Hoare Logic-based Genetic Programming

He Pei[1, 2] Kang Lishan[1] Colin G. Johnson[3] Ying Shi[1]

[1]State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, P. R. China

[2]School of Computer and Communication Engineering, Changsha University of Science and Technology Changsha 410076, P. R. China

[3]Computing Laboratory, University of Kent, Canterbury, CT2 7NF, England

Abstract. Almost all existing genetic programming systems deal with fitness evaluation solely by testing. In this paper, by contrast, we present an original approach that combines genetic programming with Hoare logic with the aid of model checking and finite state automata, henceby proposing a brand new verification-focused formal genetic programming system that makes it possible to evolve reliable programs with mathematically-verified properties.

Keywords. Genetic programming; program verification; Hoare logic; model checking; finite state automata.

Charles Rich and Richard C. Waters [1] have classified automatic programming techniques into four kinds: procedure, deduction, transformation, and inspection. Although deduction methods are the most important for dealing with simple problems, they also claim that these deduction methods cannot play an important role in more complex automatic programming challenges until they are combined with other methods.

Genetic programming (GP) was one of the most important automatic programming approaches, and was first studied in detail by John R. Koza [2] in 1992. It is based on John Holland's idea of genetic algorithms (GAs) [3-4]. Subsequently, a number of variants [5-6] have been developed, including MEP(Multi-Expression Programming), GEP(Gene Expression Programming), ADF-GP(Automatic Defined Function Genetic Programming), STGP (Strongly Typed Genetic Programming), LGP (Linear Genetic Programming), etc. Applications of GP are manifold: automatic design, pattern recognition, circuit design, cognitive theory, robot control, to name just a few, as well as multi-objective optimization problems [2, 5-15].

GP, it could be said, is an *illogical* method. Although there are many variants on GP, as far as the core problem solving processes are concerned, these methods all base their fitness evaluation on testing the programs in the population on a sample of test data [16-17]. If such methods are going to be developed further and be applied to safety-critical domains, then it is important to combine these approaches with logic-based approaches for proving and verifying program properties.

This paper is dedicated to introducing a novel verification-focused GP method: *Hoare Logic-based*

Received: 2009-
*This work was supported by the National Natural Science Foundation of China under Grant No. 60473081 and the State Key Laboratory of Software Engineering, Wuhan University under Grant No. SKLSE 20080701.
E-mail: bk_he@126.com

Genetic Programming (*HGP*). Very few previous GP systems have taken this approach, but it is beginning to be recognised. The authors of this paper first noted the heavy dependence of GP upon test and proposed alternative approaches [17-18]. Colin G. Johnson first introduced model checking into GP [17] in 2007, and this approach has been taken further by Gal Katz and Doron A. Peled [19]. In the present paper, we collaborate together to elaborate on our original studies on formal GP by linking GP with formal approaches such as Hoare logic, model checking, and finite state automaton. This kind of formal GP possesses good features of both deductive and evolutionary methods, and is therefore sharply different from traditional "illogical" GP systems [2, 7-9]. In addition, this new system also allows GP systems to incorporate concepts such as *components*, which are widely used in software specification. Since the novel GP system, Hoare logic-based GP, is essentially a program generation method based on Hoare's semantics, we call it *HGP* for brevity in subsequent discussion.

1 Motivation and Related Works

GP is essentially a GA [13] which applies evolutionary operators to populations consisting of computer programs; most typically, these programs are represented as parse trees. Papers [7-9, 11, 13] introduce many human-competitive results from real-world applications of GP, and paper [7], specifically, gives eight criteria deciding whether the product of a GP system should be regarded as human-competitive. With automatic programming, John Koza believes that search-based processes based on evolution are more more effective and fruitful than logic-based approaches.

GP breeds computer programs to solve given problems as follows [8].

（1） Generate an initial population of programs using random composition of the functions and terminals drawn from a function set and terminal set.

（2） Iteratively perform the following substeps until the terminal criterion has been satisfied:

（A） Execute each program in the population and assign it a fitness value using the fitness measure, which will depend on the problem at hand.

（B） Create a new population of computer programs by applying the following operations. The operations are applied to computer programs selected from the population with a probability based on fitness (a number of different selection schemes are found in the literature):

（i） Reproduction: Copy an existing program to the new population.

（ii） Crossover: Create new offspring program(s) for the new population by recombining randomly chosen parts of two existing programs.

（iii） Mutation: Create one new offspring program for the new population by randomly mutating a randomly chosen part of one existing program.

（3） The program that has the highest fitness in the final population is designated as the result of the genetic programming system for the run. This result may be a solution (or approximate solution) to the problem.

All other types of GPs such as MEP, GE, GEP, ADF-GP, STGP, LGP, etc. are derivations from this classical model. For instance, when expressing individual with concepts such as genotype and phenotype, we get MEP, GE, GEP and LGP; and focusing on either functional reusability or the "closure property" [16] of canonical GP, we naturally introduce ADF-GP or STGP.

To verify whether a program or an approximate solution obtained from evolutionary methods satisfies some pre-given requirement, traditional GP relies on executing the programs in the population using a set of test data, and then comparing the end results of that execution with a set of expected outputs for that given set of test data. From the point of view of software engineering, this is not *verification* of the program, but rather software testing. So, evolving programs this way is unable to safeguard

software reliability.

Software reliability [20] is an important issue of common concern among researchers all over the world. The most common solutions to it include software testing, component oriented development, and formalisation of the software engineering process. As E. W. Dijkstra [21] put it, program testing can be used to show the presence of bugs, but never to show their absence. So safety-critical software systems depend much more on mathematical proofs, i.e. formal verifications, to guarantee their soundness.

Generally speaking, computer aided proving approaches can be put into two kinds: the proof based approaches and those based on model checking [22-24]. The strength of the latter is their high level of automation. Since they have a close relationship with certain temporal logic languages [25], their expressive power is weakened to some extent. For example, they are not suited to handling changes in values [26]. Consequently, Willem Visser et al. has pointed out in [27] that model checking can be best applied to the designs rather than the implementations.

Hoare logic [28] is the most important representative of proof based approaches. It describes program properties in the first-order predicate logic, relying strongly on automated theorem proving (ATP) techniques, and therefore is inferior to model checking in terms of automation. The major reasons for choosing Hoare logic as our work basis are its strong expressiveness, deducibility, and applicability [29-30].

In Hoare logic, a *Hoare formula* or *triple* is of the form $\{P\}S\{Q\}$. Where P, Q are first-order predicates, called pre- and post-conditions; S stands for a program segment. $\{P\}S\{Q\}$ means: given that P holds before execution of S, and that the execution of S can terminate, then Q will hold. Hoare logic [28, 31] (figure 1) includes six proof rules from which program verifications can be carried out. In practical applications, however, we often use proof tableaux [25] in place of the tree-like style of proofs. Figure 2 gives an example of proof tableaux.

Skip Statement:	$\{P\}\ Skip\ \{P\}$
Assignment:	$\{\ P[t/x]\}x := t\{P\}$
If-statement:	$\dfrac{\{P \wedge e\}S1\{Q\},\ \{P \wedge \neg e\}S2\{Q\}}{\{P\}\ \text{if}\ e\ \text{then}\ S1\ \text{else}\ S2\ \{Q\}}$
Repetition:	$\dfrac{\{P \wedge e\}\ S\ \{P\}}{\{P\}\ \text{while}\ e\ \text{do}\ S\ \{P \wedge \sim e\}}$
Composition:	$\dfrac{\{P\}S1\ \{R\}, \{R\}\ S2\ \{Q\}}{\{P\}\ S1; S2\ \{Q\}}$
Rewriting:	$\dfrac{P \rightarrow P1, \{P1\}\ S\ \{Q1\}, Q1 \rightarrow Q}{\{P\}\ S\ \{Q\}}$

Figure 1. Proof rules of Hoare logic. Where *skip*, also denoted ε, stands for empty statement.

