INVITED PAPER *Special Section on Networking Technologies for Dependable Networks*

On Applicability of Formal Methods and Tools to Dependable Services

Fuyuki ISHIKAWA[†a)], ***Member and*** **Shinichi HONIDEN**[†b)], ***Nonmember***

SUMMARY As a variety of digital services are provided through networks, more and more efforts are made to ensure dependability of software behavior implementing services. Formal methods and tools have been considered as promising means to support dependability in complex software systems during the development. On the other hand, there have been serious doubts on practical applicability of formal methods. This paper overviews the present state of formal methods and discusses their applicability, especially focusing on two representative methods (SPIN and B Method) and their recent industrial applications. This paper also discusses applications of formal methods to dependable networked software.

key words: *dependability, software engineering, formal methods*

1. Introduction

Nowadays our daily lives rely more and more heavily on computer systems. We cannot know how many computer systems we use explicitly or implicitly in a day, as so many essential activities are supported by them (banking, transport, etc.). The notion of *dependability* has therefore attracted considerable attention of both research and practice communities. Dependability is defined as "the trustworthiness of a computing system which allows reliance to be justifiably placed on the service (system functions) it delivers" [1]. Dependability contains significant attributes such as availability, reliability, safety, integrity and maintainability.

Efforts on dependability have investigated means to avoid *failures*, where the delivered service deviates from the "correct" one, even with existence of various kinds of *faults* (e.g., hardware component breakdown) that can lead to *errors* (parts of the system states that may lead to failures). Failures should be avoided, as many as possible, by letting the system to prevent, tolerate, remove, and estimate/predict faults. As one of the most significant areas for dependability, dependable networking technology has been investigated by many researcher, as most of computer systems today heavily rely on the Internet infrastructure. These technologies include mechanisms for tolerance of multiple node/link/switch faults, routing reconfiguration for QoS guarantees, transaction protocols that can handle physical faults, and so on [2]. In this way, availability and reliability, or readiness and continuity of "correct" services, even under physical faults, are typically focused on as significant aspects of dependability.

It is now well-known that it is very difficult to implement "correct" services, preventing or removing design faults while engineering the system. This aspect of dependability has been discussed in the Software Engineering area. Here it should be noted that the "correctness" of services (or software behavior) is relative to the specifications describing requirements or intentions about them. There can be (and have been) incidences due to design faults (not due to physical faults). Such incidences are often very critical because design faults can lead to not only unavailability of the services but also their harmful behaviors. Dependability of any networked applications thus requires developers to tackle the problem of ensuring correctness of software implementing the networked services.

Among various approaches to the problem in the Software Engineering area, this paper overviews and discusses formal methods. Formal methods are mathematically-based techniques for the specification, development and verification of software and hardware systems [3]–[5]. This paper discusses formal methods for the following reasons.

- Formal methods have been believed to be effective when high dependability is required. Formal methods are therefore getting widespread attention as a possible measure for preventing design faults and resulting service failures, whose criticality has been recognized and often emphasized recently.
- On the other hand, formal methods are not known well. Especially, it is difficult to find hints to examine their practical applicability, while generally they are believed to be too costly.
- Formal methods are relevant to the dependability community, as they can be used to support approaches to dependable mechanisms, protocols, and so on (e.g., verification of Byzantine consensus protocols [6]).

The purpose of this paper is to introduce formal methods to people who are not engaged in formal methods, focusing on practical tools rather than theoretical foundations. Section 2 first overviews some tools to describe what formal methods are about. Sections 3 and 4 then summarize two representative methods/tools, SPIN and B Method, citing statistics from their recent industrial applications. SPIN is an illustrative example of model checking to find possible deficiencies by exhaustive exploration of possible state transitions. SPIN has been used for verification of network protocols and networked software. B Method is one of for-

Manuscript received April 25, 2008.
Manuscript revised August 25, 2008.
[†]The authors are with National Institute of Informatics, Tokyo, 101-8430 Japan.
a) E-mail: f-ishikawa@nii.ac.jp
b) E-mail: honiden@nii.ac.jp
DOI: 10.1587/transcom.E92.B.9

mal specification to increase confidence in earlier phases by clarifying and verifying functional specifications. A variation of B Method, called Event B [7], is attracting more and more attentions in Europe to handle event-based distributed systems based on B Method. Section 5 finally discusses applicability of formal methods in general, as well as their applications to dependable networked software.

