

A Tool for Automated Test Code Generation from High-Level Petri Nets

Dianxiang Xu

National Center for the Protection of the Financial Infrastructure,
Dakota State University
Madison, SD 57042, USA
dianxiang.xu@dsu.edu

Abstract. Automated software testing has gained much attention because it is expected to improve testing productivity and reduce testing cost. Automated generation and execution of tests, however, are still very limited. This paper presents a tool, ISTA (Integration and System Test Automation), for automated test generation and execution by using high-level Petri nets as finite state test models. ISTA has several unique features. It allows executable test code to be generated automatically from a MID (Model-Implementation Description) specification - including a high-level Petri net as the test model and a mapping from the Petri net elements to implementation constructs. The test code can be executed immediately against the system under test. It supports a variety of languages of test code, including Java, C/C++, C#, VB, and html/Selenium IDE (for web applications). It also supports automated test generation for various coverage criteria of Petri nets. ISTA is useful not only for function testing but also for security testing by using Petri nets as threat models. It has been applied to several industry-strength systems.

Keywords: Petri nets, predicate/transition nets, software testing, model-based testing, security testing.

1 Introduction

Software testing is an important means for quality assurance of software. It aims at finding bugs by executing a program. As software testing is labor intensive and expensive, it is highly desirable to automate or partially automate testing process. To this end, model-based testing (MBT) has recently gained much attention. MBT uses behavior models of a system under test (SUT) for generating and executing test cases. Finite state machines [9] and UML models [5] are among the most popular modeling formalisms for MBT. However, existing MBT research cannot fully automate test generation or execution for two reasons. First, tests generated from a model are often incomplete because the actual parameters are not determined. For example, when a test model is represented by a state machine or sequence diagram with constraints (e.g., preconditions and postconditions), it is hard to automatically determine the actual parameters of test sequences so that all constraints along each test sequences are satisfied [9]. Second, tests generated from a model are not immediately executable

because modeling and programming use different languages. Automated execution of these tests often requires implementation-specific test drivers or adapters.

To tackle these problems, this paper presents ISTA (Integration and System Test Automation)¹, a tool for automated generation of executable test code. It uses high-level Petri nets for specifying test models so that complete tests can be generated automatically. It also provides a language for mapping the elements in test models (Petri nets) to implementation constructs. This makes it possible to convert the model-based tests into code that can be executed against the SUT.

ISTA can reduce a lot of testing workload by supporting various testing activities, such as the following:

- **Functional testing:** ISTA can generate functional tests to exercise the interactions among system components.
- **Security testing:** ISTA can generate security tests to exercise potential insecure behaviors.
- **Regression testing:** Regression testing is conducted when system requirements or implementation are changed. If test cases are not completely generated, tester needs to determine whether they have become invalid and whether they have to be changed. Using ISTA, however, only needs to change the specification for test generation.

The remainder of this paper is organized as follows. Section 2 introduces ISTA's architecture, its input language MID (Model-Implementation Description), and test generation features. Section 3 reports several real world applications of ISTA. Section 4 reviews and compares related work. Section 5 concludes this paper.

2 Automated Test Code Generation in ISTA

2.1 Architecture

Figure 1 shows the architecture of ISTA. The input to ISTA is a MID specification, consisting of a Predicate/Transition (PrT) net, a MIM (model-implementation-mapping), and HC (helper code). Section 2.2 will elaborate on MID.

The main components of ISTA are as follows:

- **MID editor:** a graphic user interface for editing a MID specification. A MID file can also be edited directly through MS Excel.
- **MID parsers:** The PrT net parser (2.1 in Figure 1) is used to parse the textual representation of the PrT net in a MID specification. The MIM parser (2.2 in Figure 1) is used to parse the MIM specification and helper code. The parsers check to see if there are any errors in the given specification.
- **Reachability analyzer:** it verifies whether or not a given goal marking can be reached from the initial marking in the PrT net of a given MID specification. The reachability analysis can help detect problems of the specification. For example, if a goal marking is known to be reachable (or unreachable) from the initial marking but the reachability analysis reports a different result, then

¹ Interested readers may contact the author for an evaluation version.

the PrT net models is not specified correctly (e.g., a precondition is missing). Reachability verification is conducted through the fast planning graph analysis adapted for the particular PrT nets in ISTA [8].

