**PEMBANGUNAN *DOMAIN SPESIFIC LANGUAGE* (DSL) UNTUK MENUNJANG VERIFIKASI FORMAL DAN PENGUJIAN PADA *BEHAVIOR-DRIVEN DEVELOPMENT***

**PROPOSAL TESIS**

**Disusun sebagai syarat kelulusan matakuliah**

**IF5099 Metodologi Penelitian/Tesis 1**

**Oleh**

**JESSIE ANDIKA SETIADY**

**NIM : 23516064**

**(Program Studi Magister Informatika)**

****

**Tanggal: 4 November 2017**

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**

**INSTITUT TEKNOLOGI BANDUNG**

**2017**

# 

# LEMBAR PENGESAHAN

**PEMBANGUNAN *DOMAIN SPESIFIC LANGUAGE* (DSL) UNTUK MENUNJANG VERIFIKASI FORMAL DAN PENGUJIAN PADA *BEHAVIOR-DRIVEN DEVELOPMENT***

**PROPOSAL TESIS**

**Oleh**

**Jessie Andika Setiady**

**23516064**

**Program Studi Magister Informatika**

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Telah disetujui sebagai Proposal Tesis

di Bandung, pada November 2017

|  |  |
| --- | --- |
| **Pembimbing I** | **Pembimbing II** |
| Dr. Ir. M. M. Inggriani Liem  NIP: 195301161979032001 | Yudistira Dwi Wardhana Asnar, S.T., Ph.D  NIP : 198008272015041002 |

# DAFTAR PERUBAHAN

| **Versi** | **Tanggal** | **Perubahan** |
| --- | --- | --- |
| 0.1 | 27-02-2017 | Versi inisial |
| 0.2 | 13-03-2017 | Penambahan / perbaikan:   1. Menambahkan latar belakang, tujuan, dan batasan masalah 2. Menambahkan ilustrasi diagram untuk metode yang yang diusulkan |
| 0.3 | 18-03-2017 | Penambahan / perbaikan:   1. Perbaikan notasi diagram pada metodologi penelitian 2. Perbaikan kalimat pada latar belakang dan metodologi penelitian 3. Menambahkan tabel pelaksanaan tesis |
| 0.4 | 31-07-2017 | Penambahan / perbaikan:   1. Menambahkan referensi pada bagian latar belakang 2. Menghilangkan kalimat bermakna subjektif pada latar belakang 3. Mengubah kalimat rumusan masalah 4. Memperjelas bagian metodologi penelitan 5. Menambahkan studi literatur 6. Menambahkan usulan penelitian, yang terdiri dari rancangan / gambaran umum penelitan, rancangan alur kerja penelitian, dan kebaruan serta orisinalitas |
| 0.5 | 12-09-2017 | Penambahan / perbaikan:   1. Perbaikan komposisi kalimat pada latar belakang 2. Memperbaiki metoologi penelitian 3. Menambahkan studi literatur |
| 0.6 | 18-09-2017 | Penambahan / perbaikan:   1. Format penulisan referensi menjadi format IEEE 2. Penambahan studi literatur mengenai metode-metode dari penelitian sebelumnya yang digunakan untuk verifikasi perangkat lunak 3. Perbaikan kalimat di bagian rumusan masalah 4. Perbaikan kalimat di bagian tujuan 5. Perbaikan metodologi penelitian 6. Perbaikan jadwal pengerjaan menjadi 6 bulan |
| 0.7 | 03-10-2017 | 1. Perbaikan latar belakang, 2. Perbaikan bagian rumusan masalah dan tujuan 3. Penambahan rincian alur metodologi penelitian 4. Perbaikan jadwal pengerjaan |
| 0.8 | 05-10-2017 | 1. Perbaikan formulasi rumusan masalah 2. Perubahan bagian metodologi penelitian menjadi studi literatur, analisis permasalahan, pengembangan kakas, eksperimen, dan penarikan kesimpulan 3. Penambahan *workflow* usulan penelitian pada lampiran |
| 0.9 | 04-11-2017 | 1. Perbaikan judul 2. Perbaikan bagian latar belakang: 3. Menambahkan referensi term verifikasi dan validasi dari IEEE (paragraf 2) 4. Perbaikan bagian rumusan masalah dan tujuan 5. Memperjelas bagian cakupan (pembangkitan, ekstraksi, BDD) dan batasan penelitian (kompleksitas perangkat lunak, *source code* yang diverifikasi/diuji, tahap perangkat lunak pengembangan/pemeliharaan) |

# DAFTAR ISI

[LEMBAR PENGESAHAN i](#_Toc494952739)

[DAFTAR PERUBAHAN ii](#_Toc494952740)

[DAFTAR ISI iv](#_Toc494952741)

[DAFTAR TABEL vi](#_Toc494952742)

[DAFTAR GAMBAR vii](#_Toc494952743)

[BAB I PENDAHULUAN 1](#_Toc494952744)

[I.1 Latar Belakang 1](#_Toc494952745)

[I.2 Rumusan Masalah 4](#_Toc494952746)

[I.3 Tujuan Penelitian 4](#_Toc494952747)

