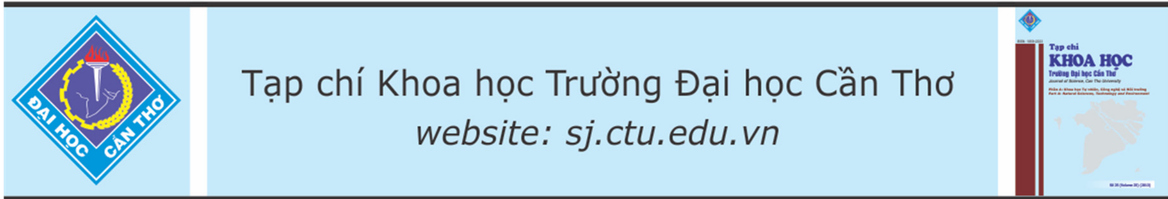



# MỘT KHUNG THỨC TÌM KIẾM VÀ TÁI SỬ DỤNG HÀM API TỰ ĐỘNG DỰA TRÊN ĐẶC TẢ HÌNH THỨC

Huỳnh Tấn Khải, Bùi Hoài Thắng, Quản Thành Thơ và Nguyễn Minh Thông[1]

[1] *Khoa Khoa học & Kỹ thuật Máy tính, Trường Đại học Bách Khoa Thành phố Hồ Chí Minh*

***Thông tin chung:***
*Ngày nhận: 03/09/2013*
*Ngày chấp nhận: 21/10/2013*

***Title:***
*A framework for automatic search and reusage of API functions based on formal specification*

***Từ khóa:***
*Tìm kiếm hàm tự động, gọi hàm tự động, tái sử dụng thành phần, đặc tả hình thức, thiết kế thoả thuận*

***Keywords:***
*Automatic API functions searching, automatic function composition, component reuse, formal specification, design by contract*

**ABSTRACT**

*To help programmers quickly develop their software systems, research communities have proposed some approaches for automatic search of function APIs, mostly based on keywords used in the API descriptions. However, the accuracy of those approaches is still limited. Formal specification with the rigor of the syntax and semantics can be applied to improve the research results of this direction. In this paper, we firstly present a survey on formal specification techniques for software design. Next, we propose a framework for searching and reusing API functions automatically based on formal specification described by JML. In addition, the framework also supports automatic composition of API functions to meet the requirement demand. Some case studies are also presented to imply the practical potential of our approach.*

**TÓM TẮT**

*Để giúp cho các lập trình viên phát triển nhanh hệ thống phần mềm của họ, cộng đồng nghiên cứu đã đề xuất một số cách tiếp cận cho việc tìm kiếm tự động các hàm API. Hầu hết các nghiên cứu này đều dựa trên các từ khóa được sử dụng trong các mô tả của các hàm API. Tuy nhiên, độ chính xác của các hướng tiếp cận này vẫn còn hạn chế. Đặc tả hình thức với sự chặt chẽ trong cú pháp và ngữ nghĩa có thể được áp dụng để cải thiện kết quả của các hướng nghiên cứu này. Trong bài báo này, đầu tiên, chúng tôi trình bày một khảo sát về các kỹ thuật đặc tả hình thức cho thiết kế phần mềm. Tiếp theo, chúng tôi đề xuất một khung thức cho việc tìm kiếm và tái sử dụng các hàm API tự động dựa trên đặc tả chính thức được mô tả bởi JML. Ngoài ra, khung thức còn hỗ trợ việc tổ hợp tự động các hàm API để đáp ứng các yêu cầu phần mềm. Một số kết quả thử nghiệm cũng được trình bày để cho thấy khả năng thực tế trong hướng tiếp cận của chúng tôi.*

## 1 GIỚI THIỆU

Quá trình thiết kế nói chung trong hầu hết các ngành kỹ thuật đều có sự tái sử dụng các *thành phần* (component) đã được xây dựng sẵn. Trong công nghệ phần mềm, việc tái sử dụng các thành phần phần mềm là điều đã được đề cập và áp dụng từ rất sớm [1]. Việc xây dựng phần mềm dựa trên cơ sở tái sử dụng các thành phần đã có sẽ làm cho chi phí và thời gian xây dựng thấp hơn và chất lượng cũng tăng lên [2]. Tuy nhiên, bên cạnh các lợi điểm, việc tái sử dụng các thành phần hiện tại cũng gặp phải một số hạn chế trong việc ứng dụng thực tế. Trong đó khó khăn thường gặp nhất là việc lựa chọn thành phần tái sử dụng. Do số lượng các

thành phần được xây dựng sẵn trong các thư viện là rất lớn, các lập trình viên thường phải tốn khá nhiều công sức để tìm hiểu và lựa chọn thành phần phù hợp với nhu cầu của mình.

Trong lĩnh vực tái sử dụng thành phần thì việc tìm kiếm và gọi thực thi các hàm thư viện API (Application Programming Interface) là công việc được thực hiện thường xuyên nhất. Đã có nhiều nghiên cứu nhằm hỗ trợ việc tìm kiếm hàm thư viện đáp ứng yêu cầu của người lập trình. Tuy nhiên, hầu hết các hướng nghiên cứu này chủ yếu dựa trên việc so trùng chuỗi (từ khóa), nghĩa là dựa vào các mô tả bằng ngôn ngữ tự nhiên của hàm thư viện và của lời yêu cầu gọi hàm. Chẳng hạn như [3] nghiên cứu về máy tìm kiếm hàm dựa trên truy vấn ngôn ngữ tự nhiên; Assieme [4] là máy tìm kiếm mã nguồn dựa trên từ khóa; SNIFF [5] thực hiện các truy vấn bằng ngôn ngữ tự nhiên và trả về các đoạn trích mã nguồn phù hợp với mục đích của câu truy vấn. Với các cách tiếp cận trên thì chúng ta nhận thấy một điều là độ chính xác của các công cụ tìm kiếm này rất khác nhau, thậm chí với một công cụ tìm kiếm, nhưng trong các tình huống khác nhau cũng sẽ cho mức độ đúng đắn khác nhau. Mức độ chính xác của các công cụ trên có sự khác nhau là do chúng làm việc trên ngôn ngữ tự nhiên, vốn luôn chứa đựng sự nhập nhằng và đa nghĩa.