2. Brief Review of Formal Methods

This section overviews formal methods. The purpose of this section is not to cover many available formal methods but to give intuitions about formal methods in general.

Formal methods target development of systems, especially software, and provide solid, scientific foundation based on mathematical logic. Formal methods use languages with clear and strict semantics to model the targets (typically designs), as well as techniques (e.g., theorem proving) based on the languages. Through such languages and techniques, formal methods provide systematic means to verify correctness of the formalized models, which leads to high dependability of the systems.

The term "formal methods" is very general, and there are a variety of its instances. The differences come from different targets and purposes, such as characteristics of target systems (e.g., concurrency) and properties to be ensured (e.g., deadlock-freeness), development phases to be covered (e.g., the design phase), underlying theories, and so on.

Table 1 illustrates several tools for formal methods. Although they are only a small part of available tools, they illustrate how formal methods vary, as described below.

Modeling Target Each method has its modeling target. VDM and B Method target sequential processing of data structures such as lists, while SPIN and FDR discuss concurrent processes. SPIN and FDR differ in their abstraction level: SPIN discusses state transition in terms of variable values while FDR rather focuses on observable interactions. Although both B Method and FDR mention the term"refinement," they actually refer to different aspects: roughly speaking, B Method discusses removal of nondeterministic behavior while FDR discusses component decomposition.

Verified Properties SPIN allows for verification of temporal properties (such as safety and liveness) through state transition, while VDM discusses effects of operation execution.

Techniques In broad terms, there are two approaches to verification, namely, model checking and theorem proving. Model checking examines all the possible (finite) state transitions in a defined model to see whether given properties are satisfied. So it tries to gather observations enough to give confidence in desired properties. For example, model checking tools can explore all the possible state transitions to see whether there can be deadlock. On the other hand, theorem proving derives properties from not observations but properties that are already known to hold. For example, from hypotheses $x > 0$ and $y = x + 1$, a property $y > 0$ can be derived by using axioms about numbers. In any case, completeness is often emphasized as an advantage of formal methods compared to traditional testing. Sometimes lightweight usages are additionally considered: VDM allows for execution and (incomplete) testing of formal specifications to verify and validate specifications in early phases. It is notable language models may provide foundations for several techniques: VDMTools primarily support specification execution but the underlying VDM languages allow for proof as well.

Target Phases Java PathFinder targets verification of implementation codes, while SPIN targets design models. Such classification is not strict, for example, it is possible to use additional tools to extract SPIN models from implementation codes. B Method is quite different as it targets a process of stepwise refinement from early design to implementation.

The most popular usage of formal methods is verification of useful properties, or finding possible causes of bugs, that are difficult to handle by program testing. Typically models of concurrent/distributed systems are verified by means of model checking, as they include complex state transitions depending on the order in which actions and events occur in multiple threads/processes. As it is usually not possible to control such an execution order in actual implementation, program testing fail to provide confidence in absence of bugs. Model checking is thus popular, providing focused solution to the limitation of testing methods.

As another usage, formal specification is considered to

Table 1 Differences in tools for formal methods.

Tool Name	Modeling Target	Verified Properties	Techniques
VDMTools (Vienna Development Method) [8]	system structures (data structures of variables and operations on them)	invariants and operation pre- and post-conditions	specification testing
B4Free/Atelier B (for B Method) [9]	system structures and their stepwise refinement	invariants and operation pre-conditions as well as refinement relationships	theorem proving
SPIN (Simple PROMELA Interpreter) [10]	state transitions of concurrent processes through their interactions	temporal properties	model checking
FDR (Failures-Divergences Refinement) [11]	observable interactions between concurrent processes	refinement relationships and determinism	model checking
Java PathFinder [12]	control structures of Java programs	deadlocks, unhandled exceptions and other properties defined as classes	model checking

clarify and validate system specifications often for highly-dependable systems. As requirements or specifications described in natural languages or languages without formal semantics (e.g., UML) can include ambiguity, incorrectness, and incompleteness. Such informal specifications do not facilitate scientific validation and can lead to misunderstanding based on personal interpretation, often causing costly fix in later phases. Methods for formal specification thus provide rigorous approaches especially in earlier phases, sometimes supporting the process toward implementation in executable codes.

