KIỂM CHỨNG CHƯƠNG TRÌNH DỰA TRÊN SINH ĐIỀU KIỆN KIỂM CHỨNG VÀ CHỨNG MINH ĐỊNH LÝ

Nguyễn Ngọc Cương[1], Nguyễn Trường Thắng[2], Trần Mạnh Đông[2]

[1]Khoa Toán - Tin học, Học viện An ninh nhân dân

[2]Viện Công nghệ thông tin, Viện Hàn lâm Khoa học và Công nghệ Việt Nam

cuongnn.hvan@gmail.com, ntthang@ioit.ac.vn, dongtm@ioit.ac.vn

TÓM TẮT *- Phần mềm ngày càng đóng vai trò rất quan trọng trong hầu hết các lĩnh vực của đời sống xã hội. Từ những trang thông tin điện tử, các hệ thống quản lý nghiệp vụ ngân hàng, các thiết bị di động, các thiết bị y tế, đến các thiết bị gia dụng sử dụng hàng ngày… Ở đâu cũng có sự góp mặt của phần mềm. Tuy nhiên, phát triển và bảo trì phần mềm ngày càng trở nên phức tạp và tốn rất nhiều chi phí. Sự phức tạp và chi phí có thể được giảm nếu trong pha phát triển phần mềm công việc kiểm chứng chương trình được thực hiện bên cạnh đó. Bài báo này trình bày một cách tiếp cận kiểm chứng phần mềm dựa trên hai kỹ thuật sinh điều kiện kiểm chứng và kỹ thuật hình thức chứng minh định lý tự động.*

Từ khóa *- Phương pháp hình thức, kiểm chứng chương trình, chứng minh định lý, điều kiện kiểm chứng.*

I. GIỚI THIỆU

Trong cuộc cách mạng công nghệ thông tin (CNTT) trên thế giới hiện nay, phần mềm là cốt lõi, chiếm tỷ trọng và vai trò ngày càng lớn trong nền kinh tế. Chính vì xu thế này, sự phụ thuộc ngày càng tăng của hoạt động kinh tế-xã hội vào chất lượng phần mềm. Tuy nhiên, có một thực tế là phần mềm và quy trình phát triển phần mềm ngày càng trở nên phức tạp và đắt đỏ. Nguyên nhân là do ngày càng có nhiều yêu cầu khắt khe đặt ra cho phần mềm, gồm các yêu cầu chức năng và phi chức năng. Do vậy, yêu cầu bức thiết của ngành công nghiệp phần mềm hiện đại là việc kiểm soát chất lượng phần mềm. Các phương pháp kiểm chứng phần mềm hình thức (*formal software verification*) là một trong những kỹ thuật đang được giới nghiên cứu cũng như giới công nghiệp tập trung nghiên cứu và phát triển vào lĩnh vực này.

Phương pháp hình thức là một kĩ thuật toán học, thường được hỗ trợ bởi các công cụ, để phát triển hệ thống phần mềm và phần cứng. Sự chặt chẽ của toán học cho phép người dùng phân tích và kiểm tra các mô hình ở bất kì phần nào vòng đời của chương trình: yêu cầu kĩ thuật, đặc tả, kiến trúc, thiết kế, thực thi, kiểm chứng, bảo trì và cải tiến. Bước đầu tiên quan trọng trong đảm bảo nâng cao chất lượng phần mềm là yêu cầu kĩ thuật. Phương pháp hình thức có thể dùng để tìm, ghép nối và miêu tả các yêu cầu. Các công cụ có thể được tự động cung cấp hỗ trợ cần thiết để kiểm tra tính đầy đủ, khả năng truy vết, khả năng kiểm chứng, và khả năng dùng lại và hỗ trợ các yêu cầu phát triển, các quan điểm khác nhau và mâu thuẫn trong quản lý. Vì thế, phương pháp hình thức đã trở thành phương pháp tiếp cận quan trọng để kiểm chứng phần mềm. Các mảng nghiên cứu này bao gồm các phương pháp chủ đạo như:

- Dựa trên kiểm chứng mô hình (*model checking*); (i)
- Dựa trên chứng minh định lý (*theorem proving*) (ii)