Một nhược điểm khác của các nghiên cứu này là chúng chỉ có thể tìm ra một hàm cụ thể đáp ứng yêu cầu (phù hợp theo từ khóa truyền vào). Khả năng kết hợp nhiều hàm lại với nhau để tổng hợp thành một lời giải hiện vẫn đang gặp nhiều khó khăn, vì để làm được điều này đòi hỏi chương trình tổng hợp phải "hiểu" được ngữ nghĩa của các hàm thư viện, thay vì chỉ có thể nhận dạng được các từ khoá mô tả các hàm thư viện này.

Một trong những hướng tiếp cận có thể vượt qua các khó khăn trên là sử dụng các *đặc tả hình thức* (formal specification) [6] để tìm kiếm và tổng hợp các hàm thư viện. Với cách tiếp cận này, các đặc tả hình thức, thực chất là các mệnh đề logic, với sự hỗ trợ của các công cụ suy luận và chứng minh định lý có thể cho được một lời giải đúng và đầy đủ cho một yêu cầu tìm kiếm hàm thư viện. Bên cạnh đó, việc sử dụng ngôn ngữ đặc tả hình thức sẽ loại bỏ được tính nhập nhằng và đa nghĩa của ngôn ngữ tự nhiên. Đồng thời, hướng tiếp cận này có thể hỗ trợ việc đưa ra một lời giải tổng hợp bao gồm một tập các hàm lồng ghép với nhau để có thể đáp ứng được yêu cầu, điều mà các phương pháp tìm kiếm dựa trên từ khóa là không thể làm được.

Trong các phương pháp phát triển phần mềm dựa trên đặc tả hình thức hiện nay thì phương pháp *Design by contract* (DBC) [7] nổi lên như một trong những phương pháp phát triển nhanh và chặt chẽ. Ý tưởng chính của phương pháp này là một lớp hoặc một phương thức và khách hàng của nó (đối tượng sử dụng) có một "*thỏa thuận*" (contract) với nhau. Thỏa thuận này thể hiện thông qua *tiền điều kiện* (pre-condition) và *hậu điều kiện* (post-condition). Khách hàng phải đảm bảo các điều kiện trước khi gọi một phương thức và kết quả trả về từ phương thức cũng phải thỏa được các yêu cầu mà khách hàng cần. Trong bài báo này, chúng tôi sẽ đưa ra một khung thức dựa trên hướng tiếp cận "thoả thuận" để tìm kiếm và tổng hợp các hàm thư viện nhằm đáp ứng một yêu cầu lập trình ban đầu.

Các đóng góp của chúng tôi bao gồm:

– Chúng tôi đưa ra một khảo sát chi tiết về các hướng tiếp cận đặc tả hình thức theo phương pháp "thoả thuận".

– Chúng tôi đưa ra một khung thức để tự động tìm kiếm và tổng hợp các hàm API dựa trên các đặc tả hình thức theo phương pháp "thoả thuận". Kết quả trả về có thể là một hàm đáp ứng yêu cầu của người gọi, hoặc cũng có thể là một chuỗi các hàm gọi tuần tự hoặc lồng nhau.

– Trong khung thức đề nghị, chúng tôi đưa ra hướng tiếp cận tính toán độ tương tự giữa các đặc tả hình thức của các API. Nhờ đó, khung thức cho phép sử dụng các kỹ thuật xử lý thông minh của lĩnh vực trí tuệ nhân tạo như *lập kế hoạch* (planning) [8] để tìm kiếm và tổng hợp các API cần thiết một cách hiệu quả.

Phần còn lại của bài báo này được tổ chức như sau: Phần 2 trình bày về các ngôn ngữ đặc tả hình thức theo phương pháp "thỏa thuận". Phần 3 trình bày ý tưởng về khung thức cho việc tìm kiếm hàm tự động dựa trên đặc tả hình thức JML. Phần 4 trình bày về một số kết quả thí nghiệm. Cuối cùng, Phần 5 là kết luận của bài báo.

## 2 CÁC NGÔN NGỮ HÌNH THỨC ĐẶC TẢ THEO PHƯƠNG PHÁP THỎA THUẬN

### 2.1 Đặc tả hình thức và hướng tiếp cận thoả thuận

Đặc tả hình thức cho phần mềm đã được quan tâm từ lâu trong lĩnh vực khoa học máy tính. Từ cuối những năm 1940, Turing đã cho rằng các chương trình tuần tự có thể được tạo ra một cách đơn giản bằng cách đánh chú thích các trạng thái của các thuộc tính của hệ thống tại các thời điểm

cụ thể [6]. Vào cuối những năm 1960, Floyd, Hoare và Naur đề xuất kỹ thuật tiền đề để chứng minh sự nhất quán giữa các chương trình tuần tự và các mô tả của chúng, được gọi là các đặc tả [9][10][11]. Sự quan tâm về các đặc tả hình thức và các ứng dụng đa dạng của chúng trong lĩnh vực công nghệ phần mềm đã phát triển liên tục cho đến hiện nay [12][13].