To take a closer look at each of the two typical usages of formal methods, the next section focuses on SPIN and B Method respectively. Especially, in order to discuss applicability of formal methods, statistics in their recent industrial applications are cited.

3. Model Checking

This section focuses on one of tools for model checking, SPIN, describing its essence and citing statistics in its recent application to an industrial project.

3.1 Overview of SPIN

SPIN (Simple PROMELA Interpreter) [10] is a tool for model checking of distributed systems. It has been refined since its first release in 80's and received ACM Software System Award in 2001. In SPIN, concurrent processes are modeled in an abstract way using a dedicated language, PROMELA (PROcess MEta-LAnguage). PROMELA facilitates modeling of significant notions in concurrent/distributed systems, such as synchronous/asynchronous communications and blocking behavior. Besides fundamental properties to be verified, such as deadlock-freeness, SPIN allows for property definition in LTL (Linear Temporal Logic). LTL provides temporal operators to describe properties about execution paths such as "some property *eventually* holds," "some property *always* holds," and "some property holds *until* another property holds." Although the SPIN tool provides a variety of useful functionalities, the primary functionality is generation of C codes to efficiently run exhaustive verification of given PROMELA and LTL descriptions. When the property turns out not to hold, a counterexample is provided by the SPIN tool to help users track the scenario (execution path) where the property does not hold.

Figure 1 shows a simple example of description of concurrent processes in PROMELA. It models two simple processes, *producer* and *consumer*. The producer produces some resource and the consumer consumes it. After the producer produces the resource, it is blocked until the consumer consumes it. The consumer is blocked when the resource has not been produced yet after the last consumption. For simplicity, this example only considers one producer and one consumer. As a result, they run one after the other. The description starts with declaration of an enumeration type

```
mtype = {P, C};
mtype turn = P;
active proctype producer()
{
  do
  :: (turn == P) -> printf(''produced\n''); turn = C
  od
}
active proctype consumer()
{
  do
  :: (turn == C) -> printf(''consumed\n''); turn = P
  od
}
END
```

Fig. 1 Example of process description in PROMELA.

(*mtype*) that takes one of the two values *P* and *C* (producer and consumer). A variable *turn* is also declared to denote which of the producer and the consumer can run. Two processes corresponding to the producer and the consumer are declared (*active proctype* is for definition of a process that is instantiated automatically). The producer process has a loop (the *do* ... *op* statement). In this simple example, there is only one execution sequence (defined with *::*) inside the loop. The execution sequence has a guard *(turn == P)*, and is only executed when the guard condition holds. When the guard condition holds, the execution sequence produces something (abstracted away in this model) and modifies the variable *turn* to let the consumer execution sequence run.

As the space is limited, the above example only models concurrent processes with a shared variable and blocking behavior. In realistic scenarios, each process has conditional and non-deterministic behavior and multiple processes interact with each other using synchronous/asynchronous communication channels. PROMELA facilitates modeling of such essential aspects in distributed systems.

Besides deadlock-freeness and inserted assertions, properties to be verified are described in LTL and verified by using SPIN. Typical properties are liveness, i.e., some event will ultimately occur under certain condition, and safety, i.e., some event never occurs under certain conditions. For example, a liveness property "any request will ultimately be satisfied" is described in LTL as follows.

$$[]\ (req\ =>\ <>\ sat)$$

(*req* and *sat* are labels attached at execution points in PROMELA)

The [] and <> operators mean "always" and "eventually," respectively. So the above formula means: at any state (always), if *req* holds at that state, then *sat* will hold at some future state (eventually).

3.2 Recent Case: Groupware Protocols

There have been many applications of SPIN, typically to network protocols and networked software. In the case of SPIN, or model checking tools in general, the primary

advantage is discovery of possible deficiencies in concurrent/distributed systems, which cannot be explored by program testing in a controllable way. On the other hand, size of the problem that could be verified is discussed as model checking use exhaustive exploration of possible state transitions. Below describes one of the recent studies, reported in 2005, using SPIN for verification of groupware protocols [13].