{ y=1 ∧ 5 = 5 }
z := 5;
{ y=1 ∧ z=5 }
{ y+z =6 }
y := y+z;
{ y=6 }

Figure 2. Sample of proof tableaux

A major task for automatic programming is reusability [1]. As a key technology in the development of software industry and economies of scale [32], component approaches are no doubt important practical activities in this aspect. *HGP* also considers components and reuse [33]. As proof and verification are so complicated, it is *unwise to prove everything from scratch*. Consequently, *HGP* is based on the principle that code should be reused, and more importantly, so should the proofs.

In recent papers [17,18] we have explored the use of various approaches (Hoare logic, model checking, and the theory of automata [34,35]) as a way of formalising the process of fitness evaluation in GP. In this paper we extend the work introduced in [18] that uses Hoare logic as the basis of fitness evaluation. Here we weaken some of its restrictions to obtain, on one hand, scalability, and on the other to present a distributed parallel algorithm for fitness evaluation. All of these are original. In fact, all

traditional GPs whose fitness calculations are based on the principle of executions cannot do this. In terms of fitness evaluation, *HGP* uses *verification*: this is its essential difference when compared to traditional GP. *HGP* first accepts pre- and post-conditions, then evolves Hoare formulae based on requirements specifications. Once an evolved result like $\{P_1, P_2, \ldots, P_n\}\ \{f\}_S\ \{Q_1, Q_2, \ldots, Q_m\}$ is found in the search space, it can tell us with certainty that $\{ P_1 \wedge P_2 \wedge \cdots \wedge P_n \}\ f\ \{ Q_1 \wedge Q_2 \wedge \cdots \wedge Q_m \}$ is a Hoare triple, i.e. f is correct with respect to its pre-condition $\{ P_1 \wedge P_2 \wedge \cdots \wedge P_n \}$ and post-condition $\{ Q_1 \wedge Q_2 \wedge \cdots \wedge Q_m \}$. As for fitness evaluation, it relies directly on relation calculations, supporting distributed parallel evaluation of fitness at an arbitrarily fine granularity. *HGP* will now be introduced: first the language, then the verification framework, followed by the GP concepts.

2 The Language of Components

The language of components used in *HGP*, denoted by $L_C(F)$, is a language of *while programs* [28] restricted to a given set of components $F = \{ f_i \mid i = 1, 2, \ldots, n\}$. Its grammar is as follows. Note that the components in F can be regarded as either a while program, a program in some other language, or even an executable code. In short, they are transparent.

$P \rightarrow f_1 \mid f_2 \mid \ldots \mid f_n \quad \mid \quad$ if C $\{P\}$ else $\{P\} \quad \mid \quad$ while C $\{P\} \quad \mid \quad P\ ;\ P$

Where C stands for Boolean expression.

3 Search Space

In the following, we will define the search space or verification task, denoted *STP**, under a closed environment for the component language given above. What we have done here is to weaken the restriction in [18]; the result, nevertheless, is scalable. This extension to Hoare's convention by introducing the so called generalized concept is done only for proof reuse and evolutionary generation of programs. For convenience, the discussion proceeds in functional form rather than assignments.

Definition 3.1 (Scalable formula) A formula of the form $P\ \{f\}_S\ Q$ is a scalable (Hoare) formula, if f is a program segment, and P, Q sets of logic formulas satisfying the following: there exist some q in Q, and at least a p in P such that $\{p\}\ f\ \{q\}$ forms a Hoare triple. In this case, P, Q are called the generalized pre- and post-conditions respectively; $\{p\}\ f\ \{q\}$ an instance of $P\ \{f\}_S\ Q$.

Definition 3.2 (Instance) Let H be a set of Hoare triples, and $\wedge X$ mean a conjunction of all elements (logic formulas) in X. A Hoare triple $\{\wedge R\}\ f\ \{\wedge K\}$ is an instance of some scalable formula $P\ \{f\}_S\ Q$ under H, if R, K are nonempty subsets of P and Q satisfying that for any $k \in K$, there exists a $r \in R$ such that $\{r\}\ f\ \{k\} \in H$.

Definition 3.3 (Scalable representation) Let H be a set of Hoare triples. A set S_H of scalable formulas is a scalable representation of H, if each $h \in H$ is an instance of some $s \in S_H$ under H and no two distinct elements in S_H share the same program segment.

Obviously, the scalable representation for a given set H of Hoare triples is not unique, but is uniquely defined with respect to a given program segment.

Definition 3.4 (Search space) Given a set H of Hoare triples and its scalable representation S_H, the search space *STP** is a set constructed by applying the following rules a finite number of times.

1) $\{/\backslash M\}\varepsilon\ \{/\backslash M\} \in STP^*$ for $P\ \{f\}_S\ Q$ in S_H with $M \subseteq P$ or $M \subseteq Q$;
2) Instances of some scalable formula in S_H under H are all in *STP**;
3) $\{/\backslash P\}\ f\ ;\ g\ \{/\backslash W\}$ ($\{/\backslash P\}\ fg\ \{/\backslash W\}$ for short) in *STP**, if $\{/\backslash P\}\ f\ \{/\backslash Q\}$, $\{/\backslash R\}\ g\ \{/\backslash W\}$ in *STP** satisfy that for each $r \in R$, there exists a $q \in Q$ such that $\mid\!- q \rightarrow r$.

In this case, elements in *STP** are either called the generalized results of H or provable under S_H, a gen-

eralized closed environment with scalability.

Theorem 1 Given H, a set of Hoare triples, and its scalable representation S_H, *STP** under H is a set of Hoare triples.

This is easy to demonstrate. In fact, for case 1 of definition 3.4, the proof is trivial; for cases 2 and 3, we can consult definition 3.2, and apply both composition and rewriting rules of figure 1.

4 Model of Search Space

In this section we will introduce a model based approach to verification and generation of reliable programs in *STP**. In order, the topics are the modeling principle, method, proof, parallel verification, and scalability.

4.1 Modeling Approach

4.1.1 Principle

As discussed [25] by Michael Huth and Mark Ryan, verification techniques can be thought of as comprising three parts:

- A framework for modeling systems, typically a description language of some sort;
- A specification language for describing the properties to be verified;
- A verification method to establish whether the description of a system satisfies the specification.

From this point of view, we can compare the two commonly used formal approaches like Hoare logic and model checking in columns 2 and 3 of table 1. As we know, to conduct proofs for some assertions often requires sophisticated guidance and expertise from the user. Since model checking is decidable, we consider it advantageous to functionally employ these two approaches to give us the advantages of both. Based on these ideas, the formal basis for the *HGP* system uses the three ideas outlined in column 4 of table 1. Now let's illustrate how *HGP* is developed in the context of them.

Table 1 Comparison of relevant formal approaches

	Hoare Logic	Model Checking	Formal basis of *HGP*
The formal framework	Hoare triples	Finite state transition system	Finite state automaton
The specification language	Hoare triple	Temporal logic	Hoare triple
The verification method	A calculus	Model checking algorithm	Verification algorithm

First of all, we must clarify what are transition system and model checking. As far as transition system is concerned, we mean a finite state automaton depicted by a mathematical model (or a labeled directed graph called a transition diagram) that consists of [34, 35]: 1) a set of states S; 2) a set of input symbols; 3) transitions among states in response to inputs; 4) a start state $s_0 \in S$, and 5) a set of final states in S. Transition systems are useful for compiler implementations and protocol verifications, etc.

In general, model checking is a model-based approach to program verification in which the system and the property of concern are represented by a model and a statement ϕ of some specification language; and the task is to compute whether a model M satisfies ϕ (written $M \models \phi$). When a transition system is chosen for M, this can more explicitly be restated as the following proposition [25]: model checking is the process of computing an answer to the question of whether $M, s \models \phi$ holds, where ϕ is a formula of some logics, M is an appropriate model of the system under consideration, s is a state of that model and $\models$ is the underlying satisfaction relation. So substituting some temporal logic for ϕ, we will get the commonly used model checking of table 1.