Một trong những kết quả mới và có tính thực tiễn nhất trong hướng tiếp cận này là việc đưa ra các "thoả thuận" giữa yêu cầu của một hàm và hiện thực của hàm đó [7]. Như đã giới thiệu trong Phần 1, một "thoả thuận" được biểu diễn thông qua các tiền điều kiện và hậu điều kiện. Ví dụ, một "thỏa thuận" cho phương thức *sqrt* (nhận vào một số và trả về căn bậc hai của nó) được biểu diễn bằng ngôn ngữ JML (*Java Modeling Language*) [14] như mô tả trong Hình 1. Trong ví dụ này, mệnh đề *require* thể hiện tiền điều kiện, là biến *x* truyền vào phải lớn hơn hoặc bằng *0.* Mệnh đề *ensure* thể hiện hậu điều kiện. Ở ví dụ này, giá trị trả về (*\result*) phải thỏa điều kiện là bình phương của nó phải bằng lại giá trị của biến *x* truyền vào.

```
//@ requires x >= 0.0;
//@ ensures \result * \result = x
public static double sqrt(double
x)
{ /*...*/ }
```

**Hình 1: "Thỏa thuận" của phương thức *sqrt* được biểu diễn bằng ngôn ngữ JML**

### 2.2 Các ngôn ngữ đặc tả hình thức theo hướng tiếp cận "thoả thuận"

#### *2.2.1 Ngôn ngữ JML*

JML (Java Modeling Language) [14] là ngôn ngữ đặc tả hành vi, được sử dụng để đặc tả hành vi của các mô đun trong ngôn ngữ Java. Nó là sự kết hợp phương pháp "thỏa thuận" với phương pháp đặc tả dựa trên mô hình của ngôn ngữ Larch [15]. JML sử dụng logic Hoare [16] kết hợp với việc đặc tả các tiền/ hậu điều kiện (pre-/post-condition) và các bất biến (invariant) của các thuộc tính, các cấu trúc.

Các đặc tả JML được thêm vào mã Java dưới dạng các chú giải (annotation) bên trong các chú thích (comment). Các chú thích trong ngôn ngữ Java được hiểu như là các chú giải JML khi nó bắt đầu bằng ký tự "@". Ví dụ trong Hình 2 minh họa việc sử dụng JML đặc tả cho lớp *Purse*. Trong đó, trường *balance* được đặc tả với bất biến (***invariant***) là *balance* luôn trong đoạn từ 0 đến *MAX_BALANCE*. Trường *pin* là một mảng thuộc kiểu ***byte*** với bất biến là *pin* có đúng 4 phần tử và các phần tử chỉ có giá trị từ 0 đến 9.

```
public class Purse {
  final int MAX_BALANCE;
  int balance;
  //@ invariant 0 <= balance &&
           balance <= MAX_BALANCE;

  byte[] pin;
  /*@ invariant pin != null &&
        pin.length == 4 &&
       (\forall int i; 0<=i&&i<4;
         0<=pin[i]&&pin[i]<=9);
*/
  /*@ requires amount >= 0;
   @ assignable balance;
   @ ensures
      balance==\old(balance)-
amount
       && \result == balance;
   @ signals (PurseException)
       balance == \old(balance);
*/
   int debit(int amount) throws

PurseException;
   }
```

**Hình 2: Ví dụ minh họa về đặc tả JML**

Phương thức *debit* nhận vào đối số *amount*. Phương thức này sẽ trừ *amount* vào *balance* và trả về giá trị của *balance* sau khi trừ trong trường hợp *balance* lớn hơn hoặc bằng *amount*. Trong trường hợp *balance* nhỏ hơn *amount*, một ngoại lệ *PurseException* sẽ được trả về. Tiền điều kiện (***requires***) của phương thức này là *amount* phải lớn hơn hoặc bằng 0; hậu điều kiện (***ensures***) là *balance* bị trừ đi một lượng *amount* và trả về kết quả sau khi trừ. Phương thức này còn có một hậu điều kiện nữa cho trường hợp phương thức trả về ngoại lệ (***signals***).

#### *2.2.2 Spec#*

Spec# [17] là một ngôn ngữ hình thức cho các "thỏa thuận" API (có sự ảnh hưởng từ JML), nó là sự mở rộng của ngôn ngữ C# với việc bổ sung các tiền/ hậu điều kiện và bất biến cho các đối tượng. Spec# là ngôn ngữ đặc tả cho phép kiểm tra động cũng như kiểm tra tĩnh chương trình. Hệ thống Spec# cung cấp một kiến trúc hoàn chỉnh, bao gồm trình biên dịch, thư viện hỗ trợ và các công cụ phát triển. Các đặc tả của Spec# cũng được biên dịch thành một phần của chương trình thực thi, trong đó

chúng sẽ được kiểm tra tự động. Đi kèm với Spec# là một bộ kiểm tra chương trình tự động, có tên là Boogie [18]. Nó cho phép kiểm tra các đặc tả theo kỹ thuật phân tích tĩnh [19].

Hình 3 minh họa về đặc tả của Spec# cho lớp *ArrayList*. Trong đó, phương thức *Insert* được đặc tả với các tiền điều kiện (***requires***), hậu điều kiện (***ensures***). Đặc biệt, Spec# còn cho phép các đặc tả có thể thừa kế lẫn nhau. Điều này giúp cho các đặc tả của Spec# rõ ràng, đáng tin cậy và dễ đọc hơn so với các ngôn ngữ đặc tả khác. Chi tiết về cách thức sử dụng Spec# chúng ta có thể tham khảo trong [17].

```
class ArrayList {
  public void Insert
        (int index, object
value)
  requires 0 <= index &&
                index <= Count
;
  requires !IsReadOnly &&

!IsFixedSize;
  ensures Count == old(Count)+1;
  ensures value == this[index];
  ensures foral{int i in
0:index;

old(this[i])==this[i]};
  ensures foral{int i in
            index:old(Count);
       old(this[i])==this[i
+1]};
  { /*...*/ }
  //… …
}
```

**Hình 3: Minh họa về đặc tả Spec#**

*2.2.3 Frama-C và ngôn ngữ ACSL*

Frama-C [20] là một nền tảng dành riêng cho việc phân tích tĩnh mã nguồn chương trình được viết bằng ngôn ngữ C. Frama-C tập hợp một số kỹ thuật phân tích tĩnh [19] vào trong một khung thức duy nhất. Để được phân tích bởi Frama-C, các chương trình C cần được đặc tả bởi ngôn ngữ ACSL (ANSI/ISO C Specification Language) [21]. Đây là ngôn ngữ đặc tả hình thức cho phép chúng ta đặc tả các thuộc tính của một chương trình C. ACSL cũng được "lấy cảm hứng" từ ngôn ngữ JML.