3.2.1 Process

The study considers introduction of publish/subscribe protocols into a groupware. The protocols define user operations and their concurrency control, e.g., a user conducts a "checkin" operation (commit a modified file) and interested users are notified of the event. SPIN is used to validate the protocol design before actual implementation in order to avoid costly fix in implementation.

The protocols are modeled carefully in PROMELA to avoid state explosion. For example, the constructed model only includes one file at any moment. Message loss and blocking of notification operations are modeled not to happen. The following properties are described by using LTL with 2-4 processes corresponding to users.

concurrency control e.g., every lock on a file must eventually be released.

expected functionality e.g., every checkout must eventually lead to a notify to all (and only those) users that are registered for the checked out file.

denial of service No user can be denied a service forever.

3.2.2 Statistics

Below describes statistics about full state space searches for deadlock. In any cases, no deadlock was found.

Users	State Vector	Depth Reached	Memory Used	Run-time
2	84 byte	4423	37Mb	1 sec.
3	108 byte	434033	114 Mb	3 min.
4	132 byte	10484899	916 Mb	8 hour

Verification of LTL properties was conducted for 3 users. It took about 40 min. at the most. Two of nine properties were found not to hold. One of the two properties shown not to hold is the example of properties about the concurrency control described above. The property is not satisfied when there is a user that keeps a lock forever. It was determined to make the property hold not by protocols but by user guidelines. The other of the two properties shown not to hold is the property about denial of service. It is possible that a user never get his turn when other users keep the controller busy. This problem was determined to be solved in future extensions using reservation.

4. Formal Specification

This section focuses on one of methods for formal specification, B Method, describing its essence and citing statistics in its recent application to an industrial project.

4.1 Overview of B Method

B Method [9] is a formal method to derive executable codes from requirement documents. It was introduced in the late 80s', and established for a large industrial project in late 90's [14]. In B Method, the data structure of system states is modeled as a set of variables and operations on them, which typically appear in many kinds of information systems. In addition, invariants, properties expected to always hold through state transitions caused by operation executions, are described by means of first order logic and set-theoretic constructs.

Tools such as B4Free and Atelier B [9] provide functionality to generate properties that need to be verified, as proof obligations. Proof obligations include the following properties.

- There is at least one set of variable values that satisfy the invariants (the invariants do not contain inconsistency).
- The initial values of the variables satisfy the invariants.
- For each operation, if the precondition of the operation and the invariants hold before execution of the operation, the invariants still hold after the execution.

The tools also provide (semi-)automated proof functionality. They can conduct automated proof in most cases, but developers sometimes need to help the tool with interactive proof interfaces.

In addition, B Method considers to derive accurate, detailed models gradually, while starting with a simple, abstract model. B Method allows for such stepwise refinement of models until executable codes are obtained (Fig. 2). In B Method, details of the problem are first extracted from the requirement documents, gradually through stepwise refinement, and formalized as Abstract Models in the B language. The obtained Abstract Models are then transformed into Concrete Models, again gradually through stepwise refinement. Abstract Models may contain nondeterministic behaviors or abstract operations, which are excluded through the stepwise refinement toward Concrete models. In Concrete Models, transitions are completely deterministic and data are in a one-to-one correspondence with computerizable objects (scalar, arrays, etc.). Finally, the Concrete Models are transformed into executable codes by using translators.

In each refinement step, proof obligations are examined about the correctness of the refinement, that is, the obtained model does not invalidate the properties proved in the previous model. In this way, B Method involves verification of correctness of each model as well as correctness of each refinement into a more concrete model. B Method does not expect testing of the generated codes, as it conducts verification by theorem proving throughout the process.

