PENILAIAN OTOMATIS TUGAS PEMROGRAMAN DENGAN PENDEKATAN SEMANTIK

Laporan Tugas Akhir

Disusun sebagai syarat kelulusan tingkat sarjana

Oleh

M. RIFKY I. BARIANSYAH

NIM : 13517081

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

Juni 2021

PENILAIAN OTOMATIS TUGAS PEMROGRAMAN DENGAN PENDEKATAN SEMANTIK

Laporan Tugas Akhir

Oleh

M. RIFKY I. BARIANSYAH

NIM : 13517081

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Telah disetujui dan disahkan sebagai Laporan Tugas Akhir

di Bandung, pada tanggal 21 Juni 2021

Pembimbing I,	Pembimbing II,
Riza Satria Perdana S.T., M.T.	Satrio Adi Rukmono S.T., M.T.
NIP. 197006091995121002	NIP. 198809272019031007

LEMBAR PERNYATAAN

Dengan ini saya menyatakan bahwa:

1. Pengerjaan dan penulisan Laporan Tugas Akhir ini dilakukan tanpa menggunakan bantuan yang tidak dibenarkan.
2. Segala bentuk kutipan dan acuan terhadap tulisan orang lain yang digunakan di dalam penyusunan laporan tugas akhir ini telah dituliskan dengan baik dan benar.
3. Laporan Tugas Akhir ini belum pernah diajukan pada program pendidikan di perguruan tinggi mana pun.

Jika terbukti melanggar hal-hal di atas, saya bersedia dikenakan sanksi sesuai dengan Peraturan Akademik dan Kemahasiswaan Institut Teknologi Bandung bagian Penegakan Norma Akademik dan Kemahasiswaan khususnya Pasal 2.1 dan Pasal 2.2.

Bandung, 21 Juni 2021

M. Rifky I. Bariansyah
NIM 13517081

ABSTRAK

PENILAIAN OTOMATIS TUGAS PEMROGRAMAN DENGAN PENDEKATAN SEMANTIK

Oleh

M. Rifky I. Bariansyah

NIM : 13517081

Pada bidang ilmu komputer, pemrograman digunakan oleh peserta didik sebagai media untuk mengimplementasikan pengetahuan teoritis ke dalam bentuk program. Untuk memfasilitasi penilaian pada kelas dengan jumlah peserta didik yang banyak, digunakan sistem penilaian otomatis atau *automatic grading*. Pendekatan yang populer digunakan dalam membangun sistem penilaian otomatis saat ini adalah *black-box testing*. Penilaian dilakukan dengan menuliskan kumpulan uji kasus untuk mengeksekusi program peserta didik. Namun, untuk menuliskan kumpulan kasus uji yang komplit, dalam hal ini mencakup sebanyak mungkin *edge cases* atau *corner cases*, merupakan hal yang sulit untuk dilakukan. Oleh karena itu, perlu dilakukan eksplorasi pendekatan selain *black-box testing* pada sistem penilaian otomatis, seperti *white-box testing*. Secara teori, dengan memperhatikan isi kode kita dapat mengulas semua jalur eksekusi yang mungkin, menghasilkan sistem penilaian yang lebih lengkap.

Pada tugas akhir ini dilakukan penelitian dalam penggunaan analisis semantik sebagai solusi untuk menentukan kebenaran implementasi program peserta didik. Pada sistem di tugas akhir ini akan dilakukan perekaman jalur eksekusi implementasi program referensi dan implementasi program peserta didik, pendeteksian *path deviation,* dan pendeteksian *path equivalence* untuk membandingkan semantik kedua implementasi. Sistem dibangun dengan memanfaatkan metode *concolic execution* untuk eksplorasi dan SMT *solver* untuk menyelesaikan formula.

Pengujian sistem menunjukkan bahwa penilaian dapat dilakukan dengan melakukan perbandingan semantik implementasi program referensi dan peserta didik. Berdasarkan perbandingan dengan pendekatan pembangkitan uji kasus secara acak, ditemukan kasus–kasus dimana sistem pada tugas akhir dapat memberikan penilaian yang lebih lengkap.

Kata kunci: penilaian otomatis, pembangkitan uji kasus, *concolic execution*

KATA PENGANTAR

Puji syukur Penulis panjatkan ke hadirat Tuhan Yang Maha Esa karena atas karunia-Nya, Penulis dapat menyelesaikan tugas akhir yang berjudul "Penilaian Otomatis Tugas Pemrograman dengan Pendekatan Semantik" untuk memenuhi syarat kelulusan tingkat sarjana. Penulis juga ingin mengucapkan terima kasih kepada pihak-pihak yang telah memberikan dukungan dalam pengerjaan tugas akhir ini:

1. Bapak Riza Satria Perdana S.T., M.T. dan Bapak Satrio Adi Rukmono S.T., M.T. selaku dosen pembimbing yang telah memberikan arahan, nasehat, dan bantuan selama pengerjaan tugas akhir.
2. Bapak Yudistira Dwi Wardhana Asnar, S.T., Ph.D. dan Ibu Yani Widyani, S.T., M.T. selaku dosen penguji yang telah memberikan evaluasi dan saran terhadap tugas akhir.
3. Bapak Nugraha Priya Utama, S.T., M.A., Ph.D, Ibu Latifa Dwiyanti, S.T., M.T., Ibu Ginar Santika Niwanputri, S.T., M.Sc., Bapak Adi Mulyanto, S.T., M.T., dan segenap tim mata kuliah Tugas Akhir yang telah memberikan dukungan dan arahan selama pengerjaan tugas akhir
4. Ibu Yanti Rusmawati, S.T., M.Sc., Ph.D. selaku dosen wali Penulis yang telah mendampingi Penulis selama berada di Program Studi Teknik Informatika.
5. Segenap anggota keluarga Penulis yang telah mendukung dan mendoakan Penulis selama berada di Program Studi Teknik Informatika.
6. Staf Tata Usaha Program Studi Teknik Informatika yang senantiasa membantu proses administrasi tugas akhir.
7. I Putu Gede Wirasuta, Pandyaka Aptanagi, Gardahadi dan rekan-rekan mahasiswa lainnya yang telah saling memberi dukungan dan saran dalam mengerjakan tugas akhir.

Penulis berharap tugas akhir ini dapat memberikan manfaat terutama dalam mendukung pendidikan di bidang informatika. Penulis menyadari tugas akhir ini

jauh dari sempurna, maka darinya Penulis mengharapkan kritik dan saran konstruktif terhadap tugas akhir ini.

Bandung, 21 Juni 2021

Penulis

DAFTAR ISI

DAFTAR LAMPIRAN

DAFTAR GAMBAR

DAFTAR TABEL

BAB I

PENDAHULUAN

I.1 Latar Belakang

Pemrograman digunakan oleh peserta didik sebagai media untuk implementasi pengetahuan teoritis ke dalam bentuk program. Dalam penggunaannya, diperlukan penilaian yang dapat dijadikan panduan belajar oleh peserta didik dan sebagai umpan balik mengenai ketercapaian proses belajar. Namun, memberikan penilaian dengan kualitas yang tinggi secara manual akan memakan banyak waktu dan tidak memungkinkan untuk latar kelas dengan jumlah peserta didik yang banyak. Semakin besar jumlah peserta didik didalam kelas semakin besar pula kemungkinan terjadinya kesalahan dalam proses penilaian. Pada hakikatnya, ketika terdapat banyak peserta didik didalam kelas, usaha pengajar dalam melakukan penilaian harus bisa dikurangi.

Permasalahan tersebut telah mendorong pengembangan sistem penilaian otomatis atau *automatic grading*. Dengan penilaian otomatis dapat diberikan umpan balik secara instan untuk peserta didik. Karakteristik instan ini dapat mendukung proses belajar peserta didik dengan mempercepat proses pengumpulan ulang bila terjadi kesalahan. Dengan penilaian otomatis, pengajar dapat melakukan penilaian secara berskala. Cukup banyak publikasi yang berkaitan dengan menggunakan penilaian otomatis dalam membantu pengajar untuk menilai implementasi program peserta didik. Dari Ihantola dkk. (2010), mayoritas sistem penilaian otomatis menggunakan pendekatan *black-box testing*. Dengan pendekatan ini pengajar menuliskan sekumpulan kasus uji serta keluaran yang diharapkan dari implementasi program. Kemudian implementasi program peserta didik dinilai berdasarkan kebenaran fungsionalitasnya. Namun, menuliskan kumpulan kasus uji yang lengkap, dalam hal ini mencakup sebanyak mungkin *edge cases* atau *corner cases*, merupakan hal

yang sulit untuk dilakukan. Kasus uji yang tidak lengkap dapat menyebabkan kesalahan dalam penilaian.

Diperlukan pendekatan penilaian otomatis yang dapat mencakup *edge cases* pada implementasi program, dimana hal tersebut mahal dan sulit untuk dilakukan dengan penulisan kasus uji secara manual. Salah satu metode yang dapat diteliti untuk mencapai hal ini adalah dengan mengukur perbedaan antara implementasi program referensi dengan implementasi program peserta didik. Perbedaan antara implementasi mungkin menunjukkan perbedaan semantik antara keduanya yang dapat mengarah ke keluaran yang berbeda. Hasil observasi perbedaan tersebut kemudian bisa dijadikan masukan sebuah sistem penilaian otomatis. Oleh karena itu, pada tugas akhir ini dilakukan penelitian dalam penggunaan analisis semantik sebagai solusi untuk menentukan kebenaran implementasi program peserta didik.

I.2 Rumusan Masalah

Berdasarkan latar belakang yang telah dijelaskan, rumusan masalah tugas akhir ini adalah:

1. Bagaimana cara menentukan kebenaran implementasi program peserta didik berdasarkan implementasi program referensi dengan memperhatikan perbedaan semantik di antara keduanya?

I.3 Tujuan

Tujuan dari tugas akhir ini adalah membuat sistem penilaian dengan pembangkitan kasus uji yang lengkap dan efisien dengan memperhatikan perbedaan semantik di antara implementasi program peserta didik dan implementasi program referensi.

I.4 Batasan Masalah

Untuk lingkup pekerjaan yang spesifik dan jelas, perlu didefinisikan beberapa batasan masalah. Batasan masalah untuk tugas akhir ini adalah sebagai berikut:

1. Hasil merupakan solusi dalam bentuk program penilaian otomatis program berbahasa Python

2. Umpan balik yang dihasilkan program adalah nilai implementasi program peserta didik, bila implementasi program peserta didik tidak benar dikembalikan pula kasus uji yang menunjukkan kesalahan

I.5 Metodologi

Metodologi yang digunakan pada pengerjaan tugas akhir ini adalah sebagai berikut.

1. Pemilihan Pendekatan

   Setelah ditemukan rumusan masalah, dilakukan pemilihan pendekatan yang cocok untuk menyelesaikan masalah tersebut. Pemilihan pendekatan dilakukan dengan perbandingan hasil dari studi literatur.

2. Perancangan Solusi dan Cara Penilaian

   Berdasarkan pendekatan yang telah dipilih, dilakukan perancangan solusi. Perancangan solusi terdiri dari perancangan algoritma dan perancangan program. Perancangan algoritma dilakukan dengan pemilihan beberapa alternatif algoritma yang paling sesuai dengan pendekatan yang telah dipilih, termasuk didalamnya algoritma untuk menentukan nilai berdasarkan hasil kebenaran dari implementasi program peserta didik. Perancangan program terdiri dari perancangan komponen, antarmuka, dan arsitektur program.

3. Implementasi Solusi

   Pada tahap ini dilakukan implementasi dari desain hasil rancangan solusi. Yang dihasilkan adalah program yang dapat melakukan penilaian implementasi program peserta didik berdasarkan implementasi program referensi.

4. Pengujian

   Dilakukan pengujian terhadap sistem yang telah diimplementasikan. Sumber yang digunakan dalam pengujian adalah kumpulan implementasi program referensi dan implementasi program peserta didik. Aspek yang diujikan adalah efektifitas sistem dalam menilai dengan pendekatan

semantik. Kemudian dilakukan evaluasi dan penarikan kesimpulan berdasarkan hasil pengujian program.

I.6 Sistematika Pembahasan

Sistematika pembahasan dari Laporan Tugas Akhir ini adalah sebagai berikut.

1. BAB I PENDAHULUAN

   Bab Pendahuluan dituliskan untuk memberikan gambaran landasan dan arahan kerja penulis dalam penulisan tugas akhir. Pada bab ini dipaparkan latar belakang, rumusan masalah, tujuan, batasan masalah, metodologi, dan sistematika pembahasan tugas akhir.

2. BAB II STUDI LITERATUR

   Bab Studi Literatur dituliskan untuk memberikan pemahaman kepada pembaca menenai teori yang berkaitan dengan persoalan tugas akhir. Pada bab ini dipaparkan hasil studi literatur yang telah dilakukan oleh penulis mengenai penilaian tugas pemrograman dengan pendekatan semantik. Didalamnya berisi pembahasan terkait penilaian otomatis, deteksi perbedaan semantik program, dan *symbolic execution*.

3. BAB III ANALISIS DAN PERANCANGAN SISTEM

   Bab Analisis dan Perancangan Sistem dituliskan untuk memaparkan analisis mendalam mengenai persoalan yang diangkat pada tugas akhir ini beserta uraian rancangan solusi untuk penyelesaian masalah. Pada bab ini dibahas analisis masalah, analisis solusi, dan rancangan solusi.

4. BAB IV IMPLEMENTASI DAN PENGUJIAN

   Bab Implementasi dan Pengujian dituliskan untuk menjelaskan proses implementasi sistem dan hasil pengujian terhadap sistem. Pada bab ini dibahas lingkungan implementasi dan pengujian, implementasi tiap komponen sistem, skenario pengujian, dan hasil pengujian.