Những xu thế trên cũng chính là nguồn gốc của những sản phẩm phục vụ kiểm chứng phần mềm nổi tiếng hiện nay. Đó là SLAM [14], BLAST [13] dành cho mảng nghiên cứu (i), ESC/Java [3,12], HAVOC [15] dành cho mảng nghiên cứu (ii). Thay vì xu thế (i) tập trung xử lý mức mô hình dành sự quan tâm của giới công nghiệp, những kỹ thuật kiểm chứng phần mềm thường do giới học thuật nghiên cứu phát triển lại quan tâm vào các quá trình xử lý ở mức mã nguồn chương trình (source code) hệ thống thay vì mô hình. Do vậy, hiện nay rất ít kỹ thuật kiểm chứng tự động dành cho chương trình. Đây chính là nguyên nhân chính mà nhóm nghiên cứu tại Viện Công nghệ thông tin lựa chọn tập trung tiếp cận nghiên cứu kỹ thuật kiểm chứng chương trình dựa trên kết hợp sinh điều kiện kiểm chứng và chứng minh định lý tự động. Đặc điểm chính của kỹ thuật này là:

1. Chuyển mã nguồn chương trình và các đặc tả trong chú giải (*annotation*) thành công thức (*formulas*);
2. Sinh điều kiện kiểm chứng;
3. Điều kiện kiểm chứng sau khi được sinh ra có thể thực hiện kiểm chứng tính hợp lệ tự động thông qua công cụ chứng minh định lý tự động.

Việc tiếp cận nghiên cứu, thực nghiệm và áp dụng kỹ thuật kiểm chứng chương trình là một hướng đi phù hợp với hoàn cảnh công nghệ phần mềm (CNPM) Việt Nam, tiếp cận nghiên cứu với trình độ tiên tiến trên thế giới để giải quyết những nhu cầu cấp thiết trong xã hội.

II. BỐI CẢNH

A. Các xu thế của kiểm chứng phần mềm hình thức

Tình hình nghiên cứu trong lĩnh vực kiểm chứng phần mềm hiện nay của CNPM trên thế giới có thể tóm tắt vào 2 nhóm chính như sau:

Thứ nhất, **kiểm chứng phần mềm dựa trên kiểm chứng mô hình** (*model checking*) liên quan đến việc xây dựng một mô hình trừu tượng M, trong hình thức biến thể trên ôtômát hữu hạn trạng thái, và xây dựng các công thức đặc tả φ, trong hình thức của các biến thể trên logic thời gian [1]. Bài toán kiểm chứng dựa trên mô hình liên quan đến việc thiết lập các mô hình ngữ nghĩa đòi hỏi các đặc tả:

$$M \models \varphi$$

Các thuật toán kiểm chứng được sử dụng trong kiểm chứng mô hình liên quan đến việc khai thác các tập trạng thái của mô hình để đảm bảo rằng các công thức φ thỏa mãn. Nếu φ là một khẳng định bất biến (invariant), cách tiếp cận kiểm chứng mô hình xác định toàn bộ không gian trạng thái để đảm bảo rằng công thức thỏa mãn trong tất cả các trạng thái. Kỹ thuật này rất phù hợp cho các ứng dụng "control-intensive" nơi phần lớn các mã chương trình phần mềm được viết bằng hình thức cấu trúc điều khiển (ví dụ, lệnh *if*) hoạt động trên các kiểu dữ liệu đơn giản (ví dụ, biến *int*), các ứng dụng bao gồm giao thức truyền thông và bộ điều khiển nhúng,... Ví dụ: SPIN [8], SLAM [14], BLAST [13],…

Thứ hai, nhóm kỹ thuật liên quan tới **chứng minh định lý** (*theorem proving*). Kỹ thuật này dùng để kiểm chứng phần mềm dựa trên các ý tưởng của hai bài báo chuyên đề. Đầu tiên bởi C. A. R. Hoare mô tả một tính toán để lập luận về tính đúng của chương trình về tiền và hậu điều kiện (pre and post conditions) [2]. Thứ hai là E. G. Dijkstra mở rộng ý tưởng Hoare trong khái niệm "chuyển vị từ" (*predicate transformers*), thay vì bắt đầu với một tiền điều kiện và hậu điều kiện, bắt đầu với một hậu điều kiện và sử dụng các mã chương trình để xác định tiền điều kiện cái mà cần để thỏa mãn, để chứng minh các tiền điều kiện [7].