Figure 3 shows a simple example of B models. It describes a system that manages points (or coupons) given to

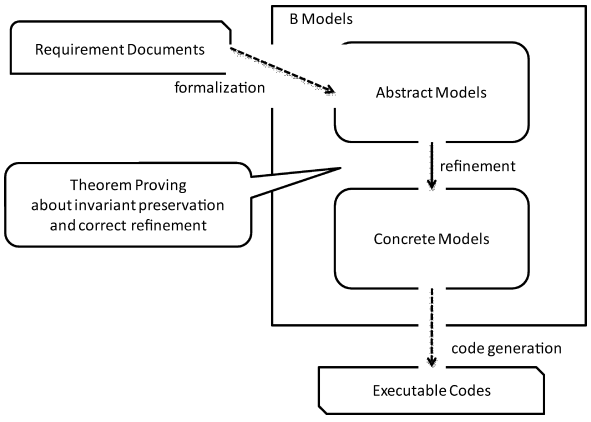

Fig. 2 Overview of B method.

```
MODEL
  PointSystem
VARIABLES
  granted, used
INVARIANT
  granted : NATURAL & used : NATURAL
  & used <= granted
INITIALISATION
  granted, used := 0, 0
OPERATIONS
  sp <-- use_a_point =
    PRE used < granted
    THEN sp, used := used+1, used+1
    END;
  ap <-- grant_a_point =
    BEGIN ap, granted := granted+1, granted+1 END
END
```

Fig. 3 Example of B abstract model.

and used by the users (e.g., a point is given when a user buys something, and later can be exchanged with some gift.). The model defines the name of the model, and then variables that define the system state: the number of granted points and used ones (*granted* and *used*, respectively). Invariants, properties expected to always hold, are then declared. In the figure, the first and second properties (separated by the logical AND operator &) denote type constraints. The last one denotes an expected property that the number of used points cannot be more than that of granted points. The model also defines the initial state where both the variable values are 0. Finally the model defines two operations. The operation *use_a_point* has an output variable *sp*, preconditions that need to hold before its execution, and the operation itself where the output variable *sp* and the variable *used* are both updated to the new value of *used + 1* (*grant_a_point* is defined similarly). In this way, invariants and operation preconditions are clarified for rigorous understanding and verification.

In the case of this simple model, the tools generate 5 proof obligations in addition to trivial (quite obvious) ones, which can then be proved automatically. For example, one of the generated proof obligations is "if the precondition of the *use_a_point* operation holds, then execution of the operation always leads to states that satisfy the invariants." This can be proved easily: intuitively, as we have $used < granted$ as the precondition, we still have the invariant $used \leq granted$ after increment of the *used* variable.

4.2 Recent Case: Airline Shuttles

B Method was applied to development of airline shuttle management system [15] (presented in 2005). Below describes the development process and notable statistics in their experience.

4.2.1 Process

Before applying B Method, requirements were examined in the develop process. Here activities of the following were involved in the requirements phase and construction of the initial B models.

- In the requirements phase, data flow diagrams and pseudo codes were used to discuss essential parts in order to reduce gaps between the requirements description and the B models following them.
- Many ambiguous points were found through formalization into B models, and resolved by questions.
- Review process was introduced because construction of the initial B models fully relies on human developers.
- As it was not always clear how to formalize desired properties, provisional properties were defined and then polished by trying to prove them. Through this process, errors in the property descriptions, errors in specifications, and unrealizable requirements were discovered and corrected.

The process after these activities is defined in B Method, as described in Sect. 4.1. A notable point in this application case is that tools for (semi-)automatic refinement were introduced, which were not used in the previous application case [14]. The tools handled refinement of abstract set-theoretic data into computerizable objects.

4.2.2 Statistics

The project provides some interesting statistics. The project had finally about 180,000 lines of B models, including the following.

Abstract Model	manual	28,000 lines
	generated	10,000 lines
Concrete Model	manual	28,000 lines
	generated	118,000 lines

Tools were used to generate typical Abstract Models as well as to obtain refined Concrete Models. Approximately, 30% of the B models were constructed manually.

These B models required proof of about 43,000 lemmas generated by the tools. About 1,400 lemmas (3%) required manual, interactive proof while the others (97%) were proven automatically. Manual proof was conducted

with an average rate of 15 lemmas by man-day.

The number of ADA code lines generated finally was about 160,000. The number was so large due to use of safety-enhanced version of ADA, duplicated codes, and so on. The number of "optimal" ADA code lines was estimated at 60,000.

Manpower cost rate was given as follows.