5. BAB V KESIMPULAN DAN SARAN

   Bab Kesimpulan dan Saran dituliskan untuk menyimpulkan hasil implementasi dan pengujian sistem yang telah dibangun beserta saran untuk pengembangan lebih lanjut.

BAB II

STUDI LITERATUR

Bab ini berisi pemaparan hasil studi literatur tentang penilaian otomatis dengan deteksi perbedaan semantik pada program.

II.1 Penilaian Otomatis

Penilaian otomatis atau *automatic grading* adalah sistem yang disajikan untuk menilai implementasi program di bidang ilmu komputer (von Matt, 1994). Penilaian otomatis dapat digunakan oleh pengajar untuk menentukan kebenaran dari program buatan peserta didik. Sistem ini ditujukan untuk mempercepat dan meningkatkan kapasitas penilaian. Sherman dkk. (2013) melalui risetnya menemukan bahwa penilaian otomatis juga dapat memberikan efek positif terhadap proses belajar peserta didik saat digunakan sebagai sistem umpan balik, hal ini ditunjukkan dengan meningkatnya jumlah percobaan oleh peserta didik yang mengindikasikan peserta didik memanfaatkan umpan balik untuk memperbaiki implementasi program. Secara garis besar, cara yang umum digunakan dalam melakukan penilaian otomatis ada dua yakni pendekatan *black-box* dan pendekatan *white-box*.

II.1.1 Penilaian Otomatis dengan Pendekatan *Black-Box*

Black-box testing atau *functional testing* adalah teknik pengujian dengan membuat kasus uji berdasarkan informasi dari spesifikasi. Pada teknik ini penguji tidak memiliki akses ke sumber kode internal program. Teknik ini tidak memperhatikan mekanisme internal dari suatu sistem, melainkan hanya fokus terhadap keluaran yang dihasilkan berdasarkan masukan yang dipilih dan kondisi eksekusi (Liu & Kuan Tan, 2009).

Menurut studi yang dilakukan oleh Ihantola dkk. (2010) mayoritas dari kakas penilaian otomatis dibangun dengan berbasiskan pengujian. Pada hasil studi

tersebut ditemukan bahwa hasil evaluasi fungsionalitas dari kode peserta didik masih paling sering digunakan untuk menilai program. Evaluasi fungsionalitas bisa dilakukan dengan penggunaan kakas pengujian industri dan solusi khusus. Contoh penggunaan kakas pengujian industri adalah:

1. Kakas *unit testing* seperti XUnit dan JUnit
2. *Acceptance testing frameworks* seperti EasyAccept dimana tes dituliskan dalam bahasa *scripting natural-language-like* yang mudah dibaca
3. *Web testing framework* seperti Selenium

Contoh penggunaan solusi khusus adalah:

1. Perbandingan keluaran, pendekatan tradisional yang banyak ditemukan pada sistem penilaian otomatis. Salah satu variasi dari solusi ini adalah menjalankan program referensi dan program peserta didik bersebelahan dan menggunakan *regular expression* untuk membandingkan keluaran program.
2. *Scripting*, pendekatan yang paling sering digunakan pada sistem penilaian otomatis. Contoh implementasi solusi ini adalah dengan *shell script* yang melakukan kompilasi program, menjalankan program dan kemudian membandingkan keluaran program dengan *file* berisi keluaran yang diharapkan

II.1.2 Penilaian Otomatis dengan Pendekatan *White-Box*

Penilaian otomatis dengan *black-box testing* dapat tentunya dapat mempercepat dan meningkatkan kapasitas penilaian, serta dapat mengembalikan umpan balik berupa masukan yang gagal, namun kakas yang dibangun dengan pendekatan ini bergantung terhadap kualitas kasus uji yang dibuat oleh pengajar. Membuat kumpulan tes yang memiliki kualitas tinggi membutuhkan usaha yang besar dan kemungkinan besar akan tidak mencakup *edge cases* tertentu sehingga menyebabkan kesalahan dalam penilaian. Fenomena tersebut kemudian mendorong perluasan penelitian ke metode *white-box testing* untuk penilaian program secara otomatis.

White-box testing atau *glass box testing* atau *structural testing* merupakan teknik pengujian dengan membuat kasus uji berdasarkan informasi dari sumber kode. Penguji memiliki akses ke sumber kode dan menuliskan kasus uji dengan mengeksekusi fungsi dengan parameter tertentu. *White-box testing* memperhatikan mekanisme internal dari suatu sistem (Liu & Kuan Tan, 2009).

Berbeda dengan penggunaan pendekatan *black-box*, masih dari studi yang dilakukan oleh Ihantola dkk. (2010), penggunaan pendekatan *white-box* tergolong eksperimental dan masih kurang awam dalam sistem penilaian otomatis. Beberapa contoh penggunaan pendekatan ini adalah:

1. Pencarian perbedaan dalam bentuk representasi abstrak program, seperti penilaian program peserta didik berdasarkan perbandingan graf buatan K.A. Naude dkk. (2010)
2. Deteksi perbedaan semantik program, penilaian program dengan memperhatikan perbedaan arti antara implementasi program peserta didik dan implementasi program referensi

II.2 Deteksi Perbedaan Semantik Program

Semantik mengacu pada arti dari program dan bukan bentuk dari program (sintaksis). Semantik memberikan aturan untuk mengartikan sintaksis yang tidak memiliki arti secara langsung, sintaksis hanya membatasi kemungkinan interpretasi dari apa yang dinyatakan (Euzenat & Shvaiko, 2007). Beberapa penelitian terkait mengenai perbedaan semantik pada program adalah sebagai berikut:

1. Semantic Diff buatan Jackson & Ladd (1994) mengambil dua versi prosedur kemudian menghasilkan laporan berisi rangkuman perbedaan semantik antara keduanya. Perbedaan kemudian disajikan dalam bentuk perbandingan pemetaan masukan dan keluaran dari kedua program.
2. SymDiff buatan Lahiri dkk. (2012), kakas *language-agnostic* untuk pengecekan persamaan dan menunjukan perbedaan semantik antara program. Kakas dapat melakukan pemisahan urusan antara bahasa dan algoritma analisis dengan menerjemahkan bahasa ke *intermediate*

verification language bernama Boogie yang memanfaatkan *Satisfiability Modulo Theories* (SMT) *solver*.

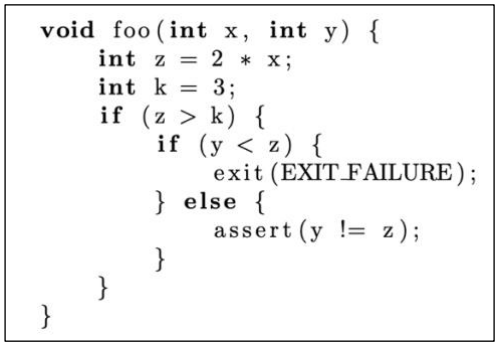
Penelitian terkait yang baru banyak memanfaatkan *symbolic execution* untuk menemukan perbedaan semantik, contohnya termasuk:

1. UCKLEE buatan Ramos & Engler (2011), kakas yang menghasilkan input secara otomatis dan menggunakan *bit-accurate symbolic execution* untuk memverifikasi dua program C menghasilkan keluaran yang sama pada jalur eksekusi terbatas.
2. iBinHunt buatan Ming dkk. (2012) melakukan *symbolic execution* inter prosedural (didalam *basic blocks*) dan memverifikasi suatu formula yang dihasilkan merepresentasikan pemetaan masukan dan keluaran. Hal ini dicapai dengan teknik *deep taint*, dimana program input dipasangkan *taint tags* kemudian diikuti propagasinya saat eksekusi program.

Pada hakikatnya, perbedaan semantik antara dua program menunjukkan kemungkinan adanya masukan yang menyebabkan eksekusi kedua program menghasilkan keluaran yang berbeda. Membuktikan perbedaan semantik antara jalur eksekusi merupakan tugas yang kompleks namun dapat dilakukan dengan berbagai macam pendekatan. Salah satu pendekatan yang paling banyak diteliti adalah dengan memanfaatkan *symbolic execution*.

II.3 *Symbolic Execution*

Symbolic Execution merupakan teknik analisis program dengan melakukan eksekusi program, perbedaannya eksekusi dilakukan dengan memasukkan simbol untuk menggantikan masukan normal (King, 1976). *Symbolic execution* dimanfaatkan untuk membantu menemukan semua jalur eksekusi yang ada pada sebuah program. Bersama dengan kumpulan jalur eksekusi itu dihasilkan masukan yang menyebabkan dilewatinya masing – masing jalur. Eksekusi akan berjalan seperti biasa dan variabel pada program akan direpresentasikan dalam bentuk ekspresi simbolik.



```
void foo(int x, int y) {
    int z = 2 * x;
    int k = 3;
    if (z > k) {
        if (y < z) {
            exit(EXIT_FAILURE);
        } else {
            assert(y != z);
        }
    }
}
```

Gambar II.3.1 Contoh Potongan Kode dalam Bahasa C (Son, 2017)

Gambar II.3.1 merupakan contoh potongan kode dalam bahasa C untuk mengilustrasikan cara kerja dari *symbolic execution*. Pada contoh ini fungsi foo menerima parameter masukan berupa *integer x* dan *integer y*. Sebagai langkah awal *x* dan *y* akan di inisialisasi dengan simbol seperti pada persamaan (II – 1).

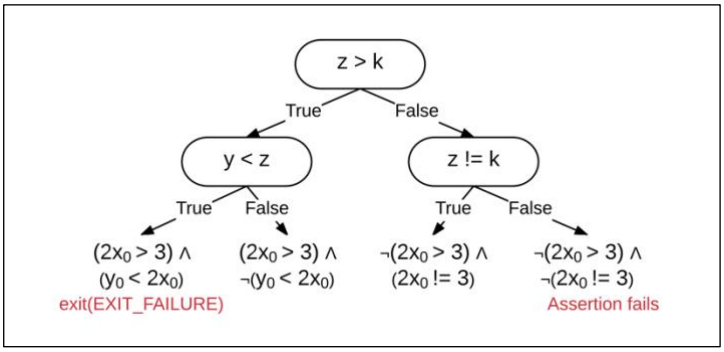
$$\{x = x_o, y = y_o\} \tag{II-1}$$

Setelah inisialisasi simbolik program akan dieksekusi. Semua kemungkinan pengambilan keputusan dalam eksekusi dapat digambarkan dalam bentuk pohon eksekusi seperti pada Gambar II.3.2.

Pohon tersebut merupakan kumpulan dari semua jalur eksekusi yang mungkin dilewati. Daun pada pohon di Gambar II.3.2 disebut sebagai *path constraint*. Setiap jalur eksekusi akan memiliki *path constraint* sendiri. *Path constraint* adalah merupakan representasi semua pilihan yang diambil untuk sebuah jalur eksekusi tertentu. Pada awalnya *path constraint* tiap jalur eksekusi adalah *true*. Saat kakas *symbolic execution* menemukan sebuah *control flow statement*, kondisi *true* dan *false* dari pernyataan tersebut akan di konjungsi ke masing – masing *path constraint*. Hal tersebut dapat digambarkan pada persamaan (II-2) dan (II-3).

$$pc_0 \supset q \tag{II-2}$$

$$pc_1 \supset \neg q \tag{II-3}$$



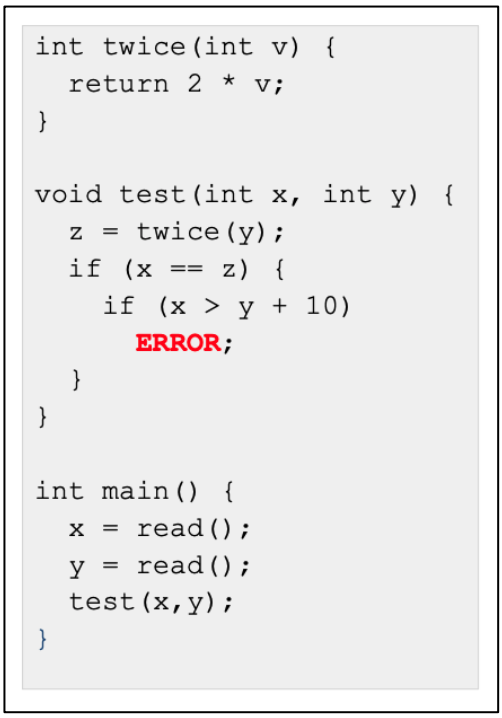
Gambar II.3.2 Pohon Eksekusi dari Contoh Potongan Kode (Son, 2017)

Path constraint juga dapat digunakan sebagai acuan masukan apa yang akan menyebabkan dilewatinya jalur eksekusi tersebut. Misalnya seperti pada Gambar II.3.2 masukkan x_o dan y_o yang memenuhi $(2x_o > 3) \wedge (y_o < 2x_o)$ akan menyebabkan program akan berakhir dengan status keluar EXIT_FAILURE.

Brumley dkk. (2007) memanfaatkan *symbolic execution* untuk mendeteksi deviasi jalur antara dua program. Penelitiannya bertujuan untuk menemukan deviasi antara implementasi protokol yang sama. Deviasi yang dimaksudkan adalah perbedaan dalam cara pengecekan dan pemrosesan masukan, yang mengungkapkan perbedaan interpretasi atau kesalahan dalam implementasi. Dalam penelitiannya, *symbolic execution* dimanfaatkan untuk menghasilkan *path constraint*. *Path constraint* ini kemudian dimasukan ke dalam sebuah formula yang dapat menunjukkan apakah terdapat deviasi jalur eksekusi diantara keduanya. *Symbolic execution* yang sebenarnya memiliki masalah cakupan pengujian atau *test coverage*, yakni capaiannya mungkin tidak terlalu dalam terutama karena *path constraint* yang tidak dapat diselesaikan oleh *constraint solver*. Dari kekurangan tersebut diteliti variasi lain dari *symbolic execution*.