- Test generator: it generates model-level tests (i.e., firing sequences) according to a chosen coverage criterion. The tests are organized and visualized as a transition tree, allowing the user to debug MID specification and manage the tests before generating executable test code. ISTA supports a number of coverage criteria for test generation from PrT nets, including reachability graph coverage, transition coverage, state coverage, depth coverage, and goal coverage (to be discussed in Section 2.3).
- Test manager: it allows the user to manage the generated tests, such as export tests to a file, import tests from a file, delete tests, change the ordering of tests, redefine actual parameters of tests, and insert extra code into tests.
- Test code generator: it generates test code in the chosen target language from a given transition tree. Currently, ISTA supports the following languages/test frameworks: Java/JUnit (or Jfcunit), C/C++, C#/NUnit, VB, and html/Selenium IDE. JUnit is a unit test framework for Java programs. Jfcunit is an extension to JUnit for GUI testing of Java programs. NUnit, similar to JUnit, is a unit test framework for C# programs. Selenium IDE (a Firefox plugin) is a test framework for web applications and thus ISTA is applicable to all web applications that can be tested with Selenium IDE, no matter what language is used to implement the web application (e.g., php or perl).

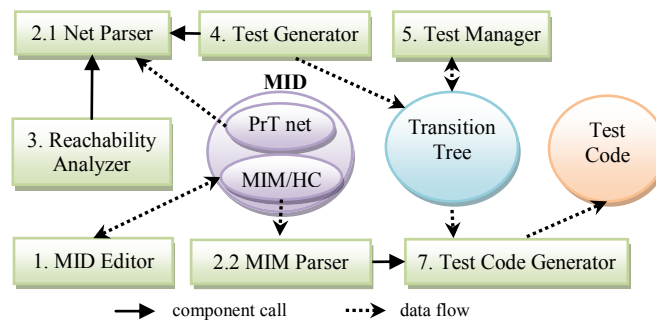


Fig. 1. The architecture of ISTA

For a complete MID specification, executable test code can be generated by just one click; it can be executed immediately against the SUT.

2.2 MID

PrT Nets as Test Models. PrT nets in ISTA are a subset of traditional PrT nets [2] or colored Petri nets [3]², where all weights in each formal sum of tokens or arc labels

² ISTA provides an interface to import CPN files. So CPN (<http://cpntools.org/>) can be used as a graphical editor for the PrT nets. Fig. 7 shows an example.

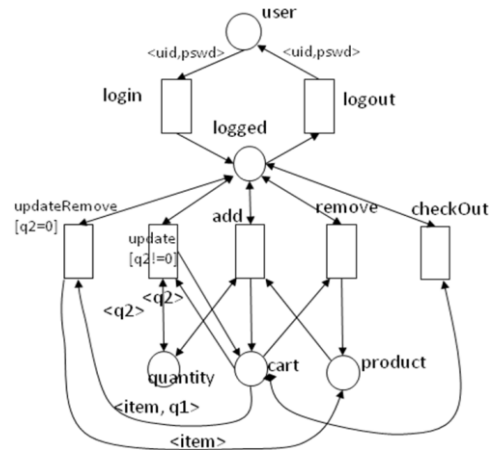


Fig. 2. Petri net for function testing of Magento

are 1. The formal definition of such PrT nets can be found in our prior work [7],[8]. The structure of such a PrT net can be represented by a set of transitions, where each transition is a quadruple $\langle \text{event, precondition, postcondition, guard} \rangle$. The precondition, postcondition, and guard are first-order logic formulas. The precondition and postcondition are corresponding to the input and output places of the transition, respectively. This forms the basis of textual descriptions of PrT nets in ISTA. Fig. 2 shows a function test model for Magento (an online shopping application to be discussed in Section 3.2). The functions to be tested include log in, log out, add items to the shopping cart, remove items from the shopping cart, update the shopping cart, and check out. For clarity, most of the arc labels are omitted. For example, the arcs between login and logged and between logged and add are actually labeled by $\langle \text{uid, pswd} \rangle$, respectively. Fig.3 presents its textual representation³.

ISTA supports inhibitor arcs and reset arcs. An inhibitor arc represents a negative precondition of the associated transition. A reset arc represents a reset postcondition of the associated transition – it removes all tokens from the associated place after the transition is fired. In an online shopping system, for example, checkout empties the shopping cart no matter how many items were in the cart.