Similarly, we can obtain another kind of model checking approach (column 4 of table 1) if we link together the ideas of a transition system M, Hoare triple $\{P\}\ f\ \{Q\}$, and some appropriate model

checker which checks the satisfaction of M, $s \models \{P\}\ f\ \{Q\}$ with respect to theorem 2 for some state s in M. To this end, we must do the following things:

a) Model the before mentioned search space *STP** using transition system or finite state automaton, arriving at a scalable model $SM(H)$ as shown in subsection 4.1.2 and section 6;
b) Describe the property of a program system f of concern using some Hoare triple h: $\{P\}\ f\ \{Q\}$.
c) Design a model checker (or verification algorithm) to check whether $SM(H)$, $s \models h$ holds (i.e. $h \in STP^*$) .

Regarding these problems, one can refer to subsections 4.1.2, 4.2, and section 6. Notice that a scalable model $SM(H)$ is a common model of all program systems involved in *STP**. As for the usage of the idea of finite state automaton, consider $SM(H)$. Treating all states of $SM(H)$, e.g. figure 5 of section 6, as final states and choosing one from them, e.g. GC_1 of figure 5, as a start state, we will obtain a finite state automaton on which the well defined model checker or verification algorithm (see subsection 4.2) computes, with inputs $SM(H)$, GC_1 and h, say $\{ P_1 \wedge P_2 \wedge P_3\}\ f_1\ ;\ f_3\ ;\ f_2\ ;\ f_4\ \{ P_1 \wedge P_5 \wedge P_7\}$, whether $SM(H)$, $GC_1 \models h$ holds. This computation process is also reflected in generation 9 of figure 6 where it means a proof of $SM(H)$, $GC_1 \models \{ \wedge \{P_1, P_2, P_3\}\}\ f_1; f_3; f_2; f_4\ \{ \wedge \{P_1, P_5, P_7\}\}$. In short, solving of the maximum expansion *ePost* for some program, say f, over a given generalized pre-condition *Gpre* is essentially a Hoare logic approach to the verification of $SM(H)$, $s \models \{ \wedge Gpre \}\ f\ \{ \wedge ePost \}$ for state s in $SM(H)$.

4.1.2 Method

Definition 4.1 (Passage) Let H be a set of Hoare triples, S_H its scalable representation. Again let $G = \langle V, E \rangle$ be a finite transition graph whose vertices represent sets (generalized pre-/post-conditions) of logic formulas, and edges are labelled either by an "f", a program segment of some scalable formula in S_H , or a "ε" symbol. A path $V_1 f_1 V_2 f_2 \ldots f_{n-1} V_n$ in G is a passage, if it defines the following maximum expansion function $m(V_i)$ for some nonempty subset P $(\subseteq V_1)$:

1) $m(V_1) = P \neq \varnothing$;

2) $$m(V_i) = \begin{cases} \{x \in V_i \mid \exists p \in m(V_{i-1})(p \to x)\} \neq \varnothing & V_{i-1} \text{ and } V_i \text{ are linked by } \varepsilon \\ \{x \in V_i \mid \exists p \in m(V_{i-1})(\{p\}\ f\ \{x\} \in H)\} \neq \varnothing & V_{i-1} \text{ and } V_i \text{ are linked by } f \end{cases} \quad \text{for } 2 \le i \le n$$

where V_i stands for vertex of G; the string α $(= f_1 f_2 \ldots f_{n-1})$ concatenated from edge labels along the passage is called a generalized body; $m(V_i)$ is the maximum expansion of $f_1 f_2 \ldots f_{i-1}$ on P.

Definition 4.2 (Scalable model) Given H, S_H , $G = \langle V, E \rangle$ as above, G is a scalable model for verification of *STP** under H , denoted by $SM(H)$, if for any program segment f and nonempty subsets P, Q of some two vertices, we have: $\{\wedge P\}\ f\ \{\wedge Q\} \in STP^* \Leftrightarrow$ there exists a passage in G with f as its generalized body and Q a subset of the maximum expansion of f on P.

Definition 4.3 (Generalized p-implication) Given two sets P, Q of logic formulas, they have generalized p-implication relationship, denoted $|\!-P \xrightarrow[p]{} Q$, if there exist two nonempty sets $S_1(\subseteq P)$, $S_2(\subseteq Q)$ such that $S_2 = \{q \in Q \mid \exists p \in S_1(p \to q)\} \neq \varnothing$.

Theorem 2 Given H, S_H as above, there exists a scalable model $SM(H)$ for *STP**.

Proof. Without loss of generality, assuming the predicates involved in H are in $\{P_1 , P_2, \ldots, P_n, Q_1, Q_2, \ldots, Q_n\}$, and $S_H = \{ R_i\ \{ f_i \}_S\ W_i \mid\ 1 \le i \le m \}$. We first construct the model and then providing the proof.

Step 1: Construction of $SM(H)$.

1) Constructing matrices of predicate relation and generalized relation in tables 2 and 3;
2) Drawing a node for each set of predicates in $\{R_i \mid 1 \le i \le m\} \cup \{W_i \mid 1 \le i \le m\}$;
3) Drawing an arrow which is labelled as "f" from R *to* W, if $R\{f\}_S W \in S_H$;

Table 2 Predicate relation

$\xrightarrow[S]{}$	P_1 ... P_n Q_1 ... Q_n
P_1 ... P_n Q_1 ... Q_n	$X_i \xrightarrow[S]{} Y_j = \begin{cases} T & \mid -X_i \to Y_j \\ F & others \end{cases}$ Where X, Y are P *or* Q, and $1 \le i, j \le n$. Technically, the calculation of table 3 relies on table 2.

Table 3 Generalized relation

$\xrightarrow[g]{}$	R_1 ... R_m W_1 ... W_m
R_1 ... R_m W_1 ... W_m	$X_i \xrightarrow[g]{} Y_j = \begin{cases} T & if \mid -X_i \xrightarrow[P]{} Y_j \\ F & others \end{cases}$ Where X , Y are generalized pre- or post- conditions R or W , and $1 \le i, j \le m$.

4) Drawing a ε arrow from X_i to Y_j , if $X_i \xrightarrow[g]{} Y_j = T$. Here X, Y stands for either R or W.

The graph obtained above is $SM(H)$.

Step 2: Showing that for nonempty subsets K, L of two generalized conditions, $\{\wedge K\} f \{\wedge L\} \in STP^* \Leftrightarrow$ there exists a passage in $SM(H)$ taking f as its generalized body and L the subset of its maximum expansion on K.

=>: By induction on the composition.