II.3.1 *Concolic Execution*

Concolic execution atau *dynamic symbolic execution* adalah variasi dari *symbolic execution*. *Concolic* merupakan kepanjangan dari *cooperative concrete and symbolic execution*. Sen dkk. (2005) dalam penelitiannya mengenai CUTE,



```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Gambar II.3.3 Contoh Potongan Kode Ilustrasi Concolic Execution (Vechev, 2020)

concolic unit testing engine for c, menjelaskan bahwa *concolic execution* merupakan *symbolic execution* yang dijalankan bersamaan dengan eksekusi konkret. Pada eksekusinya, program dijalankan dengan nilai simbolik dan konkret, *path constraint* yang sebelumnya dalam bentuk simbolik disederhanakan dengan menggunakan nilai konkret. *Path constraint* kemudian digunakan secara inkremental untuk menghasilkan masukan pengujian yang memiliki cakupan yang lebih baik. Keuntungan utama yang dapat dicapai dengan penggunaan metode ini adalah adanya nilai konkret yang dapat digunakan untuk menyederhanakan *path constraint* yang tidak dapat diselesaikan oleh *constraint solver*. Potongan kode pada Gambar II.3.3 akan digunakan untuk mengilustrasikan *concolic execution*.

Eksekusi dimulai dengan memberikan nilai simbolik untuk masukan seperti pada *symbolic execution*, misalnya pada contoh ini $\{x = x_o, y = y_o\}$. Selain itu, digunakan nilai konkret untuk masukan, misalnya pada contoh ini $\{x = 22, y = 7\}$, selama eksekusi kita akan menyimpan kedua nilai konkret dan simbolik.

Kemudian program dieksekusi seperti biasa. Eksekusi dengan masukan tersebut akan mengembalikan hasil nilai variabel konkret (II-4), nilai variabel simbolik (II-5) dan path constraint pada persamaan (II-6).

$$\{x = 22, y = 7, z = 14\} \quad \text{(II-4)}$$

$$\{x = x_o, y = y_o, z = 2 \times y_o\} \quad \text{(II-5)}$$

$$x_o \neq 2 \times y_o \quad \text{(II-6)}$$

Selanjutnya *concolic execution* dapat memilih untuk menjelajahi jalur eksekusi lain, misalnya cabang *true* dari $x == z$ dengan menghasilkan sebuah masukan konkret. Untuk mendapatkan masukan yang dimaksud, dilakukan negasi terhadap *path constraint* (II-6) yang menghasilkan persamaan (II-7).

$$x_o = 2 \times y_o \quad \text{(II-7)}$$

Hasil negasi tersebut kemudian dimasukkan ke dalam sebuah *satisfiability modulo theories solver* yang akan menghasilkan masukan konkret $\{x_0 = 2, y_0 = 1\}$ yang akan digunakan untuk mengeksekusi program kembali. Eksekusi dengan masukan baru ini akan mengembalikan hasil nilai variabel konkret (II-8), nilai variabel simbolik (II-9) dan *path constraint* pada persamaan (II-10).

$$\{x = 2, y = 1, z = 2\} \quad \text{(II-8)}$$

$$\{x = x_o, y = y_o, z = 2 \times y_o\} \quad \text{(II-9)}$$

$$(x_o = 2 \times y_o) \wedge (x_o \leq y_o + 10) \quad \text{(II-10)}$$

Pada contoh ini *concolic execution* akan memilih untuk kembali menjelajahi jalur lain misalnya cabang *true* dari $x > y + 10$ dengan menghasilkan sebuah masukan konkret. Untuk mendapatkan masukan yang dimaksud, dilakukan negasi terhadap konjungsi *path constraint* (II-10) yang menghasilkan persamaan (II-11).

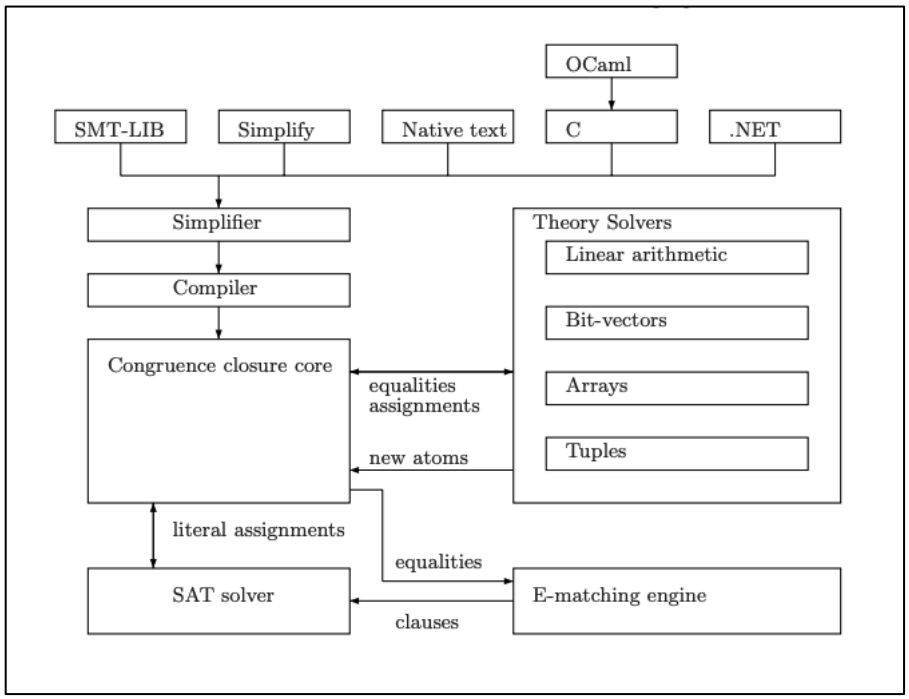
$$(x_o = 2 \times y_o) \wedge (x_o > y_o + 10) \quad \text{(II-11)}$$

Setelah negasi *path constraint* dimasukkan ke SMT *solver* didapatkan masukan konkret $\{x_0 = 30, y_0 = 15\}$. Eksekusi dengan masukan baru ini akhirnya akan menyebabkan tercapainya jalur eksekusi yang menghasilkan eror. Dapat kita lihat bahwa eksekusi konkret dapat memanfaatkan nilai simbolik untuk menghasilkan masukan yang menyebabkan dilewatinya jalur eksekusi yang baru.

II.4 *Satisfiability Modulo Theories*

Satisfiability modulo theories (SMT) mengacu pada permasalahan untuk menentukan apakah sebuah formula *first-order* dapat dipenuhi atau tidak berdasarkan *logical theory* (Barrett & Tinelli, 2018). Teori yang digunakan pada permasalahan ini seperti aritmatika, bit-vector, senarai, dan *uninterpreted function*. Dalam bidang ilmu komputer banyak masalah yang dapat disederhanakan menjadi pengecekan *satisfiability* sebuah formula dengan logika tertentu. Beberapa masalah bisa secara alami diformulasikan sebagai suatu permasalahan *satisfiability* yang dapat diselesaikan dengan SAT *solver* secara efisien pada level *boolean*. Namun terdapat juga masalah yang diformulasikan dalam bentuk *first-order logics* atau *higher-order logics* dengan bahasa yang mengandung variabel *non-boolean*. Banyak sistem yang dimodelkan seperti permasalahan tersebut dan translasi ke level *boolean* menjadi sesuatu yang mahal. Tujuan utama dari penelitian mengenai *satisfiability modulo theories* adalah untuk menciptakan mesin verifikasi yang dapat menyelesaikan masalah pada abstraksi *higher-level* selagi menjaga kecepatan dari SAT *solver* yang sudah ada. Dapat dikatakan bahwa *satisfiability modulo theories* menggeneralisir *boolean satisfiability* (SAT) dengan menambahkan penggunaan teori seperti aritmatika, bit-vector, dan teori *first-order* lainnya.

De Moura & Bjorner (2008) melakukan penelitian di bidang ini dan menghasilkan Z3, SMT *solver* yang digunakan pada berbagai macam verifikasi perangkat lunak dan analisis aplikasi. Pada subbab ini dilakukan pembahasan secara umum mengenai Z3 sebagai salah satu SMT *solver* yang efisien dan populer penggunaannya.



Gambar II.4.1 Arsitektur Z3 (De Moura & Bjorner, 2008)

Gambar II.4.1 menunjukkan gambaran skema arsitektur sistem Z3. Komponen utamanya terdiri dari:

1. *Simplifier*, pertama – tama masukan formula diproses menggunakan penyederhanaan yang tidak lengkap namun efisien. *Simplifier* ini menerapkan aturan reduksi aljabar standar, misalnya $p \wedge true \longmapsto p$. Selain itu *simplifier* juga menerapkan *contextual simplification* sederhana dengan mengidentifikasi definisi persamaan dalam sebuah konteks kemudian mereduksi sisa formula menggunakan definisi tersebut, misalnya seperti $x = 4 \wedge q(x) \longmapsto x = 4 \wedge q(4)$.
2. *Compiler*, representasi abstrak *syntax tree* dari formula yang telah disederhanakan diubah ke struktur data yang berisi kumpulan klausa dan simpul *congruence-closure*.
3. *Congruence closure core*, menerima nilai kebenaran *atoms* dari SAT *solver*. *Atoms* adalah formula atomik yang merupakan formula yang tidak mengandung subformula atau penghubung logis.

4. *Theory combination*, metode *theory combination* yang digunakan pada Z3 melakukan pencocokan model secara inkremental untuk tiap teori.
5. SAT *solver*, *solver* ini mengintegrasikan penggunaan metode *pruning* seperti pembelajaran lemma dan melakukan *backtracking* non-kronologis.
6. *Theory solver*, menggunakan *solver* aritmatika linear berdasarkan algoritma buaran Duterte dan De Moura (2006).

Seperti yang dibahas pada subbab sebelumnya SMT *solver* seperti Z3 dapat dimanfaatkan pada *concolic execution*. Untuk melakukan eksplorasi jalur lain dilakukan negasi terhadap *path constraint* untuk jalur saat ini, dengan memanfaatkan SMT *solver* dapat dihasilkan variabel yang memenuhi hasil negasi tersebut untuk digunakan sebagai masukan konkret. Selain itu, beberapa permasalahan dalam deteksi perbedaan semantik antara dua implementasi dapat direpresentasikan dalam bentuk formula. Formula ini kemudian bisa diperhatikan *satisfiability*-nya untuk menjawab pertanyaan seperti apakah terdapat deviasi antara dua jalur atau apakah dua jalur sama secara semantik seperti yang dilakukan pada penelitian Zhang dkk. (2014).

BAB III

ANALISIS DAN PERANCANGAN SISTEM

Pada bab ini dituliskan analisis mendalam mengenai masalah yang diangkat di tugas akhir ini. Kemudian diuraikan analisis dan rancangan mengenai solusi untuk penyelesaian masalah. Bab ini terdiri dari analisis masalah, analisis solusi, dan rancangan solusi.

III.1 Analisis Masalah

Metode yang populer digunakan dalam sistem penilaian otomatis masih memiliki celah kekurangan dalam efisiensi dan kelengkapan. Banyaknya penggunaan penilaian otomatis untuk persoalan pemrograman mendorong kegiatan penelitian dalam bidang ini. Dari penelitian–penelitian tersebut, dihasilkan berbagai macam kakas penilaian otomatis dengan pendekatan yang berbeda-beda. Dari Ihantola dkk. (2010), yang populer digunakan saat ini adalah pendekatan *black-box testing* khususnya pengujian program terhadap kasus uji yang dituliskan secara manual. Dalam penggunaan pendekatan penilaian otomatis tersebut, kualitas penilaian bergantung secara langsung terhadap kualitas kasus uji. Ketergantungan ini dapat menimbulkan masalah karena menuliskan kumpulan kasus uji yang cukup lengkap secara manual sulit untuk dilakukan. Selain itu penulisan kasus uji secara manual kurang efisien karena memakan banyak usaha dan waktu.

Ketidaklengkapan kumpulan kasus uji akan menyebabkan tidak dilewatinya jalur eksekusi tertentu yang mungkin menyebabkan kesalahan dalam penilaian. Kesalahan tersebut dapat diilustrasikan dengan potongan kode di Tabel III.1.1 dan contoh kumpulan kasus uji pada Tabel Tabel III.1.2. Potongan kode menunjukkan implementasi untuk mengembalikan nilai maksimum dari masukan tiga angka. Bila kita hanya menggunakan kasus uji 1 untuk pengujian, maka kedua implementasi akan mengembalikkan keluaran nilai yang sama yakni 3. Keluaran yang sama menunjukkan bahwa implementasi peserta didik benar. Namun ketika kita menambahkan kasus uji 2 untuk pengujian, keduanya akan mengembalikan