A PrT net for a function test model usually focuses on the normal behaviors of the system components. PrT nets can also be used to model security threats, which are potential attacks against a SUT. To do so, we introduce a special class of transitions, called attack transitions [7]. They are similar to other transitions except that their names start with “attack”. When a PrT net is a threat model, we are primarily interested in the firing sequences that end with the firing of an attack transition. Such a firing sequence is called an attack path, indicating a particular way to attack a SUT.

³ As indicated in Fig. 3, ISTA allows a test model to be specified as a set of contracts (preconditions and postconditions in first-order logic) or a finite state machine. Nevertheless, PrT nets are a unified representation of test models in ISTA. It automatically transforms the given contracts or finite state machine into a PrT net.

No	Transition	Precondition	Postcondition	When
1	login(uid, pswd)	user(uid, pswd)	logged(uid, pswd)	
2	logout(uid, pswd)	logged(uid, pswd)	user(uid, pswd)	
3	addItem(item, q)	logged(uid, pswd), product(item), quantity(q)	logged(uid, pswd), cart(item, q)	not equals(q, 0)
4	removeItem(item, q)	logged(uid, pswd), cart(item, q)	logged(uid, pswd), product(item)	
5	updateItem(item, q1, q2)	logged(uid, pswd), cart(item, q1), quantity(q2)	logged(uid, pswd), cart(item, q2), quantity(q2)	not equals(q2, 0)
6	updateRemove(item, q1)	logged(uid, pswd), cart(item, q1), quantity(q2)	logged(uid, pswd), product(item), quantity(q2)	equals(q2, 0)
7	checkout(uid, pswd)	logged(uid, pswd), cart(item, q)	logged(uid, pswd), reset(cart)	

Initial State [1]
user(UID, PSWD)

Fig. 3. Textual representation of the PrT net in Fig. 2

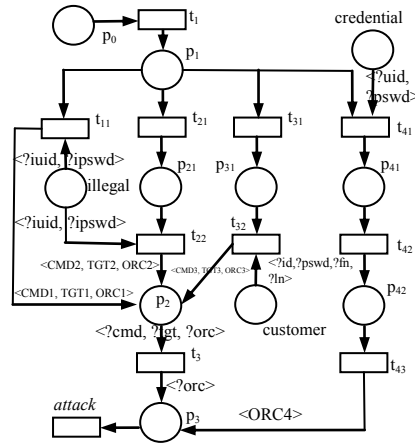


Fig. 4. Petri net for a threat model of Magento

We have demonstrated that PrT nets can be used to capture various security threats [7], such as spoofing, tampering with data, repudiation, information disclosure, denial of service, and elevation of privilege.

Fig. 4 shows a PrT net that models a group of XSS (Cross Site Scripting) threats, which are one of the top security risks in web applications. The threat model captures several ways to exploit system functions by entering a script into an input field, such as email address, password, or coupon code. The functions are log in (t1, t11, t3), create account (t1, t21, t22, t3), forgot password (t1, t31, t32, t3), and shopping with discount coupon (t1, t41, t42, t43, t3). Using formal threat models for security testing can better meet the need of security testing to consider “the presence of an intelligent adversary bent on breaking the system” [10].

MIM (Model-Implementation Mapping). A MIM specification for a test model mainly consists of the following elements:

- The identity of the SUT (URL of a web application, class name for an object-oriented program, or system name for a C program).
- A list of hidden predicates in the test model that do not produce test code (e.g., test code may not be needed for a predicate that represents an ordering constraint in the test model).
- A mapping from the objects (elements in tokens) in the test model to implementation objects (object mapping).
- A mapping from events/transitions in the test model to implementation code (method mapping).
- A mapping from the predicates (places) in the test model to implementation code for verifying the predicates in the SUT (called accessor mapping for testing object-oriented programs).
- A mapping from the predicates in the test model to implementation code for making the predicates true (e.g., for setting up an option or preference) in the SUT (called mutator mapping for testing object-oriented programs).

The screenshot shows a software interface for MIM specification. It includes a 'URL' field with 'http://www.example.com/magento/index.php/', a 'Hidden' list with 'user, logged, product, quantity, cart', and an 'Objects' table mapping model-level objects to implementation objects. Below these is a 'Method Mapping' table with 18 rows.

