BAB II TINJAUAN PUSTAKA

Bab ini memaparkan mengenai proses studi literatur yang dilakukan sebagai salah satu tahap penelitian. Secara umum, bagian ini menyajikan pembahasan mengenai topik-topik terkait yang berhubungan dengan topik utama dari penelitian ini.

Pada bagian awal bab ini, disajikan hasil dari tinjauan pustaka mengenai proses pengujian dalam rekayasa perangkat lunak berikut klasifikasi dari masing-masing proses pengujian. Di bagian berikutnya adalah pembahasan mengenai verifikasi perangkat lunak, termasuk di dalamnya adalah beberapa teori yang digunakan dalam spesifikasi perangkat lunak secara formal. Di bagian berikutnya adalah pembahasan mengenai tinjauan pustaka mengenai T2 Framework, yaitu sebuah *verification tool* yang akan dieksplorasi lebih lanjut sebagai bagian dari ruang lingkup penelitian ini. Di bagian terakhir, disajikan tinjauan terhadap penelitian-penelitian lain dalam topik terkait *verification tools*.

2.1. Pengujian dalam Rekayasa Perangkat Lunak

Pengujian (*testing*) perangkat lunak adalah bagian dari suatu rangkaian proses rekayasa perangkat lunak. Perangkat lunak kemudian memasuki tahap pengujian terhadap aplikasi yang dibangun selama masa pengembangan. Proses pengujian dilakukan dengan tujuan untuk mencari dan memperbaiki kesalahan yang terjadi. Termasuk dalam pendekatan yang dilakukan pada bagian ini adalah verifikasi perangkat lunak yang akan dijelaskan secara lebih detail pada bagian lain laporan ini.

Secara umum, pengujian perangkat lunak dapat diklasifikasikan berdasarkan teknik pengujian dan strategi pengujian. Konsep ini dikemukakan dalam **[6]**, dengan klasifikasi pertama didasarkan pada teknik pengujian perangkat lunak (*software testing techniques*). Di sisi lain, klasifikasi kedua didasarkan pada strategi pengujian perangkat lunak (*software testing strategies*).

2.1.1. *Software Testing Techniques*

Berdasarkan teknik pengujian, proses pengujian perangkat lunak dapat diklasifikasikan menjadi dua golongan, yaitu *white box testing* dan *black box* testing **[6]**. Secara khusus, penggolongan tersebut didefinisikan sebagai berikut.

- ***White Box Testing***

  White box testing adalah sebuah teknik pengujian yang memfokuskan terhadap kemungkinan-kemungkinan dari jalannya perangkat lunak yang diujikan. Hal ini meliputi penelusuran terhadap *loop*, perbandingan *logical statement*, dan kemungkinan-kemungkinan lain yang mungkin timbul dari jalannya sebuah perangkat lunak. Secara umum, dengan *white box testing* dapat dilakukan (1) pengujian terhadap seluruh kemungkinan jalannya aplikasi, (2) pengujian terhadap seluruh percabangan aplikasi, (3) pengujian terhadap *loop* berdasarkan kemungkinan data yang ada, dan (4) pengujian kebenaran dari struktur data dan data dalam aplikasi.

  Termasuk dalam *white box testing* adalah *Basis Path Testing* (dengan penggunaan *graph flow notation* dan kalkulasi *cyclomatic complexity*) dan *Control Structure Testing* (dengan pengecekan terhadap *conditional statement*, *data flow*, dan *loop*). Masing-masing dari jenis pengujian memetakan jalannya aplikasi sampai tingkat *code*, dan dengan demikian dapat diketahui dengan persis mengenai jalannya aplikasi dan kesalahan di dalamnya, bila ada.

  Hal yang perlu diperhatikan dari *white box testing* adalah bahwa umumnya pengujian jenis ini membutuhkan akses ke *sourcecode*, dan dengan demikian memungkinkan dilakukannya penjelajahan (*traversing*) terhadap kemungkinan-kemungkinan jalannya aplikasi.

- ***Black Box Testing***

  Black box testing adalah pengujian yang memfokuskan kepada perilaku perangkat lunak berdasarkan kebutuhan fungsional (*functional requirement*) dari perangkat lunak yang bersangkutan. Dalam hal ini, *testcase* yang dipergunakan adalah

testcase yang merepresentasikan kebutuhan fungsional perangkat lunak, terlepas dari jalannya perangkat lunak yang diujikan.

Termasuk dalam aplikasi *black box testing* adalah dengan *graph-based testing* (dengan merepresentasikan objek/modul dan keterhubungannya dalam *graph*) serta *comparison testing* (dengan pengujian dua versi aplikasi secara terpisah untuk kemudian diuji kesesuaian perilaku antara keduanya).

Perlu diperhatikan bahwa penggunaan *black box testing* umumnya hanya sedikit membutuhkan akses ke *sourcecode*. Hal ini sesuai dengan karakteristiknya yang tidak memperhatikan bagaimana jalannya suatu aplikasi, namun lebih memperhatikan apakah kebutuhan-kebutuhan fungsional yang dinyatakan untuk perangkat lunak tersebut telah terpenuhi.

2.1.2. *Software Testing Strategies*

Hal serupa dilakukan untuk klasifikasi berdasarkan strategi pengujian perangkat lunak, yang diklasifikasikan sebagai berikut:

- **Unit Testing**

  Unit testing adalah pengujian yang memfokuskan pengujian pada unit terkecil dari sebuah aplikasi perangkat lunak – komponen, modul, atau unit berupa *subroutine* atau *method*. Sehubungan dengan sifat unit yang umumnya tidak cukup besar, proses verifikasi terhadap unit cenderung bersifat ke arah *white box testing,* dan dengan demikian pembuktian kebenaran untuk setiap unit dapat dilakukan dengan lebih terarah.

  Umumnya *unit testing* dilakukan dengan mengisolasi suatu unit untuk diuji secara independen, dengan demikian pada tahap ini pengujian terhadap unit yang berbeda dapat dilakukan secara paralel.

- ***Integration Testing***

  Proses *integration testing* merupakan pengujian terhadap suatu sistem secara keseluruhan, dengan asumsi bahwa masing-masing unit di bawahnya telah diverifikasi dan bekerja dengan baik. Pada tahap ini, kebenaran dari perangkat lunak hasil penggabungan masing-masing unit harus kembali diuji dalam *integration testing*.

  Termasuk dalam *integration testing* adalah penggunaan teknik berupa *regression testing* dan *smoke testing*. *Regression testing* pada dasarnya adalah rangkaian pengujian dengan *testcase* yang sama, dieksekusi setiap kali satu tahap dalam proses integrasi dilakukan (misalnya, setiap penambahan satu komponen). Penggunaan *regression testing* dilakukan untuk menjamin bahwa satu tahap dalam proses integrasi tidak memberikan efek samping pada perangkat lunak secara keseluruhan.

  Di sisi lain, *smoke testing* dilakukan dengan pendekatan yang berkebalikan dengan *regression testing*. Pada *smoke testing*, pengujian dilakukan dengan mengekspos kemungkinan kesalahan pada calon aplikasi yang diintegrasikan. Proses ini diulang berkali-kali, dengan pemikiran bahwa proses *smoke testing* yang intensif akan menghasilkan versi aplikasi yang lebih stabil.