keluaran nilai yang berbeda. Implementasi referensi akan mengembalikan 2, sementara implementasi peserta didik akan mengembalikan 1. Keluaran yang berbeda menunjukkan bahwa implementasi peserta didik salah. Contoh ini memperlihatkan dengan kumpulan kasus uji yang tidak lengkap, yakni hanya menggunakan kasus uji 1, dapat disebabkan kesalahan dalam penilaian.

Terdapat usaha–usaha yang telah dilakukan untuk menangani masalah efisiensi dalam penilaian otomatis, contohnya pembangkitan kasus uji secara acak. Dengan pembangkitan kasus uji secara acak pengajar tidak perlu menuliskan kasus uji secara manual. namun usaha tersebut masih belum menyelesaikan masalah kelengkapan kasus uji. Berdasarkan analisis terhadap pendekatan sistem penilaian otomatis yang populer digunakan saat ini, muncul kebutuhan akan penelitian lebih lanjut untuk alternatif sistem penilaian otomatis yang lebih lengkap dan efisien.

Tabel III.1.1 Implementasi max_3

Implementasi referensi	**Implementasi peserta didik**

`def max_3(a: int, b: int, c: int) -> int:`
`  if a >= b and a >= c:`
`  return a`
`  elif b >= a and b >= c:`
`  return b`
`  else:`
`  return c`
	`def max_3(a: int, b: int, c: int) -> int:`
	`  if a > b and a > c:`
	`  return a`
	`  elif b > a and b > c:`
	`  return b`
	`  else:`
	`  return c`

Tabel III.1.2 Contoh kumpulan kasus uji untuk max_3

Kasus uji 1	(a=3, b=2, c=1)
Kasus uji 2	(a=2, b=2, c=1)

III.2 Analisis Solusi

Analisis masalah pada subbab sebelumnya telah menunjukkan bahwa pendekatan yang populer saat ini masih belum bisa melakukan penilaian secara lengkap. Hal ini berkaitan langsung dengan pendekatan *black-box testing* dimana kita hanya memperhatikan hasil keluaran dan tidak memperhatikan isi kode. Dengan mengabaikan isi kode seperti ini, mungkin akan ada jalur–jalur eksekusi tertentu yang tidak terlewati dalam proses penilaian. Hal ini menjadi akar penyebab ketidaklengkapan penilaian. Untuk menanggulangi hal tersebut perlu diteliti pendekatan yang berbeda, yang memperhatikan mekanisme internal dari kode. Secara teori, dengan memperhatikan isi kode kita dapat mengulas semua jalur eksekusi yang mungkin dapat dihasilkan sistem penilaian yang lebih lengkap. Pendekatan yang dimaksud adalah *white-box testing*. Lebih tepatnya, *white-box testing* digunakan untuk membangkitkan masukan yang lebih lengkap, penilaian tetap dilakukan dengan melakukan eksekusi program seperti pada pendekatan *black-box testing*.

Pendekatan *white-box testing* tergolong belum awam penggunaannya dalam penilaian otomatis. Salah satu contoh penggunaannya adalah evaluasi tugas pemrograman dengan pendekatan *control flow graph* yang merupakan bagian dari kelompok penelitian tugas akhir penulis. Terdapat juga penilaian yang menilai gaya pemrograman, contohnya menilai fitur efektivitas pada program dengan melihat lingkup variabel dan fitur tipografi dengan melihat konvensi penamaan, namun pendekatan ini tidak menilai kebenaran dari suatu program. Pada tugas akhir ini, dari mekanisme internal program akan dilakukan eksplorasi jalur eksekusi yang mungkin dan darinya dapat diukur perbedaan semantik antara implementasi referensi dan implementasi peserta didik sebagai karakteristik penilaian. Mengacu pada tujuan, sistem harus dapat menentukan kebenaran implementasi program peserta didik berdasarkan implementasi program referensi. Lebih jelasnya, ketika diberikan implementasi program peserta didik S dan implementasi program referensi P, yang ingin diketahui adalah apakah P dan S berperilaku sama. Untuk

mengetahui hal tersebut, perlu dijawab pertanyaan: apakah *P* dan *S* sama secara semantik?

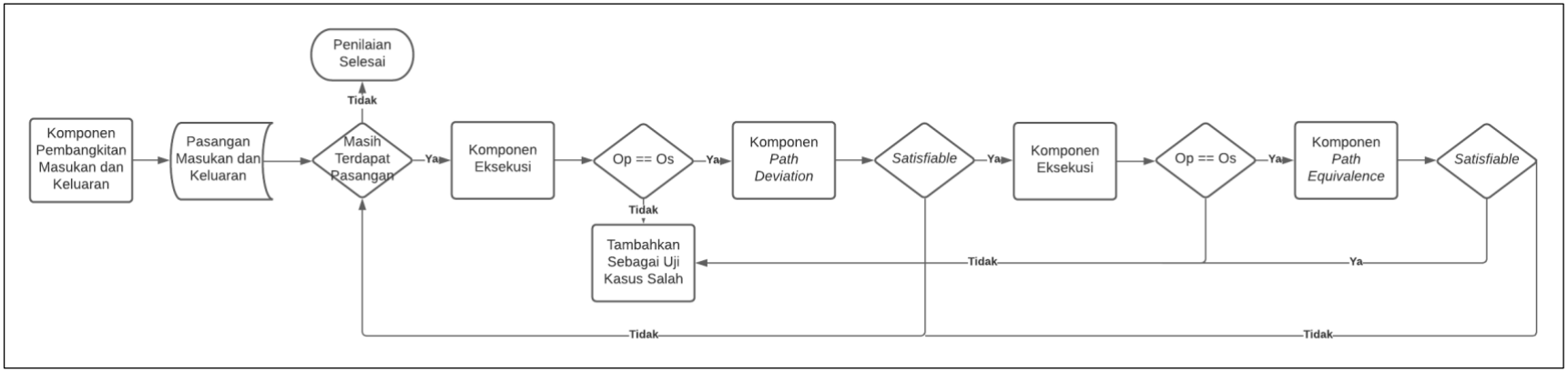
Untuk melakukan penjelajahan jalur eksekusi dapat digunakan *concolic execution*. *Concolic execution* merupakan variasi *symbolic execution* yang dapat meningkatkan cakupan eksplorasi dengan menggabungkan eksekusi dengan variabel simbolik dan variabel konkret. *Symbolic execution* telah dimanfaatkan pada beberapa penelitian lain untuk membandingkan dua buah implementasi program, contohnya Brumley dkk. (2007) yang memanfaatkan *symbolic execution* untuk menemukan deviasi antara implementasi sebuah protokol dan Zhang dkk. (2014) yang memanfaatkan *symbolic execution* untuk mendeteksi plagiarisme antara dua program. Penelitian–penelitian tersebut menunjukkan kemampuan *symbolic execution* dalam menjelajahi jalur eksekusi program. Eksplorasi yang saksama diharapkan dapat mendukung pengukuran perbedaan semantik antara implementasi referensi dan implementasi peserta didik yang tepat. Perbedaan semantik semata dapat digunakan sebagai penilaian (berbeda berarti salah, sama berarti benar). Namun, hal tersebut kurang mencukupi untuk penilaian di kelas. Diperlukan persentase kebenaran dari eksekusi uji kasus. Maka dari itu, pengukuran perbedaan semantik digunakan untuk menghasilkan kasus uji yang lengkap dan menghilangkan usaha untuk menuliskan kasus uji secara manual. Kasus uji yang dihasilkan akan di eksekusi seperti pada sistem penilaian pada umumnya. Dengan pengukuran perbedaan semantik yang tepat kita dapat menghasilkan sistem penilaian otomatis yang memiliki kasus uji yang lengkap dan efisien.

III.3 Rancangan Solusi

Pada subbab ini dijelaskan rancangan solusi yang dibuat berdasarkan analisis pada subbab sebelumnya. Bagian ini terdiri dari gambaran diagram alir umum solusi dan penjelasan komponen–komponennya.

III.3.1 Diagram Alir Umum Solusi

Ide utama dari solusi pada tugas akhir ini adalah untuk mencari perbedaan semantik dari implementasi program referensi dan implementasi program peserta didik. Hal



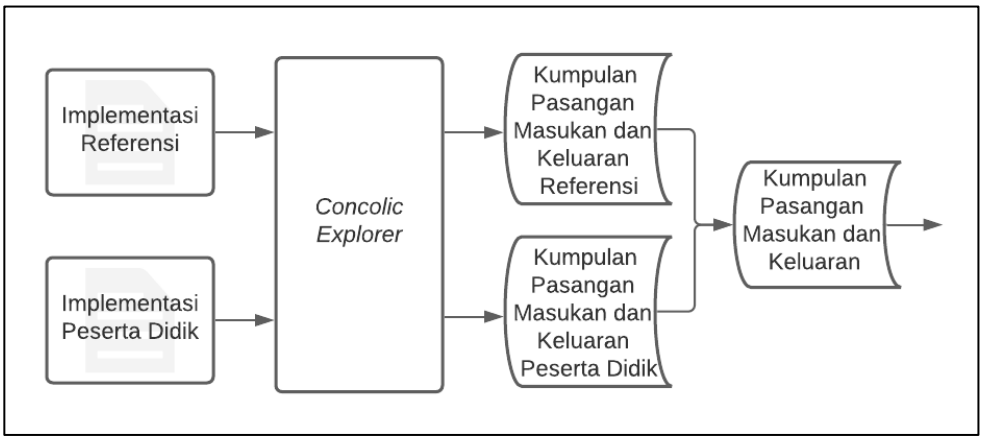
Gambar III.3.1 Diagram Alir Proses Solusi

tersebut dilakukan dengan melakukan *concolic exploration*, mendeteksi keberadaan *path deviation*, dan mendeteksi *path equivalence* di antara kedua implementasi program. *Path deviation* pertama kali diusulkan oleh Brumley dkk. (2007) untuk menemukan deviasi antara implementasi sebuah protokol komunikasi yang sama. *path equivalence* pertama kali diusulkan oleh Zhang dkk. (2014) dalam penelitiannya untuk mendeteksi plagiarisme antara dua program.

Pada tugas akhir ini, konsep *path deviation* dan *path* equivalence diadaptasi untuk penilaian otomatis implementasi program. *Path* deviation terjadi jika dua buah implementasi program dieksekusi dengan sebuah masukan *a* menghasilkan jalur eksekusi masing-masing, ketika dieksekusi dengan masukan *b* program pertama akan mengikuti jalur eksekusi asli dan program kedua mengikuti jalur eksekusi yang berbeda dari sebelumnya. *Path equivalence* diperlukan untuk memastikan deviasi bukan hanya disebabkan karena perbedaan sintaksis tetapi memang akan menunjukkan ke keluaran yang berbeda.

Solusi ini dibangun memanfaatkan *concolic execution* dan SMT *solver*. Seperti yang dijelaskan pada dasar teori, *concolic execution* merupakan variasi *symbolic execution* yang dapat meningkatkan cakupan eksplorasi. *Concolic execution* digunakan untuk menangkap semantik dari implementasi program yang kemudian disimpan sebagai *path constraint*. SMT *solver* digunakan untuk memecahkan beberapa permasalahan formula pada deteksi *path deviation*, deteksi *path equivalence*, dan pembangkitan masukan di tahap awal. Pemanfaatan kedua teknologi ini dibahas lebih lanjut pada penjelasan tiap komponen.

Gambar III.3.1 menunjukkan diagram alir proses solusi. Proses penilaian dimulai dengan pembangkitan kumpulan pasangan masukan dan keluaran berdasarkan implementasi program referensi dan implementasi program peserta didik. Selama masih terdapat pasangan masukan dan keluaran maka proses penilaian akan terus berjalan. Kemudian akan dilakukan eksekusi dengan salah satu pasangan masukan dan keluaran. Setiap dilakukannya eksekusi, jika keluaran yang dihasilkan berbeda maka akan dianggap implementasi program peserta didik tidak benar dan akan dikembalikan masukan yang menyebabkan kesalahan. Jika keluaran pada tahap

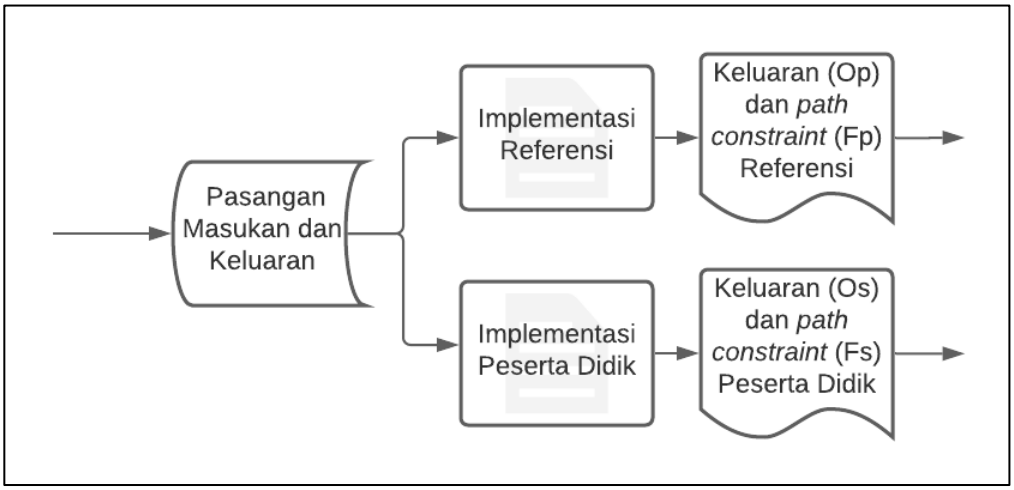


Gambar III.3.2 Komponen Pembangkitan Masukan dan Keluaran

eksekusi awal sama maka akan dilakukan deteksi *path deviation* dengan keluaran dan *path constraint* dari hasil eksekusi. Bila terdapat *path deviation* maka akan dilakukan eksekusi dari *counterexample* yang ditemukan. Jika keluaran pada tahap ini sama maka akan dilakukan deteksi *path equivalence* dengan keluaran dan *path constraint* dari hasil eksekusi. Bila *path equivalence* ditemukan maka implementasi peserta didik dianggap benar dan penilaian akan dilanjutkan ke iterasi berikutnya. Setiap komponen dijelaskan lebih lanjut pada beberapa bagian berikutnya.

III.3.2 Komponen Pembangkitan Kumpulan Masukan dan Keluaran

Komponen ini diilustrasikan pada Gambar III.3.2. Tujuan dari komponen ini adalah melakukan eksplorasi implementasi program referensi dan implementasi program peserta didik untuk menghasilkan kumpulan pasangan masukan dan keluaran pada awal penilaian. Komponen ini memanfaatkan sebuah *concolic explorer* untuk mengeksplorasi kedua implementasi program dengan *concolic execution*. Hasil dari komponen ini adalah pasangan masukan dan keluaran yang nantinya akan digunakan sebagai masukan komponen penilai.



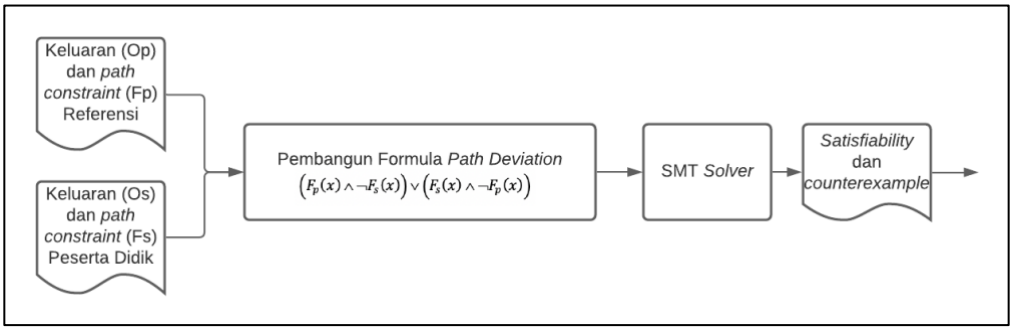
Gambar III.3.3 Komponen Eksekusi

III.3.3 Komponen Eksekusi