Khái niệm quan trọng nhất trong ACSL đó là *thỏa thuận* hàm. Một thỏa thuận hàm cho một hàm *f* trong ngôn ngữ C là một tập các yêu cầu trên các đối số của *f* và một tập các thuộc tính cần được đảm bảo sau khi kết thúc việc thực hiện *f*. Chúng ta hãy xem xét ví dụ về hàm *max* (Hình 4). Hàm này không cần tiền điều kiện, không cần bất kỳ sự ràng buộc nào trên các đối số truyền vào. Hậu điều kiện của chúng mô tả rằng kết quả trả về của hàm *max* luôn lớn hơn hoặc bằng giá trị của *x*, *y* và sẽ bằng một trong hai tham số *x* hoặc *y* truyền vào.

```
/*@ ensures \result >= x &&
                    \result >=
y;
    ensures \result == x ||
                 \result == y;
*/
int max (int x, int y)
{ return (x > y) ? x : y; }
```

**Hình 4: Minh họa về đặc tả ACSL**

Ngoài tiền và hậu điều kiện, ACSL còn hỗ trợ đặc tả bất biến cho các đối tượng và vòng lặp. Tuy nhiên, ACSL lại không có cơ chế xử lý ngoại lệ như JML. Chi tiết về ACSL chúng ta có thể tham khảo trong [21].

*2.2.4 Jahob*

Jahob [22] là một hệ thống kiểm tra dành cho các chương trình viết bằng ngôn ngữ Java (tập con của ngôn ngữ Java). Sử dụng Jahob, các nhà phát triển có thể chứng minh theo phương pháp phân tích tĩnh rằng các phương thức hiện thực có phù hợp với các "thỏa thuận" đặc tả của chúng hay không. Jahob còn cho phép đặc tả các bất biến về các cấu trúc dữ liệu quan trọng cũng như các ràng buộc thiết kế. Ý tưởng cơ bản của Jahob là mô hình hóa các trạng thái của chương trình và các cấu trúc dữ liệu mà nó sử dụng thành các tập trừu tượng các đối tượng và các mối quan hệ giữa các đối tượng. Jahob sẽ hỗ trợ phát biểu các ràng buộc trên tập các đối tượng này. Chúng ta xét ví dụ về đặc tả của Jahob cho lớp *List* như được mô tả trong Hình 5.

Trong ví dụ trên, đặc tả của lớp *List* sử dụng biến đặc tả *content* để chỉ tập các đối tượng thể hiện trong danh sách. Tập này không tồn tại khi chương trình chạy – nó đơn giản là một sự trừu tượng. Chương trình Jahob chỉ sử dụng nó cho mục đích đặc tả và bộ kiểm tra Jahob sử dụng nó để kiểm tra chương trình. Chi tiết về cấu trúc và cú pháp của ngôn ngữ đặc tả cho hệ thống Jahob chúng ta có thể tham khảo trong [22].

```
class List
{
  /*: public static specvar
           content :: objset;
*/

  public List()
  /*: modifies content
      ensures "content = {}"  */

  public void add(Object o)
  /*: requires "o ~: content &
                        o ~=
null"
      modifies content
      ensures "content =
          old content Un {o}"
*/

  public void remove (Object o)
  /*: requires "o : content"
      modifies content
      ensures "content =
           old content - {o}"
*/
}
```

**Hình 5: Minh họa đặc tả cho lớp *List* theo hệ thống Jahob**

*2.2.5 ADA 2012*

Ada [23] là ngôn ngữ lập trình do Bộ quốc phòng Mỹ đầu tư phát triển vào khoảng nửa đầu thập niên 80 của thế kỷ 20. Qua nhiều năm phát triển, phiên bản Ada 2012 [24] có rất nhiều thay đổi, trong đó có một số bổ sung đáng kể như sau: hỗ trợ đặc tả hình thức trong chương trình, cho phép khai báo các tiền và hậu điều kiện cho các thủ tục, các bất biến của kiểu dữ liệu. Tất cả các sự bổ sung này nhằm mục đích kiểm tra tính đúng của chương trình cần xây dựng.

Hình 6 mô tả một ví dụ về đặc tả cho gói *Stack* bằng Ada 2012. Trong ví dụ này, biến *Stack* được khai báo với bất biến kiểu (*Type_Invariant*) là các phần tử trong Stack không trùng nhau. Các hàm được đặc tả ngay sau phần khai báo và bắt đầu bằng từ khóa *with*. Tiền điều kiện được khai báo với từ khóa *Pre* theo cú pháp *Pre* => <*expression*>. Tương tự, hậu điều kiện được khai báo với từ khóa *Post*.

```
package Stacks is
  type Stack is private;
    with Type_Invariant =>

Is_Unduplicated(Stack);
  function Is_Empty(S: Stack)
                return
Boolean;
  function Is_Full(S: Stack)
                return
Boolean;
  function Is_Unduplicated
        (S: Stack) return
Boolean;
  procedure Push(S: in out
Stack;
               X: in Item)
    with
      Pre => not Is_Full(S),
      Post => not Is_Empty(S);
//...
end Stacks;
```

**Hình 6: Minh họa đặc tả cho gói Stack bằng ngôn ngữ Ada 2012**

### 2.3 So sánh các ngôn ngữ đặc tả

Bảng 1 tổng kết và so sánh lại các ngôn ngữ đặc tả dựa trên một số các tiêu chí. Trong số các ngôn ngữ đặc tả theo phương pháp "thỏa thuận" nêu trên thì cách sử dụng, cách thức làm việc của các phương pháp là khá giống nhau.

Hầu hết các ngôn ngữ trên đều bắt đầu từ ý tưởng nền tảng của JML, đây là ngôn ngữ tiên phong cho các ngôn ngữ đặc tả theo phương pháp "thỏa thuận". JML có ưu điểm là cấu trúc và cú pháp rõ ràng, đặc tả dựa trên ngôn ngữ chủ là ngôn ngữ Java đã trở nên rất phổ biến. Sự tách biệt giữa mã đặc tả và mã xử lý giúp cho hai công việc này hầu như tách biệt nhau. JML cho phép lồng ghép các đặc tả phi hình thức chung với các đặc tả hình thức nhằm hỗ trợ nhiều mức đặc tả cho chương trình.