Abstract Model	55%
- questions/answers and document analysis	18%
- inspections	5%
- proof	16%
Concrete Model	24%
- proof	11%
Others (project management, documentation, etc.)	21%

Many efforts were put on construction of the Abstract Models (55%), most of which were on their verification and validation (39% in total, about 70% of the efforts put on Abstract Models).

5. Discussion

5.1 Applicability of Formal Methods

Generally, advantages of development using formal methods are believed as follows.

- The process of formalization helps understanding of the problem, leading to discovery of ambiguity, incorrectness, and inconsistency typically in informal requirements or specifications.
- Obtained formal models allow for tool verification of system behavior and desirable properties, typically providing completeness for defined criteria.

These are about increasing confidence in the output of (a) certain phase(s). It will prevent discovery of some errors in later steps (after implementation and testing, or even after deployment), which has recently been recognized to be very costly. With these advantages, formal methods seem promising especially when successful applications such as the ones presented in Sects. 3.2 and 4.2 are focused on.

On the other hand, limitations of formal methods are generally believed as follows.

- Formal methods are too costly unless the target system requires very high dependability.
- They are very theoretical, too difficult for common developers to understand and use.
- They provide little support for practical use.

Below these disadvantages are discussed further, citing the application case presented in Sect. 4.2, recent studies on B Method,

5.1.1 Too Costly?

Regarding the first limitation mentioned above, i.e., costs, it is necessary to clarify what kinds of costs are discussed. In the case of model checking, it is often thought adequate to take additional cost to explore deficiencies that cannot be revealed by program testing. In the case of formal specification, which makes the development process rigorous, costs are difficult to examine. The advantages mentioned at the beginning of this section mention decrease in costs in later phases, but implicitly mentioning increase in costs in earlier phases. In the case of B Method presented in Sect. 4.2, most efforts were put on the initial construction of B Abstract Models from requirement documents, while later phases (implementation and testing) did not basically require any efforts.

Regarding this point, there was a study that compared cases where B Method is used with a case where an informal method (SDL) is used [16]. It reported decrease in overall development time in 30–50% (depending on the way B Method is used) due to remodelling work required to counteract errors due to misspecification in the SDL case. There has also been studies to examine systematic ways to balance effectiveness and cost in introduction of formal methods, e.g., by using metrics to find out complex part of the system [17].

Even if formal methods provide overall cost-efficiency in some situations, there is a problem that it is difficult for developers (or rather managers) to learn cost-efficiency of formal methods. It is necessary to learn involved tradeoffs in order to use formal methods, as it changes the way of thinking, e.g., exclusion of the testing phase. In other words, the problem is unavailability of reference methodologies/guidelines about incorporation of formal methods into development processes and about selection of formal methods, in order to allow for meaningful discussion on introduction of some formal methods.

5.1.2 Too Difficult?

Regarding the second limitation, i.e., difficulty, it depends on whether it is necessary to know the inside of the tools. Readers might find the example SPIN and B models easy to read. SPIN requires understanding of automata theory and temporal logic. B Method requires basic mathematics, first order logic and set theory. In the case of SPIN, developers can rely much on the automated verification by the tool. In the case of B Method, developers need to help the tool through interactive proof, which is not so easy for common developers. It is notable that anyway it is still necessary to know the underlying theory to deal with troubles, to make more use of the tools, and be confident of what are being done by the tools.

Besides theoretical foundations it is absolutely necessary to carefully understand practical ways of thinking for each of formal methods and tools. For example, the presented application of SPIN (Sect. 3.2) considered abstraction (simplification, or approximation) of the targeted system in order to reduce the number of states to be explored. It is up to human developers to define problems that are han-

dled by formal methods and tools, as well as to interpret obtained results.

It is not so reasonable not to learn difficult things without discussion of its effectiveness. In this sense, formal methods are believed to be worth learning. Formal methods and their underlying theories are being incorporated into education programs, such as an emerging standard curriculum for teaching Software Engineering in universities [18].

Another point is that it is not necessary for all the developers to know details of the applied formal methods. In development of a transport system in New York using B Method, only one of four developers knew underlying theories of B Method while two of them had not joined development using B Method [19].

Despite the above discussion, it should be noted that, again, it is necessary to have some guidelines regarding teaching or managing development using formal methods.