No	Model-Level Object	Implementation Object
1	UID	xu001@gannon.edu
2	PSWD	password
3	SONYLAPTOP	Sony VAIO VGN-TXN27N/B 11.1" Notebook PC

9	login(?uid, ?pswd)	clickAndWait	link=Log In
10	login(?uid, ?pswd)	waitForElementPresent	//form[@id='login-form']/div/div[2]/div[1]/h4
11	login(?uid, ?pswd)	type	email ?uid
12	login(?uid, ?pswd)	type	pass ?pswd
13	login(?uid, ?pswd)	clickAndWait	send2
14	logout(?uid, ?pswd)	clickAndWait	link=Log Out
15	additem(?item, ?q)	clickAndWait	//img[@alt='Magento Commerce']
16	additem(?item, ?q)	clickAndWait	link=?item
17	additem(?item, ?q)	type	qty ?q
18	additem(?item, ?q)	clickAndWait	//button[@onclick='productAddToCartForm.submit()']

Fig. 5. Portion of a MIM specification

Fig. 5 shows portion of the MIM for the test model in Fig. 2. UID and PSWD are two objects in the test model, representing user id and password. When they appear in a test case, they refer to xu001@gannon.edu and password in the SUT, respectively. Rows 9-18 are part of the method mapping. Rows 9-13 are Selenium IDE commands for login, and row 14 is the Selenium IDE command for logout. These commands can be easily obtained by using the record function of Selenium IDE.

Helper Code. Helper code refers to the code that needs to be provided by tester in order to make the generated test code executable. It includes the header (e.g., package/import statements in Java and #include statements in C), alpha/omega segments (invoked at the beginning/end of a test suite), setup/teardown methods (invoked at the beginning/end of each test case), and extra code to be inserted into the generated test code (called local code).

2.3 Test Generation

Test generation consists of two components: test sequence generation and test code generation. Test sequence generation produces a test suite, i.e., a set of test sequences

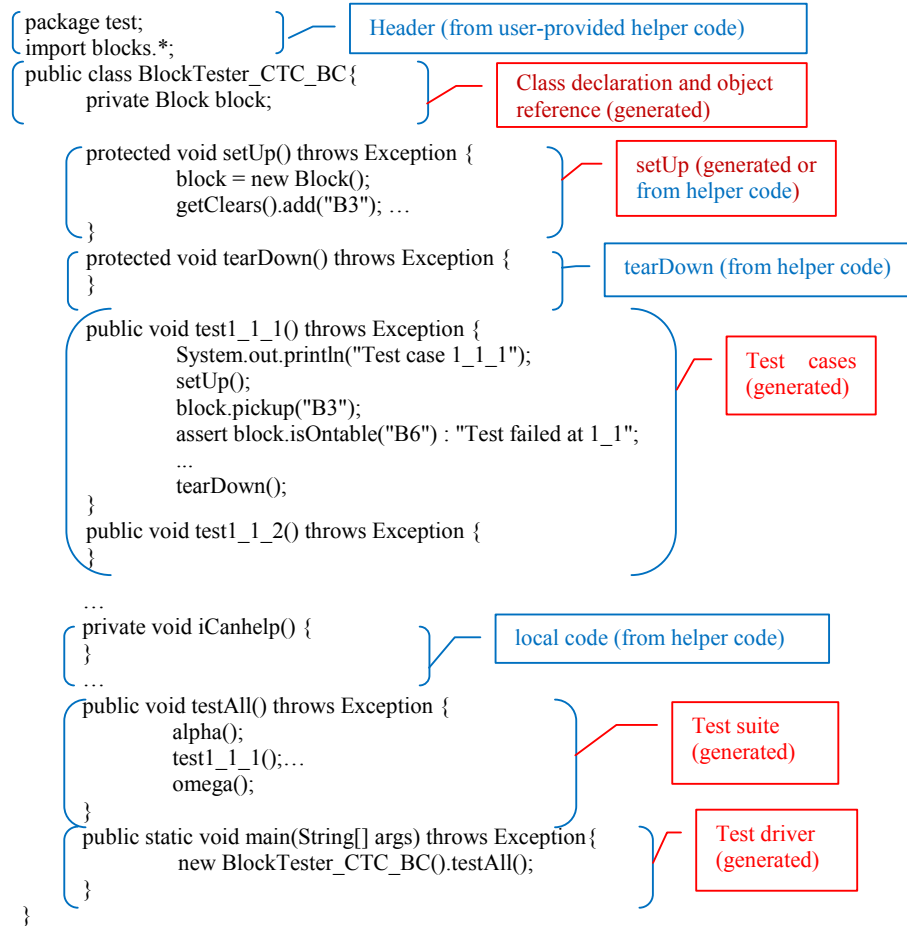


Fig. 6. Structure of the test code in Java

(firing sequences) from a test model according to a chosen coverage criterion. The test sequences are organized as a transition tree (also called test tree). The root represents the initial state (resulting from the new operation, like object construction in an object-oriented language). Each path from the root to a leaf is a firing sequence.