Ngoài các ưu điểm của bản thân ngôn ngữ, JML còn có một số ưu điểm khác mà chúng tôi quyết định sử dụng nó cho khung thức của mình. Đó là: thư viện API của ngôn ngữ Java đã được đặc

tả đầy đủ bằng ngôn ngữ JML; JML có nhiều công cụ hỗ trợ cho việc kiểm tra chương trình như ESC/Java [25], KeY [26],... Do vậy, trong phần tiếp theo của bài báo này, JML là ngôn ngữ đặc tả hình thức được chúng tôi lựa chọn để tiếp tục phát triển cho khung thức tìm kiếm, tổng hợp và tái sử dụng hàm API tự động được đề cập trong các phần sau.

**Bảng 1: So sánh các ngôn ngữ đặc tả hình thức theo hướng thoả thuận**

| | JML | Spec# | ACSL | Jahob | ADA 2012 |
|---|---|---|---|---|---|
| **Ngôn ngữ chủ** | Java | Mở rộng của C# | C | Tập con của Java | ADA 2012 |
| **Cách thức đặc tả** | Chèn vào dưới dạng chú giải trong các chú thích | Đưa thêm từ khóa vào ngay trong ngôn ngữ | Chèn vào dưới dạng chú giải trong các chú thích | Chèn vào dưới dạng chú giải trong các chú thích | Đưa thêm từ khóa vào ngay trong ngôn ngữ |
| **Các thành phần đặc tả** | Tiền điều kiện<br>Hậu điều kiện<br>Bất biến<br>Ngoại lệ | Tiền điều kiện<br>Hậu điều kiện<br>Bất biến<br>Ngoại lệ | Tiền điều kiện<br>Hậu điều kiện<br>Bất biến<br>Hỗ trợ con trỏ | Tiền điều kiện<br>Hậu điều kiện<br>Bất biến<br>Cấu trúc dữ liệu | Tiền điều kiện<br>Hậu điều kiện<br>Bất biến |
| **Mục đích chính** | Kiểm tra tính đúng của phương thức được hiện thực so với đặc tả | Kiểm tra tính đúng của phương thức được hiện thực so với đặc tả | Kiểm tra tính đúng của hàm được hiện thực so với đặc tả | Kiểm tra tính đúng của cấu trúc dữ liệu được hiện thực so với đặc tả | Kiểm tra tính đúng của hàm/ thủ tục được hiện thực so với đặc tả |
| **Hỗ trợ đặc tả phi hình thức** | Có | Không có | Không có | Không có | Không có |
| **Số lượng công cụ hỗ trợ/ sử dụng** | Nhiều (ECS/Java, KeY, JMLUnint,…) | Boogie | Frama-C | Hệ thống Jahob | Bản thân trình biên dịch ADA 2012 |

## 3 VÀ TÁI SỬ DỤNG HÀM API TỰ ĐỘNG

Hình 7 mô tả khung thức cho việc tìm kiếm và tái sử dụng hàm API tự động. Khung thức này có ba khối chức năng chính sau đây:

### 3.1 Bộ đánh giá sự tương tự giữa các đặc tả

Khối chức năng này giải quyết bài toán cơ bản để thực hiện việc tìm kiếm các đặc tả phù hợp với một yêu cầu đầu vào. Đó là đánh giá sự tương tự giữa các công thức logic đặc tả các hàm. Nhờ vậy, chúng ta có thể thu hẹp không gian tìm kiếm hàm. Việc thu hẹp này dựa trên sự tính toán mức độ tương tự của đặc tả yêu cầu với các đặc tả của hàm API có trong thư viện.

Nguyên lý cho việc phân loại hàm dựa trên học máy là với một tập các hàm đã cho (*P*) và một lời yêu cầu (*c*), giải thuật sẽ *tiên đoán* những hàm từ *P* có khả năng thỏa mãn được yêu cầu của *c*. Hướng tiếp cận này dựa trên một số định nghĩa sau:

***Định nghĩa 1***: Ma trận phụ thuộc

Gọi $\Gamma$ là một tập đặc tả được biểu diễn dưới dạng logic của các hàm thư viện. Khi đó chúng ta định nghĩa:

$$\mu = \Gamma \times \Gamma \rightarrow \{0,1\}$$

$$\mu(c,p) = \begin{cases} 1 & \text{nếu } p \text{ có khả năng thỏa mãn được yêu cầu của } c \\ 0 & \text{ngược lại} \end{cases}$$

***Định nghĩa 2***: Ma trận đặc trưng

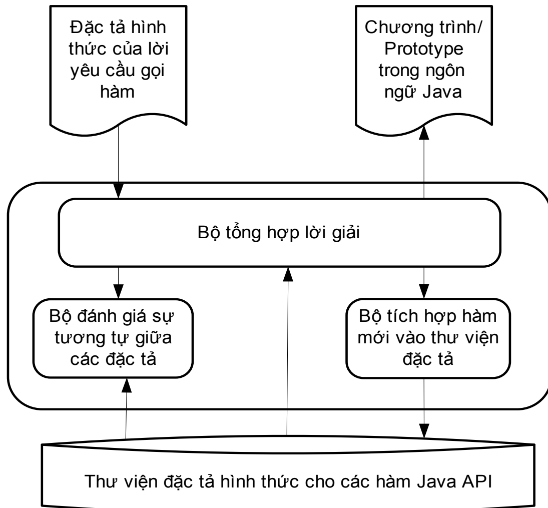
Gọi $T=\{t_1, ..., t_m\}$ là tập các mệnh đề (term) đặc tả trong $\Gamma$. Chúng ta định nghĩa hàm Φ như sau:

$$\Phi = \Gamma \times \{1, ..., m\} \rightarrow \{0,1\}$$

$$\Phi(c,i) = \begin{cases} 1 & \text{nếu } t_i \text{ xuất hiện trong } c \\ 0 & \text{ngược lại} \end{cases}$$

***Định nghĩa 3***: Hàm đặc trưng

Hàm đặc trưng φ: Γ → $\{0,1\}^m$, với mỗi *c* thuộc Γ thì chúng ta có vector $\varphi^c \in \{0,1\}$ thỏa điều kiện $\varphi_i^c = 1 \Leftrightarrow \Phi(c,i) = 1$.