5.1.3 Little Support for Practical Use?

Regarding the third limitation, i.e., lack of practical, user-friendly tools, it is often true though some tools such as SPIN are very sophisticated. For example, B4Free, the free version of the B tools, is based on XEmacs and not so user-friendly (though the commercial version is sophisticated).

It is well known that tools for B methods have been sophisticated through application to actual, industrial projects, by strong collaboration of a tool company and a transport company. It is necessary for formal methods to have user-friendly, practical tools in order to become widespread, matching technical seeds with practical needs typically through collaboration of the academic and the industry.

5.2 Toward Dependability in Highly Networked Society

As a variety of digital services are provided through networks, more and more efforts are required to ensure dependability of software behavior implementing services. Difficulty in networked software primarily comes from autonomy and concurrency. Each process acts autonomously while interacting with one another. The resulting system consists of non-deterministic, complex state transitions, which cannot be controllably explored in actual implementation. As described in the previous sections, model checking tools such as SPIN can deal with this difficulty directly. So model checking has been made use of for verification of networked software, especially essential protocols to coordinate distributed processes. Network faults are sometimes modeled when targeted protocols essentially handle reliability, or message reachability properties.

This paper has discussed another approach, formal specifications such as B Method. As formal specifications typically focus on data structures and functional aspects of systems, existence of networks is often abstracted away. So they can be still used to increase confidence in the essential functionalities exchanged via networks. In addition, recent studies have considered adaption of formal specifications to concurrent or distributed systems. It is notable that a variation of B Method, called Event B [7], is attracting more and more attentions in Europe to handle event-based distributed systems based on B Method.

This paper has only focused on formal methods and tools that are used before implementation. There is another approach to extract models to analyze or verify from executable codes. The approach is getting popular because it can enhance the testing phase without changing much of the development process. For networked software, it is necessary to use dedicated mechanisms to extract models to be analyzed, as programs are constructed so that they are only connected indirectly through RPC or RMI. Recently there have been studies on such mechanisms to gather distributed programs, which are specified to run with existence of networks, and construct an adequate model of concurrent processes combined with a model that denotes network faults [20].

It is also notable that specific tools have been developed to adopt formal methods to popular application areas in networked software. Recently, formal methods for Web Services, or Service-Oriented Architecture, are representative. For example, a sophisticated tool has been provided for verification and analysis of business processes and service choreographies [21]. Business processes denote local behavior of each participant while service choreographies denote expected collaboration protocols from the global viewpoint. They are specified by standard languages called BPEL and WS-CDL, respectively. The tool allows for verification and analysis of descriptions in these languages, on the basis of a formal model, Labeled Transition System. Such tools facilitate to make use of formal methods, as they provide fine support for the target area and eliminate the necessity to construct models from scratch, which is required when general-purpose tools are used.

6. Conclusion

This paper has overviewed and discussed the present state of formal methods, citing their recent industrial applications. The essence of formal methods should provide great effectiveness when the scope of their contributions are clarified and properly positioned in the development process. However, there have still been obstacles, primarily unavailability of reference methodologies/guidelines about incorporation of formal methods into development processes, about selection of formal methods, and about education of formal methods. There has also been the problem of limitation in practical applicability of tools. These issues need to be tackled through collaboration of the academic and the industry, matching technical seeds with practical needs, for complex but dependable software systems that provide the basis of our lives in highly networked society.