[I.4 Batasan Masalah 4](#_Toc494952748)

[I.5 Metodologi Penelitian 4](#_Toc494952749)

[I.6 Asumsi 7](#_Toc494952750)

[I.7 Hipotesis 7](#_Toc494952751)

[I.8 Kontribusi Penelitian 8](#_Toc494952753)

[I.9 Pelaksanaan Tesis 8](#_Toc494952754)

[BAB II STUDI LITERATUR 11](#_Toc494952755)

[II.1 Verifikasi Formal 11](#_Toc494952756)

[II.2 Verifikasi Perangkat Lunak 11](#_Toc494952757)

[II.3 Model Checking 15](#_Toc494952758)

[II.4 *Linear Temporal Logic* (LTL) 15](#_Toc494952759)

[II.4.1 Sintaks LTL 15](#_Toc494952760)

[II.4.2 Semantik LTL 16](#_Toc494952761)

[II.5 Test-Driven Development 16](#_Toc494952762)

[II.6 Behavior-driven Development (BDD) 18](#_Toc494952763)

[II.6.1 Perbedaan BDD dan TDD 19](#_Toc494952764)

[II.7 Kakas pada Behavior-driven Development 19](#_Toc494952765)

[II.7.1 Cucumber 19](#_Toc494952766)

[II.7.2 Gherkin 19](#_Toc494952767)

[DAFTAR PUSTAKA 22](#_Toc494952768)

[LAMPIRAN 1: *Workflow* Usulan Penelitian 24](#_Toc494952769)

# DAFTAR TABEL

**Tabel- 1** Pelaksanaan Tesis 8

# DAFTAR GAMBAR

Gambar- I.1 Tahap Utama Metodologi Penelitian 4

Gambar- I.2 Subtahap pada Pengembangan Kakas 5

Gambar- I.3 Aktivitas pada Tahap Eksperimen 7

Gambar- II.1 Contoh Berkas dengan Bahasa Gherkin 20

Gambar- II.2 Contoh Berkas *Feature* 21

Gambar- II.3 Contoh *Step Definition* pada Gherkin 21

Gambar-Lampiran 1.1 *Workflow* usulan penelitian 24

# PENDAHULUAN

Pada bagian ini dijabarkan hal-hal yang menjadi latar belakang penelitian, rumusan masalah dari persoalan yang akan diteliti, dan tujuan penelitian.

## Latar Belakang

Bagian yang tidak terpisahkan dari pengembangan perangkat lunak adalah proses mendeteksi, menemukan, dan mengkoreksi kesalahan. Pada survei yang dilakukan oleh Hailpern dan Santhanam (Hailpern & Santhanam, 2002), biaya untuk menjamin bahwa perangkat lunak akan memenuhi spesifikasi fungsional dan non-fungsional pada lingkungan *deployment* yang disepakati adalah antara 50 hingga 75% dari total biaya pengembangan. Terlepas dari usaha tersebut, *bugs* masih mungkin ditemukan oleh pengguna dari perangkat lunak, dan berpotensi menimbulkan kerugian yang lebih besar dari biaya yang diinvestasikan untuk pengembangan (Hailpern & Santhanam, 2002). Penyebab utama kegagalan perangkat lunak berdasarkan (Hailpern & Santhanam, 2002) yaitu: gagal memenuhi spesifikasi atau standar.

Mereduksi biaya pengembangan perangkat lunak, dan meningkatkan kualitas saat ini menjadi perhatian utama. Proses verifikasi dan validasi (V&V) perangkat lunak dilakukan untuk mencapai tujuan tersebut. V&V adalah proses untuk menentukan apakah kebutuhan untuk sistem atau komponen lengkap dan benar, produk yang dihasilkan oleh setiap fase pengembangan memenuhi kebutuhan atau kondisi yang ditetapkan, dan sistem atau komponen akhir sesuai dengan kebutuhan yang dispesifikasikan. (IEEE, 1990). Proses V&V meliputi banyak pendekatan dan kakas. Berdasarkan (Collofello, 1988) ada 5 pendekatan V&V yang saling melengkapi satu sama lain, yaitu: *technical review*, pengujian perangkat lunak, verifikasi program, simulasi dan *prototyping*, dan *requirement tracing*.

Perangkat lunak umumnya diverifikasi dan divalidasi dengan cara diuji. Pengujian perangkat lunak bertujuan untuk mencari kesalahan, namun belum mampu menjamin perangkat lunak bebas dari kesalahan (Dijkstra, n.d.) (Liu, 2016). Tantangan utama pada pengujian perangkat lunak adalah kompleksitas perangkat lunak. Cakupan pengujian yang terbatas sehingga tidak mampu mengeksplorasi seluruh *state* dan input yang mungkin pada perangkat lunak. Tantangan lainnya pada pengujian perangkat lunak adalah perubahan spesifikasi yang cepat mengakibatkan kasus uji dan skrip untuk pengujian otomatis harus diperbarui agar tetap selaras dengan spesifikasi. Pembaruan ini sering kali diabaikan, dan hanya dilakukan pada tahap akhir menjelang tenggat waktu (Wilcox, n.d.). Pada kasus ekstrim, pengujian bahkan sangat minim dilakukan karena keterbatasan waktu atau anggaran (Wilcox, n.d.).