Komponen ini diilustrasikan pada Gambar III.3.3. Tujuan dari komponen ini adalah untuk melakukan eksekusi implementasi program referensi dan implementasi program peserta didik. Masukan dari komponen ini adalah pasangan masukan hasil pembangkitan komponen III.3.2, *counterexample* dari komponen deteksi *path deviation,* atau *counterexample* dari komponen deteksi *path equivalence*. Komponen ini akan mengembalikan keluaran dan *path constraint* dari hasil eksekusi.

III.3.4 Komponen Deteksi *Path Deviation*

Komponen ini diilustrasikan pada Gambar III.3.4. Tujuan dari komponen ini adalah untuk mendeteksi apakah terdapat *path deviation* di antara jalur eksekusi referensi dan peserta didik. Jika diberikan dua formula *path constraint*, masukan yang menunjukkan *path deviation* akan dapat memenuhi salah satu formula *path constraint* namun tidak yang lainnya. Dengan ini dilakukan pengecekan *satisfiability* dari formula (III-1) (Zhang dkk., 2014) dimana $F_p(x)$ adalah formula *path constraint* program referensi dan $F_s(x)$ adalah formula *path constraint* program peserta didik. Masukan dari komponen ini adalah keluaran dan *path*



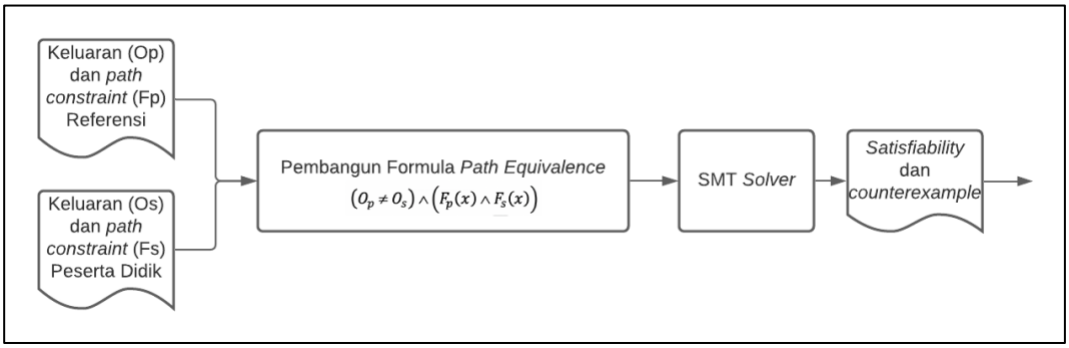
Gambar III.3.4 Komponen *Path Deviation*

constraint kedua program yang dihasilkan oleh komponen eksekusi. Keluaran dari komponen ini adalah *satisfiability* dari pengecekan formula dan *counterexample* bila formula dapat dipenuhi. Prosesnya meliputi pembangunan formula *path deviation* dalam bentuk ekspresi yang dapat diterima SMT *solver* dan penyelesaian formula. Secara umum, ketika formula tidak dapat dipenuhi, maka akan dianggap tidak ada *path deviation* dan penilaian akan dilanjutkan ke iterasi berikutnya. Ketika formula dapat dipenuhi, maka akan dianggap terdapat *path deviation* dan akan dikembalikan *counterexample* sebagai masukan dari komponen eksekusi.

$$\left(F_p(x) \wedge \neg F_s(x)\right) \vee \left(F_s(x) \wedge \neg F_p(x)\right) \quad \text{(III-1)}$$

III.3.5 Komponen Deteksi *Path Equivalence*

Komponen ini diilustrasikan pada Gambar III.3.5. Tujuan dari komponen ini adalah untuk mendeteksi apakah terdapat *path equivalence* di antara jalur eksekusi referensi dan peserta didik. Hal ini dilakukan untuk memastikan *path deviation* yang ditemukan memang akan menunjukkan ke keluaran yang berbeda. Dari dua formula *path constraint* akan dibentuk konjungsi untuk mencari sebuah masukan yang dapat menghasilkan keluaran yang berbeda dari kedua program. Untuk melakukan hal tersebut, akan dilakukan pengecekan *satisfiability* dari formula (III-2) (Liu dkk., 2019) dimana O_p adalah keluaran simbolik program referensi, O_s adalah keluaran simbolik program peserta didik, $F_p(x)$ adalah formula *path constraint* program referensi, dan $F_s(x)$ adalah formula *path constraint* program



Gambar III.3.5 Komponen Path Equivalence

peserta didik. Masukan dari komponen ini adalah keluaran dan *path constraint* kedua program yang dihasilkan oleh komponen eksekusi. Keluaran dari komponen ini adalah *satisfiability* dari pengecekan serta *counterexample* bila formula dapat dipenuhi. Seperti pada komponen *path* deviation, prosesnya meliputi pembangunan formula *path equivalence* dalam bentuk ekspresi yang dapat diterima SMT *solver* dan penyelesaian formula. Secara umum, ketika formula tidak dapat dipenuhi, maka akan dianggap terdapat *path equivalence* dan penilaian akan dilanjutkan ke iterasi berikutnya. Ketika formula dapat dipenuhi, maka akan dianggap tidak terdapat *path equivalence* dan akan dikembalikan *counterexample* sebagai masukan yang menyebabkan keluaran yang berbeda.

$$(O_p \neq O_s) \wedge (F_p(x) \wedge F_s(x)) \quad \text{(III-2)}$$

BAB IV

IMPLEMENTASI DAN PENGUJIAN

Pada bab ini dituliskan proses implementasi dari rancangan solusi yang dibahas pada Bab III dan pengujian hasil implementasi. Pembahasan pada bab ini terdiri dari lingkungan implementasi dan pengujian, implementasi setiap komponen, dan pengujian serta evaluasi hasil implementasi.

IV.1 Lingkungan Implementasi dan Pengujian

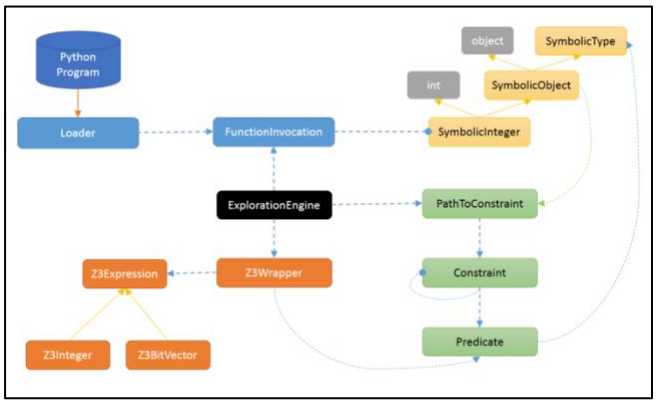
Implementasi program dan pengujian dituliskan dalam bahasa Python versi 3.9.0. Pustaka yang digunakan adalah PyExZ3 versi 1.0.0 dan Z3 versi 4.8.10. Implementasi dan pengujian dilakukan dalam lingkungan dengan spesifikasi pada Tabel IV.2.1.

IV.2 Implementasi

Terdapat empat komponen utama yang diimplementasikan pada tugas akhir ini, yakni komponen pembangkitan kumpulan masukan dan keluaran, komponen eksekusi, komponen deteksi *path deviation*, dan komponen deteksi *path equivalence*. Untuk mendukung pembangunan komponen tersebut digunakan SMT *solver* Z3 dan pustaka *concolic execution*, PyExZ3 yang dijelaskan lebih lanjut pada subbab ini.

Tabel IV.2.1 Spesifikasi Lingkungan Implementasi dan Pengujian

Sistem Operasi	macOS Catalina 10.15.7
Prosesor	2,3 GHz Dual-Core Intel Core i5
Memori	8 GB 2133 MHz LPDDR3
IDE	Visual Studio Code



Gambar IV.2.1 Arsitektur PyExZ3 (Ball dkk., 2015)

IV.2.1 Pustaka *Concolic Execution* PyExZ3

Untuk melakukan *concolic execution* pada tugas akhir ini, dimanfaatkan PyExZ3 oleh Ball dkk. (2015) sebagai pustaka eksplorasi. PyExZ3 adalah kakas *concolic execution* (disebut juga sebagai *dynamic symbolic execution* pada penelitiannya) untuk program berbahasa Python. Selain untuk melakukan hal tersebut, teknik perekaman *path constraint* PyExZ3 digunakan untuk mendukung komponen eksekusi. Arsitektur PyExZ3 digambarkan pada Gambar IV.2.1.

Kakas ini memiliki kelas buatan yang fungsinya adalah menyimpan simbol dan nilai konkret dari suatu variabel. Mengikuti prinsip *concolic execution* yang dijelaskan pada subbab II.3, secara garis besar yang dilakukan oleh PyExZ3 adalah sebagai berikut:

1. Menginisiasi variabel masukan sebagai objek yang menyimpan simbol dan nilai konkret
2. Melakukan eksekusi program. Saat eksekusi, setiap dilakukan pengecekan nilai kebenaran (object.__bool__ dipanggil) akan disimpan *constraint* ke dalam *path constraint*
3. Melakukan pembangkitan masukan yang baru dengan memanfaatkan Z3 sebagai SMT *solver*
4. Melakukan eksplorasi lebih lanjut berdasarkan masukan yang baru

Kakas ini memanfaatkan Z3 yang dibahas pada subbab II.4. Perlu diperhatikan bahwa untuk menggunakan Z3 akan dilakukan translasi dari *constraint* yang didapatkan ke dalam bentuk ekspresi yang dapat diterima oleh Z3.

IV.2.2 Implementasi Komponen Pembangkitan Kumpulan Masukan dan Keluaran

Komponen ini diimplementasikan untuk melakukan eksplorasi implementasi program referensi dan implementasi program peserta didik untuk menghasilkan kumpulan pasangan masukan dan keluaran hasil. *Concolic exploration* dilakukan dengan menggunakan PyExZ3 sebagai pustaka *concolic execution*. Komponen ini menerima masukan berupa implementasi fungsi dalam bentuk objek yang menyimpan fungsi dan variabel masukan fungsi. Eksplorasi diawali dengan menginisialisasi variabel ke dalam bentuk objek yang menyimpan simbol variabel dan nilai variabel. Contohnya, pada fungsi *foo(a)*, variabel *a* akan memiliki simbol '*a*' dan nilai 0. Hingga eksplorasi selesai akan dilakukan eksekusi implementasi program dengan tiap masukan yang baru dibangkitkan. Semua *constraint* yang ditemukan selama eksekusi akan disimpan ke dalam senarai kumpulan *constraint*. Pada tiap iterasi akan diambil *constraint* dari kumpulan *constraint* untuk membangkitkan masukan baru dengan menggunakan Z3 sebagai SMT *solver*. Sampai senarai kumpulan *constraint* kosong maka eksplorasi akan terus berjalan. Hasil eksplorasi yang dilakukan pada implementasi referensi dan peserta didik kemudian akan digabungkan. Komponen ini akan mengembalikan senarai kumpulan masukan (berupa *tuple* simbol variabel dan nilai variabel) dan pasangan keluarannya.

IV.2.3 Implementasi Komponen Eksekusi

Komponen ini diimplementasikan untuk melakukan eksekusi implementasi fungsi referensi dan implementasi fungsi peserta didik. Eksekusi diawali dengan memperbarui nilai variabel dengan masukan yang baru. Kemudian fungsi akan dipanggil dengan variabel yang telah diperbarui. Dalam proses eksekusi, setiap dilakukan pengecekan nilai kebenaran (*if*, *while*, *and*, dan *or*) *constraint* akan

disimpan ke dalam *path constraint*. Setiap eksekusi memiliki sebuah *path constraint*. *Path constraint* kemudian akan diubah ke dalam ekspresi yang dibangun dengan objek Z3. Hal ini dilakukan agar ekspresi dapat diterima oleh Z3 sebagai SMT *solver*. Komponen ini akan mengembalikan *path constraint* yang dapat diterima Z3 serta keluaran hasil eksekusi implementasi referensi dan peserta didik.

IV.2.4 Implementasi Komponen Deteksi *Path Deviation*

Komponen ini diimplementasikan untuk mendeteksi apakah terdapat *path deviation* di antara *path constraint* referensi dan peserta didik. Deteksi diawali dengan membangun formula yang dijelaskan pada bagian III.3.4 ke dalam ekspresi yang dapat diterima oleh Z3 seperti pada kode IV.2.1. Kemudian dilakukan penyelesaian formula oleh Z3 sebagai SMT *solver*. Komponen ini akan mengembalikan *satisfiability* dan jika formula dapat dipenuhi maka akan dikembalikan pula masukan baru yang dibangkitkan dari penyelesaian formula. Bila tidak dapat dipenuhi, maka dapat dianggap tidak ada *path deviation* dan penilaian akan dilanjutkan ke iterasi selanjutnya tanpa pengecekan *path equivalence*.

