Kỷ yếu Hội nghị KHCN Quốc gia lần thứ XII về Nghiên cứu cơ bản và ứng dụng Công nghệ thông tin (FAIR); Huế, ngày 07-08/6/2019
DOI: 10.15625/vap.2019.00051

SECOND ORDER MUTATION TESTING FOR LUSTRE PROGRAMS

Le Van Phol[1] , Nguyen Thanh Binh[2], Le Thi Thanh Binh[3]
[1] Information Technology Department, Travinh University, Travinh, Vietnam
[2] The University of Danang - University of Science and Technology, Danang, Vietnam
[3] Quang Nam University, Quang Nam, Viet Nam

phol@tvu.edu.vn, ntbinh@dut.udn.vn, thanhbinh.le@qnamuni.edu.vn

ABSTRACT: *Lustre is synchronous language, widely used for the development of reactive systems, control systems and monitoring systems, such as nuclear reactors, civil aircraft, automobile vehicles... In particular, Lustre is suitable for developing real-time systems. In such applications, testing activities for fault detection play a very important role. Mutation testing is one of the most commonly used techniques for evaluating the probability of fault detection of test data. However, there are three main challenges: (1) a large number of mutants (leading to a very high computational cost); (2) real fault simulation by generated mutants; and (3) equivalent mutant problems. There are many different approaches which have been proposed for overcoming the above-mentioned problems of mutation testing. Each solution has its own advantages and disadvantages. Among them, higher order mutation testing, which was first introduced in 2009, is the only approach abling to address of all three mutation testing problems, and the only one to deal with the problem of realism of injected defects. In this paper, we propose an application of second order mutation testing for Lustre program. The experimentation of second order mutation testing on some Lustre programs shows the better result in terms of mutation score compared to first order mutation testing. This opens a promising direction in improving mutation testing for Lustre programs.*

Keywords: *Mutation testing, Second order mutation, Reactive system, Lustre language.*

I. INTRODUCTION

Software testing is one of important steps in software development process, it consumes an enormous proportion (sometimes as much as 50%) of the effort for developing a system [1]. In particular, some critical systems require high reliability and safety, like reactive systems, so testing such systems costs more time and resource.

A reactive system is a system which reacts to events that are produced by an external environment. Reactive systems are core components of many safety critical systems, such as the ABS system of cars, flight control systems in aircrafts... [2]. Synchronous programming languages are often used to develop reactive systems.

Synchronous programming [3] was introduced in the late 1980s as an approach to the design of reactive systems. Some synchronized programming languages have been developed, such as Esterel, Signal or Lustre [4]. Among these languages, Lustre [5] is a synchronous data flow language, designed in 1984 by the IMAG Institute in Grenoble. Lustre is widely used for building models, control system designs in a number of industrial fields such as electronics, automobiles and power.

Mutation testing [6] is a technique for evaluating the capability of test data to uncover faults made by programmers. Mutation testing generates the faulty versions (mutants) from the original program based on a set of mutation operators. The goal is to select a set of test data that is capable to detect (kill) mutants and, hence, to perform a quality assessment of that set of test data.

At the same time, many studies are proposed to improve the quality of the application of mutation testing. In particular, the recently proposed higher order mutation (HOM) [7] allows solving mechanical limitations of the traditional mutation testing, such as the large number of mutants or the equivalent mutants.

High order mutation testing as mentioned above is a solution that is used to improve the effectiveness of mutation testing. Instead of using only a simple change in first order mutation testing, higher order mutants [8] are generated by inserting two or more faults into original program to generate mutants. There are many recent studies on higher order mutation, these studies are divided into two groups [7]: Second Order Mutation Testing (SOM) and Higher Order Mutation Testing (HOM).

Recently, in our previous work, we studied mutation testing for Lustre applications, initial research results were presented in [20] [21].

In [20], we proposed the set of mutation operators, the mutation testing process, the manual generation of mutants, the random generation of test data for Lustre programs. The big limitation in our first work on mutation testing for Lustre programs is to generate manually mutants, this is costly in terms of time.

To solve this problem, in [21], we have developed the automatic generator of mutants. Then, we have done the experimentation on more and bigger Lustre programs. However, the experimentation showed that mutation score was low and number of alive mutants was high.

As mentioned above, higher order mutation was showed as effectiveness in terms of detecting equivalent mutants and simulating real faults. In this paper, we propose to apply higher order mutation testing, specifically, second order mutation testing for Lustre programs. We reuse the set of mutation operators proposed in [20] for Lustre and apply different strategies to generate second order mutants. The experimentation will be conducted on a set of Lustre programs and the result will be compared with the experimental result of first order mutation.