1) Base: it is trivial for rules 1) and 2) in definition 3.4.
2) Induction step: assuming $\{\wedge K\} f_1 \{\wedge P\}$, $\{\wedge Q\} f_2 \{\wedge L\} \in STP^*$, by induction hypothesis, there exist two passages, say $R_1 e_1 R_2 e_2 \cdots e_{u-1} R_u$ and $W_1 g_1 W_2 g_2 \cdots g_{n-1} W_n$, which take $f_1 = e_1 e_2 \cdots e_{u-1}$, $f_2 = g_1 g_2 \cdots g_{n-1}$ as their generalized body on the one hand, $m(R_i)$ $(\varnothing \ne m(R_i) \subseteq R_i, 1 \le i \le u)$ and $m'(W_j)$ $(\varnothing \ne m'(W_j) \subseteq W_j$, $1 \le j \le n)$ the maximum expansion of $e_1 e_2 \cdots e_{i-1}$ and $g_1 g_2 \cdots g_{j-1}$ on K and Q on the other. As such, $K = m(R_1)$, $Q = m'(W_1)$ and $P \subseteq m(R_u)$, $L \subseteq m'(W_n)$. Thus $\varnothing \ne Q = m'(W_1) \subseteq \{w \in W_1 \mid \exists p \in P(p \to w)\} \subseteq \{w \in W_1 \mid \exists p \in m(R_u)(p \to w)\} = m(W_1)$, if the two Hoare formulas can be combined into $\{\wedge K\} f_1 f_2 \{\wedge L\} \in STP^*$, i.e. for each $q \in Q$, there is a $p \in P$ such that $\mid -p \to q$. Again by definition 4.3 and the drawing method of $SM(H)$, we have $R_u \xrightarrow[p]{} W_1$, which means there exists a ε arrow between R_u and W_1. Consequently, we can construct a new maximum expansion $m(W_j)$ for $g_1 g_2 \cdots g_{j-1}$ on $m(W_1)$ such that $\varnothing \ne m'(W_j) \subseteq m(W_j)$ $(1 \le j \le n)$. Hence combining them with all of those maximum expansions related to f_1, say m (R_i) s, we will get from induction hypothesis the proof for $R_1 e_1 R_2 e_2 \cdots e_{u-1} R_u$ ε $W_1 g_1 W_2 g_2 \cdots g_{n-1} W_n$ being a passage satisfying $m(R_1) = K$ and $L \subseteq m(W_n)$. The composition of two passages is shown in figure 3.

$$
\begin{array}{ccccccccccccc}
R_1 & e_1 & \cdots & e_{u-1} & R_u & & & & W_1 & g_1 & \cdots & g_{n-1} & W_n \\
\cup \mid & & \cdots & & \cup \mid & & & & \cup \mid & & \cdots & & \cup \mid \\
m(R_1) & & \cdots & & m(R_u) & \{w \in W_1 \mid \exists p \in m(R_u)(p \to w)\} & = & & m(W_1) & & \cdots & & m(W_n) \\
\| & & \cdots & & \cup \mid & \cup \mid & & & \cup \mid & & \cdots & & \cup \mid \\
K \ne \varnothing & e_1 & \cdots & e_{u-1} & P \ne \varnothing & \{w \in W_1 \mid \exists p \in P(p \to w)\} & \supseteq & & m'(W_1) & & \cdots & & m'(W_n) \\
 & & & & & & & & \| & & \cdots & & \cup \mid \\
 & & & & & & & & Q \ne \varnothing & g_1 & \cdots & g_{n-1} & L \ne \varnothing
\end{array}
$$

Figure 3. Composition of Two Passages

<=: By induction on the number of edges in a passage.

1) Base: when the passage contains only one edge, the proof is trivial.
2) Induction step: suppose that V_1 $e_1 V_2$ $e_2 ... e_{n-1} V_n$ $x V_{n+1}$ is a passage in $SM(H)$ taking

$f = e_1 e_2 \cdots e_{n-1} x$ as its generalized body such that $m(V_1) = K \neq \varnothing$, $\varnothing \neq L \subseteq m(V_{n+1})$. Where V_i stands for the vertex or generalized condition, e_i and x for edges. By induction hypothesis, we have $\{/\backslash K\}\ e_1 e_2 ... e_{n-1} \{/\backslash m(V_n)\}, \{/\backslash m(V_n)\} x\ \{/\backslash L\} \in STP^*$. Again by the definition of *STP**, it follows easily $\{/\backslash K\} f \{/\backslash L\} \in STP^*$. This completes the proof.

4.2 Parallel Verification Algorithm

A parallel verification algorithm carried out on the concerned model is given below. For the sequential algorithm, one can refer to [18].

Algorithm 4.1 Given $H = \{\{X_j\} f_j \{Y_j\} \mid 1 \le j \le k\}$, $S_H = \{Z_i \{f_i\}_S W_i \mid 1 \le i \le q\}$, $SM(H)$ as above, the algorithm for parallel verification of $\{P\}\alpha\{Q\} \in STP^*$ ($\alpha = f_1 \cdots f_n \in \{f_1, f_2, \cdots, f_q\}^*$) is as follows.

1) Solving $m(Z_1) = \{x \in Z_1 \mid P \rightarrow x\}$ for $Z_1 \{f_1\}_S W_1 \in S_H$. Because there is only one edge in $SM(H)$ annotated by "f";
2) Solving R[f_i]= $\{(t, e_t) \mid t \in Z_i$, e_t is the maximum expansion of f_i on $\{t\} \subseteq Z_i\}$ for $f_i \in \{f_1, f_2, ..., f_q\}$
3) Solving R[α] for $\alpha = f_1 \cdots f_n$ ($|\alpha| \ge 1$) of the form $\bigcup_{t \in Z_1} \{(t, e_t)\}$ (e_t is the maximum expansion of α on $\{t\}$) in parallel with the algorithm of figure 4.

```
CalculateExpansion(α, H, SM(H))
begin
  Let |α| be the length of α;
  if α ∈ {f1, f2, ..., fq} then return(R[α])
    else
      begin
        divide α into 2 halves: lHalf= f1 ... f⌊|α|/2⌋ and rHalf= f⌊|α|/2⌋+1 ... fn;
        if there is no ε arrow linking W⌊|α|/2⌋ and Z⌊|α|/2⌋+1
            then return( {"No"})
            else
              begin
                Solve R1 and R2 in parallel for lHalf and rHalf as follows:
                    R1= CalculateExpansion(lHalf, H, SM(H)),
                    R2= CalculateExpansion(rHalf, H, SM(H));
                R[α]:= {};
                for each (t, et) ∈ R1 do
                    if there are e∈ et and (u, v) ∈ R2 such that e → u
                        then R[α]:=R[α] ∪ {(t, ⋃_{∃e∈et :(e→t')∧(t', et', )∈R2} et' )} ;
                return( R[α] )
              end
      end
end;
```

Figure 4. Parallel algorithm for calculation of the maximum expansion

4) Interpreting the returned result as follows. If the result is {No}, then the goal to be verified is wrong with respect to S_H; if the result is R[α] and Q'= $\bigcup_{\exists t\in m(Z_1):(t\to u)\wedge(u,e_u)\in R[\alpha]} e_u$ satisfying $|-\wedge Q'\to Q$, then $\{P\}\alpha\{Q\}$ is correct; otherwise, the goal to be verified is unprovable under S_H.

4.3 Verification of $L_C(F)$

The principle for adapting the linear model to verify both the branch and iteration statements in $L_C(F)$ is the multilayer strategy. That is, we verify program segments layer by layer: first proving some inner segments, and then their immediate outsides. Algorithm 4.2 is based on this principle. Of course, we should treat "if" and "while" statements as the same control structure when dealing with such concepts as layer number and nested depth.

Algorithm 4.2 Given *SM*(*H*), to verify an arbitrary program *P* composed of components from *H*, we proceed in the following way:

(1) Initialize *k* (nested depth) as 1, and gathering all iterations and branch statements involved in *P*, denoted *IB*(*P*), based on the following formula;

$$IB(P)=\begin{cases}\{\} & \textit{neither} \text{ iteration nor branch statement in } P\\ \{\textit{while } C\{P_1\}\}\cup IB(P_1) & P \text{ is while } C\ \{\ P_1\ \}\\ \{\text{if } C\ \{P_1\}\ \text{else}\ \{P_2\}\}\cup IB\{(P_1)\cup IB(P_2) & P \text{ is if } C\{P_1\}\text{else}\{P_2\}\\ IB(P_1)\cup IB(P_2) & P \text{ is } P_1;P_2\end{cases}$$

(2) Verify all program segments of depth *k* in *IB*(P) based on the algorithm given in the previous subsection under the current *SM*(*H*);

(3) Maintain *SM*(*H*)by adding what were achieved in step (2) either as scalable formulas or properties into the current *SM*(*H*).

(4) *k*:= *k*+1; if there still remains some program segment of depth *k* in *IB*(P) untouched, go back to step (2).