Given a finite state test model, ISTA can generate a transition tree to meet the following criteria:

- **Reachability graph coverage:** the transition tree actually represents the reachability graph of the PrT net for a function test model. If the PrT net is a threat model, however, the transition tree consists of all attack paths, i.e., firing sequences that end with the firing of an attack transition.
- **Transition coverage:** each transition in the PrT net is covered by at least one firing sequence in the transition tree.

- State coverage: each state (marking) reachable from the initial marking is covered by at least one firing sequence in the transition tree.
- Depth coverage: the transition tree consists of all firing sequences whose size (number of transition firings) is no greater than the given depth.
- Goal coverage: the transition tree consists of a firing sequence for each of the goal markings reachable from the initial marking.

Test code generation is to convert a transition tree to test code according to the MIM specification and helper code. The entire tree represents a test suite and each firing sequence from the root to a leaf is a test case. Fig. 6 shows the general structure of Java test code. It is similar for other object-oriented languages(C#, C++, and VB). The structure for a procedural (C) or scripting language (html) is much simpler.

3 Applications

3.1 Self-Testing of ISTA

ISTA is implemented in Java. It has been developed in an incremental fashion - new functions and examples were added from time to time. The current version has 35K LOC (lines of code) in 195 classes. Every time ISTA was updated, we had to run all the examples to check if the changes have inadvertently affected the existing code. Such regression testing would be very tedious if it were not automated. We created a test model for ISTA itself so that it can generate executable test code to run all the examples with various system options (e.g., coverage criteria). Fig. 7 shows the test model edited with CPN, where for clarity the initial marking is simplified. The automated self-testing not only saved us a lot of time, but also helped us find various problems in the development process.

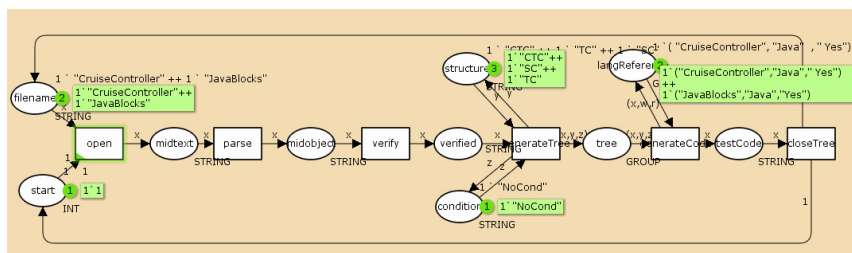


Fig. 7. PrT net for the function testing of ISTA

3.2 Function and Security Testing of Online Shopping Systems

ISTA has been applied to two real-world online shopping systems, Magento⁴ and Zen-cart⁵. They are both implemented in php. Magento has 52K LOC whereas Zen-cart has 96K LOC. They both are being used by many live online stores. For

⁴ www.magentoocommerce.com

⁵ www.zen-cart.com

Magento, more than 2,100 function tests with a total of 608K line of html code were generated and executed. For Zen-cart, more than 20,000 function tests (12 million lines of html code) were generated and executed. We have also built 19 PrT nets for various security threats to Magento. They produced 103 security tests that killed 56 of the 62 security mutants of Magento created according to OWASP's top 10 security risks of web applications and the security requirements of online shopping. A security mutant is a variant of the original version with a vulnerability injected deliberately. A test is said to kill a security mutant if it is a successful attack against the mutant.

3.3 Security Testing of an FTP Server Implementation

FTP (File Transfer Protocol) is a widely used method for working with remote computer systems and moving files between systems. FileZilla Server⁶ is a popular FTP server implementation. Currently, it is the seventh most downloaded program on SourceForge⁷. FileZilla Server version 0.9.34 used in our study has 88K lines of C++ code in 107 classes. We created 8 PrT nets to specify various security threats to FileZilla Server. 76 security tests were generated. They killed 35 of the 38 security mutants of FileZilla Server created according to the security requirements of FTP services and the common vulnerabilities of C++ programs.