The paper is organized into 5 sections. The next section presents the background of the mutation testing, the Lustre language, and the higher order mutation testing. The proposed second order mutation testing for Lustre programs with random test data is presented in Section 3. The experimentation and analysis on SOM and FOM for some Lustre programs are showed in Section 4. Finally, the paper finishes by the conclusion and future work.

II. BACKGROUND

A. Mutation testing

Mutation testing was proposed in the 1970s by Richard DeMillo [6]. Mutation operators are applied to the original program to generate mutants (faulty programs). Then, the original and mutant program are executed on the same set of test data. If the outputs of the original and mutant programs are different, the mutant will be killed (by the test data set), otherwise, the mutant is alive.

Mutation testing is based on two hypotheses: the competent programmer hypothesis and the coupling effect [6]. Theoretical and empirical research has shown that mutation testing is an effective way to evaluate the quality of test data [10]. However, mutation testing has some limitations, such as big number of generated mutants, equivalent mutant problem…

In Figure 1, we give an example for a FOM by replacing operator ">" by operator ">=".

Orignal program (P)	**First order mutant (P')**

node sum(a,b:int) returns (s:int);
let
s= if (a>b) then a+b else a-b;
tel;
	node sum(a,b:int) returns (s:int);
	let
	s= if (**a**>=b) then a+b else a-b;
	-- replace > by >=
	tel;

Figure 1. Example of first order mutant.

In this example, if the test data {a = 1, b = 1} are used then this mutant is killed because output s in the original program is 0 while output s in the mutant program is 2. However, if we use the test data {a = 0, b = 0}, this mutant is not killed because both mutant and original program produce the same output.

To evaluate the quality of a set of test data T for a program P, the mutation score is defined as follows: $MS(P,T) = \frac{D}{M-E}$, where, D is the number of mutants killed; M is the number of generated mutants; E is the number of equivalent mutants.

Mutation score indicates that the best set of test data (that is, the MS equals to 1) will kill all mutants, i.e. it reveals all faults in the program.

B. The Lustre Language

Lustre [4][5] is a data flow synchronous specification and programming language. Lustre program is described by a network of nodes, represented by the relationships between input and output flows. These relationships are expressed by operators, intermediate variables and constants. Inputs, outputs, variables and constants are represented by data flows such as (e0, e1, e2, ...).

Lustre is widely used for the development of reactive systems, control systems and monitoring systems, such as nuclear reactors, civil aircrafts, automobile vehicles, etc. In particular, Lustre is suitable for the development of real-time systems [5].

All variables in a Lustre program must be declared explicitly with a specified data type. There are three basic types of data, including logic, real numbers, integers (bool, real, int) with their associated arithmetic operations: +, -, *, /,

mod, div; logical operators: and, or, not, xor; relational operations: =, <, <=,>,> =, <> and conditional expression if then else.

In addition, the Lustre language has three temporal operators: *precedence*, *initialization*, and clock operator.

The *precedence* operator is denoted by *pre*(). The *pre*() operator denotes the value of the argument on the previous clock cycle, so it will return the value *nil* at the first clock cycle.

The *initialization* operator is denoted by -> or *fby*. It is used to provide the initial value of a data flow. This operator is often used in conjunction with the *pre*() operator because the first string value of a *pre*() expression is *nil*.

Operator *when* used to change the clock frequency. The *when* operator is also called a filter.

They work specifically on data flows and can be used to build more complex new operators.

C. Higher Order Mutation

According to Harman *et al*. [7], mutants are classified into two types: first-order mutants (FOM) and higher-order mutants (HOM). FOM is created by applying only one mutation operator and used in a first-order mutation testing, while HOM uses more than one mutation operator and is used in higher-order mutation testing.

In Figure 2, from original program P, FOM P' is generated by replacing ">" by ">=" from expression "a>b"; HOM P'' is generated by replacing ">" by ">=" from expression "a>b" and "+" by "*" from expression "a+b".