- ***Validation Testing***

  Validation testing adalah tahap akhir dari pengujian perangkat lunak. *Validation testing* dilakukan untuk menjamin kesesuaian perangkat lunak dengan desain yang telah ditetapkan sebelumnya. Pada tahap ini perangkat lunak telah melalui tahap pengujian yang telah didiskusikan sebelumnya, untuk kemudian menjalani validasi akhir terhadap jalannya perangkat lunak.

  Termasuk dalam tahap ini adalah *alpha testing* dan *beta testing*, yaitu pengujian akhir sebelum perangkat lunak dinyatakan selesai dikembangkan. Pada tahap ini pula dilakukan modifikasi pada aplikasi berdasarkan masukan yang diterima pada masa *alpha testing* dan *beta testing*.

2.2. Verifikasi Perangkat Lunak

Bagian ini menjelaskan mengenai definisi umum dari verifikasi terhadap perangkat lunak dan tinjauan pustaka dalam topik tersebut. Secara khusus, pembahasan pada bagian ini meliputi pendekatan formal dengan logika Hoare yang menjadi dasar proses verifikasi oleh T2 Framework.

2.2.1. Definisi Verifikasi Perangkat Lunak

Verifikasi perangkat lunak adalah suatu proses pengujian terhadap perangkat lunak dengan pendekatan formal matematis, yang dilakukan terhadap perangkat lunak itu sendiri dan spesifikasi dari perangkat lunak yang bersesuaian **[3]**. Dalam perspektif ini, sebuah perangkat lunak adalah sebuah objek dari persamaan formal matematis, dengan masing-masing objek memiliki *properties* yang dapat diuji dengan pembuktian matematis.

Dengan pembuktian matematis, dapat dibuktikan bahwa perangkat lunak yang dikembangkan tidak memiliki kesalahan berdasarkan spesifikasi yang diberikan. Hal ini juga meliputi pembuktian bahwa perangkat lunak tersebut memiliki semua properti yang diharapkan, sehingga dapat dinyatakan sebagai aplikasi yang valid.

2.2.2. Verifikasi dengan Prinsip Logika Hoare

Dalam penelitian ini, verifikasi yang dilakukan meliputi penggunaan Logika Hoare sebagai landasan teori pengujian terhadap perangkat lunak. Logika Hoare adalah sebuah sistem formal (*formal system*) yang dikembangkan oleh C.A.R. Hoare **[4]** untuk pembuktian kebenaran suatu perangkat lunak dengan prinsip logika matematis. Bagian ini mendiskusikan beberapa teori dasar dalam Logika Hoare yang digunakan sebagai landasan dari teknik verifikasi dalam penelitian ini.

Notasi Dasar

Logika Hoare didasarkan pada suatu bentuk dasar yang dikenal juga sebagai *Hoare triple,* yang dinyatakan sebagai berikut.

$$\{P\}S\{Q\}$$

Di mana *P* dan *Q* secara berturut-turut menyatakan *precondition* dan *postcondition* dari suatu eksekusi perintah *S. S* di sini dapat berupa program, modul, atau *method*. Sebuah *Hoare triple* dapat juga dipandang sebagai 'bila *P* dipenuhi, *S* akan dieksekusi dan kemudian *Q* akan dipenuhi'.

Dalam penulisan aturan-aturan Logika Hoare, digunakan notasi yang didasari *natural deduction* bahwa 'apabila diketahui P_1 maka P_2 terpenuhi', dinyatakan sebagai berikut.

$$\frac{P_1}{P_2}$$

Di mana masing-masing P_1 dan P_2 merupakan suatu ekspresi logika. Penggunaan *Hoare triple* dan notasi deduksi tersebut merupakan dasar dari pembentukan aturan-aturan Hoare yang digunakan dalam proses verifikasi perangkat lunak.

Empty Statement Axiom

Salah satu ekspresi dalam Logika Hoare adalah pernyataan *empty statement*, atau *skip*. Pada kasus ini tidak dilakukan eksekusi perintah, dan dengan demikian tidak terjadi perubahan dari kondisi awal.

$$\{P\}skip\{P\}$$

Dalam sebuah *empty statement,* suatu *precondition P* juga berlaku sebagai *postcondition.* Perlu diperhatikan bahwa hal ini adalah karakteristik dari perangkat lunak; keadaan dalam aplikasi tidak akan berubah apabila tidak terdapat aksi yang

dilakukan (di sini dimodelkan dengan *skip*), dan dengan demikian tidak terjadi perubahan antara *precondition* dan *postcondition*.

Assignment Axiom

Dalam Logika Hoare, sebuah operasi *assignment* dinyatakan dalam notasi sebagai berikut.

$$\overline{\{P[x/E]\}x := E\{P\}}$$

Dengan `{P[x/E]}` menyatakan sebuah ekspresi matematis *P* di mana seluruh variabel *x* dalam *P* digantikan dengan sebuah ekspresi matematis *E*.

Sebagai contoh, ekspresi di bawah adalah sebuah contoh *assignment schema* yang valid dalam Logika Hoare.

$$\{y = 5\}y := y + 3\{y = 8\}$$

Dari contoh tersebut, dapat diperhatikan *precondition* dan *postcondition* dari sebuah operasi *assignment* sederhana. Misalkan pada keadaan awal diketahui variabel *y* bernilai 5, maka dengan operasi *assignment* seperti pada contoh, keadaan akhir yang terjadi adalah variabel *y* memiliki nilai 8.

Sebuah operasi *assignment* dapat juga dinyatakan dengan deduksi sebagai berikut.

$$\frac{P \Rightarrow Q[x/E]}{\{P\}x := E\{Q\}}$$

Dengan implikasi 'jika `P` maka `Q[x/E]`' di mana `Q[x/E]` menyatakan sebuah ekspresi matematis, maka dapat dideduksi sebuah *Hoare triple* `{P}x:=E{Q}`. Penggunaan notasi tersebut untuk operasi *assignment* sederhana dinyatakan sebagai berikut.

$$\frac{(y = 5) \Rightarrow (y + 3 = 8)}{\{y = 5\} y := y + 3 \{y = 8\}}$$

Dapat diperhatikan bahwa contoh di atas merupakan deduksi untuk sebuah operasi *assignment* sederhana.

Consequence Rule

Consequence rule adalah aturan deduksi terkait *precondition* dan *postcondition*, khususnya untuk deduksi sebuah *Hoare triple*. *Consequence rule* berlaku apabila diketahui suatu keadaan (*precondition* atau *postcondition*) yang lebih lemah dari keadaan yang dimiliki oleh suatu *Hoare triple.*

Consequence rule untuk *precondition* berlaku sebagai berikut: apabila diberikan sebuah *Hoare triple* {P}S{Q} dan sebuah *precondition P'* di mana *P* lebih lemah daripada *P'*, maka {P'}S{Q} adalah valid.

$$\frac{P' \Rightarrow P, \{P\} S \{Q\}}{\{P'\} S \{Q\}}$$

Sebuah *precondition P* dinyatakan sebagai lebih lemah daripada *P'* apabila kondisi yang terpenuhi oleh *P* lebih banyak daripada kondisi yang terpenuhi oleh *P'*. Dengan demikian, untuk seluruh kondisi yang terpenuhi oleh *P'* dan *statement S* yang berhenti dengan benar (*halt*), maka dijamin bahwa kondisi akhir *Q* dipenuhi. Hubungan antara *P'* dan *P* dinyatakan dengan implikasi dalam proses deduksi.