Pada awal proyek perangkat lunak, dan pada setiap perubahan perangkat lunak, dilakukan analisis kebutuhan dan diskusi mengenai perilaku perangkat lunak yang akan dibangun. Diskusi dilakukan antara pemesan perangkat lunak dan tim pengembang. Pada pendekatan BDD, pemesan perangkat lunak menspesifikasikan perilaku yang diinginkan pada perangkat lunak yang dibangun, tim pengembang kemudian melakukan klarifikasi dan menuliskan perilaku tambahan yang dibutuhkan berdasarkan perspektif pengembangan perangkat lunak. Dengan demikian, baik pemesan maupun tim pengembang akan mengacu ke daftar spesifikasi yang sama (Sharma, 2014).

Spesifikasi pada pendekatan BDD meliputi fitur, dan perilaku yang dideskripsikan dalam sekumpulan skenario. Setelah spesifikasi disepakati, skrip pengujian fungsional dikembangkan berdasarkan skenario tersebut (Wynne & Hellesoy, 2012). Dengan pendekatan BDD, spesifikasi dan implementasi pengujian akan saling terkait, dan mudah dilakukan pelacakan (Wynne & Hellesoy, 2012). BDD difasilitasi dengan penggunaan *Domain-spesific Language* (DSL) pada spesifikasi yang mengekspresikan *behavior* dan hasil yang diharapkan [Dan North]. DSL, yang berisi spesifikasi dalam bahasa natural yang sudah terstruktur tersebut, kemudian dikonversikan menjadi skrip pengujian. Dengan demikian, pengujian akan selalu selaras mengikuti perubahan spesifikasi.

Pengujian perangkat lunak belum cukup untuk menjamin bahwa perangkat lunak sudah bebas dari kesalahan (*correctness*). Perangkat lunak dapat menghasilkan bermacam-macam reaksi, namun pengujian untuk setiap kemungkinan input tidak dimungkinkan. Verifikasi formal, yang telah menjadi metode yang diterima secara luas untuk memeriksa kesesuaian rancangan atau model dengan spesifikasi (Richards, 2003), dapat melengkapi pengujian untuk menjamin kesesuaian perangkat lunak dengan spesifikasi. Metode yang digunakan untuk memverifikasi berbasis pada logika matematika dan teori otomata (Dill, 1998). Salah satu metode untuk memverifikasi adalah *model checking*, yang dapat secara otomatis memverifikasi apakah desain mencakup, atau tidak mencakup perilaku yang dispesifikasikan secara formal. Pada verifikasi dengan *model checking*, model dari perangkat lunak dibangun secara formal misalnya dengan *Finite State Automata*. Model tersebut kemudian diverifikasi terhadap spesifikasi yang didefinisikan secara formal, misalnya dalam bentuk *temporal logic*.

Namun model dari sistem telah terverifikasi belum menjamin akan menghasilkan perangkat lunak yang benar, karena sangat tergantung dari implementasinya. Model yang telah terverifikasi sebelumnya mungkin menjadi tidak relevan lagi. Agar model yang diverifikasi selalu relevan dengan implementasi aktual dari perangkat lunak, pada penelitian ini diusulkan model yang diekstraksi dari *source code* perangkat lunak. Model hasil ekstraksi kemudian diverifikasi terhadap spesifikasi yang telah diformalkan dan dianotasikan pada *source code*.

Penelitian ini menyajikan pendekatan untuk verifikasi dan pengujian perangkat lunak, yaitu dengan menggabungkan pengujian dengan *Behavior-Driven Development* dan verifikasi formal, yang diterapkan pada *source code* perangkat lunak. Dengan demikian, manfaat dari kedua pendekatan ini dapat diperoleh sekaligus. Verifikasi formal dilakukan berdasarkan *source code*, yang telah diekstraksi menjadi model yang mampu diverifikasi, terhadap spesifikasi yang telah diformalkan dan dianotasikan pada *source code*. Model tersebut kemudian diperiksa apakah mampu memenuhi properti yang telah dispesifikasikan. Kemudian skrip pengujian *Behavior-Driven Development* dibangkitkan berdasarkan skenario yang dituliskan pada spesifikasi. Hasil penelitian ini adalah spesifikasi formal yang dianotasikan pada *source code*, model dari *source code*, skrip pengujian, hasil verifikasi, dan hasil pengujian. Dengan cara demikian, verifikasi dan pengujian menjadi relevan karena yang diverifikasikan adalah implementasinya, dan skrip pengujian selalu selaras dengan spesifikasi.

## Rumusan Masalah

Masalah yang dikaji pada penelitian ini adalah verifikasi dan validasi perangkat lunak dengan pendekatan verifikasi formal dan pengujian, yang sulit untuk diselaraskan dengan perubahan spesifikasi dan implementasi perangkat lunak. Diperlukan strategi untuk membuat spesifikasi perangkat lunak yang mencakup kebutuhan untuk verifikasi formal dan pengujian, serta strategi untuk mengekstraksi model formal dari *source code*. Dengan demikian, verifikasi formal dan pengujian akan selalu relevan dengan spesifikasi dan implementasi.