4 Related Work

ISTA is related to testing with Petri nets and MBT. Zhu and He [11] have proposed a methodology for testing high-level Petri nets. The methodology consists of four testing strategies: transition-oriented testing, state-oriented testing, data flow-oriented testing, and specification-oriented testing. Each strategy is defined in terms of an adequacy criterion for selecting test cases and an observation scheme for observing a system's dynamic behavior during test execution. It is not concerned with how tests can be generated to meet the adequacy criteria. Desel et al. [1] have proposed a technique to generate test inputs (initial markings) for the simulation of high-level Petri nets. Wang et al. [6] have proposed class Petri net machines for specifying inter-method interactions within a class and generating method sequences for unit testing. Manual work is needed to make the sequences executable.

There is a large body of literature in MBT, particularly with finite state machines and UML models. Existing MBT tools are limited in the generation of executable tests. We refer the reader to [9] for a brief survey of MBT research. To the best of our knowledge, ISTA is a unique tool for generating executable test code in various languages from the specifications of complex SUTs. It is the only tool that can generate executable security tests from formal threat models. These features are made possible because of the expressiveness of high-level Petri nets for capturing both control and data flows of complex software.

⁶ <http://filezilla-project.org/>

⁷ <http://sourceforge.net/top/topalltime.php?type=downloads>

5 Conclusions

We have presented the ISTA tool for the automated generation of executable tests by using high-level Petri nets to build the test models. Using ISTA for its own testing has proven to be an effective approach in its incremental development process. As demonstrated by several case studies, ISTA is applicable to both function testing and security testing of various software systems. It can improve testing productivity and reduce testing cost due to automated generation and execution of tests.

ISTA has been adopted by a globally leading company in high-tech electronics manufacturing and digital media. We are currently collaborating on integrating ISTA with their testing process and extending ISTA to generate test code for concurrent programs. With the advances in multi-core technologies, concurrent software is becoming more and more ubiquitous (e.g., in cell phones and home appliances). As an excellent formalism for modeling concurrent systems, Petri nets are expected to play an important role in quality assurance of concurrent software.

Acknowledgments. This work was supported in part by the National Science Foundation of USA under grant CNS 0855106. Dr. Weifeng Xu, Michael Sanford, Lijo Thomas, Daniel Woodraska, and Jayanth Gade contributed to the application, experimentation, and documentation of ISTA.

References

1. Desel, J., Oberweis, A., Zimmer, T., Zimmermann, G.: Validation of Information System Models: Petri Nets and Test Case Generation. In: Proc. of SMC 1997, pp. 3401–3406 (1997)
2. Genrich, H.J.: Predicate/Transition Nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 254, pp. 207–247. Springer, Heidelberg (1987)
3. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer* 9, 213–254 (2007)
4. Murata, T.: Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE* 77(4), 541–580 (1989)
5. Utting, M., Legeard, B.: *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, San Francisco (2006)
6. Wang, C.C., Pai, W.C., Chiang, D.-J.: Using Petri Net Model Approach to Object-Oriented Class Testing. In: Proc. of SMC 1999, pp. 824–828 (1999)
7. Xu, D., Nygard, K.E.: Threat-Driven Modeling and Verification of Secure Software Using Aspect-Oriented Petri Nets. *IEEE Trans. on Software Engineering*. 32(4), 265–278 (2006)
8. Xu, D., Volz, R.A., Ioerger, T.R., Yen, J.: Modeling and Analyzing Multi-Agent Behaviors Using Predicate/Transition Nets. *International Journal of Software Engineering and Knowledge Engineering* 13(1), 103–124 (2003)
9. Xu, D., Xu, W., Wong, W.E.: Automated Test Code Generation from Class State Models. *International J. of Software Engineering and Knowledge Engineering* 19(4), 599–623 (2009)
10. Xu, D.: Software Security. In: Wah, B.W. (ed.) *Wiley Encyclopedia of Computer Science and Engineering*, vol. 5, pp. 2703–2716. John Wiley & Sons, Inc, Hoboken (2009)
11. Zhu, H., He, X.: A Methodology for Testing High-Level Petri Nets. *Information and Software Technology* 44, 473–489 (2002)