**Hình 7: Khung thức cho việc tìm kiếm và tái sử dụng hàm API tự động**

***Định nghĩa 4***: Hàm phân loại

Cho mỗi *p* trong Γ, hàm phân loại được định nghĩa như sau $C_p(.)$: Γ → *R*, với mỗi lời yêu cầu *c* thì hàm phân loại $C_p(c)$ sẽ tính toán mức độ phù hợp của hàm *p* so với lời yêu cầu *c*.

Ví dụ, với các hàm *f*1, *f*2 và *f*3 được mô tả trong Hình 8, nếu chúng ta ký hiệu các mệnh đề $p_1$, $p_2$, $p_3$, $p_4$ tương ứng với các tập toán tử {+,-} {*}, {>, ≤}, {*log*}[1], chúng ta sẽ có ma trận đặc trưng như sau:

| $\varphi$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|
| $f_1$ | 0 | 1 | 0 | 0 |
| $f_2$ | 1 | 0 | 0 | 0 |
| $f_3$ | 1 | 0 | 1 | 1 |

```
//@ ensures \result == 2 * x;
double f1 (double x);

//@ ensures \result == - x;
double f2 (double x);

//@ ensures
//@   (x>0)==>(\result==x+1) &&
//@
(x≤0)==>(\result==log(x2));
double f3 (double x);
```

**Hình 8: Một số đặc tả hàm bằng JML**

[1] Đây là một sự phân loại toán tử thường gặp dựa trên độ ưu tiên,

Khi đó, chúng ta sẽ tính được vector đặc trưng cho từng lời gọi hàm dựa vào $\varphi_c$. Ví dụ dựa vào ma trận trên, chúng ta có vector đại diện cho lời gọi hàm $f_2$ là $\varphi_{f_2}$=[1,0,0,0] (giá trị 1 ở cột thứ *i* nghĩa là $c_2$ có đặc trưng *i*). Với vector $\varphi$ vừa tính được, chúng ta có thể phát triển các phương pháp tính toán độ tương tự. Các phương pháp này có thể đơn giản là phép tích góc giữa hai vector, hoặc có thể là một phương pháp học máy như Naive Bayes [27], tuỳ theo độ phức tạp của bài toán.

### 3.2 Bộ tổng hợp lời giải

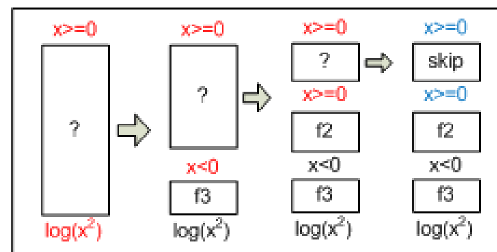
Khối chức năng thứ hai là bộ tổng hợp lời giải. Đây là bộ phận quan trọng nhất trong khung thức. Nhờ vào việc các hàm thư viện được đặc tả bằng các hàm logic và phương pháp tính độ tương tự giữa các đặc tả xây dựng được, chúng ta có thể phát triển một cơ chế tìm kiếm và tổng hợp các hàm API đáp ứng nhu cầu ban đầu dựa trên kỹ thuật *lập kế hoạch* trong lĩnh vực Trí tuệ Nhân tạo (AI Planning).

Bộ tìm kiếm hàm sử dụng kỹ thuật lập kế hoạch dựa trên trí tuệ nhân tạo hoạt động theo nguyên tắc sau: Xuất phát từ yêu cầu ban đầu (*c*) và tập các hàm *P* có độ tương tự trong ngưỡng quy định đối với (*c*), bộ lập kế hoạch sẽ chọn ra một hàm *p* từ *P* (theo độ ưu tiên về mức độ phù hợp), nếu tiền điều kiện của *p* phù hợp với tiền điều kiện của *c* thì quá trình tìm kiếm kết thúc và trả kết quả về. Ngược lại, giải thuật xem tiền điều kiện của *p* như là một hậu điều kiện mới và lặp lại quá trình tìm kiếm. Trong trường hợp quá trình tìm kiếm bế tắc, giải thuật sẽ quay lui lại với các hàm *p'* trong *P* (*p'*≠*p*).

Với các hàm thư viện như mô tả trong Hình 8, giả sử chúng ta có một lời yêu cầu gọi hàm được đặc tả như sau:

```
//@ requires x ≥ 0
//@ ensures \result == log(x2)
```

Với thư viện hàm và lời yêu cầu như trên, quá trình *lập kế hoạch* của chúng ta được thực hiện thông qua các bước như trong Hình 9.




**Hình 9: Quá trình tìm kiếm hàm dựa trên lập kế hoạch**

Ban đầu, hệ thống sẽ cố gắng tìm một hàm có khả năng đưa ra kết quả là $log(x^2)$ từ yêu cầu ban đầu là $x \geq 0$. Sử dụng cơ chế tính toán tương tự giữa các đặc tả như mô tả trong Phần 3.1, bộ lập kế hoạch xác định được hàm phù hợp nhất là $f_3$. Để có thể tạo ra kết quả là $log(x^2)$, hàm $f_3$ yêu cầu ngõ nhập phải thoả điều kiện $x<0$. Như vậy bộ lập kế hoạch cần tìm một hàm thoả mãn *subgoal* : $\{x \geq 0 \Rightarrow x<0\}$. Một lần nữa, việc tính toán độ tương tự giữa các hàm sẽ xác định hàm phù hợp nhất cho subgoal này là $f_2$[2]. Đến đây, một lời giải tổng hợp đã được hình thành đầy đủ như sau: $f_3(f_2(x))$.