## Tujuan Penelitian

Tujuan yang ingin dicapai dari penelitian ini adalah verifikasi dan validasi perangkat lunak dengan dua pendekatan, yaitu verifikasi formal dan pengujian, yang bermanfaat dalam penjaminan *correctness* implementasi perangkat lunak terhadap spesifikasinya. Tujuan spesifik dari penelitian ini adalah menghasilkan teknik yang mampu membangkitkan skrip pengujian dari spesifikasi, mampu memformalkan spesifikasi, dan memodelkan *source code* agar dapat diverifikasi secara formal.

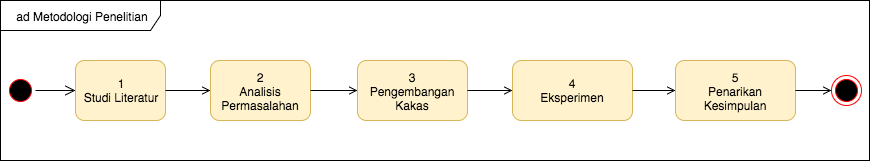
## Batasan Masalah

Berikut adalah batasan masalah dari penelitian ini:

1. Batasan masalah dari penelitian ini adalah hanya menggunakan teknik *model checking* untuk melakukan verifikasi formal.

## Metodologi

Metodologi pada penelitian ini meliputi 5 tahap utama, yaitu studi literatur, analisis permasalahan, pengembangan kakas, eksperimen, dan penarikan kesimpulan. Berikut pada Gambar- I.1 adalah metodologi pada penelitian ini.



Gambar- I.1 Tahap Utama Metodologi Penelitian

1. Studi Literatur

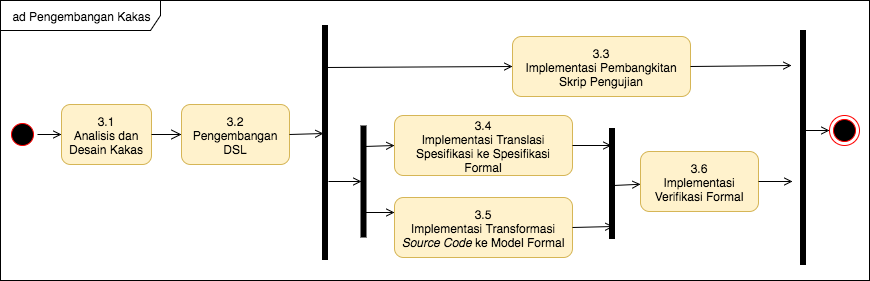
Pada tahap ini dilakukan studi literatur terkait verifikasi dan validasi perangkat lunak, verifikasi formal, ekstraksi model dari *source code*, pengujian perangkat lunak, pengujian dengan pendekatan *behavior-driven* pada perangkat lunak, dan pembangkitan skrip pengujian dari spesifikasi.

1. Analisis Permasalahan

Pada tahap ini dilakukan analisis terhadap permasalahan yang diangkat pada penelitian. Tahap ini akan menghasilkan usulan solusi untuk permasalahan pada penelitian.

1. Pengembangan Kakas

Tahap ini meliputi 5 subtahap yang ditunjukkan pada Gambar- I.2 berikut.



Gambar- I.2 Subtahap pada Pengembangan Kakas

Berikut adalah rincian tahap-tahap yang dilakukan pada pengembangan kakas

* 1. Analisis dan Desain Kakas

Pada tahap ini dilakukan analisis dan desain kakas verifikasi formal dan pengujian yang akan dikembangkan. Analisis dan desain meliputi analisis mengenai DSL yang perlu dikembangkan agar mampu menjadi spesifikasi perangkat lunak, mampu ditranslasikan ke spesifikasi formal, dan mampu dijadikan input untuk pembangkitan skrip pengujian. Serta analisis bagaimana mentransformasikan *source code* menjadi model yang mampu diverifikasi secara formal. Tahap ini akan menghasilkan daftar kemampuan kakas, dan rancangan kakas yang akan diimplementasikan.

* 1. Pengembangan DSL

Tahap ini menghasilkan DSL yang mampu dijadikan spesifikasi perangkat lunak, dan mencakup kebutuhan verifikasi formal dan pengujian.

* 1. Implementasi Pembangkitan Skrip Pengujian

Tahap ini akan menghasilkan modul untuk membangkitkan skrip pengujian berdasarkan spesifikasi dalam DSL yang dihasilkan pada tahap 2.2.

* 1. Implementasi Translasi Spesifikasi ke Spesifikasi Formal

Tahap ini akan menghasilkan modul untuk mentranslasikan spesifikasi dalam DSL yang dihasilkan pada tahap 2.2, menjadi spesifikasi formal untuk kebutuhan verifikasi formal.