B. Công cụ chứng minh định lý tự động

Chứng minh định lý tự động (*Automated Theorem Proving-ATP*) là một phần của lập luận tự động và logic toán học giải quyết chứng minh một tập có giới hạn của định lý toán học bởi chương trình máy tính. Lập luận tự động thông qua chứng minh toán học là một động lực chính cho sự phát triển của khoa học máy tính. Cách tiếp cận bộ chứng minh định lý có thể suy luận về không gian trạng thái vô hạn và không gian trạng thái liên quan đến các kiểu dữ liệu phức tạp và đệ quy. Điều này có thể đạt được vì một lý do lý bộ chứng minh định lý về những ràng buộc trạng thái, không thể hiện các trạng thái. Các công cụ chứng minh định lý tìm kiếm các chứng minh trong miền cú pháp (*syntactic domain*), mà thường là nhỏ hơn nhiều so với miền ngữ nghĩa (*semantic domain*) tìm kiếm bởi các công cụ kiểm chứng mô hình. Do đó, các công cụ chứng minh định lý rất phù hợp cho các hệ thống lập luận tập trung vào dữ liệu (*data-intensive*) với các cấu trúc dữ liệu phức tạp nhưng luồng thông tin đơn giản. Mặc dù các công cụ chứng minh định lý hỗ trợ phân tích hoàn toàn tự động trong trường hợp bị giới hạn [5], một hệ thống lý luận chứng minh ở mức cao hơn có thể cung cấp một mức độ chấp nhận được của tự động hóa [4]. Lập luận về cấu trúc quy nạp với kích thước tùy ý (ví dụ: cây, danh sách, ngăn xếp) có truy cập được thông qua quy nạp toán học nhưng không thể được tự động. Tuy nhiên, sự cân bằng này là chấp nhận được trong trường hợp nhất định kể từ khi kiểu phân tích này không thể được thực hiện bởi công cụ kiểm chứng mô hình, nhưng vẫn còn quan trọng đối với nỗ lực kiểm chứng.

ESC/Java là một ví dụ của một hệ thống phổ biến rộng cho tính đúng chương trình dựa trên chứng minh định lý [3]. ESC/Java giấu hầu hết các chi tiết của các chứng minh định lý tương tác từ người sử dụng và đưa ra các "cảnh báo" (warning) cho người dùng khi một tính chất không thể được suy ra để thỏa mãn. Tuy nhiên, những cảnh báo này được cung cấp trong một kiểu tương tự như một trình biên dịch. Người dùng có thể loại bỏ các cảnh báo bằng cách chú thích mã nguồn Java với các đặc tả/chú thích (*annotations*) đặc tả ESC, được lấy từ công thức của các vị từ trên các giá trị của biến. Các công cụ chứng minh định lý sử dụng đằng sau là hệ thống chứng minh định lý tự động Simplify - được phát triển tại trung tâm nghiên cứu Compaq [4].

C. Sinh điều kiện kiểm chứng

Sinh điều kiện kiểm chứng là việc tạo ra những điều kiện kiểm chứng (công thức logic toán học) từ mã nguồn chương trình. Nếu điều kiện kiểm chứng hợp lệ thì chương trình đúng, ngược lại điều kiện kiểm chứng chương trình không hợp lệ thì có thể có lỗi trong chương trình. Sinh điều kiện kiểm chứng sử dụng một chương trình giải tích như một tiền điều kiện yếu nhất (*weakest precondition*) để tạo ra một điều kiện kiểm chứng (*Verification Condition*), một công thức logic thuần túy có hiệu lực đòi hỏi sự chính xác của các chương trình đề cập tới đặc tả của nó. Sau đó điều kiện kiểm chứng được đưa vào bộ chứng minh định lý. Nhiều bộ kiểm chứng chương trình sinh điều kiện kiểm chứng hiện đại bằng cách đầu tiên chuyển các chương trình và các đặc tả hành vi của chương trình thành một ngôn ngữ trung gian như **Guarded** [6], **BoogiePL** [11] hoặc **Why** [9] và sau đó tính toán điều kiện kiểm chứng trên các ngôn ngữ trung gian.