### 3.3 Bộ tích hợp hàm mới vào thư viện đặc tả

Chức năng này đóng vai trò giúp cho thư viện đặc tả các hàm của chúng ta phong phú thêm, giúp cho những lần sau khi gặp những yêu cầu tương tự, hệ thống sẽ có sẵn các hàm/ chuỗi các hàm đáp ứng yêu cầu, giúp tối ưu hiệu suất của hệ thống.

Kết quả đầu ra của khung thức này là một đoạn mã hoặc một nguyên mẫu phần mềm (prototype) trong đó có chứa các lời gọi hàm cần thiết theo yêu cầu ban đầu của người sử dụng. Ví dụ, sau khi tìm ra lời giải tổng hợp cho yêu cầu được mô tả trong Phần 3.2, đặc tả và lời giải này lại được tiếp tục lưu trữ vào thư viện như một hàm mới (hàm $f_4$) để phục vụ cho những yêu cầu tìm kiếm và tổng hợp lời giải mới sau này.

## 4 MỘT SỐ BÀI TOÁN MINH HOẠ

Sau đây là một số ví dụ minh họa việc sử dụng đặc tả JML để đặc tả cho các yêu cầu tìm kiếm và gọi hàm tự động từ một số trường hợp thực tế.

***Trường hợp 1**: Tìm ngay được một hàm phù hợp, đáp ứng yêu cầu.*

Yêu cầu 1: "Cho một hình lập phương, cạnh có độ dài đại số là *a*. Tính thể tích của hình lập phương đó".

Chúng ta đặc tả yêu cầu trên như sau:

```
//@ requires a > 0
/*@ ensures \result > 0 &&
 @JMLDouble.approximatelyEqualTo
  @    (a^3, \result, 0.001);
 */
```

Bằng cách phân tích độ tương thích giữa hậu điều kiện (***ensure***) của yêu cầu với hậu điều kiện của các hàm có trong thư viện thì chúng ta nhận được một hàm phù hợp ngay, đó là hàm p*ow()* của lớp *java.lang.Math*. Hàm *pow()* này có đặc tả JML như sau[3]:

```
/*@ ensures
 @JMLDouble.approximatelyEqualTo
 @    (a^b, \result, 0.000001);
*/
public static double pow
           (double a, double b);
```

Tiền điều kiện của hàm *pow()* được bỏ qua nghĩa là tiền điều kiện luôn đúng. Đó đó, nó thỏa mãn với tiền điều kiện của yêu cầu. Như vậy *pow(a,3)* là hàm cần gọi.

Yêu cầu 2: "Chèn một phần tử *x* vào đầu Stack *S*".

Đặc tả JML cho yêu cầu:

```
//@ requires S.size<S.maxSize-1;
/*@ ensures
S.size==\old(S.size+1)
  @         &&
S.firstElement()==x;
 */
```

Với phương pháp tương tự như yêu cầu 1, khung thức cũng dễ dàng tìm được 1 hàm thỏa mãn yêu cầu trên là hàm *push()* của lớp Stack, với đặc tả như sau:

```
/*@ requires size < maxSize - 1;
 @ assignable size;
 @ ensures size = \old(size + 1)
         && S.firstElement()==x;
 @ ensures_redundantly (\forall
 @       int i; 0≤i && i<size-1;
 @element(i)==\old(element(i)));
 @*/
public void push(Object x);
```

***Trường hợp 2**: Cần tinh chế đặc tả mới có thể tìm ra lời gọi hàm.*

Yêu cầu 3: "Trả về vị trí của phần tử *x* trong mảng *a*. Nếu phần tử *x* không tồn tại trong mảng *a* thì trả về -1".

Đặc tả JML cho yêu cầu trên như sau:

---

[2] Để tính toán được rằng đặc tả của $f_2$ ($x \Rightarrow -x$) là tương đương với đặc tả của subgoal, chúng ta sẽ cần đến một *công cụ chứng minh* (prover). Đã có rất nhiều prover được phát triển trong cộng đồng nghiên cứu, nhưng trong khuôn khổ của bài báo này, chúng tôi không thảo luận đến các prover.

[3] Đặc tả này đã có trong thư viện về đặc tả của các hàm thư viện của Java.

```
//@ requires a != null;
/*@ ensures \result==-1 ||
 @  (\result≥0 &&
\result<a.length
 @        && a[\result] == x);
*/
```

Căn cứ vào hậu điều kiện của đặc tả yêu cầu trên, chúng ta sẽ tìm được hàm *binarySearch()* của lớp *java.util.Arrays* với đặc tả như sau:

```
/*@ requires a != null &&
  @  (\forall int i; 0<i &&
  @     i<a.length; a[i-1]≤a[i]);
  @ ensures \result==-1 ||
  @
(\result≥0&&\result<a.length
  @       && a[\result]==key);
*/
public static int binarySearch
              (int[] a, int
key);
```

Tuy nhiên, tiền điều kiện của hàm *binarySearch()* lại chưa phù hợp với tiền điều kiện của yêu cầu. Do đó, quá trình tìm kiếm và tổng hợp lời giải lại tiếp tục như cơ chế đã được mô tả trong phần 3.2. Quá trình tìm này sẽ thu được hàm *sort()* của lớp *java.util.Arrays*. Hàm *sort()* có đặc tả như sau:

```
/*@ requires a != null;
  @ ensures  a != null &&
  @  (\forall int i; 0<i &&
      i<a.length; a[i-
1]≤a[i]);*/
public static void sort(int[]
a);
```

Đến đây, tiền điều kiện của hàm *sort()* này hoàn toàn phù hợp với tiền điều kiện của yêu cầu. Như vậy quá trình tìm kiếm sẽ kết thúc và chúng ta có một kế hoạch gọi thực thi các hàm như sau:

```
java.util.Arrays.sort(a);
java.util.Arrays.binarySearch(a,
x);
```