(5) Verify the original program in *SM*(*H*)

4.4 Scalability

From what was discussed above, it follows that a model may contain many states or ε arrows. To overcome this shortcoming, we can apply the following property.

Definition 4.4 (Subformula) Given two scalable formulas F_1: $P\{f_1\}_S Q$ and F_2: $R\{f_2\}_S W$, F_1 is a *subformula* of F_2, denoted $F_1\subseteq F_2$, if $P\subseteq R$, $Q\subseteq W$, and $f_1 = f_2$.

Definition 4.5 (Subrepresentation) Let *H* be a set of Hoare triples and S_H, S'_H its scalable representations. S_H is a *subrepresentation* of S'_H, denoted $S_H\subseteq S'_H$, if for each $S\in S_H$ there exists a S'$\in S'_H$ such that $S\subseteq S'$.

Similarly, we can define the concept *submodel*. Furthermore, according to these definitions, we have:

Theorem 3 Given *H* such that $S_H\subseteq S'_H$, if *STP**, *STP'** are the search spaces of S_H and S'_H, then *STP**$\subseteq$*STP'**.

5 *HGP*: A Novel Formal GP

By the model's existence theorem that given a set *H* of Hoare triples and its scalable representation S_H, there must exists a scalable model *SM*(*H*) for *STP**, we can infer that the formal model obtained

from the use of the multilayer principle can not only be used to verify, but also to generate numerous reliable programs in the well defined search space. As we know, to automatically generate the desired programs is far more difficult than to verify them. Since the longest path problem (ND29 [36], i.e. whether for two given points and an integer *k* there will exist a simple path of edge number over *k* in a directed graph) is a NP-Complete problem, we can search the most suitable solution or approximate program in the formal model through the use of *GA*. In this case, the string concatenated from edge labels along a passage is just a correct program with respect to its pre-/post-conditions.

HGP as a member of the *GP* family naturally shares with its brethren many general characteristics. Because section 1 has given an overview of *GP*, in the following we will introduce the novel *GP* in terms of its distinctive features. For other related aspects, the reader can refer to [13].

The representation is one of the major differences between classical *GP* and *HGP*. Since edge labels in the formal model stand for the names of components, a string concatenated from edge labels along a path naturally forms a program. Furthermore, if the path is a passage with respect to its input/output conditions, this string must be a correct program with respect to its pre-/post-conditions. So given a set *H* of (verified) Hoare triples and its scalable model *SM*(*H*), populations can be defined as sets of programs comprised of only components in *H*. This certainly tells them apart.

In regard to fitness, *HGP* first calculates the maximum expansion for a randomly generated program (passage) on some given pre-condition, then checking the similarity between the target condition (as the post-condition) and the evolved maximum expansion. This leads to the following fitness function:

$$f(Gpre, S, Gpos) = n(m(S, Gpre),\ Gpos)$$

where meanings of the symbols are:

S: a program segment or a sequence of components.

Gpre: generalized pre-condition.

Gpos: target requirement as generalized post-condition. *HGP* first accept *Gpre, Gpos* as inputs, then automatically search reliable programs in search space.

n(*P, Q*): *n* is a function of sets *P, Q* of predicates calculating the order of $\{q \in Q \mid \exists p \in P(p \to q)\}$.

m(*S, Gpre*): *m* is a function solving the maximum expansion of generalized body *S* on a generalized pre-condition *Gpre* based on a given scalable model.

Note that the case of breakpoint calculation in [18] can technically be avoided by using valid genetic operation based on *SM*(*H*). The efficiency, however, is raised dramatically using this method.

Clearly, when evaluating the randomly generated programs, the greater the returned value the better. Ideally, the returned value should be *n*(*m*(*S*, *Gpre*), *Gpos*)=|*Gpos*|.

So, another major difference between *HGP* and *GP* lies in their fitness evaluation. The latter applies such a strategy as firstly executing the randomly generated programs on a sample data set, then checking the approximation between the returned value and the target requirement. From the viewpoint of software engineering, this method can only be categorized as testing rather than verification. Besides the advantages of being a verification-based method, the method used here also brings with it such new properties such as closure, sufficiency, etc. [13]

HGP solves the fitness based only on property relations rather than execution or test. Worth noticing is that the formal framework discussed above can also support distributed parallel evaluation at arbitrarily fine granularity. Additionally, *HGP* also differs from the formal *GP* of paper [17]. The latter has paved the way for introducing formal method into *GP*, focusing on the combination of model checking and *GP* — in this system we use Hoare logic instead of the temporal logic of model checking as a specification language. This can help to extend the expressiveness of the system.

Genetic operators are integral part of evolutionary computation. *HGP* has such operators as reproduction, crossover, mutation, etc. They are not applied to tree-like individuals but to sequences of components. The genetic strategy involved in the evolutionary process is similar to that of classical *GA*.

What follows is the comparison between classical *GP* and *HGP* (table 4). As for the pseudo-algorithmic description of *HGP*, one can refer to the *GP* framework of section 1. They are similar in principle.

Table 4 Classical *GP* and *HGP*

	Classical *GP*	*HGP*
Representation	Parse tree	Sequence of justified components
Fitness evaluation	Execution and comparison	Direct computation
Based on Logic	No	Yes
Soundness	Software test	Software verification
Underlying search space	Sets of terminals and functions	Hoare triples
Operators	Similar to *GA*	Similar to *GA*
Application areas	Expression, Lisp	Arbitrary programming language
Solution and result	Approximation	Both accuracy and approximation

To facilitate the understanding of why *HGP* works effectively, we will go into more detail about genetic operations.

Definition 5.1 (Context) Given that H is a set of Hoare triples, $S_H = \{P_i \{f_i\}_S Q_i \mid 1 \le i \le n\}$ is its scalable representation, $SM(H)$ represents the corresponding scalable model, and $S = \{f_i \mid 1 \le i \le n\}$ is all the program components involved in S_H, a context for f in S with respect to $SM(H)$, denoted $C(SM(H), f)$, is a 2-tuple $C(SM(H), f) = (front, rear)$ such that $front$, $rear$ are subsets of S; and that $front = \{ g \in S \mid$ there exists a ε arrow in $SM(H)$ linking g (i.e. the generalized post-condition of g) to f (i.e. the generalized pre-condition of f) $\}$ and $rear = \{h \in S \mid$ there exists a ε arrow in $SM(H)$ linking f (i.e. the generalized post-condition of f) to h (i.e. the generalized pre-condition of h) $\}$.

Definition 5.2 (Crossable space) Given H, S_H, $SM(H)$ and S as above, the crossable space for two strings $\alpha, \beta \in S^*$, denoted $CS(\alpha, \beta)$, is defined as $CS(\alpha, \beta) = \{ f \in S \mid f$ appears in both α and $\beta \}$.

Now, it is time to algorithmically depict the semantic-based genetic operations in terms of definitions 5.1 and 5.2. As for the initialization step, individuals (or sequences of program components) consistent with the concerned scalable model $SM(H)$ for *STP** can incrementally be generated through the use of *context*. For example, having figured out the ith component f_i of some individual along with $C(SM(H), f_i) = (front, rear)$, *HGP* will proceed to generate f_{i+1} based on the set *rear*.

Mutation: 1) Let $P = f_1 f_2 \cdots f_m$ be an individual to be mutated;

2) Choose a position, say i (i.e. f_i), in the sequence P for mutation;

3) Define *mutation_space* for the position i as follows:

```
case i of
  1:  let C(SM(H), f2) = (front, rear) in
          mutation_space:= front
      end;     //all possible program components which can be linked to f2
  m:  let C(SM(H), fm-1) = (front, rear) in
          mutation_space:= rear
      end;   // all possible program components to which fm-1 can be linked
  1 < i < m :  let C(SM(H), fi-1) = (front, rear) and
                   C(SM(H), fi+1) = (front2, rear2)   in
                       mutation_space:=  rear ∩ front2
               end
end;
```

4) Replace f_i of P with some randomly chosen program component, say f, in *mutation_space* $-\{f_i\}$ if *mutation_space* $-\{f_i\} \neq \varnothing$.