Sementara itu, *consequence rule* untuk *postcondition* berlaku sebagai berikut: apabila diberikan sebuah *Hoare triple* {P}S{Q} dan sebuah *postcondition Q'* di mana *Q*‘ lebih lemah daripada *Q*, maka {P}S{Q'} adalah valid.

$$\frac{Q \Rightarrow Q', \{P\} S \{Q\}}{\{P\} S \{Q'\}}$$

Sebuah *precondition Q'* dinyatakan sebagai lebih lemah daripada *Q* apabila kondisi yang terpenuhi oleh *Q'* lebih banyak daripada kondisi yang terpenuhi oleh *Q*. Untuk seluruh kondisi yang terpenuhi oleh *P* dan *statement S* yang berhenti dengan benar (*halt*), secara definisi dijamin bahwa kondisi akhir *Q* dipenuhi – dan dengan demikian *Q'* dipenuhi. Hubungan antara *Q* dan *Q'* dinyatakan dengan implikasi.

Rule of Composition

Rule of composition dalam Logika Hoare berlaku untuk lebih dari satu *statement* program yang dieksekusi secara berurutan. Misalkan diketahui dua buah *statement* program yang dapat dipastikan akan dieksekusi secara berurutan, maka kedua *statement* tersebut dapat didefinisikan dalam notasi *composition* dalam Logika Hoare.

Berikut adalah notasi untuk *composition* pada Logika Hoare, dengan dua buah *statement* yaitu *S* dan *T*.

$$\frac{\{P\}S\{Q\},\{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

Notasi tersebut menyatakan bahwa apabila terdapat dua buah program *S* dan *T* dengan masing-masing *precondition* dan *postcondition* seperti dinyatakan di atas, maka keadaan tersebut dapat dinyatakan dengan *precondition* pertama dan *postcondition* terakhir. Perlu diperhatikan bahwa *postcondition* dari *S* haruslah setara dengan *precondition* dari *T* dalam deduksi dengan *rule of composition*. Notasi `S;T` menyatakan bahwa *S* dieksekusi tepat sebelum *T*.

Sebagai contoh, misalkan dua buah operasi *assignment* dalam Logika Hoare:

$$\{x+1=54\}y := x+1\{y=54\} \ , \ \{y=54\}z := y\{z=54\}$$

Memperhatikan bahwa *postcondition* pertama setara dengan *precondition* kedua, dapat dideduksi dengan *rule of composition* bahwa:

$$\{x+1=54\}y := x+1; z := y\{z=54\}$$

Secara umum, notasi tersebut dapat digeneralisasi untuk lebih dari satu operasi dalam logika Hoare, dengan syarat yang telah dijelaskan.

Weakest Precondition

Weakest precondition adalah pendekatan yang digunakan untuk menentukan sebuah *precondition* dari suatu rangkaian *statement* pada program. Diberikan suatu rangkaian *statement* dan sebuah *postcondition* sebagai keadaan akhir, dapat ditentukan sebuah *precondition* dengan notasi penurunan *weakest precondition*.

Notasi dari *weakest precondition* didefinisikan sebagai berikut.

$$wp(S, R)$$

Dengan *S* adalah rangkaian *statement* dari program dan *R* adalah *postcondition* dari program yang diberikan. Notasi ini dapat dipandang sebagai menyatakan predikat yang menjadi *precondition* dari program yang bersangkutan.

Berikut adalah contoh bentuk umum dari penggunaan *weakest precondition*.

$$wp(x := E, R) = R^x{}_E$$

Dengan *E* adalah sebuah ekspresi *assignment* untuk variabel x, dan *R* adalah sebuah *postcondition* dari program. Di sini, sebuah nilai $R^x{}_E$ dapat dipandang sebagai *postcondition* dengan seluruh variabel *x* digantikan oleh ekspresi *E*.

Misalkan diketahui sebuah program dalam notasi Logika Hoare sebagai berikut:

$$\{P\}x := x + 4\{x = 10\}$$

Dengan {*P*} adalah *precondition* yang akan ditentukan, {x = 10} adalah *postcondition* yang telah diketahui, dan x := x+5 adalah *statement* dari program. Menggunakan pendekatan *weakest precondition*, maka:

$$wp(x := x + 4, x = 10) = (x + 4 = 10) = (x = 6)$$

Dengan pendekatan *weakest precondition* di atas, diperoleh {*P*} adalah `{x=6}`, sehingga program tersebut dapat dinyatakan dalam notasi Logika Hoare sebagai berikut:

$$\{x = 6\}x := x + 4\{x = 10\}$$

Secara umum, pendekatan *weakest precondition* dapat digeneralisasi untuk lebih dari satu *statement* pada sebuah program, khususnya untuk pembuktian terkait *rule of composition*. Dalam konteks ini, penggunaan *weakest precondition* dapat dilakukan dengan deduksi sebagai berikut.

Misalkan sebuah *Hoare triple* `{P}S{Q}` dengan `wp(S, Q)` sebagai *weakest precondition*. Dari keadaan tersebut, dapat dilakukan deduksi dengan *consequence rule* sebagai berikut:

$$\frac{P \Rightarrow wp(S,Q)}{\{P\}S\{Q\}}$$

Secara informal, hal ini berarti bahwa apabila diberikan sebuah *precondition P* dan sebuah *weakest precondition* `wp(S,Q)`, maka apabila *P* dan `wp(S,Q)` memiliki implikasi $P \Rightarrow wp(S,Q)$ dapat dideduksi bahwa `{P}S{Q}` adalah *Hoare triple* yang valid. Dengan demikian, proses pengujian kebenaran dari sebuah *precondition* dapat dilakukan dengan pendekatan *weakest precondition*.

Dalam penggunaan terkait *rule of composition*, proses deduksi dengan *weakest precondition* dilakukan dengan membuktikan bahwa sebuah *precondition* pada awal program memiliki hubungan implikasi terhadap *weakest precondition* yang diperoleh.

Pembuktian Sederhana dengan Logika Hoare

Misalkan diketahui sebuah program sederhana *P* sebagai berikut. *P* memiliki dua buah variabel *x* dan *y* yang hendak ditukar nilainya, tanpa menggunakan variabel tambahan.

```
{x = A} /\ {y = B}
x := x+y;
y := x-y;
x := x-y;
{x = B} /\ {y = A}
```

Program tersebut dapat dibuktikan dengan menggunakan *assignment axiom* dan *rule of composition* yang telah dibicarakan sebelumnya. Pembuktian dimulai dari *statement* dari akhir program ke awal program.

Untuk proses pembuktian ini, dilakukan perhitungan *weakest precondition* dari *statement* dan *postcondition* yang diberikan. Selanjutnya, ditunjukkan bahwa *precondition* memiliki implikasi sebagai *weakest precondition* yang diperoleh.

Sebagai contoh, dengan *statement* `(x := x-y)` dan *postcondition* `(x = B /\ y = A)` diperoleh:

$$wp(x := x - y, x = B \wedge y = A) = (x - y = B \wedge y = A)$$

Dari *weakest precondition* tersebut, diperoleh deduksi sebagai berikut.

$$\frac{x - y = B \wedge y = A \Rightarrow x - y = B \wedge y = A}{\{x - y = B \wedge y = A\} x := x - y \{x = B \wedge y = A\}}$$

..…..… (1)

Dari keadaan yang sama pada (1), pembuktian dilanjutkan ke *statement* sebelumnya.

$$\frac{x - (x - y) = B \wedge x - y = A \Rightarrow x - (x - y) = B \wedge x - y = A}{\{x - (x - y) = B \wedge x - y = A\} y := x - y \{x - y = B \wedge y = A\}}$$

……..…… (2)

Pada tahap ini, diperoleh *Hoare triple* sebagai hasil deduksi pada (2), untuk *statement* (`y := x - y`). Di sini, dapat diperhatikan bahwa *postcondition* yang diperoleh

sebagai hasil deduksi $\{x-y=B \wedge y=A\}$ adalah setara dengan *weakest precondition* yang diperoleh pada (1).