## 5 KẾT LUẬN

Việc áp dụng ngôn ngữ đặc tả hình thức vào cơ chế tìm kiếm và tái sử dụng thành phần tự động là một hướng nghiên cứu mới, giúp cho kết quả tìm kiếm sẽ tốt hơn so với các nghiên cứu dựa trên từ khóa. Cho đến nay, các nghiên cứu về việc ứng dụng đặc tả hình thức, đặc biệt là với hướng tiếp cận thoả thuận, chỉ chủ yếu tập trung vào việc kiểm chứng tính đúng của phần mềm. Trong bài báo này, chúng tôi đề xuất sử dụng các đặc tả hình thức vào việc tìm kiếm và tổng hợp các thành phần cần thiết theo nhu cầu của người lập trình. Từ đó, chúng tôi trình bày một khung thức để tìm kiếm và tổng hợp các API được mô tả bằng ngôn ngữ JML. Hơn thế nữa, cơ chế hoạt động của khung thức này cho phép lưu trữ và bổ sung các hàm thư viện mới từ kết quả tổng hợp được. Một số thí nghiệm ban đầu với những bài toán lập trình thường gặp đã thu được kết quả khả quan trên khung thức của chúng tôi. Điều này khích lệ chúng tôi công bố kết quả nghiên cứu của mình để thu thập thêm ý kiến đóng góp từ cộng đồng.

Hiện tại, khung thức do chúng tôi đề nghị chủ yếu dựa trên đặc tả JML và sử dụng để tìm kiếm và tổng hợp các hàm API cho ngôn ngữ Java. Tuy nhiên, do bản chất của hình thức đặc tả trong khung thức là các công thức logic Hoare [16] nên khung thức này hoàn toàn có thể tuỳ biến hiệu quả để áp dụng cho các ngôn ngữ lập trình dạng câu lệnh (imperative) khác như C, C++, C#, Scala, Pascal …

## TÀI LIỆU THAM KHẢO


1. M. D. McIlroy, J. M. Buxton, P. Naur and B. Randell, 1968. Mass-produced software components. *In Proceedings of the 1st International Conference on Software Engineering*, pp: 88–98.
2. Sametinger, Johannes, 1997. *Software engineering with reusable components*. Springer, pp: 11-15.
3. Tran DinhHuy and HuaPhung Nguyen, 2013. API specification-based function search engine using natural language query. *Computing, Management and Telecommunications, International Conference on IEEE*, pp:140-145.
4. Hoffmann, Raphael, J. Fogarty, and D.S. Weld, 2007. Assieme: finding and leveraging implicit references in a web search interface for programmers. *Proc. of the 20th annual ACM symposium on User interface software*, pp:13-22.
5. Chatterjee, Shaunak, S. Juvekar, and K. Sen, 2009. Sniff: A search engine for java using free-form queries. In: *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, pp: 385-400.
6. B. Randell, 1973. *The Origin of Digital Computers*. Springer-Verlag, p. 298.

7. Meyer, Bertrand, 2002. *Design by Contract*. Prentice Hall.
8. D.S. Weld, 1999. Recent Advances in AI Planning. *AI Magazine*, Vol.20: 2-93.
9. R. Floyd, 1976. Assigning Meanings to Programs. *In Mathematical Aspects of Computer Science. Proc. Syrup. Appl. Maths*. Vol.19:19-32.
10. C.A.R. Hoare, 1969. An Axiomatic Basis for Computer Programming. In: *Comm. ACM*, Vol. 12: 576-583.
11. P. Naur, 1969. Proofs of algorithms by General Snapshots. BIT Numerical Mathematics. Vol.6.4: 310-316.
12. E.M. Clarke, J.M. Wing *et al*, 1996. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, Vol.28: 626-643.
13. J.M. Wing, J. Woodcock and J. Davies, 1999. FM-99: Worm Conference on Formal Methods in the Development of Computing Systems. *LNCS 1708 and 1709*, Springer-Verlag.
14. Burdy, Lilian, *et al*., 2005. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*. Vol.7: 212-232.
15. J.V. Guttag and J.J. Horning, 1993. *LARCH: Languages and Tools for Formal Specification*. Springer-Verlag.
16. Jacobs, Bart, and E. Poll, 2001. *A logic for the Java Modeling Language JML*. Springer Berlin.
17. Barnett, et.al, 2005. The Spec# programming system: An overview. *Construction and analysis of safe, secure, and interoperable smart devices*. Springer Berlin Heidelberg, pp:49-69.
18. Barnett, Mike, *et al*., 2006. Boogie: A modular reusable verifier for object-oriented programs. *Formal methods for Components and Objects*. Springer Berlin Heidelberg. pp: 364-387.
19. Gopan, Denis, and Thomas, 2007. Guided static analysis. In: *Static Analysis*, Springer Berlin Heidelberg, pp: 349-365.
20. L. Correnson, *et al*., 2013. *Frama-C User Manual*, Software Safety Laboratory, Saclay.
21. P. Baudin, et.al, 2012. *ACSL: ANSI/ISO C Specification Language*. Version 1.6.
22. Kuncak, Viktor, and M. Rinard, 2006. An overview of the Jahob analysis system: project goals and current status. In: *Parallel and Distributed Processing Symposium*. 8 pp
23. Watt, A. David, B.A. Wichmann and W. Findlay, 1987. *Ada language and methodology*. Prentice Hall International (UK).
24. Barnes and John, 2011. Rationale for Ada 2012: Contracts and aspects. *ADA USER*, Vol.32 (4): 247-264.
25. C. Flanagan, *et.al*, 2002. Extended static checking for Java. *Proc. of the Conference on Programming Language Design and Implementation*, pp: 234-245.
26. Beckert, Bernhard, R. Hähnle, and P.H. Schmitt, 2007. *Verification of object-oriented software: The KeY approach*. Springer-Verlag.
27. Rish, Irina. An empirical study of the naive Bayes classifier. *IJCAI 2001 workshop on empirical methods in artificial intelligence*. Vol.3: 41-46.