Crossover: 1) Let $P_1 = f_1 f_2 \cdots f_m$, $P_2 = h_1 h_2 \cdots h_n$ with $CS(P_1, P_2) \neq \varnothing$ be two individuals to crossover;

2) Determine the crossover positions in P_1, P_2 with substeps a) to b):

   a) Randomly choose some program component, say g, from $CS(P_1, P_2)$;

   b) Randomly choose some positions, say i and j, in P_1, P_2 with $g = f_i = h_j$ as the crossover positions.

3) Conduct crossover on P_1, P_2 through constructing such semantic-allowed individuals as $f_1 f_2 \cdots f_i h_{j+1} \cdots h_n$ and $h_1 h_2 \cdots h_j f_{i+1} \cdots f_m$ for further use.

Consequently, combining these techniques with the algorithm of *HGP* can give birth to an effective approach to reliable program generation. In fact, if the parents reflect some paths in *SM*(*H*), so do the results obtained from either the crossover or the mutation. Of course, it is permissible to make the genetic operators more complicated, but for the sake of the fact that these studies do not benefit the framework of *HGP* fundamentally, we wouldn't like to discuss it deeply.

6 Experiment and Analysis

In this section we will elaborate on parallel evaluation, simulation experiments, and scalability through the use of the example of [18].

Problem. Given a set of Hoare triples *H* (table 5) and a predicate relation matrix (table 6), generating a program which is correct with respect to the pre-condition ($P_1 \wedge P_5 \wedge P_7$) and the post-condition ($P_1 \wedge P_5 \wedge P_7 \wedge (u=0 \vee r < z)$).

6.1 Theoretical Analysis

Thought: from Hoare logic, if there exists a program *X* which together with ($P_1 \wedge P_5 \wedge P_7$) $\wedge$ ($u \neq 0 \wedge r \geq z$) and ($P_1 \wedge P_5 \wedge P_7$) forms a Hoare triple $\{ (P_1 \wedge P_5 \wedge P_7) \wedge (u \neq 0 \wedge r \geq z) \}\ X\ \{P_1 \wedge P_5 \wedge P_7\}$, then:

$$\{P_1 \wedge P_5 \wedge P_7\}\ \textit{while}\ (u \neq 0 \wedge r \geq z)\ \{X\}\ \{P_1 \wedge P_5 \wedge P_7 \wedge (u=0 \vee r < z)\}.$$

So the desired program is : *while* $(u \neq 0 \wedge r \geq z)\ \{X\}$. Now we solve to find the value for *X*.

Method:

1) According to table 5 we have a scalable representation $S_H = \{\{P_1, P_2, P_3\}\{f_1\}_S\{P_2, P_3, P_4\}, \{P_2, P_4, P_6\}\{f_2\}_S\{P_1, P_5, P_6\}, \{P_2, P_3, P_4\}\{f_3\}_S\{P_2, P_4, P_6\}, \{P_1, P_5, P_6\}\{f_4\}_S\{P_1, P_5, P_7\}\}$ for *H*.

2) Constructing the generalized relation matrix (table 7) and the scalable mode *SM*(*H*) (figure 5) from S_H and predicate relation matrix (table 6) in the same way as that of theorem 2.

3) By theorem 2, searching for a desired passage in *SM*(*H*).

Because there is a passage in *SM*(*H*) with $f_1 f_3 f_2 f_4$ as the generalized body, $\{P_1, P_5, P_7\}$ as its maximum expansion on $\{P_1: y + uz = xz,\ P_2:\ u > 0,\ P_3: x = r + qz \wedge r \geq z \wedge z > 0\}$ verifying $\{P_1 \wedge P_2 \wedge P_3\}\ f_1; f_3; f_2; f_4\ \{P_1 \wedge P_5 \wedge P_7\} \in STP^*$, and satisfying $\vdash (P_1 \wedge P_5 \wedge P_7 \wedge u \neq 0 \wedge r \geq z) \rightarrow (P_1 \wedge P_2 \wedge P_3)$, we have $\{(P_1 \wedge P_5 \wedge P_7) \wedge (u \neq 0 \wedge r \geq z)\}\ f_1; f_3; f_2; f_4\ \{P_1 \wedge P_5 \wedge P_7\}$, *i.e.* $\{P_1 \wedge P_5 \wedge P_7\}$ *while* $(u \neq 0 \wedge r \geq z)\ \{f_1; f_3; f_2; f_4\}\ \{P_1 \wedge P_5 \wedge P_7 \wedge (u = 0 \vee r < z)\}$.

Table 5 Set of Hoare triples. Each row stands for a Hoare formula

Pre-condition		Function	Post-condition	
P_1	$y+uz=xz$	f_1	$y+(u-1)z=xz$	P_4
P_2	$u>0$	f_1	$u>0$	P_2
P_3	$x=r+qz \wedge r \geq z \wedge z>0$	f_1	$x=r+qz \wedge r \geq z \wedge z>0$	P_3
P_4	$y+(u-1)z=xz$	f_2	$y+uz=xz$	P_1
P_2	$u>0$	f_2	$u \geq 0$	P_5
P_6	$x=r+(q+1)z \wedge r \geq 0 \wedge z>0$	f_2	$x=r+(q+1)z \wedge r \geq 0 \wedge z>0$	P_6
P_3	$x=r+qz \wedge r \geq z \wedge z>0$	f_3	$x=r+(q+1)z \wedge r \geq 0 \wedge z>0$	P_6
P_4	$y+(u-1)z=xz$	f_3	$y+(u-1)z=xz$	P_4
P_2	$u>0$	f_3	$u>0$	P_2
P_6	$x=r+(q+1)z \wedge r \geq 0 \wedge z>0$	f_4	$x=r+qz \wedge r \geq 0 \wedge z>0$	P_7
P_1	$y+uz=xz$	f_4	$y+uz=xz$	P_1
P_5	$u \geq 0$	f_4	$u \geq 0$	P_5

Table 6 Predicate relation

$\xrightarrow{s}$	P_1	P_2	P_3	P_4	P_5	P_6	P_7
P_1	T						
P_2		T			T		
P_3			T				T
P_4				T			
P_5					T		
P_6						T	
P_7							T

Table 7 Generalized relation

$\xrightarrow{g}$	GC_1	GC_2	GC_3	GC_4	GC_5
GC_1					
GC_2					
GC_3					
GC_4		F	F		
GC_5		F			

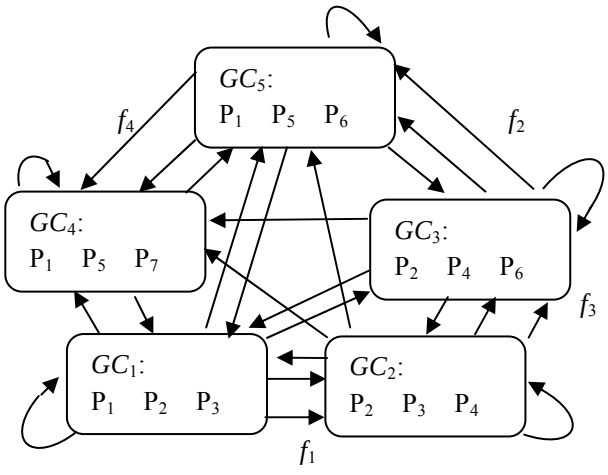

Figure 5 The scalable model *SM(H)* of *STP under *H*. Here each GC_i stands for a generalized condition, and edges without labels are ε arrows.**

Formal principle: The verification process of f_1 ; f_3 ; f_2 ; f_4 in *SM*(*H*) is as follows.