Menggunakan *rule of composition*, (1) dan (2) dapat dideduksi sebagai sebuah *sequence*, sehingga:

$$\frac{(1),(2)}{\{x-(x-y)=B \wedge x-y=A\}y:=x-y;x:=x-y\{x=B \wedge y=A\}}$$

..... (a)

Dari keadaan pada (2), pembuktian dilanjutkan ke *statement* sebelumnya.

$$\frac{y=B \wedge x=A \Rightarrow (x+y)-((x+y)-y)=B \wedge (x+y)-y=A}{\{y=B \wedge x=A\}x:=x+y\{x-(x-y))=B \wedge x-y=A\}}$$

..... (3)

Pada (3), dapat diperhatikan bahwa *precondition* dari model program yang diberikan ($y=B \wedge x=A$) memiliki implikasi sebagai *weakest precondition* yang diperoleh. Pada tahap ini kemudian diperoleh sebuah *Hoare triple* yang dinyatakan sebagai hasil deduksi di atas. Kembali dapat diperhatikan, bahwa *postcondition* pada (3) setara dengan *weakest precondition* yang terdapat pada (a) – *weakest precondition* ini diperoleh pada (1).

Kemudian, (a) dan (3) dapat dideduksi sebagai sebuah *sequence* dengan menggunakan *rule of composition*, sehingga:

$$\frac{(1),(2),(3)}{\{y=B \wedge x=A\}x:=x+y;y:=x-y;x:=x-y\{x=B \wedge y=A\}}$$

Dengan demikian, telah dibuktikan bahwa program tersebut valid dalam fungsinya untuk melakukan penukaran nilai dua buah variabel.

2.3. T2 Framework

Bagian ini menyajikan pembahasan mengenai hasil studi literatur yang telah dilakukan terhadap T2 Framework. Pembahasan pada bagian ini meliputi kapabilitas dan karakteristik dari T2 Framework yang didefinisikan dalam beberapa publikasi terkait yang memperkenalkannya sebagai sebuah *verification tool*.

2.3.1. Definisi T2 Framework

T2 Framework (selanjutnya disebut sebagai T2) adalah sebuah *verification tool* yang dikembangkan untuk melakukan verifikasi terhadap perangkat lunak berorientasi objek, khususnya yang dibangun dengan Java. T2 dirancang sebagai *trace-based testing tool* [1] yang bekerja berdasarkan pemanggilan dan spesifikasi *method* berikut *invariant* dari aplikasi yang diverifikasi. Proses pengujian dilakukan dengan menggunakan *testcase* yang di-*generate* oleh T2 secara random.

T2 adalah sebuah *testing tool* yang melakukan pengujian berdasarkan satu *sequence* pemanggilan terhadap *method* yang didefinisikan dalam suatu *class*. Untuk setiap tahap pengujian, dilakukan pengecekan terhadap spesifikasi *method* dan *class invariant*. T2 kemudian melaporkan mengenai hasil pengujian, dengan laporan yang terdiri atas pelanggaran terhadap spesifikasi (*violation*) yang timbul selama proses pengujian.

Berdasarkan strategi pengujian, secara umum T2 dapat digolongkan sebagai sebuah *unit testing tool*, dengan kemampuan untuk melakukan pengecekan terhadap spesifikasi dari perangkat lunak yang diverifikasi. Termasuk dalam kemampuan T2 adalah pengecekan terhadap spesifikasi *Hoare triple*, *class invariant*, dan *temporal properties* dalam spesifikasi yang ditulis dalam Java pada *class* yang bersesuaian (*in-code specification*).

Dari segi teknik pengujian, T2 memiliki karakteristik yang lebih mendekati sebuah *black box testing tool*. T2 hanya membutuhkan akses ke *sourcecode* untuk menuliskan spesifikasi yang kemudian akan di-*compile* bersama aplikasi yang diujikan. Secara umum, hal ini memiliki implikasi bahwa T2 tidak dapat dikatakan sebagai sebuah *white box testing tool*. Meskipun demikian, perlu ditekankan bahwa akses terhadap *sourcecode* dilakukan sebatas untuk kebutuhan penulisan spesifikasi, tanpa memperhatikan segmen *code* yang lain dari aplikasi yang diujikan.

2.3.2. *In-code Specification*

T2 melakukan verifikasi terhadap aplikasi berdasarkan spesifikasi yang ditulis dalam Java, di dalam *class* yang akan diverifikasi. Spesifikasi dari aplikasi ini meliputi dua bagian utama, yaitu *class invariant* dan *method specifications*. masing-masing spesifikasi ini harus diterjemahkan dari notasi formal ke dalam Java, untuk kemudian diverifikasi oleh T2.

Penggunaan *in-code specification* memberikan kemudahan dalam melakukan *maintenance* terhadap spesifikasi dari aplikasi yang dibangun. Pada umumnya, *maintenance* dari spesifikasi perangkat lunak memiliki hambatan tersendiri karena spesifikasi yang didefinisikan dalam bahasa yang berbeda. Hal ini memberikan hambatan dalam proses sinkronisasi antara spesifikasi yang dirancang dan aplikasi yang dibangun, khususnya ketika terjadi perubahan terhadap aplikasi. Penerapan *in-code specification* secara umum tidak memiliki masalah dengan hal tersebut. Hal ini mengingat bahwa spesifikasi dan aplikasi yang dibangun berada pada *code* yang sama, dan proses verifikasi mengharuskan bahwa spesifikasi di-*compile* bersama *class* sebagai prasyarat.

Hal lain yang juga perlu diperhatikan adalah bahwa *in-code specification* memiliki sifat sebagai media spesifikasi yang formal (didefinisikan dalam Java), deklaratif (menyatakan keadaan yang harus dipenuhi), dan pada saat yang sama *powerful* (dengan eksploitasi terhadap *library* yang dimiliki oleh Java).

2.3.3. *Class Invariant*

Class invariant pada dasarnya adalah sebuah *invariant*, yaitu suatu predikat yang harus selalu terpenuhi dalam suatu *class.* Setiap *public method* (i.e. *method* yang bebas diakses dari objek tersebut) harus mempertahankan predikat dari *class invariant*, dan dengan demikian suatu *class invariant* dijamin untuk selalu terpenuhi pada setiap objek yang merupakan *instance* dari *class* yang bersesuaian.

Pada verifikasi dengan T2 Framework, sebuah *class invariant* didefinisikan dalam sebuah method dengan *return type* boolean dengan nama `classinv()`. *Method* ini didefinisikan di bawah *class* yang akan diverifikasi, dengan *return type yang* merepresentasikan hasil pengujian terhadap *class invariant* yang bersesuaian.