Orignal program (P)	First order mutant (P')	Second order mutant (P'')

node sum(a,b:int) returns (s:int);
let
s= if (a>b) then a+b else a-b;
tel;
	node sum(a,b:int) returns (s:int);
	let
	s= if (**a**>=b) then a+b else a-b;
	-- replace > by >=
	tel;
		node sum(a,b:int) returns (s:int);
		let
		s= if (a>=b) then a*b else a-b;
		-- replace > by >= and "+" by "*"
		tel;

Figure 2. Example of higher order mutant

D. Work related to higher order mutation testing

In the past, mutation testing was understood as inserting only one fault into original program to create mutants. Since higher order mutation testing was first introduced, traditional mutation testing is considered as first order mutation testing. First order mutation takes care only single fault, while most real faults are complex faults [9]. A complex fault is a fault that cannot be solved by making a single change in program. Complex faults are simulated better by higher order mutation testing. In addition, the recent empirical research results showed that higher order mutants have reflected better actual fault [7]. In this paper, we focus on higher order mutation testing.

Higher order mutation testing applies two or more mutation operators to original program to create mutants. The important problems of higher order mutation testing is to reduce the number of mutants and to generate mutants reflecting most real faults of original program, i.e. reducing the cost and increasing the quality of test data set. Many recent researches have investigated higher order mutation testing. They are divided into two groups: Second Order Mutation Testing (SOM) and High Order Mutation Testing (HOM).

Polo *et al.* [11] introduced three different algorithms (Last2First, DifferentOperators and RandomMix) to combine first order mutations (FOMs) to create SOMs. Experimental results showed that their approaches have reduced the number of mutants by about 50%, without reducing the quality of the test set.

The algorithms proposed by Polo *et al*. in [11] have been more studied by Papadakis and Malevris [24] in order to improve the equivalent mutant problem. Their experimentations showed the promising results: the number of equivalent mutants is reduced to 85.65-87.77% and loss of error detection capacity between 11.45-14.57%. In addition, their studies indicate that the number of mutants is decreased to approximately 50%, this deduces the decrement of execution and computation time.

In [12], the authors presented an empirical study of higher order mutation testing. They focused on analyzing second order mutation and they found that SOM achieved a higher coverage compared to HOM. They proposed the SOM testing strategies (*H Dom* and *SDHomF*) and got the most interesting experimental results when applying the

hybrid strategies. Decrease of equivalent mutants varies between 65.5% for the *H Dom*(50%) and 86.8% for the *SDomF* strategy, with a loss of test effitiveness from 1.75% for *HDom*(50%) to 4.2% for *SDomF*.

In [13], the authors proposed to apply the multi objective Parejo optimal genetic programming to generate HOMs. They utilized to fitness functions: semantic difference and syntactic difference. Their experimental results showed that realistic HOMs were harder to be killed than first order mutants.

Besides, in [14], [15], [16], [17], [18], [19], the authors proposed a classification of higher order mutants that cover all possible cases of generated mutants. These higher order mutants are created by combining the first order mutants, and they are more difficult to be killed than the first order mutants creating them and a set of test cases that can kill these higher order mutants will also kill all first order mutants creating them. These higher order mutants can replace all of their first order mutants without diminishing the effectiveness of mutation testing. This allows to apply higher order mutation testing to solve problem the equivalent mutant problem and describing the actual faulty of a program.

III. SECOND ORDER MUTATION TESTING FOR LUSTRE PROGRAMS

In [20], [21], the results of experimentations with random test data showed some problems, such as low mutation score, big number of alive mutants and equivalent mutant problem.

In this work, we focus on improving mutation score by simulating better the actual faults by generated mutants. In order to tackle this problem, we propose to apply higher order mutation to Lustre programs. Concretely, in this paper, we apply the second order mutation strategies to generate mutants for Lustre programs. In this section, we first present the second order mutant generator and then the process of second order mutation testing.

A. Generation of second order mutants for Lustre programs

We propose the schema to generate second order mutants for Lustre programs by combining first order mutants (Figure 3). In this schema, FOMs are generated by applying mutation operators proposed in our previous work [20], ơ21]. Next, we apply the algorithms for combining FOMs to create HOMs. In this paper, we first implemented the DifferentOperators algorithm [11] for the experimentation, i.e. two FOMs generated by two different mutation operators are used to generate SOMs.