Step 1: solving $m(GC_1)=\{x \in GC_1 \mid (P_1 \wedge P_5 \wedge P_7) \wedge (u \neq 0 \wedge r \geq z) \rightarrow x\}=\{P_1, P_2, P_3\}$

Step 2: solving R[f_1]={(P_1,{P_4}), (P_2,{P_2}),(P_3,{P_3})), R[f_2]= {(P_2,{P_5}), (P_4,{P_1}),(P_6,{P_6})}, R[f_3]= {(P_2,{P_2}), (P_3,{P_6}), (P_4,{P_4})} and R[f_4]= {(P_1,{P_1}), (P_5,{P_5}), (P_6,{P_7})};

Step 3: invoking *CalculateExpansion*(f_1 ; f_3 ; f_2 ; f_4 , H, *SM*(H)), we have R[α]={(P_1,{P_1}), (P_2, {P_5}), (P_3, {P_7})} ;

Step 4: solving Q'= $\bigcup_{\exists t \in m(Z_1):(t \to u) \wedge (u, e_u) \in R[\alpha]} e_u$ = {P_1, P_5, P_7} from R[α] above. This means { $P_1 \wedge P_2 \wedge P_3$} f_1 ; f_3 ; f_2 ; f_4 { $P_1 \wedge P_5 \wedge P_7$} $\in$ *STP**. By what was analyzed in 3), we get the desired result.

Apparently, the graph *SM*(H) is rather complicated. For this, we can use theorem 3 for the simplification. For example, solving the problem based on such a scalable representation S_H ={{P_1, P_2, P_3}{f_1}$_S$ { P_2, P_3, P_4, P_6 }, { P_2, P_3, P_4, P_6 }{ f_2}$_S${P_1, P_5, P_6},{ P_2, P_3, P_4, P_6 }{ f_3}$_S$ { P_2, P_3, P_4, P_6 }, {P_1, P_5, P_6}{ f_4}$_S${P_1, P_5, P_7}} for H , the result is still correct.

6.2 Experimental Analysis

The simulation includes 2 steps.

1) For each randomly generated programs, using {$x \in GC_1$| ($P_1 \wedge P_5 \wedge P_7 \wedge u \neq 0 \wedge r \geq z$) $\to x$ }= { P_1, P_2 , P_3 }= GC_1 and *tPost* ={P_1, P_5 , P_7 } as the generalized pre-condition *Gpre* and the target requirement *Gpos* to invocate the fitness function;
2) Let the population size be 8. Then pressing the button "*Run*", we will get Figure 6, a screenshot of the *HGP*. With the progress in solution evolution, the result approaches gradually to our target, say 157 in figure 6, in terms of fitness. *HGP* must terminate some time under the case of either the maximum number of generations or the given requirement being reached. Thus it is effective for generation of solutions of both precision and approximation. With precision, we mean Hoare formulas can be obtained through evolutionary approaches; with approximation, search methods are employed.

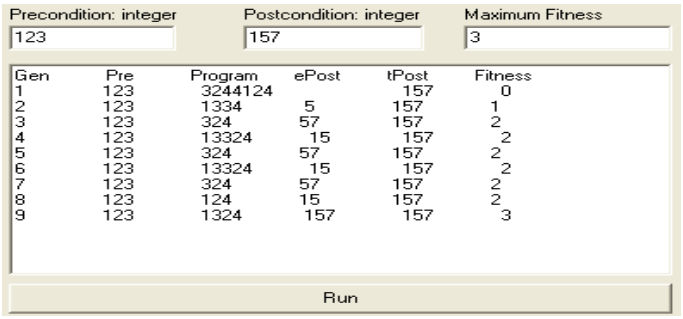

Figure 6 Screenshot of result

To better understand *HGP*, we annotate figure 6 as follows. Each line in figure 6 reflects the best solution of the population of programs at that moment. The data under the names *Pre* (Precondition), *ePost* (the maximum expansion of a program on the pre-condition) and *tPost* (the target requirement or post-condition) represent properties of programs. For example, the data under the name Pre "123" stands for {P_1, P_2, P_3}. Similarly, the data under the name Program like "132" stands for "f_1; f_3; f_2", a sequence of components, i.e. a program. As for the fitness value 0 of the first generation in figure 6, we can deduce it from the fact that the corresponding maximum expansion is an empty set, therefore implying no element of { P_1, P_5, P_7}, denoted *tPost* = 157. The fitness values of generation 3 through 8 are consistent, i.e. 2, because 15 shares two digits, i.e. the "1" and the "5" with 157; and so does 57 with 157. Obviously, the desired result {$\wedge$ {P_1, P_2, P_3}} f_1; f_3; f_2; f_4 { $\wedge$ {P_1, P_5, P_7}} appears in generation 9. This is shown by the value for *ePost* and *tPost* being the same, i.e. 157. By (1), we have $\vdash$(P_1

$\wedge P_5 \wedge P_7 \wedge \ u \neq 0 \ \wedge \ r \geq z \ ) \ \rightarrow (P_1 \wedge P_2 \wedge P_3)$. Consequently, the desired program is:

$$while\ (u \neq 0 \wedge r \geq z)\ \{ f_1 ; f_3 ; f_2 ; f_4 \}.$$

which agrees with the result from the theoretical analysis in the previous section.

7 Discussion

It is really hard to make a precise comparison between search techniques of different natures. Apart from the objective factors, we are subjectively dedicated to the establishment of *HGP* recently, therefore having not explored the efficient issues comprehensively and deeply. However *HGP* along with its search technique has the following characteristics.

a) Usefulness

Verification and testing are two major kinds of approaches to software reliability. A very fundamental problem with software testing is that testing under all combinations of inputs and preconditions (initial state) is not feasible, even for simple examples [37]. Consequently, classical GP cannot establish that its result functions properly under all conditions, because it works according to executions of members of the population over some limited sample dataset.

However this is not the case for *HGP*. This approach searches the desired computer programs through the use of Hoare logic style reasoning. Once the result, say the Hoare formula $\{P_1 \wedge P_5 \wedge P_7 \}$ $while \ \ (u \neq 0 \wedge r \geq z) \ \{X\} \{P_1 \wedge P_5 \wedge P_7 \wedge (u = 0 \vee r < z)\}$ in subsection 6.1, is obtained, we can say with certainty that the program $while\,(u \neq 0 \wedge r \geq z)\{\ X\ \}$ is correct with respect to $P_1 \wedge P_5 \wedge P_7$ and $P_1 \wedge P_5 \wedge P_7 \wedge$ (u=0 $\vee$ r < z). This means for any values of x, y, z, u, r, q such that $P_1 \wedge P_5 \wedge P_7$ holds before the program runs, so will the post condition $P_1 \wedge P_5 \wedge P_7 \wedge (u = 0 \vee r < z)$ after that program's termination for their returned values. As such, it is in this sense that *HGP* is superior to classical *GP*. Of course, each method has its own strong points, and so we maintain that both of them merit deep study. Indeed, one interesting area for future study is hybridizing logic-based and testing-based approaches to *GP*.

b) Scope

Whether a probabilistic approach is useful should depend, first, on its *effectiveness*, and then its technical *efficiency*. In view of the following analysis, we have reason to claim that *HGP* gives more scope to us than the standard search.