* 1. Implementasi Transformasi *Source Code* ke Model Formal

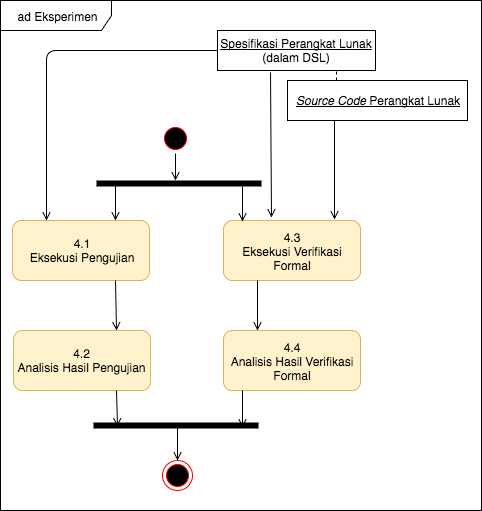
Tahap ini akan menghasilkan modul untuk mentransformasikan *source code* menjadi model formal yang mampu diverifikasi.

* 1. Implementasi Verifikasi Formal

Tahap ini akan menghasilkan modul untuk memverifikasi perangkat lunak secara formal, berdasarkan spesifikasi formal yang dihasilkan oleh modul 3.4 dan model formal yang dihasilkan oleh modul 3.5.

1. Eksperimen

Pada tahap ini dilakukan eksperimen terhadap kakas yang dihasilkan pada tahap 3. *Input* dari eksperimen adalah sebuah perangkat lunak, serta spesifikasinya yang dituliskan dalam DSL. Kakas akan menghasilkan hasil verifikasi formal dan pengujian pada perangkat lunak. Dilakukan analisis terhadap hasil verifikasi formal dan pengujian yang dilakukan untuk mengukur kinerja kakas. Rincian aktivitas yang dilakukan pada tahap eksperimen ditunjukkan pada Gambar- I.3. Adapun *workflow* dari verifikasi formal dan pengujian ditunjukkan pada Lampiran 1.



Gambar- I.3 Aktivitas pada Tahap Eksperimen

1. Penarikan Kesimpulan

Pada tahap ini akan dilakukan evaluasi kinerja dari kakas yang dihasilkan pada penelitian, kemudian ditarik kesimpulan terkait apakah pendekatan yang diusulkan pada penelitian ini mampu memberikan hasil evaluasi *correctness* implementasi perangkat terhadap spesifikasinya.

## Asumsi

Asumsi pada penelitian ini adalah spesifikasi perangkat lunak yang dijadikan input untuk penelitian telah mencakup kebutuhan pembangkitan skenario pengujian dan spesifikasi formal.

## Hipotesis

Gabungan dari verifikasi formal pada *source code* dan pengujian *Behavior-driven* *Development* mampu memverifikasi dan memvalidasi kesesuaian implementasi perangkat lunak terhadap spesifikasinya.

## Kontribusi Penelitian

Diharapkan penelitian ini dapat memberikan kontribusi antara lain, pertama yaitu DSL yang mampu menjadi spesifikasi yang mampu diformalkan dan dijadikan input untuk pembangkitan skrip pengujian, teknik pemodelan *source code* sehingga mampu diverifikasi secara formal, dan teknik untuk membangkitkan skrip pengujian berdasarkan spesifikasi.

## Pelaksanaan Tesis

Penelitian ini direncanakan dimulai pada bulan September 2017 dan dilaksanakan selama 6 bulan, sehingga akan selesai pada bulan Februari 2018. Rencana tahapan penelitian terhadap waktu pengerjaan dapat dilihat pada Tabel- 1 berikut.

**Tabel- 1** Pelaksanaan Tesis

| **Kegiatan** | **Sept** | | | | **Okt** | | | | **Nov** | | | | **Des** | | | | **Jan** | | | | **Feb** | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **1** | **2** | **3** | **4** | **1** | **2** | **3** | **4** | **1** | **2** | **3** | **4** | **1** | **2** | **3** | **4** | **1** | **2** | **3** | **4** |
| * + 1. Penyusunan proposal dan rencana penelitian |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * + 1. Studi Literatur dan penyusunan Bab II |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * + 1. Analisis Permasalahan |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * + 1. Pengembangan Kakas |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Analisis dan Desain Kakas |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Pengembangan DSL |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Implementasi Pembangkitan Skrip Pengujian |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Implementasi Translasi Spesifikasi ke Spesifikasi Formal |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Implementasi Transformasi *source code* ke model formal |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Implementasi verifikasi formal |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Penyusunan Bab III |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * + 1. Eksperimen |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Penentuan *input* (perangkat lunak studi kasus), dan identifikasi properti dan perilaku yang harus dipenuhi |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Eksekusi Pengujian |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Analisis Hasil Pengujian |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Eksekusi Verifikasi Formal |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Analisis Hasil Verifikasi Formal |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * 1. Penyusunan Bab IV |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| * + 1. Penarikan Kesimpulan dan penyusunan Bab V |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

# STUDI LITERATUR

Bab ini menyajikan hasil dari kajian literatur yang relevan dengan penelitian mengenai verifikasi formal, ekstraksi model dari *source code*, dan pengujian *behavior-driven*.