Sebagai contoh, berikut adalah sebuah *class invariant* yang didefinisikan untuk menjamin keterurutan elemen-elemen yang ada dalam sebuah *array*.

```
private boolean classinv() {
     for (int i=0; i<arr.size-1;i++) {
          if (arr[i].compareTo(arr[i+1]) > 0) return false;
      }
      return true;
}
```

Dalam *class invariant* tersebut, dilakukan pengujian keterurutan isi dari sebuah *array*. Pada *code* tersebut, masing-masing elemen di indeks ke-*i* harus memiliki nilai yang lebih kecil atau sama dengan elemen di indeks ke-(*i+1*). Dalam kasus ini, apabila terdapat sebuah elemen pada indeks ke-*i* dengan nilai yang lebih besar daripada elemen pada indeks ke-(*i+1*), *class invariant* akan mengembalikan nilai *false* – hal ini merepresentasikan terjadinya pelanggaran terhadap spesifikasi *class invariant*.

Class invariant yang didefinisikan dalam method `classinv()` akan diverifikasi oleh T2 pada saat pertama kali sebuah objek diinstansiasi dari kelas *C*, dan untuk selanjutnya dilakukan setiap kali terjadi pemanggilan *method* yang berada di bawah *C*.

2.3.4. *Method Specifications*

Spesifikasi *method* dalam T2 dilakukan dengan berdasarkan fasilitas *assertion* yang disediakan oleh Java. Spesifikasi *method* ini memiliki fungsi untuk melakukan pengecekan terhadap *precondition* dan *postcondition* dari eksekusi sebuah method dalam *class*. Secara teknis, spesifikasi *method* dalam T2 dapat dilaksanakan secara internal (dari dalam method yang bersesuaian) maupun melalui sebuah *method* yang menjadi perantara antara spesifikasi dengan *code* dari *method* yang bersesuaian.

Sebagai contoh, berikut adalah spesifikasi *method* yang dilakukan secara internal untuk fungsi penarikan uang sederhana.

```
public class Bank {

private int JUMLAH_SALDO_MINIMAL;

...
      //keseluruhan method yang dispesifikasikan
      public void withdraw(int rp) {
            //pre-condition:
            //jumlah nilai tabungan harus nonnegatif
            assert rp > 0 : "PRE"

            //...(aplikasi method)

            //post-condition:
            //jumlah saldo yang tersisa harus lebih besar daripada
            //jumlah saldo minimal yang ditetapkan bank
            assert saldo >= JUMLAH_SALDO_MINIMAL : "POST"

            return;
      }
}
```

Meskipun demikian, penggunaan teknik tersebut tidak disarankan mengingat implementasinya yang dituliskan langsung di dalam *code* yang tersedia. Untuk keperluan ini, T2 menyediakan teknik untuk mendefinisikan spesifikasi secara lebih bersih, yaitu dengan mendefinisikan method `m_spec()` dalam *class* yang bersesuaian. `m_spec()` merupakan sebuah method perantara yang digunakan untuk melakukan verifikasi terhadap suatu method *m*.

Penggunaan `m_spec()` dilakukan dengan pembuatan sebuah *method* dengan parameter dan *return type* yang sama, dengan nama method yang diberi *suffix* `_spec`.

Dalam *method* tersebut dilakukan pemanggilan ke *method* yang akan diverifikasi, untuk kemudian me-*return* hasil yang sama dengan *method* yang diverifikasi.

Berikut ini adalah spesifikasi *method* dengan contoh yang sama, menggunakan `m_spec()`.

```
public class Bank {

private int JUMLAH_SALDO_MINIMAL;

...
      //method khusus untuk penulisan spesifikasi
      public void withdraw_spec(int rp) {
            //pre-condition:
            //sesuai penjelasan sebelumnya
            assert rp > 0 : "PRE"

            //pemanggilan method yang dispesifikasikan
            withdraw(rp);

            //post-condition:
            //sesuai penjelasan sebelumnya
            assert saldo >= JUMLAH_SALDO_MINIMAL : "POST"

            return;
      }
}
```

Dengan penggunaan teknik ini, spesifikasi dari sebuah *method* dapat didefinisikan secara lebih rapi. Untuk definisi spesifikasi ini, informasi yang dibutuhkan hanya meliputi nama dari *method* yang akan dispesifikasikan serta parameter dan *return type* dari *method* yang bersangkutan. Dapat diperhatikan dari *code* tersebut bahwa penulisan spesifikasi dilakukan secara terpisah dengan *method* yang dispesifikasikan. Penulisan spesifikasi dengan teknik ini menghasilkan penulisan spesifikasi yang lebih bersih, tanpa akses langsung ke definisi *method* yang bersangkutan – hal ini menjadi faktor yang membedakan dengan penulisan spesifikasi pada contoh pertama.

Perlu diperhatikan bahwa penggunaan penanda `"PRE"` dan `"POST"` merupakan variabel yang diolah oleh T2 dalam melakukan pengujian, secara berturut-turut merepresentasikan *precondition* dan *postcondition* dari eksekusi sebuah *method*.

2.3.5. APPMODEL

Selain *class invariant* dan *Hoare triple* (dalam implementasi untuk verifikasi *precondition* dan *postcondition*), T2 memiliki kemampuan menangani verifikasi terhadap asumsi-asumsi mengenai jalannya aplikasi dengan menerapkan pendekatan berupa definisi *application model*. Sebagai contoh, misalkan diketahui bahwa aplikasi yang menggunakan suatu *class* tertentu memiliki pola atau kemungkinan yang spesifik, yaitu bahwa nilai variabel tertentu selalu lebih besar daripada yang lain. Hal ini merupakan suatu asumsi akan jalannya sebuah aplikasi, dan dinyatakan sebagai *application model*.

Berikut ini adalah contoh penggunaan *application model* untuk verifikasi oleh T2, dengan asumsi bahwa variabel *x* selalu lebih besar atau sama dengan *y*. Dapat diperhatikan bahwa *application model* didefinisikan terpisah dari *class invariant* utama pada contoh di bawah.

```
private boolean classinv() {
    assert x >= y : "APPMODEL"
    return 25*x + 3*y <= 200;
}
```

Sehubungan dengan sifatnya yang harus selalu dijamin kebenarannya, *application model* didefinisikan dalam *method* `classinv()` berupa *assertion* dengan penanda "APPMODEL". Perlu juga diperhatikan bahwa penamaan dengan "APPMODEL" merupakan variabel yang diolah oleh T2 dalam melakukan pengujian, dan dengan demikian *assertion* dengan penanda "APPMODEL" akan dikenali oleh T2 sebagai sebuah *application model.*

2.3.6. *Temporal Properties*

T2 sebagai sebuah *verification tool* memiliki kemampuan untuk melakukan pengujian terhadap *temporal properties*, yaitu *properties* dalam aplikasi yang berubah terhadap waktu ketika aplikasi dijalankan. Dalam pembahasan ini, sebuah *property* didefinisikan sebagai suatu predikat yang memiliki nilai kebenaran tertentu seiring jalannya aplikasi. Perlu diperhatikan bahwa definisi *property* dalam pembahasan ini

tidak berkaitan dengan definisi *property* sebagai anggota sebuah *class* dalam paradigma *object-oriented programming*.

T2 tidak memiliki kemampuan untuk melakukan pengujian terhadap *properties* untuk aplikasi yang bersifat *concurrent*, dan dengan demikian *temporal properties* di sini dibatasi sebagai *properties* yang dapat diamati dalam satu *sequence* jalannya aplikasi (atau dalam T2, satu *sequence* pemanggilan method dari suatu *class*). Perlu diperhatikan bahwa *temporal properties* yang dapat diobservasi dalam T2 adalah *properties* yang diamati sebelum atau setelah satu pemanggilan *method*. Dengan demikian, T2 tidak dapat mengamati *properties* yang berubah dalam pemanggilan *method*.