Khi mà một điều kiện kiểm chứng là một công thức logic thuần túy, nó phải bao gồm tất cả các kiến thức cần thiết để chứng minh tính đúng đắn của chương trình. Kiến thức này bao gồm nhiều tính chất của ngữ nghĩa ngôn ngữ lập trình, ví dụ, về các giá trị (*values*) và các kiểu (*types*). Với ngôn ngữ mệnh lệnh, nó cũng chứa một mô hình vun đống (heap), thường được tính theo ánh xạ toàn cục từ các ô nhớ tới để các giá trị. Tất cả các khía cạnh của kiểm chứng, bao gồm cả lập luận về tính chất đống (*heap*) như bí danh (*aliasing*), sau đó để lại cho công cụ chứng minh định lý.

Một đặc điểm của phương pháp tiếp cận trong bộ công cụ VCG [12], nó tính toán chỉ một điều kiện kiểm chứng mỗi mô-đun, và do đó, sẽ gọi công cụ chứng minh định lý chỉ một lần mỗi mô-đun. Ở trên, giải quyết với một điều kiện kiểm chứng lớn cho phép các công cụ chứng minh định lý để áp dụng tối ưu hóa. Mặt khác, điều kiện kiểm chứng có xu hướng phức tạp ngay cả đối với các chương trình nhỏ, làm tăng sự phức tạp cho công cụ chứng minh định lý và làm phức tạp vì công cụ chứng minh sẽ tổng quát tìm các mẫu phù hợp hơn. Một nhược điểm của những điều kiện kiểm chứng rộng là không thể giải được với con người, vì vậy gỡ lỗi dựa trên sinh điều kiện kiểm chứng cần sự hỗ trợ thêm trong VCG.

III. KIỂM CHỨNG HÌNH THỨC DỰA TRÊN SINH ĐIỀU KIỆN KIỂM CHỨNG VÀ CHỨNG MINH ĐỊNH LÝ

A. Logic Hoare và điều kiện kiểm chứng

Việc sinh các điều kiện kiểm chứng cho một chương trình và đặc tả của chương trình là một khâu rất quan trọng trong kiểm chứng chương trình. Kiểm chứng chương trình là một phương pháp hình thức, tức là sử dụng các công thức toán học để đặc tả các yêu cầu nói riêng và một chương trình nói chung. Trong phương pháp hình thức, logic Hoare để chứng minh tính đúng đắn thông qua các khái niệm về một "bộ ba Hoare" (*Hoare triple*), đó là một công thức trong các hình thức sau đây:

{P}c{Q}

Trong đó P và Q là các mệnh đề logic, c là lệnh chương trình. Công thức này có thể được hiểu như sau "nếu tính chất p thỏa mãn trước khi chương trình P bắt đầu, tính chất Q thỏa mãn khi chương trình thực thi". Chương trình c có thể tham chiếu toàn bộ một chương trình hoặc một hàm đơn, tùy thuộc vào đơn vị đang được kiểm chứng. Trong giải tích Hoare, những tiên đề và những quy tắc suy luận được sử dụng để nhận được Q dựa trên P và c. Cú pháp của c được mô tả bởi Hoare tương ứng với một ngôn ngữ mệnh lệnh đơn giản với các cấu trúc thông thường (phép gán, điều kiện rẽ nhánh, vòng lặp và các lệnh tuần tự). Một nguyên tắc suy luận mẫu được hiển thị dưới đây cho một lệnh rẽ nhánh if:

$$\frac{\{P \wedge b\}c_1\{Q\}\ \{P \wedge \neg b\}c_2\{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\{Q\}}$$
$$\frac{\{P \wedge b\}c\{Q\}\quad (P \wedge \neg b) \Rightarrow Q}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ c\{Q\}}$$

Hình 3.1. Lệnh *if* trong logic Hoare

Chứng minh định lý sử dụng để chứng minh tính chất chương trình dựa trên các biến đổi của logic Hoare. Tuy nhiên, ngôn ngữ của các bộ chứng minh định lý thường là một phương ngữ (dialect) của LISP, dựa trên bài báo chuyên đề của McCarthy [10].