```
Or(And(fp, Not(fs)), And(fs, Not(fp)))
```

Kode IV.2.1 Formula *Path Deviation* dalam ekspresi Z3

IV.2.5 Implementasi Komponen Deteksi *Path Equivalence*

Komponen ini diimplementasikan untuk mendeteksi apakah terdapat *path equivalence* di antara *path constraint* referensi dan peserta didik. Deteksi diawali dengan membangun formula yang dijelaskan pada bagian III.3.5 ke dalam ekspresi yang dapat diterima oleh Z3 seperti pada kode IV.2.2. Kemudian dilakukan penyelesaian formula oleh Z3 sebagai SMT solver. Komponen ini akan mengembalikan *satisfiability* dan jika formula dapat dipenuhi maka akan dikembalikan pula masukan baru yang dibangkitkan dari penyelesaian formula. Masukan baru ini akan menyebabkan eksekusi yang menghasilkan keluaran yang berbeda diantara implementasi referensi dan peserta didik.

```
And(op!=os, And(fp, fs))
```

Kode IV.2.2 Formula *Path Equivalence* dalam ekspresi Z3

IV.3 Batasan Implementasi

Program yang dihasilkan pada tugas akhir ini memiliki batasan implementasi sebagai berikut:

1. Implementasi yang dinilai dalam bentuk fungsi dengan *return statement* masukan tipe data *integer*.

IV.4 Pengujian

Pada subbab ini dilakukan pembahasan mengenai pengujian terhadap sistem hasil implementasi tugas akhir. Yang dibahas adalah tujuan pengujian, skenario pengujian, dan hasil pengujian dan evaluasi.

IV.4.1 Tujuan Pengujian

Pengujian ini dilakukan untuk tujuan berikut:

1. Untuk mengetahui apakah sistem hasil implementasi tugas akhir dapat menentukan kebenaran implementasi program peserta didik berdasarkan implementasi program referensi dengan memperhatikan perbedaan semantik di antara keduanya.
2. Untuk menguji jika umpan balik yang dihasilkan program adalah nilai implementasi program peserta didik dan bila implementasi program peserta didik tidak benar dikembalikan pula masukan yang menunjukkan kesalahan
3. Untuk membandingkan sistem hasil implementasi tugas akhir dengan sistem dengan pendekatan pembangkitan masukan secara acak

Implementasi dapat dikatakan berhasil jika dapat menentukan kebenaran implementasi program peserta didik berdasarkan implementasi program referensi. Umpan balik yang dihasilkan sistem harus sesuai dengan yang dinyatakan pada batasan masalah. Perbandingan dengan pendekatan lain dilakukan untuk melihat apakah ada *edge cases* yang tidak ditangani sistem dengan pendekatan

pembangkitan masukan secara acak namun ditangani oleh sistem hasil implementasi tugas akhir.

IV.4.2 Skenario Pengujian

Untuk melakukan pengujian terhadap sistem diperlukan contoh – contoh persoalan pemrograman serta implementasi program peserta didik. Contoh persoalan diambil dari praktikum mata kuliah IF1210 Dasar Pemrograman tahun ajaran 2019/2020 dan latihan mata kuliah KU1102 Pengenalan Komputasi. Contoh implementasi program peserta didik diambil dari submisi praktikum dan buatan penulis. Pengujian diawali dengan pengumpulan contoh tersebut. Karena sistem hasil implementasi tugas akhir memiliki batasan seperti yang dijelaskan pada subbab IV.3, implementasi peserta didik dimodifikasi ke dalam bentuk fungsi yang menerima masukan *integer* dan memiliki *return statement*. Implementasi peserta didik yang digunakan dalam pengujian berbeda-beda, tidak ada dua implementasi dengan cara yang sama. Setelah ini sistem hasil implementasi tugas akhir akan disebut juga sebagai sistem penilaian dengan semantik.

Setelah contoh persoalan terkumpul, dilakukan pembuatan implementasi referensi untuk tiap persoalan. Setelah kita memiliki implementasi referensi dan peserta didik untuk tiap persoalan kita dapat mulai menguji sistem. Antarmuka pada *command-line* ditunjukkan pada Kode IV.4.1.

```
python grade.py <implementasi_referensi>.py
      <implementasi_peserta_didik>.py
```

Kode IV.4.1 Antarmuka *command-line* pengujian

Perintah tersebut dijalankan untuk tiap pasang implementasi referensi dan implementasi peserta didik. Untuk setiap perintah yang dijalankan kita dapat mencapai dua tujuan pengujian pertama dengan menjawab pertanyaan berikut:

1. Apakah sistem hasil implementasi tugas akhir dapat menentukan kebenaran implementasi program peserta didik berdasarkan implementasi program referensi dengan memperhatikan perbedaan semantik di antara keduanya?

2. Apakah umpan balik yang dihasilkan program adalah nilai implementasi program peserta didik dan bila implementasi program peserta didik tidak benar dikembalikan pula masukan yang menunjukkan kesalahan?

Untuk mencapai tujuan pengujian ketiga digunakan penilaian dengan pendekatan pembangkitan uji kasus secara acak. Setiap pasang implementasi referensi dan implementasi peserta didik dinilai dengan menggunakan pembangkitan uji kasus secara acak kemudian dibandingkan dengan hasil dari penilaian oleh sistem hasil implementasi tugas akhir untuk menjawab pertanyaan berikut:

1. Apakah ada *edge cases* yang tidak ditangani sistem penilaian dengan pendekatan pembangkitan kasus uji secara acak namun ditangani oleh sistem hasil implementasi tugas akhir?

Tabel IV.4.1 menunjukan contoh persoalan yang digunakan dalam pengujian beserta dengan penjelasan singkat mengenai persoalan.

IV.4.3 Hasil Pengujian dan Evaluasi

Hasil pengujian dan evaluasi dibagi menjadi dua bagian, yakni penilaian menggunakan sistem hasil implementasi tugas akhir dan perbandingan dengan penilaian dengan pembangkitan kasus uji secara acak.

Tabel IV.4.1 Persoalan yang Diujikan

Nama Persoalan	Penjelasan Singkat
max_3	Mencari nilai maksimal dari tiga integer
segiempat	Membangun *string* berbentuk segiempat dengan karakter ‘*’ dan ‘#’
student_grade	Mengembalikan indeks berdasarkan nilai siswa
is_allowed_to_buy	Mengembalikan boleh atau tidak berdasarkan daftar barang belanja
no_of_triangle	Mengembalikan jumlah susunan segitiga yang dapat dibentuk dengan tiga panjang sisi
air	Menentukan bentuk air dengan masukan suhu

Tabel IV.4.2 Contoh Implementasi max_3

Implementasi Referensi	**Implementasi Peserta Didik**

`def max_3(a, b, c):`
`  if a >= b and a >= c:`
`  return a`
`  elif b >= a and b >= c:`
`  return b`
`  else:`
`  return c`
	`def max_3_1(a, b, c):`
	`  if a > b and a > c:`
	`  return a`
	`  elif b > a and b > c:`
	`  return b`
	`  else:`
	`  return c`

Tabel IV.4.3 Pasangan Masukan dan Keluaran dari *concolic exploration* max_3

Masukan			**Keluaran**
a	**b**	**c**	
0	0	0	0
0	2	0	2
0	0	1	1
1	2	3	3
1	0	2	2
2	0	0	2
4	5	0	5
2	0	8	8
0	1	2	2

IV.4.3.1 Penilaian Menggunakan Sistem Hasil Implementasi Tugas Akhir

Pengujian dilakukan terhadap semua contoh persoalan pada Tabel IV.4.1. Untuk menjelaskan hasil pengujian digunakan salah satu pasangan implementasi program referensi dan peserta didik untuk persoalan "max_3". Tabel IV.4.2 menunjukkan pasangan yang kita gunakan sebagai contoh. Contoh ini sama dengan yang digunakan untuk menjelaskan analisis masalah. Dengan menjalankan sistem dengan pilihan *logging* secara *verbose* kita dapat melihat secara detail proses yang dilewati selama penilaian. Pada awal penilaian dihasilkan pasangan masukan dan keluaran berdasarkan *concolic exploration* pada Tabel IV.4.3. Dari kumpulan ini akan diambil salah satu pasangan untuk menjalankan iterasi penilaian. Dengan menggunakan masukan a = 0, b = 0, c = 1 kedua implementasi dieksekusi dan akan dikembalikan hasil yang sama, yakni 1. Setiap eksekusi akan merekam pilihan yang diambil ke dalam bentuk *path constraint*. *Path constraint* dari eksekusi pertama ini dapat dilihat pada Tabel IV.4.4. Karena eksekusi pertama mengembalikan nilai yang sama, akan dilanjutkan deteksi *path deviation*.

Tabel IV.4.4 *Path Constraint* Eksekusi max_3 Pertama

Implementasi Referensi	`And(a < b, b >= a, b >= c)`
Implementasi Peserta Didik	`And(a <= b, b <= a)`

Tabel IV.4.5 *Path Deviation* max_3

Formula *Path Deviation*

`Or(And(And(a >= b, a < c, b >= a, b < c),`
`  Not(And(a <= b, b <= a))),`
`  And(And(a <= b, b <= a),`
`  Not(And(a >= b, a < c, b >= a, b < c))))`

Berdasarkan *path constraint* pada Tabel IV.4.4, dihasilkan formula *path deviation* pada Tabel IV.4.5. Setelah formula *path deviation* dimasukkan ke dalam SMT *solver*, hasilnya adalah *satisfiable* dan dihasilkan masukan a = 0, b = 0, c = 0. Dengan masukan tersebut dilakukan eksekusi dan akan dikembalikan hasil yang sama, yakni 0. *Path constraint* dari eksekusi kedua ini dapat dilihat pada Tabel IV.4.6. Karena eksekusi setelah deteksi *path deviation satisfiable*, maka penilaian akan dilanjutkan ke deteksi *path equivalence*.

Berdasarkan *path constraint* pada Tabel IV.4.6, dihasilkan formula *path equivalence* pada Tabel IV.4.7. Setelah formula *path equivalence* dimasukkan ke dalam SMT *solver*, hasilnya adalah *satisfiable* dan dihasilkan masukan a = 0, b = 0, c = -1. Dengan masukan tersebut dilakukan eksekusi dan akan dikembalikan hasil yang berbeda, yakni 0 untuk implementasi referensi dan -1 untuk implementasi peserta didik. Penilaian akan dilanjutkan untuk uji kasus lainnya. Dari sini dapat dilihat sistem hasil implementasi tugas akhir dapat menentukan kebenaran implementasi program peserta didik berdasarkan implementasi program referensi dengan memperhatikan perbedaan semantik di antara keduanya.

Tabel IV.4.6 *Path Constraint* Eksekusi max_3 Kedua

Implementasi Referensi	`And(a >= b, a >= c)`
Implementasi Peserta Didik	`And(a <= b, b <= a)`

Tabel IV.4.7 *Path Equivalence* max_3

Formula *Path Equivalence*
`And(a != c, And(And(a >= b, a >= c), And(a <= b, b <= a)))`

Penilaian dilanjutkan hingga pasangan masukan dan keluaran yang dibangkitkan di awal habis. Setelah penilaian berakhir akan dikembalikan hasil seperti pada Tabel IV.4.8. Dari sini dapat dilihat umpan balik yang dihasilkan program adalah nilai implementasi program peserta didik. Bila implementasi program peserta didik tidak benar dikembalikan pula masukan yang menunjukkan kesalahan. Penilaian juga dapat dilakukan pada implementasi yang mengandung *loop.* Contohnya pada persoalan arithmetic_seq yang mengembalikan nilai jumlah deret aritmatika. Tabel IV.4.9 menunjukkan dua implementasi *loop* dengan menggunakan ekspresi *while* dan *for*. Berdasarkan implementasi referensi, implementasi peserta didik ini tidak mengandung kesalahan. Hasil penilaian lengkapnya dapat dilihat pada lampiran A.1.

Tabel IV.4.8 Hasil Penilaian max_3

Umpan Balik dalam Bentuk JSON