## Behavior-Driven Development

## Domain-Spesific Languages (DSLs)

Domain-spesific languages (DSLs) adalah bahasa yang disesuaikan untuk domain aplikasi yang spesifik [When and how to develop Domain-Spesific Language]. *Domain-spesific language* dikenal juga dengan nama *application-oriented*, *special purposes*, *specialized*, *task-specific*, atau *application language*. DSLs menukarkan generalitas ekspresi dalam domain yang terbatas, dengan menyediakan notasi dan konstruksi yang disesuaikan dengan domain aplikasi tertentu, DSLs menawarkan peningkatan dalam mengekspresikan dan kemudahan penggunaan, jika dibandingkan dengan GPLs (*General Programming Languages*) untuk domain yang dimaksud, sekaligus peningkatan produktivitas dan mereduksi biaya pemeliharaan. Selain itu, dengan mereduksi jumlah domain dan pengalaman pemrograman yang dibutuhkan, DSL membuka domain aplikasi ke kelompok pengembang perangkat lunak yang lebih luas. Berikut pada tabel x adalah beberapa DSL yang digunakan secara luar beserta domain aplikasinya.

**Tabel x.** Beberapa DSL yang digunakan secara luas [X]

| **DSL** | **Domain Aplikasi** |
| --- | --- |
| BNF | Spesifikasi sintaks |
| Excel | Spreadsheets |
| HTML | Halaman web *hypertext* |
| LATEX | Typesetting |
| Make | *Software building* |
| MATLAB | Komputasi teknis |
| SQL | Query basis data |
| VHDL | Perancangan perangkat keras |

Dengan mengkombinasikan dengan librari aplikasi, GPL apapun dapat berperan sebagai sebuah DSL. Librari

Berikut adalah manfaat penggunaan DSL yang menjawab masalah pada GPL.

|  |  |
| --- | --- |
|  |  |
| Notasi spesifik domain yang cocok biasanya diluar notasi operator yang ditawarkan oleh GPL | Sejak awal, DSL menawarkan notasi yang cocok untuk domain spesifik. Penggunaan DSL ini berdampak pada peningkatan produktivitas. |
| Konstruksi dan abstraksi dari spesifik domain tidak selalu dapat dipetakan secara langsung ke fungsi atau objek yang dapat digunakan pada librari | DSL dapat menginkorporasi konstruksi domain spesifik sejak awal |
|  | DSL menawarkan kemungkinan untuk analisis, verifikasi, optimisasi, paralerisasi dan transformasi dalam hal konstruksi DSL |

Fitur dari DSL adalah sebagai berikut [B]:

DSL dirancang sederhana, dengan tujuan mereduksi waktu *learning* pengguna DSL

DSL dibangun diatas kosakata pengguna dari domain

Sintaks yang disediakan DSL menyembunyikan aspek inheren pemrograman aplikasi dari *client*

Berikut adalah alasan-alasan dari kebutuhan pembuatan dan penggunaan DSL:

Pembuatan sebuah domain-spesific language dapat

## Verifikasi dan Validasi Perangkat Lunak

Berikut adalah 5 teknik yang pernah digunakan untuk memverifikasi perangkat lunak berdasarkan literatur (Harisson, 2008), (Liu, 2016):

1. *Formal Proof*

*Formal proof* adalah teknik matematis yang digunakan untuk verifikasi perangkat lunak (Harisson, 2008). Ide utamanya adalah untuk memformalisasi properti dari perangkat lunak kemudian membuktikan validitasnya dengan menggunakan *rules* dan *axioms*. Ada beberapa logic yang pernah digunakan dalam literatur, yaitu *propositional logic*, *first-order predicate logic*, *high-order predicate logic*, *Hoare logic* *logic* (Liu, 2016). Setiap logic punya kegunaan yang berbeda-beda. Kebanyakan sistem program menggunakan *first-order logic* (Liu, 2016).

Kekuatan dari *formal proof* adalah kemampuannya untuk mendemonstrasikan bahwa properti akan berlaku dalam kondisi-kondisi yang ditentukan, sehingga sangat baik untuk digunakan dalam penjaminan *correctness* dari perangkat lunak dengan *requirement* yang didefinisikan dengan baik (Liu, 2016). Namun, ada keterbatasan pada pengaplikasian *formal proof* pada software development, yaitu (Liu, 2016):

1. Kompleksitas dari proses pembuktian (*proof process*)
2. Tidak memungkinkan untuk menangani fungsi logika dengan definisi matematis yang minim
3. Hanya ada sedikit praktisi yang punya kemampuan untuk melakukan formal proof
4. Umumnya membutuhkan biaya besar, sehingga tidak mampu menjamin *cost-effective benefit* pada *software verification*
5. Model Checking

Model checking adalah teknik yang mampu secara otomatis memverifikasi apakah model pada finite automata memenuhi properti yang diekspresikan dalam *temporal logic* (Liu, 2016). Karena finite automata cocok untuk memodelkan reactive system, maka model checking awalnya diusulkan untuk hardware verification. Model checking telah menunjukkan efektivitasnya dalam mendeteksi error di hardware. Tantangan model checking pada software verification adalah sulitnya mengabstraksikan implementasi dari perangkat lunak kedalam *finite automata*, dan penanganan struktur data yang kompleks (Liu, 2016).