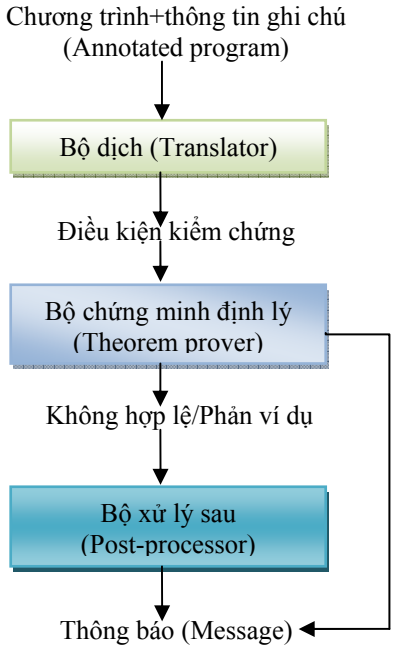
Trong ESC/Java, điều kiện kiểm chứng được sinh ra như sau:

Mã nguồn java + đặc tả	*Công thức (ngôn ngữ lệnh Guarded)*	*Điều kiện kiểm chứng*

static int abs(int x)
//@ensures \result >= 0
{
if(x < 0) {
x = -x;
//@assert x > 0;
}
if(c >0) {
c--;
}
return x;
}
	((assume $x<0$; $x=-x$; assert $x>0$)
	[](assume $\neg(x<0)$));
	((assume $c>0$; $c=c-1$;)
	[]assume $\neg(c>0)$));
	assert $x\geq 0$
		$x<0 \Rightarrow \begin{pmatrix} -x>0 \\ \wedge\ \begin{pmatrix} c>0 \Rightarrow x\geq 0 \\ \wedge\ \neg(c>0) \Rightarrow -x\geq 0 \end{pmatrix} \end{pmatrix}$
		$\wedge(x<0) \Rightarrow \begin{pmatrix} c>0 \Rightarrow x\geq 0 \\ \wedge\ \neg(c>0) \Rightarrow x\geq 0 \end{pmatrix}$

B. Kiểm chứng hình thức dựa trên sinh điều kiện kiểm chứng và chứng minh định lý

1. Lưu đồ kiểm chứng



Hình 3.2. Lưu đồ kiểm chứng chương trình dựa trên sinh điều kiện kiểm chứng

Theo lưu đồ trong Hình 3.2 thì:

- Đầu vào: Chương trình và các đặc tả/chú thích chương trình
- Đầu ra: Các thông báo/cảnh báo lỗi/vi phạm có thể xảy ra trong chương trình
- Các bước xử lý như sau:
  - Bộ dịch (*Translator*): Chuyển chương trình và các đặc tả chương trình thành các chương trình viết bằng ngôn ngữ trung gian mà dễ chuyển thành biểu thức điều kiện kiểm chứng được biểu diễn bởi các công thức toán học hình thức logic Hoare.
  - Bộ chứng minh định lý (*Theorem prover*): Giai đoạn tiếp theo là tích hợp các bộ chứng minh định lý tự động để chứng minh điều kiện kiểm chứng đã được sinh ra bởi Bộ dịch.
  - Bộ sau xử lý (*Post processor*): Đầu ra của chứng minh định lý, đưa ra các thông báo điều kiện kiểm chứng đã hợp lệ hoặc các cảnh báo không hợp lệ.

Việc chứng minh định lý sau khi sinh điều kiện kiểm chứng là để kiểm tra tính hợp lệ của các điều kiện kiểm chứng. Trong ESC/Java, công cụ kiểm tra tính hợp lệ của điều kiện kiểm chứng được hỗ trợ bởi công cụ chứng minh định lý tự động là Simplify [4]. Công cụ chứng minh định lý Simplify được dựa trên các thuật toán Nelson-Oppen cho kết hợp các thủ tục quyết định. Thủ tục quyết định trong Simplify bao gồm một Egraph cho các lý thuyết về sự phương trình trong đó có tính đóng đồng dư (congruence), một bộ giải đơn giản cho số học tuyến tính, tìm kiếm quay lui (backtracking) cho các mệnh đề, một khớp cho các công thức lượng từ, và thủ tục lý thuyết thứ tự cho các thứ tự bộ phận. Simplify được xây dựng như một phần của ESC, phát triển công nghệ tự động kiểm tra tính hợp lệ của các điều kiện kiểm chứng.