Without loss of generality, assuming g is an element of some *GP* function set which takes the form of *repeat f* ; y:= *y-4 until y=5* (where *f* stands for a program whose execution has no effect on y and particularly can terminate), it follows that *g cannot* work or contribute the search process *effectively* unless its loop control variable y satisfies $y = 4 * k + 5$ ($k \geq 1$). In other words, since *GP* relies on executions of programs (for the fitness values) to guide the evolution of populations, its probabilistic search must suffer from the endless loops for which y has been assigned values such that $y \neq 4 * k + 5$ by the previous computation steps of g, thus resulting in failures in evaluating programs as well as further searches. This makes the standard search *vulnerable*. Solving of this problem must seek help from the semantic measures. Also, *GP* search still faces the challenge of addressing the closure problem [13]. This problem, to put it simply, concerns type consistency. As such, it is necessary to introduce some mechanisms to ensure this type consistency into the standard search for program evaluations. Fortunately, *HGP* provides a means for these issues on the basis of Hoare triples. For instance, it can cope with these cases on the condition that the execution terminations have been deliberately provided for the concerned components (say g), and reflected in the pre-/post-conditions. In summary, *SM*(*H*) based search may not appear to be very efficient in all situations, but often is effective. In fact, its effort toward working out the problem is evident. Since *HGP* evaluates programs in light of the computation of *properties* of programs instead of program executions, its running, unlike that of standard *GP* which may fail in executions, will succeed all the time. Consequently, we see from the above mentioned facts that not only is

semantic-based *HGP* a search method more effective than the standard search, but rather it has a broader scope than traditional *GP*.

8 Conclusion

The work in this paper represents the first attempt to explore the use of various approaches (Hoare logic, model checking, and the theory of automata) as a way of formalising the process of fitness evaluation in *GP*; it has the following characteristics. Firstly, *HGP* has not only the capability for generation and verification of programs in the search space, but supports fitness evaluation at fine granularity. Secondly, *HGP* takes ideas from earlier work on *GP* search using model checking, but differs from it in working style. The former is a common model of Hoare semantics for verification of numerous program objects, the latter nevertheless is a solution peculiar to a concrete problem. Thirdly, *HGP* generates programs on demand, using a mixture of accuracy and approximation. These surely make it different from existing *GP*s. So, if extended with modern automated theorem proving techniques, this method may become an alternative approach to software reliability and program generations.

Our future studies will focus on such related topics as schema theory, the definition of new tasks, efficient search algorithms, the unified theory of various kinds of *GP*s, service applications, and improved implementations.

Acknowledgements

We are grateful to the reviewers for their helpful suggestions. In addition, He Pei would like to give special thanks to the late Prof. Tang Zhisong for introducing him to the area of formal methods.

References

1 Charles Rich, Richard C. Waters. Automatic programming: myths and prospects. Computer, 1988, 21(8): 40-51

2 John R. Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection, Cambridge, MA: The MIT Press, 1992.

3 John Holland. Adaptation in Natural and Artificial Systems, Michigan: The University of Michigan Press, 1975.

4 M. Mitchell. An Introduction to Genetic Algorithms, Cambridge: MIT Press, 1976.

5 Mihai Oltean, Crina Grosan. A comparison of several linear genetic programming techniques. Complex Systems. 2004, 14(4): 1-29.

6 Ajith Abraham, Nadia Nedjah, and Luiza de Macedo Mourelle. Evolutionary computation: from genetic algorithms to genetic programming. In: Nadia Nedjah et al. eds. Studies in Computational Intelligence, Springer-Verlag, 2006: 1-20.

7 John R. Koza. Human-competitive machine intelligence by means of genetic algorithms. In Lashon Booker, Stephanie Forrest, Melanie Mitchell and Rick Riolo Eds. Perspectives on Adaptation in Natural and Artificial System. Oxford University Press, 2005: 33-55

8 John R. Koza, Forrest H Bennett III, David Andre, and Martin A. Keane. Four problems for which a computer program evolved by genetic programming is competitive with human performance. In: Proceedings of IEEE International Confer. On Evolutionary Computation, 1996: 1-10.

9 John R. Koza, Martin A. Keane, Jessen Yu, Forrest H Bennett III, William Mydlowec. Automatic creation of human-competitive programs and controllers by means of genetic programming, Genetic Programming and Evolvable Machines, 2000, 1(1-2): 121-164.

10 Franck Binard, Amy Felty. An abstraction-based genetic programming system, GECCO'2007, July 7, 2007: 2415-2422.

11 Enrique Frias-Martinez, Fernand.Gobet. Automatic generation of cognitive theories using genetic programming, Minds and Machines, 2007, 17(3): 287-309..

12 Michael O'Neill, Conor Ryan. Grammatical evolution. IEEE Transactions on Evolutionary Computation. 2001, 5(4): 349-358.

13 S. Sette, L. Boullart. Genetic programming: principles and applications, Engineering Applications of Artificial Intelligence, 2001, 14(6): 727-736.

14 D. J. Montana. Strongly typed genetic programming. Evolutionary Computation, 1995, 3(2): 199-230

15 J. R. Koza. Genetic Programming II: Automatic Discovery of Reusable Programs. Cambridge, MA: MIT Press, 1994.

16 Man Leung Wang, Kwong Sak Leung. Data Mining Using Grammar Based Genetic Programming and Applications. Kluwer

Academic Publishers, 2002: 27-99.

17 Colin G. Johnson. Genetic programming with fitness based on model checking. EuroGP 2007. In Marc Ebner et al. Eds. Genetic Programming, LNCS 4445, Springer-Verlag, 2007: 114-124.

18 Pei He, Lishan Kang, Ming Fu. Formality based genetic programming. Proc. of IEEE Congress on Evolutionary Computation. Hong Kong. 2008: 4081-4088

19 Gal Katz, Doron Peled. Model checking-based genetic programming with an application to mutual exclusion. In: C. R. Ramakrishnan, J. Rebof, Eds. TACAS2008, LNCS 4963, 2008: 141-156.

20 Chen Huo-wang，Wang Ji，Dong Wei。High confidence software engineering technologies，Acta Electronica Sinica, 2003, 31(12A): 1933-1938 (in Chinese)

21 E. W. Dijkstra. A Discipline of Programming. Englewood Cliffs, NJ: Prentice-Hall, 1976.

22 Ed Brinksma. Verification is experimentation!, Int J STTT. 2001, 3(2): 107-111

23 Ralf Kneuper. Limits of formal methods. Formal Aspects of Computing. 1997, 9(4): 379-394

24 Edmund M. Clarke, Jeannette M. Wing. et.al. Formal methods: state of the art and future directions. ACM Computing Surveys. 1996, 28(4): 626-643.

25 Michael Huth, Mark Ryan. Logic in Computer Science: Modelling and Reasoning about System. Cambridge University Press, England, 2004.

26 Lin Hui-min, Zhang Wen-hui. Model checking: theories, techniques, and applications. Acta Electronica Sinica, 2002, 30(12A): 1907-1912. (In Chinese)

27 Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, Flavio Lerda. Model checking programs. Automated Software Engineering, 2003, 10(2): 203-232.

28 C. A. R. Hoare, An axiomatic basis for computer programming, CACM, 1969, 12(10):576-583.

29 Zohar Manna. Mathematical Theory of Partial Correctness. Journal of Computer and System Sciences (JCSS), June, 1971, 5(3): 239-253

30 Awadhesh Kumar Singh, Umesh Ghanekar, Anup Kumar Bandyopadhyay. Specifying mobile network using a wp-like formal approach. Revista Colombiana de Computacion, 2005, 6(2): 59-77.

31 Glynn Winskel. The Formal Semantics of Programming Language: A Introduction. The Mit Press, 1993.

32 Yang Fuqing, Wang Qianxiang, Mei Hong, Chen zhaoliang. Reuse-based software production technology. Science in China Series E. 2001, 31(4): 363-371 (In Chinese)

33 Xiong Huiming, Ying shi, Yu Lijuan, Zhang Tao. A composite reuse of architectural connectors using reflection. Journal of Software, 2006, 17(6): 1298-1306. (In Chinese)

34 John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. 3rd edition, Pearson Education Inc. 2007: 1-170.

35 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. 2nd Edition Addison Wesley, 2007

36 Michael Garey and David S. Johnson. Computers and Intractability - A Guide to the Theory of NP-completeness. San Francisco: Freeman, 1979: 213.

37 Cem Kaner, Jack Falk, Hung Q. Nguyen. Testing Computer Software. 2nd Edition, New York: John Wiley and Sons, Inc, 1999: 480