Untuk melakukan verifikasi terhadap *temporal properties* dengan T2, diperlukan pendekatan dengan merepresentasikan perubahan *state* yang ada pada aplikasi. *State* dari aplikasi di sini didefinisikan sebagai kombinasi dari nilai-nilai variabel yang berada dalam aplikasi. Perlu diperhatikan bahwa definisi ini tidak berangkat dari definisi *state* dalam topik terkait automata, walaupun dalam penggunaannya teknik ini digunakan untuk mengemulasi pendekatan tersebut.

Pendekatan ini dilakukan dengan tujuan untuk merepresentasikan *state* yang terdapat dalam aplikasi sebagai *automaton* dalam Java. Proses ini dapat dipandang sebagai penerjemahan suatu *temporal formula* dalam bentuk suatu program yang bekerja dalam Java, untuk keperluan verifikasi dengan T2.

Berikut adalah contoh verifikasi *temporal properties* dalam Java. Diketahui sebuah aplikasi pemungutan suara sederhana dalam sebuah *class* bernama `SimpleVotesManager`. **[2]**

```
public class SimpleVotesManager {
     private int numberOfBallots ;
     private int yes = 0 ;
     private int no = 0 ;
     private boolean open = true ;

     public SimpleVotesManager(int ballots){
           numberOfBallots=ballots ; }

     public void voteYes() {
           if (open && numberOfBallots>0) {
           numberOfBallots-- ;
```

```
            yes++ ;
            } ;
      }

      public void voteYes(int n) {
            if (numberOfBallots>n) {
            numberOfBallots = numberOfBallots - n ;
            yes = yes+n ;
            } ;
      }

      public void voteNo() {
            if (open && numberOfBallots>0) {
            numberOfBallots-- ;
            no++ ;
            } ;
      }

      public void close() { open = false ; }
}
```

Diketahui pula sebuah *temporal formula* sebagai berikut.

$$[](\neg open \wedge numberOfBallots = N) \rightarrow [](numberOfBallots = N)$$

Temporal formula tersebut adalah sebuah persamaan dalam *linear time logic*. Notasi dalam persamaan ini digunakan untuk merepresentasikan predikat yang berubah terhadap waktu, khususnya dalam konteks jalannya sebuah perangkat lunak yang diverifikasi.

Persamaan tersebut menyatakan bahwa apabila sesi pemungutan suara telah ditutup (`open = false`), maka nilai `numberOfBallots` tidak akan berubah untuk seterusnya. Dapat juga dikatakan, bahwa aplikasi yang valid tidak akan menerima suara yang masuk setelah waktu pemungutan suara selesai.

Temporal property ini kemudian direpresentasikan dalam Java, dan dinyatakan dalam *class invariant*. Dalam kasus ini, digunakan *auxiliary variables* untuk mendukung representasi *temporal property* tersebut. *Auxiliary variable* adalah variabel yang digunakan untuk keperluan verifikasi oleh T2, dengan konvensi penamaan berupa *prefix* `aux_`. Secara umum, peran *auxiliary variables* dapat dipandang sebagai sebuah variabel biasa, hanya saja dengan ketentuan bahwa variabel-variabel tersebut digunakan hanya untuk kebutuhan verifikasi.

Berikut adalah spesifikasi untuk pengujian *temporal property* tersebut pada *class invariant*. *Code* berikut dapat ditemukan pada **[2]**, ditampilkan dengan modifikasi seperlunya untuk laporan ini.

```
public class SimpleVotesManager {
... // definisi variabel dan method sebelumnya

// representasi state pada aplikasi:
private static final int NEVER_STATE_INIT = 0 ;
private static final int NEVER_STATE_1 = 1 ;
private static final int NEVER_STATE_ACCEPT = 2 ;

// auxiliary variables:
private int aux_state = NEVER_STATE_INIT ;
private int aux_numberOfBallots_frozen ;

  // representasi temporal properties, dalam class invariant:
  private boolean classinv() {

      if (aux_state == NEVER_STATE_INIT && open)
           return true ;

      if (aux_state == NEVER_STATE_INIT && !open) {
           aux_state = NEVER_STATE_1 ;
           aux_numberOfBallots_frozen = numberOfBallots ;
           return true ;
      } ;

      if (aux_state == NEVER_STATE_1 &&
      numberOfBallots != aux_numberOfBallots_frozen) {
           aux_state = NEVER_STATE_ACCEPT ;
           return false ;
      } ;

      if (aux_state == NEVER_STATE_1 &&
      numberOfBallots == aux_numberOfBallots_frozen) {
           return true ;
      }

      return false ;
  }
}
```

Pada *class invariant* tersebut, *state* dalam aplikasi dinyatakan dalam tiga buah *integer*:

1. NEVER_STATE_INIT – merepresentasikan keadaan pada saat sesi pemungutan suara sedang dibuka. Pada saat ini, pemilih bisa melakukan *vote.* (*method* terkait: `voteYes(), voteNo(), voteYes(int n)`)

2. NEVER_STATE_1 – merepresentasikan keadaan setelah sesi pemungutan suara ditutup. Perhatikan bahwa *update* nilai dari `aux_state` dari NEVER_STATE_INIT ke NEVER_STATE_1 hanya dilakukan dalam *class invariant*. (variabel terkait: `aux_state`, `numberOfBallots`, `numberOfBallots_frozen`)

3. NEVER_STATE_ACCEPT – merepresentasikan keadaan di mana terdapat perbedaan antara suara yang diterima pada saat sesi pemungutan suara ditutup dan setelahnya. Perhatikan bahwa hal ini menyebabkan *violation* terhadap *class invariant.* (variabel terkait: `numberOfBallots`, `numberOfBallots_frozen`)

Dengan penambahan *temporal property* tersebut pada *class invariant*, verifikasi dengan T2 akan menemukan *violation* terhadap *class invariant*. Hal ini terjadi ketika *method* `voteYes(int n)` dipanggil setelah *method* `close()`.

Terjadinya *violation* di sini disebabkan oleh implementasi dari *method* `voteYes(int n)` yang tidak memperhatikan apakah sesi pemungutan suara masih terbuka atau sudah ditutup (secara teknis, tidak memperhatikan variabel *boolean* `open`) ketika melakukan penghitungan suara. Dengan demikian, nilai suara yang telah masuk akan terus bertambah setelah sesi pemungutan suara ditutup. Hal ini merupakan pelanggaran terhadap spesifikasi yang ditetapkan dalam *class invariant,* yaitu pada *conditional statement* dengan syarat bahwa `(aux_state == NEVER_STATE_1` && `numberOfBallots != aux_numberOfBallots_frozen)`.

2.4. Analisis Klasifikasi terhadap T2 Framework

T2 Framework adalah sebuah *testing tool*, dan dengan demikian dapat digolongkan berdasarkan klasifikasi yang telah didiskusikan sebelumnya. Analisis pada laporan ini didasarkan pada kapabilitas T2 yang juga telah didiskusikan pada bagian sebelumnya dari laporan ini.