```
{
    "reference": "max_3",
    "grading": "max_3_1",
    "grade": 91.66666666666666,
    "tested_case": {
        "(('a', 0), ('b', 0), ('c', 0))": [0, 0],
        "(('a', 0), ('b', 0), ('c', 1))": [1, 1],
        "(('a', 0), ('b', 2), ('c', 0))": [2, 2],
        "(('a', -1), ('b', 0), ('c', 0))": [0, 0],
        "(('a', 0), ('b', 0), ('c', -1))": [0, -1],
        "(('a', 1), ('b', 2), ('c', 3))": [3, 3],
        "(('a', 1), ('b', 0), ('c', 2))": [2, 2],
        "(('a', 0), ('b', -1), ('c', 0))": [0, 0],
        "(('a', 2), ('b', 0), ('c', 0))": [2, 2],
        "(('a', 4), ('b', 5), ('c', 0))": [5, 5],
        "(('a', 2), ('b', 0), ('c', 8))": [8, 8],
        "(('a', 0), ('b', 1), ('c', 2))": [2, 2]
    },
    "wrong_case": {
        "(('a', 0), ('b', 0), ('c', -1))": [0, -1]
    }
}
```

Sistem juga menangani bila ada penggunaan ekspresi *break*. Misalnya bila implementasi peserta didik dimodifikasi menjadi seperti pada Tabel IV.4.10. Penambahan ekspresi *break* menyebabkan kesalahan pada uji kasus n = 8 dengan implementasi referensi mengembalikan 36 dan implementasi peserta didik mengembalikan 28. Hasil penilaian lengkapnya dapat dilihat pada lampiran A.2.

Tabel IV.4.9 Contoh Implementasi arithmetic_seq

Implementasi Referensi	**Implementasi Peserta Didik**

def arithmetic_seq_2(n):
    if n < 1:
        return 1
    ret = 0
    for i in range(1, n+1):
        ret += i
    return ret
	def arithmetic_seq_3(n):
	    if n < 1:
	        return 1
	    ret = 0
	    i = 1
	    while i <= n:
	        ret += i
	        i += 1
	    return ret

Tabel IV.4.10 Contoh Implementasi arithmetic_seq dengan Ekspresi *break*

Implementasi Peserta Didik

def arithmetic_seq_4(n):
    if n < 1:
        return 1
    ret = 0
    i = 1
    while i <= n:
        ret += i
        i += 1
        if i == 8:
            break
    return ret

Sistem penilaian dengan semantik juga dapat dilakukan pada implementasi yang mengandung rekursi. Masih dengan persoalan yang sama, implementasi dengan rekursi dapat ditunjukkan pada Tabel IV.4.11. Berdasarkan implementasi referensi, implementasi peserta didik ini tidak mengandung kesalahan. Hasil penilaian lengkapnya dapat dilihat pada lampiran A.3.

IV.4.3.2 Perbandingan dengan Penilaian dengan Pembangkitan Kasus Uji Secara Acak

Pada bagian ini dilakukan pengujian dengan penilai dengan pembangkitan kasus uji secara acak. Sistem ini membangkitkan kumpulan masukan secara acak dan melakukan perbandingan keluaran hasil eksekusi implementasi referensi dan peserta didik untuk penilaian. Hasilnya dibandingkan dengan sistem penilaian dengan semantik. Sistem penilaian dengan pembangkitan kasus uji secara acak dibangun dengan memanfaatkan beberapa komponen pada sistem penilaian dengan semantik. Sistem akan menerima masukan berupa implementasi fungsi dalam bentuk objek yang menyimpan fungsi dan variabel masukan fungsi. Penilaian dilakukan dengan membangkitkan masukan yang baru secara acak untuk tiap iterasi. Untuk mendapatkan nilai acak, dimanfaatkan modul random milik Python yang menghasilkan nilai *pseudo-random*. Hasil penilaian ditunjukkan pada Tabel IV.4.14 dan Tabel IV.4.15. Hasil juga direpresentasikan dalam bentuk grafik pada Gambar IV.4.1.

Tabel IV.4.11 Contoh Implementasi arithmetic_seq dengan Rekursi

Implementasi Peserta Didik

`def arithmetic_seq(n):`
`  if n <= 1:`
`  return 1`
`  else:`
`  return n + arithmetic_seq(n-1)`

Hasil pengujian menunjukkan secara keseluruhan sistem penilaian dengan semantik mencakup lebih banyak *edge cases* dibandingkan dengan sistem penilaian dengan pembangkitan kasus uji secara acak.

Pengujian menunjukkan pada sistem penilaian dengan pembangkitan kasus uji secara acak dengan menjalankan pengacakan kasus uji dengan jumlah yang cukup banyak maka semakin banyak pula *edge cases* yang tertangani. Banyak implementasi peserta didik seperti pada persoalan segiempat yang tercakup *edge cases*-nya dengan pembangkitan kasus uji secara acak. Namun dari Gambar IV.4.1 dapat dilihat pada banyak hasil penilaian implementasi peserta didik dengan pembangkitan kasus uji secara acak memiliki nilai yang jauh lebih tinggi dibandingkan dengan penilaian dengan semantik. Hal ini menunjukkan ada kasus-kasus yang tidak dicakup oleh penilaian dengan pembangkitan kasus uji secara acak. Hal ini terlihat semakin jelas pada persoalan yang kompleks seperti pada is_allowed_to_buy. Jalur–jalur pada persoalan ini memiliki pernyataan yang banyak sehingga semakin kecil kemungkinannya ditemukan kesalahan dengan kasus uji acak.

Pada penilaian max_3 dengan contoh implementasi Tabel IV.4.2, pembangkitan kasus uji secara acak tidak dapat mencakup *edge cases*. Sistem penilaian dengan semantik memiliki cakupan yang lebih baik pada kasus ini. Pada kasus ini sistem penilaian dengan semantik berhasil membangkitkan masukan a = 0, b = 0, c = -1 yang menunjukkan kesalahan pada implementasi peserta didik. Masukan tersebut dapat dibangkitkan karena dilakukan deteksi *path equivalence*, yang arti formulanya adalah untuk mencari masukan yang memenuhi kedua jalur eksekusi referensi dan peserta didik namun dapat mengembalikan hasil yang berbeda. Dengan menggunakan pembangkitan kasus uji secara acak, kondisi dimana masukan a = b, a > c, dan a > b belum tentu dapat ditemukan.

Kelebihan sistem penilaian dengan semantik juga dapat dilihat pada penilaian contoh implementasi Tabel IV.4.12. Pada hasil penilaian dengan pembangkitan kasus uji secara acak tidak ditemukan kesalahan pada implementasi peserta didik. Kesalahan tidak ditemukan karena tidak berhasil dibangkitkan masukan yang

mengembalikan “ANTARA CAIR-GAS.” Dengan menggunakan *concolic exploration* pada penilaian dengan semantik, masukan tersebut berhasil dibangkitkan. Pada eksplorasi dibangkitkan masukan yang memenuhi t > 100, kemudian untuk meneruskan eksplorasi dibangkitkan masukan yang dapat memenuhi negasi-nya yakni t ≤ 100. Darinya dihasilkan masukan t = 100, yang menunjukkan kesalahan pada implementasi peserta didik. Hasil penilaian selengkapnya dapat dilihat pada lampiran A.4 dan A.5.

Perlu diperhatikan dari Gambar IV.4.1 beberapa penilaian dengan pembangkitan kasus uji acak memiliki nilai yang lebih rendah dari penilaian dengan semantik. Hal ini disebabkan jumlah kasus uji acak lebih banyak, dimana diantaranya melewati jalur eksekusi yang sama, dan bukan dikarenakan kasus uji yang lebih lengkap dari penilaian dengan semantik. Misalnya pada persoalan student_grade, terdapat penilaian dengan nilai 76.9% untuk semantik sementara 57.1% untuk kasus uji acak.

Tabel IV.4.12 Contoh Implementasi air

Implementasi Referensi	**Implementasi Peserta Didik**

`def air(t):`
`  if t < 0:`
`  return "PADAT"`
`  elif t > 0 and t < 100:`
`  return "CAIR"`
`  elif t > 100:`
`  return "GAS"`
`  elif t == 0:`
`  return "ANTARA PADAT-CAIR"`
`  else:`
`  return "ANTARA CAIR-GAS"`
	`def air_1(t):`
	`  if t < 0:`
	`  return "PADAT"`
	`  elif t > 0 and t <= 100:`
	`  return "CAIR"`
	`  elif t > 100:`
	`  return "GAS"`
	`  else:`
	`  return "ANTARA PADAT-CAIR"`

Pada hasilnya, semua kasus uji acak yang dinyatakan salah hanya menunjukkan ke jalur yang sama, lebih tepatnya dari 20 kasus uji, 6 kali eksekusi menuju jalur masukan kurang dari 0. Sementara pada penilaian dengan semantik kasus uji salah mencakup semua kemungkinan kesalahan, yakni masukan kurang dari 0 atau lebih dari 100, dengan 2 kali eksekusi menuju masukan kurang dari 0 dan 1 kali eksekusi menuju masukan lebih dari 100. Jumlah dilewatinya jalur secara lengkap ditunjukkan pada Tabel IV.4.13. Pada kasus no_of_triangle dan segiempat terdapat beberapa kasus dimana kedua sistem menghasilkan kelengkapan yang sama. Namun, dengan jumlah kasus yang lebih banyak pada penilaian dengan kasus uji acak nilainya menjadi lebih rendah dibanding penilaian dengan semantik.

Dengan melihat hasil pengujian, dapat dikatakan bahwa ada *edge cases* yang tidak ditangani sistem penilaian dengan pendekatan pembangkitan kasus uji secara acak namun ditangani oleh sistem hasil implementasi tugas akhir. Dosen mata kuliah IF1210 Dasar Pemrograman, Bapak Satrio Adi Rukmono S.T., M.T., selaku ahli juga menuliskan beberapa contoh implementasi referensi untuk persoalan max_3. Tabel hasil penilaian dengan semantik dapat dilihat pada lampiran B.1. dan dengan kasus uji acak pada lampiran B.2.

Tabel IV.4.13 Jumlah Dilewatinya Jalur student_grade

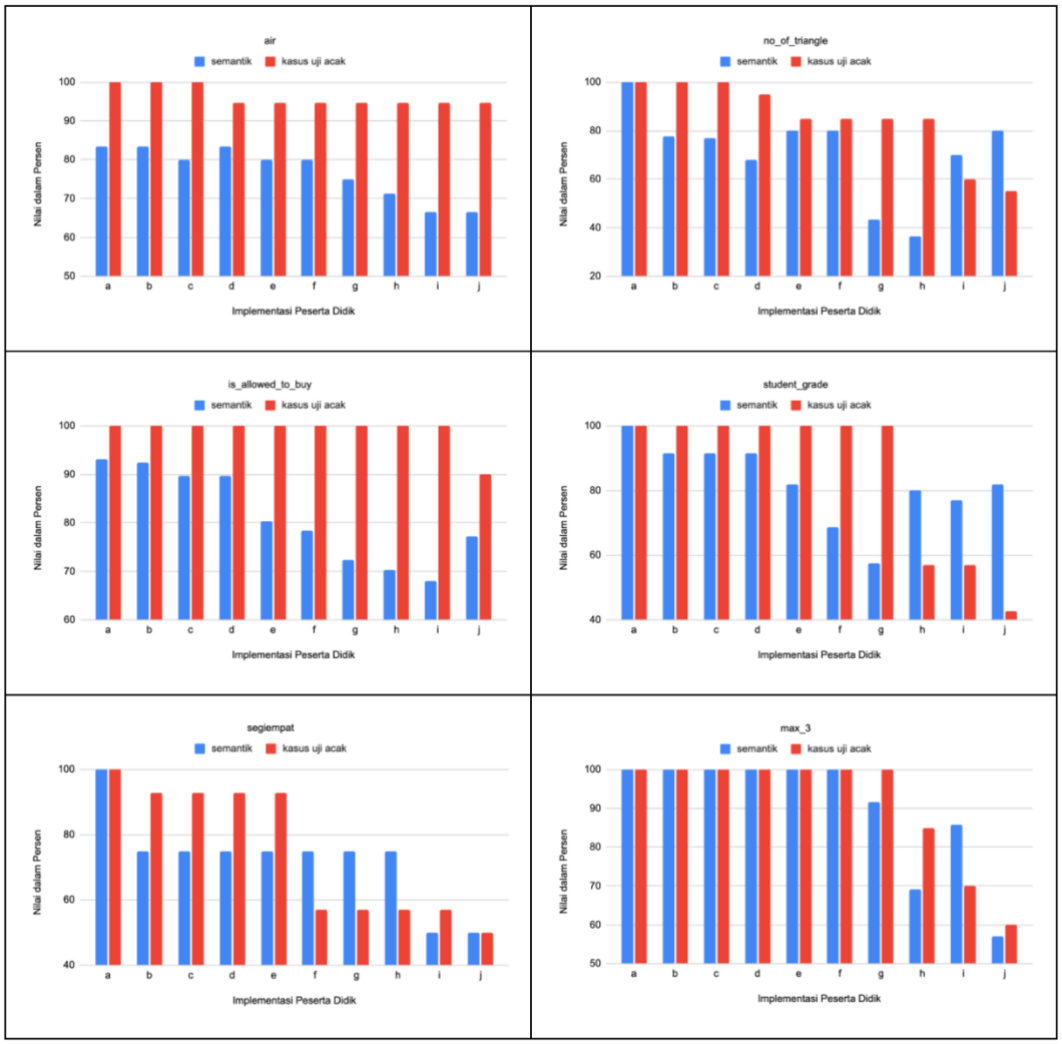
Jalur	**Sistem Penilaian Tugas Akhir**	**Sistem Penilaian Acak**
a < 0 (terjadi kesalahan pada implementasi siswa)	2 (15.3%)	6 (42.8%)
a > 100 (terjadi kesalahan pada implementasi siswa)	1 (7.6%)	0 (0%)
80 ≤ a ≤ 100	2 (15.3%)	1 (7.1%)
73 ≤ a ≤ 79	2 (15.3%)	0 (0%)
65 ≤ a ≤ 72	1 (7.6%)	1 (7.1%)
57 ≤ a ≤ 64	2 (15.3%)	4 (28.4%)
50 ≤ a ≤ 56	1 (7.6%)	1 (7.1%)
35 ≤ a ≤ 49	1 (7.6%)	1 (7.1%)
0 ≤ a ≤ 35	1 (7.6%)	0 (0%)

Tabel IV.4.14 Hasil Penilaian Persoalan Bagian 1

max_3		**segiempat**		**student_grade**	
Sistem Penilaian Tugas Akhir	Sistem Penilaian Acak	Sistem Penilaian Tugas Akhir	Sistem Penilaian Acak	Sistem Penilaian Tugas Akhir	Sistem Penilaian Acak
91.6%	100%	75%	92.8%	57.6%	100%
100%	100%	75%	92.8%	91.6%	100%
100%	100%	50%	50%	68.7%	100%
57.1%	60%	75%	57.1%	76.9%	57.1%
85.7%	70%	75%	57.1%	81.8%	42.8%
100%	100%	75%	57.1%	91.6%	100%
100%	100%	75%	92.8%	80%	57.1%
69.2%	85%	75%	92.8%	81.8%	100%
100%	100%	100%	100%	100%	100%
100%	100%	50%	57.1%	91.6%	100%

Tabel IV.4.15 Hasil Penilaian Persoalan Bagian 2

is_allowed_to_buy		**no_of_triangle**		**air**	
Sistem Penilaian Tugas Akhir	Sistem Penilaian Acak	Sistem Penilaian Tugas Akhir	Sistem Penilaian Acak	Sistem Penilaian Tugas Akhir	Sistem Penilaian Acak
93.1%	100%	80%	85%	83.3%	100%
92.5%	100%	70%	60%	80%	94.7%
89.6%	100%	43.4%	85%	75%	94.7%
68.1%	100%	36.3%	85%	83.3%	94.7%
77.2%	90%	76.9%	100%	83.3%	100%
80.3%	100%	67.8%	95%	71.4%	94.7%
70.3%	100%	80%	55%	80%	94.7%
78.5%	100%	100%	100%	80%	100%
89.6%	100%	77.7%	100%	66.6%	94.7%
72.4%	100%	80%	85%	66.6%	94.7%



Gambar IV.4.1 Hasil Pengujian dalam Persen

BAB V

KESIMPULAN DAN SARAN

V.1 Kesimpulan

Kesimpulan yang dapat ditarik dari implementasi dan pengujian pada tugas akhir adalah sebagai berikut:

1. Kebenaran implementasi program peserta didik berdasarkan implementasi program referensi dapat ditentukan dengan memperhatikan perbedaan semantik di antara keduanya dengan melakukan *concolic exploration* kemudian mencari keberadaan *path deviation* dan *path equivalence* pada setiap eksekusi.
2. Dengan memanfaatkan metode *concolic exploration*, *path deviation*, dan *path equivalence* sistem penilaian dengan semantik dapat mencakup *edge cases* dengan lebih lengkap dibandingkan sistem penilaian dengan pembangkitan kasus uji secara acak.

V.2 Saran

Saran untuk pengembangan selanjutnya mengenai topik pada tugas akhir adalah sebagai berikut:

1. Menambahkan tipe data lain untuk dapat ditangani oleh sistem penilaian.
2. Melakukan eksplorasi lebih lanjut untuk sistem dapat menerima implementasi tanpa *return statement*, pada praktiknya banyak persoalan pemrograman yang memerintahkan untuk mencetak hasil ke layar.

DAFTAR PUSTAKA

Ball, T., Daniel, J., & Irlbeck, M. (2015). Deconstructing dynamic symbolic execution. Dependable Software Systems Engineering, 40, 26.

Barrett, C., & Tinelli, C. (2018). Satisfiability modulo theories. *Handbook of Model Checking* (pp. 305-343). Springer, Cham.

Brumley, D., Caballero, J., Liang, Z., Newsome, J., & Song, D. (2007). Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *USENIX Security Symposium*, 15.

De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337-340). Springer, Berlin, Heidelberg.

Dijkstra, E. W. (1976). *A discipline of programming*. Englewood Cliffs: prentice-hall.

Euzenat, J., & Shvaiko, P. (2007). *Ontology matching* (Vol. 18). Heidelberg: Springer.

Ihantola, P., Ahoniemi, T., Karavirta, V. & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli Calling'10.*

Jackson, D., & Ladd, D. A. (1994). Semantic Diff: A Tool for Summarizing the Effects of Modifications. *ICSM (Vol. 94)*, 243-252.

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385-394.

Lahiri, S. K., Hawblitzel, C., Kawaguchi, M., & Rebêlo, H. (2012). Symdiff: A language-agnostic semantic diff tool for imperative programs. *International Conference on Computer Aided Verification*, 712-717.

Liu, H., & Kuan Tan, H. B. (2009). Covering code behavior on input validation in functional testing. *Information and Software Technology, 51(2)*, 546–553.

Liu, X., Wang, S., Wang, P., & Wu, D. (2019). Automatic Grading of programming assignments: an approach based on formal semantics. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 126-137.

Matt, U. V. (1994). Kassandra*:* The Automatic Grading System. *ACM SIGCUE Outlook*, *22(1)*, 26-40.

Ming, J., Pan, M., & Gao, D. (2012). iBinHunt: Binary hunting with interprocedural control flow. *International Conference on Information Security and Cryptology*, 92-109). Berlin, Heidelberg: Springer.

Naude, K.A., Greyling J.H., & Vogts, D. (2010). Marking Student Programs Using Graph Similarity. *Computer Education, 54(2),* 541-561.

Ramos, D. A., & Engler, D. R. (2011). Practical, low-effort equivalence verification of real code. *International Conference on Computer Aided Verification*, 669-685. Berlin, Heidelberg: Springer.

Sen, K., Marinov, D., & Agha, G. (2005). CUTE: a concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes*, 30(5), 263-272.

Sherman, M., Bassil, S., Lipman, D., Tuck, N., & Martin, F. (2013). Impact of Auto-Grading on an Introductory Computing Course. *J. Comput. Sci. Coll. 28, 6,* 69–75.

Son, J. (2017). *Symbolic Execution*. CS261: Security In Computer Systems, University of California Berkeley.

Vechev, M. (2020). *Concolic Execution*. Verimag Laboratory, Perancis.

Zhang, F., Wu, D., Liu, P., & Zhu, S. (2014). Program logic based software plagiarism detection. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 66-77. IEEE.

Lampiran A Contoh Hasil Penilaian

A.1 Penilaian arithmetic_seq_3 dengan arithmetic_seq_2 Menggunakan Sistem Penilaian dengan Semantik

Umpan Balik Penilaian arithmetic_seq_3 dengan arithmetic_seq_2 Menggunakan Sistem Penilaian dengan Semantik