1. *Review* Perangkat Lunak

Review Perangkat Lunak adalah proses atau aktivitas untuk memeriksa dokumentasi perangkat lunak, atau kode perangkat lunak oleh developer yang relevan, atau user. Aktivitas yang dilakukan pada review perangkat lunak yaitu pemberian komentar, deteksi defect, dan perbaikan dokumen atau kode. Teknik yang umumnya digunakan pada review perangkat lunak antara lain: *peer review* (Wiegers, 2001), *inspection* (D. L. Parnas, 2003), dan *walkthrough* (Ciolkowski, et al., 2002). *Peer review* adalah evaluasi perangkat lunak oleh satu atau lebih developer dengan kompetensi yang sama atau lebih tinggi dari developer yang di-*review*. Inspeksi adalah proses pemeriksaan apakah perangkat lunak memenuhi standar yang ditentukan, atau *checklist* yang telah dibuat sebelumnya. *Walkthrough* adalah proses membaca algoritma dari perangkat lunak untuk mendeteksi bugs dengan cara mencoba memahami perilaku yang diharapkan dari tujuan program.

Kelebihan dari pendekatan *review* perangkat lunak adalah mampu diaplikasikan pada dokumen dalam bentuk apapun, seperti: spesifikasi kebutuhan, desain, kode, dan pengujian. Tantangan dari pendekatan ini adalah pemeriksaan dari target yang di-*review* harus dilakukan oleh manusia dengan pengetahuan, dan kecakapan yang sesuai.

1. Pengujian Perangkat Lunak

Pengujian dilakukan untuk mendeteksi kesalahan pada pada perilaku perangkat lunak ketika runtime. Pengujian perangkat lunak umumnya terdiri dari 3 aktivitas utama, yaitu pembuatan kasus uji, eksekusi pengujian, dan analisis hasil pengujian. Tiga pendekatan yang dikenal pada pengujian perangkat lunak adalah sebagai berikut (Liu, 2016):

* 1. Pengujian White-box

Pada pengujian white-box, struktur program menjadi elemen yang digunakan untuk membuat kasus uji. Ada beberapa pendekatan yang digunakan pada pengujian white-box, yaitu: *branch coverage*, *statement coverage*, *path coverage*, dan *modified condition / decision coverage* (MC/DC). Setiap kriteria *coverage* mengindikasikan kondisi dimana pengujian dapat dihentikan. Pengujian white-box dapat menentukan bagian pada program yang perlu diperiksa. Namun, pengujian white-box mensyaratkan ketersediaan kode program untuk diperiksa oleh tester, dan hanya mampu memeriksa fungsionalitas yang sudah diimplementasikan pada program, dan tidak mampu memeriksa apakah program telah mengimplementasikan semua fungsi atau konstrain yang dispesifikasikan.

* 1. Pengujian Black-box

Pada pengujian Black-box, program yang diujian (SUT) diperlakukan sebagai black-box, yang evaluasinya dilakukan hanya berdasarkan pada input dan output, bukan berdasarkan pada isi dari program. Pengujian dilakukan dengan menentukan *initial states* dan *final states*, kemudian membandingkan *final states* dengan output dari perangkat lunak. Ada banyak teknik yang digunakan untuk membuat kasus uji, yaitu: *random testing*, *combinatorial testing*, *specification-based testing*, *vibration testing*, dan *relation-based testing*. Jika dibandingkan dengan pengujian white-box, pengujian black-box relative lebih mudah dilakukan, dan mudah diotomasi (Liu, 2016).

* 1. Pengujian Grey-box

Pengujian grey-box adalah kombinasi dari pengujian white-box dan pengujian black-box. Pembuatan kasus uji dilakukan berdasarkan struktur program dan spesifikasi kebutuhan. Namun, umumnya hanya sebagian kode yang dapat dipelajari oleh tester. Kelebihan dari pengujian grey-box adalah kemampuannya untuk menggunakan kedua informasi: spesifikasi kebutuhan dan implementasi program dalam perancangan kasus uji. Tantangan pada pengujian grey-box adalah cara agar informasi parsial mengenai struktur program dapat diutilisasi dan dikombinasikan dengan informasi pada spesifikaksi kebutuhan untuk membuat kasus uji.

## Verifikasi Formal Perangkat Lunak

### Verifikasi Formal

### Model Checking

### Ekstraksi Model dari *Source Code*

## Pengujian pada Behavior-Driven Development

### Skenario Pengujian

## Kakas pada Behavior-Driven Development

## Penelitian Terkait

## Verifikasi Formal

Verifikasi formal adalah tindakan untuk membuktikan kebenaran suatu sistem berkenaan dengan spesifikasi atau properti formal tertentu (Kukimoto, 1996). Fase perancangan adalah tahap kritis dalam pengembangan sebuah produk perangkat lunak. Rancangan sistem yang salah atau tidak konsisten akan mengakibatkan waktu yang dibutuhkan untuk tahap perancangan bertambah untuk mengakomodasi perubahan dan perbaikan rancangan. Maka dari itu, sangat penting untuk melakukan eksplorasi dari desain sedini mungkin (Kukimoto, 1996).