C. Các tiếp cận kiểm chứng hiện nay

1. ESC/Java

ESC/Java là công cụ kiểm chứng chương trình Java và các đặc tả bằng các bước sinh điều kiện kiểm chứng bởi công thức logic Hoare từ chương trình và đặc tả của nó qua ngôn ngữ Guarded Commands. Sử dụng tích hợp công cụ chứng minh định lý tự động Simplify để chứng các công thức điều kiện kiểm chứng có hợp lệ hay không. Bộ sau xử lý của ESC/Java đã xử lý các cảnh báo của điều kiện không hợp lệ. Cho một ví dụ về chương trình của Java như sau:

```
1:classBag{
2: int size;
3: int[]elements; // valid: elements[0..size-1]
4:
5: Bag(int[]input){
6: size=input.length;
7: elements=new int[size];
8: System.arraycopy(input,0,elements,0,size);
```

```
9: }
10:
11: int extractMin(){
12: int min=Integer.MAXVALUE;
13: int minIndex=0;
14: for (int i=1;i <=size; i++){
15: if (elements[i]<min){
16: min=elements[i];
17: minIndex=i;
18: }
19: }
20: size--;
21: elements[minIndex]=elements[size];
22: returnmin;
23: }
24: }
```

Đầu ra trong ESC/Java thu được các cảnh báo tương ứng cho chương trình:

Bag.java:6: Warning: Possible null dereference (Null)
size = input.length;
Bag.java:15: Warning: Possible null dereference (Null)
if (elements[i] < min) {
Bag.java:15: Warning: Array index possibly too large (...
if (elements[i] < min) {
Bag.java:21: Warning: Possible null dereference (Null)
elements[minIndex] = elements[size];
Bag.java:21: Warning: Possible negative array index (...
elements[minIndex] = elements[size];

Cảnh báo đầu tiên miêu tả hàm khởi tạo có thể tham chiếu đến một giá trị Null. Từ đó, đưa ra yêu cầu hàm khởi tạo không thể null. Và ở dòng 4 sẽ được thêm vào một câu chú thích của ESC/Java như sau:

4a: //@requiresinput !=null

Cảnh báo thứ 2 và thứ 4 cũng cảnh báo rằng hàm khởi tạo có thể tham chiếu tới giá trị null. Để cánh báo rằng thiết kế không cho phép null đưa ra cảnh báo ở dòng 3 như sau:

3': /@nonnull*/int[]elements; // ...*

ESC/Java tạo ra một cảnh báo bất cứ khi nào xuất hiện mã có thể gán null. Đồng thời kiểm tra rằng các hàm khởi tạo cũng không null. Các tham số cũng cần phải được khai báo giá trị.

5': Bag(/@nonnull*/int[]input){*

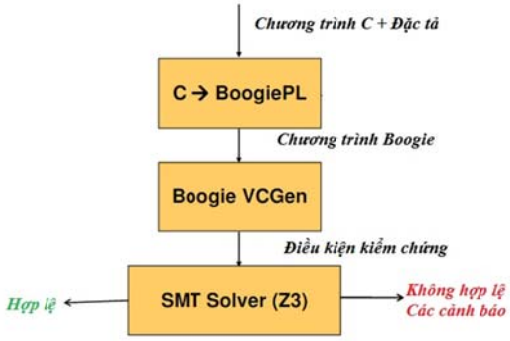
Hai cảnh báo còn lại đưa ra cảnh báo về giá trị kích thước được chọn lựa không tốt.

2a: //@invariant0<=size &&size<=elements.length

ESC/Java đưa ra các cảnh báo nhằm giúp cho chương trình tránh được các lỗi không cần thiết xảy ra. Tuy nhiên, trong ESC/Java chỉ giải quyết chương trình biên dịch (compiling-time), ngữ nghĩa của phép chuyển vị từ được định nghĩa thông qua đồ thị luồng điều khiển (*Control Flow Graph-CFG*) của chương trình sử dụng để suy luận về chương trình. Vấn đề mối quan hệ tương đương giữa các ngôn ngữ chưa được đưa ra phân tích. Trong báo cáo [16] đề cập sâu hơn về khía cạnh này. Chương trình có thể được mô hình hóa bởi cấu trúc hình thức Kripke, biểu diễn các hệ chuyển trạng thái thường được dùng khi thực thi chương trình.