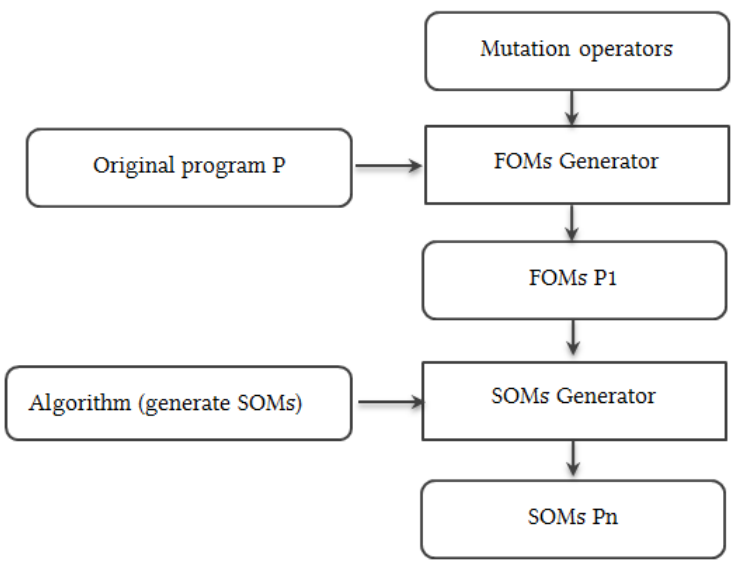

Figure 3. Second order mutants generator architecture

B. Second order mutation testing for Lustre programs

In this subsection, we present second order mutation testing process for Lustre programs in Figure 4.

In this process, both generated SOM and orginal program are executed on the same set of test data. If the output of the mutant program is different from the output of the original program, the mutant is killed. If the output of the mutant is identical to the original program, this mutant may be equivalent or can be killed by a better test data set. In this latter case, the test data set should be adjusted and repeated until testing is sufficient to kill all mutants or reach the threshold of mutation score.

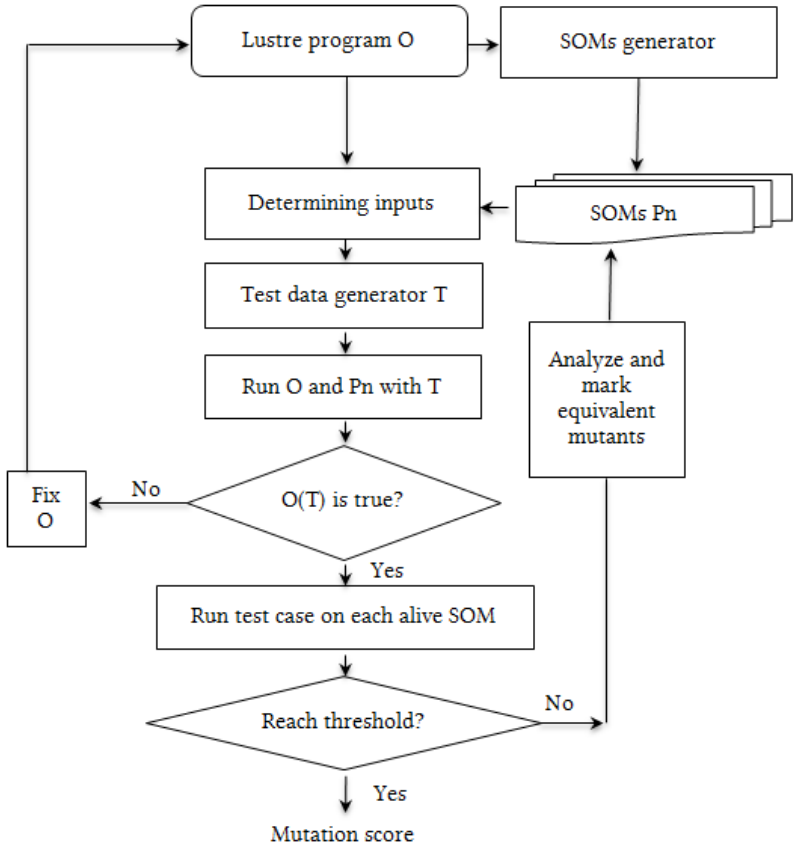

Figure 4. Second order mutation testing for Lustre programs

IV. EXPERIMENTATION

In this section, two experimentations are presented. In the first one, we applied the proposed solution of second order mutation testing for program *AC_Controller* written in Lustre language. Then, we compared with the experimentation conducted in the previous work for first order mutation testing on the same program *AC_Controller* and the same set of test data in terms of mutation score. In the second experimentation, we applied the proposed solution of second order mutation testing for different programs written in Lustre language to evaluate the number of generated mutants.

