purpose of reducing the dimension of the detection problem. This is achieved by approximating the behavior of libraries and library functions by integrated visible behavior set in the specification. In this case, the library function is replaced by a system of predicates based on contracts and error states defined in the specification. This approach is used for the BMC analyzer Borealis, developed in the Program Analysis and Verification Laboratory of the Peter the Great St. Petersburg Polytechnic University [14]. A similar approach is used for the Aegis tool based on abstract interpretation, being developed in the same laboratory [15].

The task of automated migration of software to new libraries requires not only the external specification of the library's behavior, but also a partial description of the internal semantics of the library. A semantic domain of the library is built based on the description of the internal semantics,

and can be then used for checking library compatibility and automatically constructing the migration procedure. [16]

## 6. Conclusion

The study presented the results on creating formalism for software library specification. The formalism was built taking into account the entire range of problems that could be solved through it. The main idea was in using the same formal specification as a basis for several methods of software engineering: detection of software defects, automated software migration and software documentation generation. Due to limited space, the formalism was presented without going into details.

A direction for the future research is developing language support for the proposed formalism and implementing converters of language descriptions for the existing tools of error detection and software migration.

## References

1. D. Lamb. IDL: sharing intermediate representations. ACM Trans. Program. Lang. Syst. 9, 3 (July 1987), 297-318. DOI=http://dx.doi.org/10.1145/24039.24040
2. https://msdn.microsoft.com/en-us/library/aa367091
3. http://www.omg.org/gettingstarted/omg_idl.htm
4. R. Allen and D. Garlan. Formalizing architectural connection. In Proceedings of the 16th international conference on Software engineering (ICSE '94). IEEE Computer Society Press, Los Alamitos, CA, USA, 71-80.
5. Хоар Ч. Взаимодействующие последовательные процессы. — М.: Мир, 1989. — 264 с.
6. A.W. Roscoe, Modelling and verifying key-exchange protocols using CSP and FDR, Proceedings of 1995 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 1995.
7. L. de Alfaro and T. Henzinger. Interface automata. In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-9). ACM, New York, NY, USA, 2001, 109-120. DOI=http://dx.doi.org/10.1145/503209.503226
8. M. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07). ACM, New York, NY, USA, 123-134. DOI=http://dx.doi.org/10.1145/1250734.1250749
9. S. Sankaranarayanan, F. Ivančić, and A. Gupta. Mining library specifications using inductive logic programming. In Proceedings of the 30th international conference on Software engineering (ICSE '08). ACM, New York, NY, USA, 131-140. DOI=http://dx.doi.org/10.1145/1368088.1368107

10. Thomas Ball and Sriram K. Rajamani. SLIC: a Specication Language for Interface Checking (of C). Microsoft Research, Technical Report, MSR-TR-2001-21. 2002
11. Gary T. Leavens. The future of library specification. In Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10). ACM, New York, NY, USA, 211-216. DOI=10.1145/1882362.1882407
12. Itsykson V. M., Zozulya A.V. The formalism for description of the partial specifications of program envinroment components. St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunication and Control Systems. N 4, 2011. – SPb: Publishing of Polytechnic University - pp. 81-90.
13. Itsykson V.M., Glukhikh M.I. A program component behavior specification language. St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunication and Control Systems. N 3, 2010, SPb: Publishing of Polytechnic University cc. 63-71.
14. M.Kh. Akhin, M.A. Belyaev, V.M. Itsykson. Software defect detection by combining bounded model checking and approximations of functions / Automatic Control and Computer Sciences, December 2014, Volume 48, Issue 7, pp 389-397
15. V. Itsykson, M. Moiseev ; V. Tsesko ; A. Zakharov. Automatic defects detection in industrial C/C++ software. In proceeding of Software Engineering Conference in Russia (CEE-SECR), 2009 5th Central and Eastern European. IEEE, Moscow, 07 DOI=10.1109/CEE-SECR.2009.5501189, pp 50-55
16. Itsykson V.M., Zozulya A.V. Automated Program Transformation for Migration to New Libraries. Software Engineering. 2012. N 6. pp. 8-14