2. HAVOC

HAVOC là công cụ dùng kiểm chứng mã nguồn được viết bằng C và các đặc tả, được phát triển bởi trung tâm nghiên cứu của Microsost [17]. Trong HAVOC, chương trình C và các đặc tả của nó được chuyển đổi sang chương trình BoogiePL [11], một ngôn ngữ trung gian đơn giản với ngữ nghĩa các phép toán hỗ trợ trong việc chuyển đổi thành các công thức logic hay được gọi là điều kiện kiểm chứng. Công cụ chứng minh định lý được sử dụng trong HAVOC là bộ giải SMT (*Satisfiability-Modulo-Theory*) Z3, lưu đồ cụ thể như Hình 3.3 bên dưới:

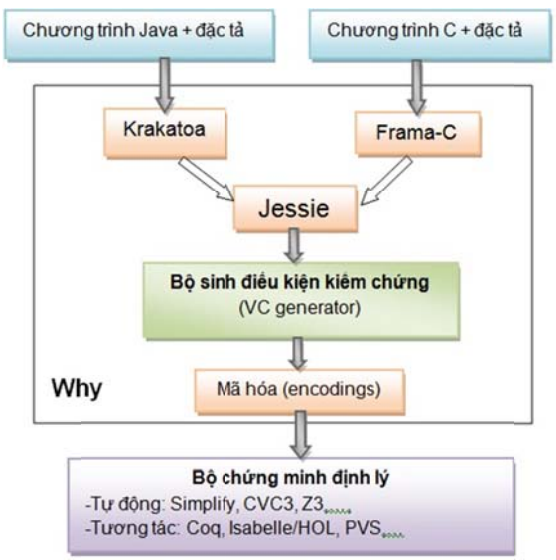


Hình 3.3. Lưu đồ kiểm chứng của HAVOC

Các ứng dụng của HAVOC chủ yếu tập trung vào giải quyết tính năng chương trình C ở mức thấp (low-level) như: Mô hình heap, biểu thức con trỏ (pointer expression), ép kiểu (casts) tính đúng của chương trình,... Tuy nhiên, một điểm giống như ESC/Java là sự chứng minh các quan hệ tương đương giữa các ngôn ngữ là vấn đề chưa được đề cập trong HAVOC.

3. Krakatoa và Jessie

Bộ công cụ Krakatoa và Jessie là công cụ kiểm chứng chương trình được viết bằng ngôn ngữ Java và ngôn ngữ C [18]. Trong đó, Krakatoa thực hiện cho các chương trình viết bằng Java và các đặc tả, Jessie thực hiện cho các chương trình viết bằng ngôn ngữ C và các đặc tả (ngôn ngữ đặc tả ANSI/ISO C - ACSL). Sự tích hợp hai công cụ này tạo nên bộ công cụ kiểm chứng chương trình có kiến trúc như Hình 3.4 bên dưới:



Hình 3.4. Kiến trúc bộ công cụ Krakatoa và Jessie

Khác với trong ESC/Java chỉ làm việc với ngôn ngữ Java và đặc tả, Krakatoa và Jessie có thêm plugin Jessie cho phép thêm xử lý kiểm chứng chương trình C bên trong Frama-C. Bộ sinh điều kiện kiểm chứng Krakatoa và Jessie sử dụng là Why [9], Why cung cấp một đầu ra đã chứng minh (multi-prover) như trong Hình 3.4. Ngoài ra, Krakatoa và Jessie cho phép lựa chọn công cụ chứng minh định lý tự động để chứng minh các điều kiện kiểm chứng sau khi được sinh ra bởi Why như Coq, Simplify, Z3,... Nhưng vì Krakatoa và Jessie sử dụng Why trong việc sinh điều kiện kiểm chứng, do đó giải quyết được các cấu trúc con trỏ, còn đối với cấu trúc dữ liệu phức tạp vẫn chưa được giải quyết trong bộ công cụ này.

D. Đánh giá

Kiểm chứng phần mềm sử dụng các phương pháp hình thức đã được nghiên cứu trong khoảng 15 năm nay. Trong khi có hàng trăm công trình nghiên cứu về lĩnh vực này thì lại có rất ít công cụ nghiên cứu hay thương mại. Một số các công cụ trong số đó như ESC/Java, HAVOC, hay Krakatoa và Jessie cũng chỉ hỗ trợ cho một hỗ trợ cho một ngôn ngữ phổ biến như C, Java. Trên thực tế, có rất nhiều ngôn ngữ lập trình cho các bài toán ứng dụng thực tế khác nhau. Do đó, với tiếp cận trong bài báo này các nghiên cứu tiếp theo cần hướng tới là các vấn đề liên quan tới sự tương đương giữa các ngôn ngữ lập trình, đây cũng chính là một pha trong việc chuyển đổi ngôn ngữ (chuyển các chương trình và đặc tả bằng các ngôn ngữ lập trình hay sử dụng sang ngôn ngữ trung gian) để sinh các điều kiện kiểm chứng. Ngoài ra, sử dụng công cụ trợ giúp chứng minh các điều kiện kiểm chứng để đưa ra các thông báo hợp lệ hay không hợp lệ cho các điều kiện kiểm chứng để hỗ trợ xác định các lỗi chương trình.