In the first experimentation, the set of FOMs was generated for the AC_Controller program, then these FOMs were run on four sets of test data *TS1, TS2, TS3,* and *TS4* to compute the number of killed mutants and the mutation score. Next, the differentOperators algorithm was applied to generate SOMs for the AC_Controller program. The SOMs were executed on the same four sets of test data *TS1, TS2, TS3,* and *TS4*. The equivalent mutants were determined manually. Table 1 shows the results of this experimentation, where *TS* is test data set, *TM* is the number of generated mutants, *KM* is the number of killed mutants, *AM* is the number of alive mutants, and *MS* is the mutation score.

Table 1. Mutation score between fom and som for *Ac_Controller* application

TS	FOM				SOM			
	TM	KM	AM	MS	TM	KM	AM	MS
TS1	29	25	4	86,21%	410	409	1	99,76%
TS2	29	27	2	93,10%	410	408	2	99,51%
TS3	29	26	3	89,66%	410	360	50	87,80%
TS4	29	20	9	68,97%	410	356	54	86,83%

From the experimental results, we can state that the capability of detection of faults of the generated SOMs are better than the generated FOMs in terms of mutation score for all four sets of test data. Based on FOMs, *TS2* is the best set of test data, but *TS1* is the best one based on SOMs. Both FOMs and SOMs show that *TS4* is the worst amongst the four sets.

In the second experimentation, FOMs and SOMs were generated for four different Lustre programs. The results are presented in Table 2 The first experimentation shows that SOMs are better than FOMs in terms of fault detection, however, the second one shows that the number of SOMs is much higher than the number of FOMs. This means that SOM increases the test cost. In the next step of this research, we intend to continue studying the reduction of number of generated SOMs but keep the quality of SOMs.

Table 2. Generated FOM and SOM

Program	**Number of expressions**	**Number of FOMs**	**Number of SOMs**
Shutter_Controller	12	58	1.282
CarLights	19	90	3.384
Stopwatch	20	86	5.618
Fibonacci	7	17	104

V. CONCLUSION

Mutation testing is applicable to many different programming languages, but it has not been fully researched to apply to the Lustre language. In our project, we intend to develop a whole framework of mutation testing for Lustre programs. Previously, we already proposed a set of mutation operators [20] and we developed the mutant generator to automate a part of mutation testing process and reduce the test cost [21].

Along with the results of our previous studies, we find that more research is needed to contribute and improve mutation testing techniques for Lustre program. Through this theoretical and empirical study, we understand that it is possible to apply higher order mutation testing techniques to Lustre programs. This is the new research direction that has positive effects on the field of mutation testing in general and the mutation test for Lustre program in particular, making an important contribution to solving difficult problems in mutation testing for Lustre application.

In this paper, we only applied the second order mutation with the DifferentOperators algorithm, and executed mutants manually. In the next step, we will develop an automatic second order mutants generation and execution. We also intend to study solutions to reduce the number of second order mutants for Lustre programs.

REFERENCES

[1]. S. Kadry. A New Proposed Technique to Improve Software Regression Testing Cost. International Journal of Security and Its Applications Vol. 5 No. 3, July, 2011.

[2]. A. Poigne, M. Morley, O. Maffe, L. Holenderski, R. Budde. The Synchronous Approach to designing Reactive Systems. Formal methods in system design, V, 1-25(199x), Kluwer Academic publishers, Boston. Manufacture in the Netherlands.

[3]. A. Benveniste, G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. Procedings of the IEEE, Volume 79, Issue 9, 1991.

[4]. Nicolas Halbwachs. Synchronous Programming of Reactive systems. Kluwer Academic Publishers, Dordrecht, 1993.

[5]. Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language Lustre. Proceedings of the IEEE, Vol. 79, Issue 9, pp. 1305-1320, Sep, 1991.

[6]. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection Help for the practicing programmer. IEEE Computer, 11(4) p. 34-41, Apr, 1978.

[7]. M. Harman, Y. Jia, and W. B. Langdon. A manifesto for higher order mutation testing. Third International Conf. on Software Testing, Verification, and Validation Workshops, 2010.

[8]. Y. Jia, M. Harman, Higher order mutation testing. Inf. Softw. Technol.51, 1379-1393, 2009.

[9]. R. Purushothaman and D. E. Perry. Toward Understanding the Rhetoric of Small Source Code Changes.IEEE Transactions on Software Engineering,31(6):511-526, 2005.

[10]. Y Jia, M Harman. An analysis and survey of the development of mutation testing, IEEE transactions on software engineering, 2011.

[11]. M. Polo, M. Piattini, and I. Garcia-Rodriguez. Decreasing the cost of mutation testing with second-order mutants. Software Testing, Verification, and Reliability, 19 (2):111 -131, 2008.