Dari segi teknik pengujian, T2 tidak melakukan pengecekan menyeluruh terhadap jalannya aplikasi. Perlu diperhatikan bahwa dalam konteks ini T2 tetap melakukan eksekusi terhadap

seluruh *statement* yang berada di dalam *method* yang dipanggil, namun pengecekan yang dilakukan terbatas kepada spesifikasi *class* dan *method.* Di sini, T2 sama sekali tidak memperhatikan atau menganalisis jalannya aplikasi pada tingkat di bawah *method*. Dalam konteks ini yang dilakukan T2 adalah melakukan pengecekan apakah aplikasi memberikan hasil sesuai dengan spesifikasi yang ditetapkan, terkait nilai kebenaran dari masing-masing spesifikasi. T2 tidak memperhatikan mengenai detail dari jalannya aplikasi di luar hal tersebut, dan dengan demikian T2 tidak dapat diklasifikasikan sebagai sebuah *white box testing tool*.

Meskipun demikian, T2 adalah sebuah *testing tool* yang membutuhkan akses terhadap *sourcecode* dari aplikasi yang diujikan; kebutuhan ini timbul karena spesifikasi dari perangkat lunak harus dituliskan dalam *class* yang bersesuaian. Hal yang perlu diperhatikan adalah bahwa walaupun T2 membutuhkan akses ke *sourcecode*, T2 tidak benar-benar membutuhkan akses terhadap segmen *code* terkait logika aplikasi. T2 hanya membutuhkan *sourcecode* sebagai media untuk menyimpan dan menuliskan spesifikasi dari aplikasi yang diujikan – perlu ditekankan bahwa T2 melakukan analisis terhadap *bytecode* yang diperoleh dari hasil kompilasi suatu *class*, dan dengan demikian kebutuhan akan akses *sourcecode* terbatas hanya sebagai media penulisan spesifikasi.

Pada tahap ini dapat disimpulkan bahwa T2 lebih condong ke arah *black box testing tool* dalam klasifikasinya, walaupun dengan kebutuhan akan akses ke *sourcecode* selama pengujian. Meskipun demikian, keadaan bahwa *code* program sama sekali tidak diperhatikan dalam verifikasi mengakibatkan T2 diklasifikasikan sebagai sebuah *grey box testing tool*, dengan karakteristik yang sangat dekat ke sebuah *black box testing tool*.

Dari segi strategi pengujian, sebagaimana dinyatakan dalam **[1]**, T2 adalah sebuah *unit testing tool* untuk perangkat lunak yang dibangun dengan Java; T2 melakukan verifikasi terhadap spesifikasi dari *method* dan *class invariant* berdasarkan pemeriksaan terhadap serangkaian pemanggilan method dari aplikasi. Klasifikasi T2 sebagai sebuah *unit testing tool* memenuhi kriteria tersebut, dengan unit terkecil yang diobservasi adalah sebuah *method* dari aplikasi yang diujikan.

T2 memiliki konfigurasi untuk melakukan *regression test*, namun kapabilitas tersebut terbatas untuk pengujian dalam satu *class*. Hal ini mengakibatkan T2 tidak tepat untuk

diklasifikasikan sebagai sebuah *integration testing tool*. Mengingat kapabilitas T2 yang terbatas pada verifikasi dalam lingkup *class* pada Java, T2 juga tidak dapat digunakan untuk mendukung *validation testing* yang bekerja pada tingkatan di atas *integration testing*.

Dengan demikian, T2 Framework dapat diklasifikasikan sebagai sebuah *unit testing tool*, dengan kemampuan verifikasi terhadap *class* dan *method* dari aplikasi yang diujikan. Di sisi lain, T2 Framework cukup sesuai untuk diklasifikasikan sebagai sebuah *grey box testing tool*, dengan karakteristik yang lebih mendekati *black box testing tool*.

2.5. Penelitian Lain Terkait *Verification Tools*

Bagian ini menyajikan pembahasan mengenai beberapa penelitian dalam topik terkait *verification tools*. Pada bagian ini disertakan perbandingan terhadap beberapa aspek dari masing-masing *verification tool* sebagai bagian dari proses studi literatur dalam penelitian ini. Sebagaimana telah dinyatakan dalam ruang lingkup penelitian, perbandingan yang disajikan dalam pembahasan ini dilakukan dalam konteks gambaran umum dan tidak merepresentasikan perbandingan yang menyeluruh terhadap masing-masing *verification tool*.

2.5.1. Lingu

Lingu adalah sebuah bahasa yang menerapkan verifikasi dan validasi program dengan bantuan *theorem prover*. Lingu dirancang untuk bekerja pada domain yang spesifik, yaitu transaksi basis data [11].

Sebagai sebuah bahasa, Lingu adalah bahasa abstrak (*abstract language*) sekaligus bahasa tingkat tinggi (*high level language*). Dalam sebuah bahasa abstrak, kode yang dituliskan tidak dapat langsung berjalan pada suatu sistem; dalam konteks ini, kode dalam Lingu harus diterjemahkan terlebih dahulu dalam bahasa yang lebih konkret (misalnya Java atau C) sebelum dapat dijalankan pada suatu sistem.

Dalam penerapannya, Lingu digunakan untuk mendefinisikan spesifikasi dari perangkat lunak dalam sebuah *script*. Proses pembuktian terhadap kebenaran terhadap model program dilakukan dengan bantuan *theorem prover* untuk *Higher Order Logic*

(HOL). Setelah proses verifikasi berhasil, barulah dilakukan transformasi dari bahasa Lingu ke bahasa konkret – saat ini, penerjemahan bahasa Lingu baru dilakukan untuk Java.

Selain pembuktian kebenaran secara formal dengan bantuan HOL, pengujian juga dilakukan dengan pembuatan *generated testcases* untuk Lingu. Pada penelitian terkait pengembangan *test generator* untuk Lingu **[12]**, telah dilakukan pengujian dengan pendekatan yang lebih teknis dengan akses terhadap struktur basis data.

Proses Verifikasi dalam Lingu

Proses verifikasi untuk model aplikasi dalam Lingu dilakukan dalam dua pendekatan, yaitu secara matematis (verifikasi) dan teknis (validasi) **[10]**. Pendekatan matematis di sini dimaksudkan bahwa model aplikasi dalam Lingu dipecah-pecah ke dalam formula preposisi matematis, untuk kemudian dibuktikan dengan bantuan *theorem prover* HOL. Di sisi lain, pendekatan teknis dimaksudkan sebagai proses untuk menghasilkan kasus uji (*testcase*) terhadap studi kasus dalam basis data pada umumnya.

Setelah pembuktian berupa induksi dengan *theorem prover*, dilakukan sebuah proses validasi untuk model yang disiapkan. Proses validasi dilakukan dengan pembuatan *script* validasi dengan berbagai kombinasi masukan dan parameter yang merepresentasikan pemakaian nyata. Di sini, proses validasi dilakukan dengan dukungan sebuah modul *test generator* yang memiliki fungsi untuk meng-*generate* berbagai kemungkinan masukan dan parameter yang mungkin **[12]**.

Secara umum, kedua pendekatan tersebut didasarkan kepada paradigma yang berbeda; pendekatan matematis mengutamakan kepada pembuktian dengan induksi dari model aplikasi, sementara pendekatan teknis lebih menekankan kepada aspek informal dan pragmatis dari sebuah aplikasi basis data.