```
{
    "reference": "arithmetic_seq_2",
    "grading": "arithmetic_seq_3",
    "grade": 100.0,
    "tested_case": {
        "(('n', 0),)": [1, 1],
        "(('n', 2),)": [3, 3],
        "(('n', 1),)": [1, 1],
        "(('n', 4),)": [0, 0],
        "(('n', 3),)": [6, 6],
        "(('n', 8),)": [6, 6],
        "(('n', 5),)": [5, 5],
        "(('n', 6),)": [1, 1],
        "(('n', 7),)": [8, 8],
        "(('n', 12),)": [8, 8]
    },
    "wrong_case": {}
}
```

A.2 Penilaian arithmetic_seq_4 dengan arithmetic_seq_2 Menggunakan Sistem Penilaian dengan Semantik

Umpan Balik Penilaian arithmetic_seq_4 dengan arithmetic_seq_2 Menggunakan Sistem Penilaian dengan Semantik


```
{
    "reference": "arithmetic_seq_2",
    "grading": "arithmetic_seq_4",
    "grade": 87.5,
    "tested_case": {
        "(('n', 0),)": [1, 1],
        "(('n', 2),)": [3, 3],
        "(('n', 1),)": [1, 1],
        "(('n', 4),)": [10, 10],
        "(('n', 3),)": [6, 6],
        "(('n', 8),)": [36, 28],
        "(('n', 5),)": [15, 15],
        "(('n', 6),)": [21, 21]
    },
    "wrong_case": {
        "(('n', 8),)": [36, 28]
    }
}
```

A.3 Penilaian arithmetic_seq dengan arithmetic_seq_2 Menggunakan Sistem Penilaian dengan Semantik

Umpan Balik Penilaian arithmetic_seq dengan arithmetic_seq_2 Menggunakan Sistem Penilaian dengan Semantik


```
{
    "reference": "arithmetic_seq_2",
    "grading": "arithmetic_seq",
    "grade": 100.0,
    "tested_case": {
        "(('n', 0),)": [1,1],
        "(('n', 1),)": [1,1],
        "(('n', 2),)": [3,3],
        "(('n', 8),)": [36,36],
        "(('n', 9),)": [45,45],
        "(('n', 3),)": [6,6],
        "(('n', 4),)": [10,10],
        "(('n', 5),)": [15,15],
        "(('n', 6),)": [21,21],
        "(('n', 7),)": [28,28],
        "(('n', 14),)": [105,105],
        "(('n', 15),)": [120,120],
        "(('n', 10),)": [55,55]
    },
    "wrong_case": {}
}
```

A.4 Penilaian air_1 dengan air Menggunakan Sistem Penilaian dengan Semantik

Umpan Balik Penilaian air_1 dengan air Menggunakan Sistem Penilaian dengan Semantik


```
{
    "reference": "air",
    "grading": "air_1",
    "grade": 83.33333333333334,
    "tested_case": {
        "(('t', 0),)": ["ANTARA PADAT-CAIR", "ANTARA PADAT-CAIR"],
        "(('t', -16),)": ["PADAT", "PADAT"],
        "(('t', 2),)": ["CAIR", "CAIR"],
        "(('t', 100),)": ["ANTARA CAIR-GAS", "CAIR"],
        "(('t', 65),)": ["CAIR", "CAIR"],
        "(('t', 102),)": ["GAS", "GAS"]
    },
    "wrong_case": {
        "(('t', 100),)": ["ANTARA CAIR-GAS", "CAIR"]
    }
}
```

A.5 Penilaian air_1 dengan air Menggunakan Sistem Penilaian dengan Pembangkitan Kasus Uji Secara Acak

Umpan Balik Penilaian air_1 dengan air Menggunakan Sistem Penilaian dengan Pembangkitan Kasus Uji Secara Acak


```
{
    "reference": "air",
    "grading": "air_1",
    "grade": 100.0,
    "tested_case": {
        "(('t', 88),)": ["CAIR", "CAIR"],
        "(('t', 97),)": ["CAIR", "CAIR"],
        "(('t', 0),)": ["ANTARA PADAT-CAIR", "ANTARA PADAT-CAIR"],
        "(('t', 56),)": ["CAIR", "CAIR"],
        "(('t', 120),)": ["GAS", "GAS"],
        "(('t', 114),)": ["GAS", "GAS"],
        "(('t', 93),)": ["CAIR", "CAIR"],
        "(('t', 67),)": ["CAIR", "CAIR"],
        "(('t', 112),)": ["GAS", "GAS"],
        "(('t', 81),)": ["CAIR", "CAIR"],
        "(('t', 45),)": ["CAIR", "CAIR"],
        "(('t', 119),)": ["GAS", "GAS"],
        "(('t', 25),)": ["CAIR", "CAIR"],
        "(('t', 62),)": ["CAIR", "CAIR"],
        "(('t', 14),)": ["CAIR", "CAIR"],
        "(('t', 54),)": ["CAIR", "CAIR"],
        "(('t', 27),)": ["CAIR", "CAIR"],
        "(('t', 69),)": ["CAIR", "CAIR"],
        "(('t', 15),)": ["CAIR", "CAIR"]
    },
    "wrong_case": {}
}
```

Lampiran B Penilaian max_3 dengan Implementasi Referensi Buatan Bapak Satrio Adi Rukmono S.T., M.T.

B.1 Tabel Hasil Penilaian max_3 dengan Implementasi Referensi Buatan Bapak Satrio Adi Rukmono S.T., M.T. Menggunakan Sistem Hasil Implementasi Tugas Akhir

Dinilai \ Penilai	max_3_v1	max_3_v2	max_3_v3	max_3_v4	max_3_v5
max_3_1	91.6%	91.6%	92.3%	92.3%	91.6%
max_3_2	100%	100%	100%	100%	100%
max_3_3	100%	100%	100%	100%	100%
max_3_4	57.1%	57.1%	60%	60%	60%
max_3_5	85.7%	85.7%	86.6%	86.6%	90%
max_3_6	100%	100%	100%	100%	100%
max_3_7	100%	100%	100%	100%	100%
max_3_8	69.2%	69.2%	81.8%	81.8%	63.6%
max_3_9	100%	100%	100%	100%	100%
max_3_10	100%	100%	100%	100%	100%

B.2 Tabel Hasil Penilaian max_3 dengan Implementasi Referensi Buatan Bapak Satrio Adi Rukmono S.T., M.T. Menggunakan Pembangkitan Masukan Secara Acak

Dinilai \ Penilai	max_3_v1	max_3_v2	max_3_v3	max_3_v4	max_3_v5
max_3_1	100%	100%	100%	100%	100%
max_3_2	100%	100%	100%	100%	100%
max_3_3	100%	100%	100%	100%	100%
max_3_4	60%	60%	60%	60%	60%
max_3_5	95%	95%	95%	95%	95%
max_3_6	100%	100%	100%	100%	100%
max_3_7	100%	100%	100%	100%	100%
max_3_8	100%	100%	100%	100%	100%
max_3_9	100%	100%	100%	100%	100%
max_3_10	100%	100%	100%	100%	100%