[12]. M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in Proc. 17th Asia Pacific Software Engineering Conf. (APSEC), pp. 300–309, 2010.

[13]. W. B. Langdon, M. Harman, and Y. Jia. Multi-objective higher order mutation testing with genetic programming. Proceedings Fourth Testing: Academic and Industrial Conference Practice and Research, 2009.

[14]. Quang Vu Nguyen and L. Madeyski. Problems of mutation testing and higher order mutation testing. Advanced Computational Methods for Knowledge Engineering, Advances in Intelligent Systems and Computing, 282, 2014.

[15]. Quang Vu Nguyen and L. Madeyski. Searching for strongly subsuming higher order mutants by applying multiobjective optimization algorithm. Advanced Computational Methods for Knowledge Engineering, Advances in IntelligentSystemsandComputing, 358:391-402,2015.

[16]. Quang Vu Nguyen and L. Madeyski. Higher order mutation testing to drive development of new test cases: an empirical comparison of three strategies. Lecture Notes in Computer Science, vol. 9621, 2016.

[17]. Quang Vu Nguyen and L. Madeyski. On the relationship between the order of mutation testing and the properties of generated higher order mutants. Lecture Notes in Computer Science, 2016.

[18]. Quang Vu Nguyen and L. Madeyski. Empirical evaluation of multiobjective optimization algorithms searching for higher order mutants. Cybernetic and Systems: An International Journal, 47 (1-2), 2016.

[19]. Quang Vu Nguyen and L. Madeyski. Addressing mutation testing problems by applying multi-objective optimization algorithms and higher order mutation. Journal of Intelligent & Fuzzy Systems, vol. 32, No 2, pp. 1173-1182, 2017.

[20]. Le Van Phol, Nguyen Thanh Binh, Ioannis Parissis. 2016. Mutation Operator for Lustre program. In Proceedings of 19th National Conference: Selected Problems about IT and Telecommunication, Vietnam, Oct, 2016.

[21]. Le Van Phol, Nguyen Thanh Binh, Ioannis Parissis. 2017. Mutants Generation For Testing Lustre Programs. In Proceedings of SoICT ’17, Nha Trang City, Viet Nam, Dec, 2017.

KIỂM THỬ ĐỘT BIẾN BẬC HAI CHO CÁC CHƯƠNG TRÌNH LUSTRE

Lê Văn Phol, Nguyễn Thanh Bình, Ioannis Parissis

TÓM TẮT: *Lustre là ngôn ngữ đồng bộ, được sử dụng rộng rãi để phát triển các hệ thống phản ứng, hệ thống điều khiển và hệ thống giám sát, như lò phản ứng hạt nhân, máy bay dân dụng, xe ô tô ... Đặc biệt, Lustre phù hợp để phát triển các hệ thống thời gian thực. Trong các ứng dụng như vậy, các hoạt động kiểm thử để phát hiện lỗi đóng vai trò rất quan trọng. Kiểm thử đột biến là một trong những kỹ thuật được sử dụng phổ biến nhất để đánh giá chất lượng của dữ liệu thử. Tuy nhiên, có ba thách thức chính cần giải quyết: (1) số lượng đột biến thường rất lớn (dẫn đến chi phí kiểm thử thường rất cao); (2) các đột biến được sinh ra có mô tả được các lỗi thực hay không; và (3) các vấn đề về đột biến tương đương. Có nhiều cách tiếp cận khác nhau đã được đề xuất để khắc phục các vấn đề nêu trên cho kiểm thử đột biến. Mỗi một giải pháp đều có ưu điểm và nhược điểm riêng. Trong số đó, kiểm thử đột biến bậc cao, lần đầu tiên được giới thiệu vào năm 2009, là cách tiếp cận duy nhất để giải quyết cả ba vấn đề của kiểm thử đột biến và là cách duy nhất để giải quyết vấn đề về mô tả được các lỗi thực. Trong bài báo này, chúng tôi đề xuất áp dụng kỹ thuật kiểm thử đột biến bậc hai cho các chương trình Lustre. Thử nghiệm kiểm thử đột biến bậc hai được thực hiện trên một số chương trình Lustre cho thấy kết quả tốt hơn về tỷ lệ đột biến so với kiểm thử đột biến bậc một. Điều này mở ra một hướng đi đầy hứa hẹn trong việc cải thiện kiểm thử đột biến cho các chương trình Lustre.*