Perbandingan Umum dengan T2 Framework

Secara umum, proses verifikasi dalam Lingu dilakukan dengan pendekatan yang didasarkan kepada penerapan metode formal. Lingu memandang sebuah model aplikasi yang didefinisikan sebagai model matematika yang terdiri atas preposisi-preposisi yang kemudian dibuktikan kebenarannya dengan *theorem prover*.

Hal ini berbeda dengan T2 Framework yang menawarkan pendekatan yang lebih pragmatis – dengan spesifikasi yang didasarkan kepada pendekatan formal. T2 tidak memfasilitasi pembuktian kebenaran terhadap model aplikasi sebagai teorema-teorema; dalam hal ini, T2 hanya melakukan pengecekan terhadap kondisi dari spesifikasi formal yang ditetapkan. Dibandingkan dengan proses verifikasi pada Lingu, proses pembuktian kebenaran dalam T2 dilakukan secara lebih informal, walaupun dengan spesifikasi yang tetap bersandar kepada kaidah-kaidah metode formal.

Hal yang perlu diperhatikan adalah perbedaan domain antara Lingu dan T2 dalam proses verifikasi. Lingu adalah sebuah bahasa yang bersifat abstrak, dan proses verifikasi dilakukan dengan tidak memperhatikan implementasi dari perangkat lunak. Di sisi lain, T2 adalah sebuah *verification tool* yang terkait erat dengan hasil implementasi perangkat lunak, dan verifikasi dilakukan dalam keadaan perangkat lunak sudah selesai dibangun. Perbedaan ini memberikan sudut pandang bahwa pembuktian dan formalisasi dalam Lingu masih harus ditelaah lebih lanjut dalam bentuk aplikasi yang sudah jadi, sementara pengujian dengan T2 masih harus mempertimbangkan kesesuaian penerjemahan formalisasi dalam bahasa pemrograman yang menjadi media spesifikasinya – dalam konteks ini, Java.

2.5.2. Java PathFinder

Java PathFinder (JPF) adalah sebuah *verification tool* yang dikembangkan untuk melakukan verifikasi terhadap program berbasis Java dalam bentuk *bytecode* **[13]**. Dalam konteks verifikasi, JPF melakukan penjelajahan terhadap semua kemungkinan *path* yang dapat dieksekusi oleh perangkat lunak. JPF kemudian akan melaporkan

pelanggaran terhadap spesifikasi yang ditemukan, termasuk dalam hal ini adalah *runtime error* dan *unhandled exception* yang mungkin terdapat pada perangkat lunak.

JPF dikembangkan sebagai sebuah *model checker* **[14]** yang memetakan seluruh kemungkinan *path* dalam perangkat lunak. Seluruh kemungkinan *path* ini dipetakan ke dalam sebuah model yang dapat dipandang sebagai sebuah *tree*; masing-masing percabangan kemungkinan dalam model inilah yang akan diselidiki dalam proses verifikasi oleh JPF.

Dalam proses verifikasi, pendekatan yang dilakukan oleh JPF analog dengan pendekatan *depth-first search* terhadap sebuah *tree*. Pendekatan yang diadaptasi oleh JPF memiliki mekanisme *backtracking* untuk melakukan pengecekan terhadap *path* yang belum dieksplorasi; dengan demikian, secara teoretis JPF dapat melakukan verifikasi terhadap sembarang perangkat lunak berbasis Java apabila diberikan waktu dan alokasi memori yang memadai.

Pengujian oleh Java PathFinder

Secara umum, pendekatan yang dilakukan oleh JPF meliputi penelusuran terhadap seluruh kemungkinan yang mungkin dari jalannya sebuah perangkat lunak – dimulai dari *method* `main` sebagai titik awal pengujian. Pengujian kemudian dilakukan dengan teknik *depth-first search* dengan strategi dan *heuristic* yang dapat dikonfigurasi oleh pengguna. Secara teknis, pendekatan ini membutuhkan alokasi memori yang sangat besar, karena dalam pendekatan ini diperlukan kemampuan penyimpanan *state* dari program. Di sisi lain, pendekatan ini juga cenderung memakan waktu verifikasi yang lebih panjang – tergantung besarnya kemungkinan-kemungkinan yang dapat muncul pada sebuah program yang diujikan, masing-masing kemungkinan ini harus ditelusuri satu per satu oleh JPF.

Perlu diperhatikan bahwa pendekatan yang dilakukan oleh JPF masih harus memperhatikan kemungkinan terjadinya *state explosion*, yaitu keadaan di mana *state* yang harus disimpan oleh JPF terus bertambah dan mengakibatkan terjadinya penggunaan memori yang berlebihan. Hal ini diantisipasi dengan teknik pemetaan

state dengan pendekatan berupa pencocokan pola (*pattern matching*) terhadap analisis bentuk struktur data dalam eksekusi program oleh JPF. Penggunaan teknik ini, dipadukan dengan konfigurasi strategi dan *heuristic* yang tepat, memberikan hasil yang signifikan dalam pengurangan jumlah state yang harus dipetakan **[14]**.

Dengan karakteristiknya sebagai *model checker*, JPF tidak secara khusus melakukan pengujian program terhadap spesifikasi formal yang diberikan. Pengujian yang dilakukan oleh JPF hanya meliputi apakah terdapat *violation* (dalam hal ini, *runtime error* dan *exception*) dalam sembarang *path* yang mungkin. Spesifikasi dapat dimasukkan untuk pengujian dengan JPF, namun untuk keperluan ini spesifikasi harus dituliskan dalam bentuk *assertion* pada Java pada lokasi yang bersesuaian pada *sourcecode*.

Perbandingan Umum dengan T2 Framework

Dalam eksperimen awal (*preliminary experiment*) untuk analisis terhadap Java PathFinder dan T2 Framework **[15]**, telah diperoleh gambaran umum mengenai karakteristik dan kapabilitas dari masing-masing sebagai *verification tool*. Secara umum, penggunaan masing-masing memiliki karakteristik yang berbeda, mengingat keduanya dikembangkan dengan berangkat dari sudut pandang yang juga berbeda.

JPF adalah sebuah *model-checker*, dan dengan demikian proses verifikasi yang dilakukan meliputi proses penelusuran terhadap jalannya program yang diujikan. Walaupun dalam penggunaannya JPF tampak mengakomodasi pendekatan yang lebih informal dan pragmatis, perlu diperhatikan bahwa JPF tetap memiliki karakteristik formal dalam pendekatannya; JPF sendiri dikembangkan sebagai sebuah *model checker* yang didasarkan kepada pendekatan dengan metode formal.

Di sisi lain, T2 menawarkan pendekatan yang lebih formal dan simbolik. T2 memiliki dukungan terhadap definisi spesifikasi yang lebih formal, dengan kemudahan tersendiri untuk penulisan dan pemeliharaan spesifikasi dari perangkat lunak – walaupun perlu diperhatikan bahwa spesifikasi formal untuk T2 masih harus diterjemahkan ke dalam Java sebagai media spesifikasinya.

Pada dasarnya, T2 dan JPF adalah *verification tool* yang dirancang untuk bekerja pada area yang berbeda; JPF menggunakan pendekatan yang lebih informal untuk penggunaannya, sementara T2 lebih mendukung untuk keperluan spesifikasi yang bersifat formal. Mempertimbangkan karakteristik yang dimiliki oleh JPF dan T2, penggunaan masing-masing lebih tepat untuk dikatakan sebagai komplemen dan tidak sebagai substitusi bagi yang lain.