IV.KẾT LUẬN

Độ tin cậy của phần mềm ngày càng tăng lên do sự phụ thuộc của hoạt động kinh tế-xã hội trên các hệ thống tính toán. Để đảm bảo độ tin cậy phần mềm, một số phương pháp được đề xuất. Trong khi hầu hết các phương pháp chính thức cố gắng để đảm bảo độ tin cậy về mức mô hình, yêu cầu về độ an toàn của mã nguồn trong ứng dụng thực tế là rất lớn trong việc bảo trì kế thừa phần mềm hiện có. Kiểm chứng phần mềm ở mức mã nguồn dựa trên kết hợp sinh điều kiện kiểm chứng và công cụ chứng minh định lý tự động giải quyết vấn đề này bằng cách áp dụng phương pháp tiếp cận hình thức hướng đến mức mã nguồn. Trong tương lai, sử dụng lý thuyết các phương pháp hình thức trong việc phân tích và kiểm chứng chương trình cũng như sự phát triển các công cụ kiểm chứng tự động sử dụng các phương pháp hình thức đem lại những ứng dụng có giá trị thực tế.

V. LỜI CẢM ƠN

Báo cáo này được hỗ trợ một phần bởi đề tài CS'15.13.

VI.TÀI LIỆU THAM KHẢO

[1] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petruccci, P. Schnoebelen, P. McKenzie: Systems and Software Verification. Springer-Verlag (2001).

[2] C. A. R. Hoare: An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10) (October 1969) 576–585.

[3] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata: Extended Static Checking for Java. In: Proceedings of the Internatioanl ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02), ACM Press (2002).

[4] Detlefs, D., Nelson, G., Saxe, J. B.: Simplify: A Theorem Prover for Program Checking. Journal of the ACM 52(3) (May).

[5] Duffy, D.: Principles of Automated Theorem Proving. John Wiley and Sons (1991).

[6] E. W. Dijkstra: A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.

[7] E. W. Dijkstra: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communications of the ACM 18(8) (1975) 453–457.

[8] G. Holzmann, R. Joshi: Model-Driven Software Verification. In: Proceedings of the 11th Spin Workshop. Volume 2989 of LNCS., Springer-Verlag (2004) 77–92.

[9] J. C. Filliatre. Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Universite Paris Sud, 2003.

[10] J. McCarthy: Checking Mathematical Proofs by Computer. Proceedings Symposium on Recursive Function Theory.

[11] K. R. M. Leino. This is Boogie 2. Working Draft - available at http://research.microsoft.com/en-us/um/people/leino/papers.html, 2008.

[12] P. Muller and J. N. Ruskiewicz. Using debuggers to understand failed verification attempts. In M. Butler and W. Schulte, editors, FM'11, volume 6664 of Lecture Notes In Computer Science. Springer-Verlag, 2011.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre: Software Verification with BLAST. In: Model Checking Software: Proceedings of the 10th International SPIN Workshop. Volume 2648 of LNCS., Springer-Verlag (2003) 235–239.

[14] T. Ball, S. K. Rajamani: Automatically Validating Temporal Safety Properties of Interfaces. In: Model Checking Software: Proceedings of the 8th International SPIN Workshop, 2001, p103–122.

[15] T. Ball, B. Hackett, S. K. Lahiri, S. Qadeer, and J. Vanegue, "Towards scalable modular checking of user-defined properties," in Verified Software: Theories, Tools, Experiments (VSTTE '10), vol. LNCS 6217, 2010, pp. 1–24.

[16] T. T. Nguuyen, M. D. Tran, "Interaction Analysis of Annotated Specification and Program Codes in Extended Static Checking" in IEEE RIVF International Conference on, 2015, pp. 144-150.

[17] http://research.microsoft.com/en-us/projects/havoc/

[18] http://krakatoa.lri.fr/