References

[1] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE

Trans. Dependable and Secure Computing, vol.1, no.1, pp.11–33, 2004.
[2] D.R. Avresky, J. Bruck, and D.E. Culler, "Introduction to the special section on dependable network computing," IEEE Trans. Dependable and Secure Computing, vol.12, no.2, pp.97–98, 2001.
[3] H.C. Michael, "Why engineers should consider formal methods," 16th DASC. AIAA/IEEE Digital Avionics Systems Conference, pp.16–22, 1997.
[4] "Formal Methods Europe," http://www.fmeurope.org/, April 2008 (Last Access).
[5] S. Nakajima, "Formal methods as software engineering tools — An exile in FM wonderland," Tech. Rep. NII-2007-007J, National Institute of Informatics, 2007.
[6] P. Zielinski, "Automatic verification and discovery of byzantine consensus protocols," 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), pp.72–81, 2007.
[7] "Rodin — Rigorous open development environment for complex systems." http://rodin.cs.ncl.ac.uk/, Aug. 2008 (Last Access).
[8] "VDM information web site." http://www.vdmtools.jp/, April 2008 (Last Access).
[9] "Bmethod: Presentation of BMethod, B language, and formal methods." http://www.bmethod.com/, April 2008 (Last Access).
[10] "SPIN — Formal verification." http://spinroot.com/spin/whatispin.html, April 2008 (Last Access).
[11] "Formal systems (Europe) Ltd." http://www.fsel.com/index.html, April 2008 (Last Access).
[12] "Java PathFinder." http://javapathfinder.sourceforge.net/, April 2008 (Last Access).
[13] M.H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, and M. Sebastianis, "A case study on the automated verification of groupware protocols," ICSE'05, Proc. 27th International Conference on Software Engineering, pp.596–603, New York, NY, USA, 2005.
[14] P. Behm, P. Benoit, A. Faivre, and J.M. Meynadier, "Météor: A successful application of B in a large project," World Congress on Formal Methods in the Development of Computing Systems (FM'99), p.712, 1999.
[15] F. Badeau and A. Amelot, "Using B as a high level programming language in an industrial project: Roissy VAL," ZB 2005: Formal Specification and Development in Z and B, pp.334–354, 2005.
[16] I. Oliver, "Experiences in using B and UML in industrial development," 7th International B Conference (B 2007), pp.248–251, 2006.
[17] Y. Zheng, J. Wang, K. Wang, and J. Xue, "Partially introducing formal methods into object-oriented development: Case studies using a metrics-driven approach," Formal Methods 2006 (FM 2006), pp.190–204, 2007.
[18] "J07-BOK." http://www.ipsj.or.jp/12kyoiku/J07/J07contents.html, April 2008 (Last Access).
[19] D. Essamé and D. Dollé, "B in large-scale projects: The Canarsie Line CBTC Experience," 7th International B Conference (B 2007), pp.252–254, 2007.
[20] C. Artho, C. Sommer, and S. Honiden, "Model checking networked programs in the presence of transmission failures," First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE), pp.219–228, 2007.
[21] H. Foster, "LTSA WS-Engineer — Implerial College London." http://www.doc.ic.ac.uk/ltsa/eclipse/wsengineer/, April 2008 (Last Access).

Fuyuki Ishikawa received his Ph.D. degree in Information Science and Technology from The University of Tokyo, Japan, in 2007. Since April 2007, he has been an assistant professor at Digital Content and Media Sciences Research Division, National Institute of Informatics, Japan. He is also an assistant professor at Department of Informatics, School of Multidisciplinary Sciences, The Graduate University for Advanced Studies (Sokendai University), Japan. His research interests include Service-Oriented Computing, Multi-Agent Systems, Ubiquitous/Pervasive Computing and Software Engineering. He is a member of the Association for Computing Machinery (ACM), Computer Society of the Institute of Electrical and Electronics Engineers (IEEE), the Information Processing Society of Japan (IPSJ), and Japan Society for Software Science and Technology (JSSST).

Shinichi Honiden received his Ph.D. degree in electrical engineering from Waseda University, Tokyo, Japan, in 1986. From 1978 to 2000 he was with Toshiba Corporation. Since April 2000, he has been a professor and a director, Information Systems Architecture Research Division, National Institute of Informatics, Japan. He has also been a professor in the Graduate School of Information Science and Technology, The University of Tokyo since April 2001. He has been a visiting professor of Waseda University since April 2006. He was a visiting researcher of University College London and Imperial College, London, from 2002 to 2003. He was an invited professor at Le Laboratoire d'Informatique de Paris 6, Pierre et Marie Curie in 2006. His research interests include agent technology, pervasive computing, and software engineering. Prof. Honiden is a member of the Institute of Electrical and Electronics Engineers (IEEE), the Information Processing Society of Japan (IPSJ), the Association for Computing Machinery (ACM), the Japanese Society for Artificial Intelligence (JSAI), the Japan Society for Software Science and Technology (JSSST).