Berdasarkan referensi (Liu, 2016), untuk dapat melakukan verifikasi formal sebuah rancangan, mula-mula rancangan harus diubah ke dalam format yang dapat diverifikasi. Rancangan dapat dipandang sebagai himpunan dari sistem yang saling berinteraksi. Pada setiap sistem interaksi, ada sejumlah *finite states*. *States* dan transisi antar *states* akan membentuk akan *Finite State Machines* (FSMs). Keseluruhan sistem dapat dipandang sebagai sebuah FSMs, yang diperoleh dengan menyusun FSMs yang berkaitan. Dengan demikian, langkah utama yang dilakukan pada verifikasi formal adalah menghasilkan deskripsi lengkap sistem dalam format FSM yang dapat diverifikasi. Setelah itu, dilakukan verifikasi model terhadap setiap formula yang mewakili properti.

## Verifikasi Perangkat Lunak

## Model Checking

Salah satu metode untuk melakukan verifikasi formal adalah *model checking* (Kukimoto, 1996). Pada model checking dengan temporal logic, sebuah finite state system direpresentasikan dalam buah graf transisi berlabel, dimana label dari *state* adalah nilai dari proposisi atomic pada label tersebut (Kukimoto, 1996). Properti dari sistem diekspresikan sebagai formula dalam *temporal logic* yang mana transisi state dianggap sebagai sebuah model (Kukimoto, 1996). *Model checking* terdiri dari menelusuri graf transisi sistem untuk memverifikasi bahwa model memenuhi sebuah formula yang merepresentasikan properti atau spesifikasi sistem.

## *Linear Temporal Logic* (LTL)

*Temporal logic* mengekspresikan urutan kejadian dalam waktu dengan menggunakan operator-operator yang mewakili properti. Salah satu *temporal* logic adalah *Linear Temporal Logic* (LTL). LTL pada penelitian ini digunakan untuk menspesifikasikan properti pada program. Program konvensional umumnya melakukan hal berikut: menerima input, melakukan komputasi, dan mengembalikan output (Mukund, 1996). Dengan demikian, program dapa dipandang sebagai fungsi yang memetakan *input domain*, ke *output domain*. *Behavior* dari program terdiri dari transformasi dari *initial states* ke *final states* (Mukund, 1996).

Pada referensi (Wolper, 1986), LTL diperluas penggunaannya ke model checking. Persoalan yang ditangani oleh *model checking* adalah verifikasi: apakah *finite-state program* memenuhi spesifikasi .

### Sintaks LTL

Berikut adalah notasi dasar dan definisi pada LTL berdasarkan referensi (Sebastiani, 2017):

1. Proposisi atomic , adalah formula LTL
2. Jika dan adalah LTL formula, maka , , , , adalah LTL formula
3. Jika dan adalah LTL formula, maka X, U, G, F adalah LTL formula, dimana X, G, F, U merepresentasikan “next”, “globally”, “eventually”, dan “until”
4. Operator lain yang digunakan pada LTL adalah R (Release) R

### Semantik LTL

Berikut adalah semantik dari LTL berdasarkan referensi (Sebastiani, 2017)::

1. *Next* (X), adalah *true* pada jika dan hanya jika true pada
2. *Finally* (F), adalah *true* pada jika dan hanya jika true pada beberapa dimana t’ > t
3. *Globally* (G), adalah *true* pada jika dan hanya jika true pada semua dimana t’ > t
4. *Until* (U), adalah *true* pada jika dan hanya jika pada beberapa state dimana t’ ≥t:

- *true* pada dan

- *true* pada semua *state* t < t’’ < t’

1. *Release* (R), adalah *true* pada jika dan hanya jika untuk semua state dimana t’ ≥t:

- *true* pada dan

- *true* pada beberapa *state* t < t’’ < t’

akan *false* jika bernilai *true* lebih dulu

## Test-Driven Development

Perangkat lunak saat ini melakukan banyak operasi kompleks seperti menampilkan antar muka yang kompleks, *multithreading*, menyimpan data di database lokal, pemutaran media, dan mengkonsumsi RESTful web API. Dibutuhkan cara untuk mengetahui bahwa kode yang ditulis akan menghasilkan perilaku yang sesuai dengan ekspektasi (spesifikasi kebutuhan). Cara yang dapat digunakan adalah dengan melakukan pengujian unit dan pengujian *behavior-driven* (Mishra, 2017).

*Test-Driven Development* (TDD) dirancang untuk menyediakan *developers* cara untuk membuktikan bahwa kode yang dituliskan berperilaku sesuai dengan yang diharapkan. Berdasarkan (Mishra, 2017), inti dari pendekatan TDD adalah konsep bahwa *developer* tidak hanya menulis kode untuk mengimplementasi fungsionalitas aplikasi, tapi juga sekaligus menguji kode agar kode berlaku seperti yang diharapkan.

**Terminologi pada TDD**

Berikut adalah terminologi pada TDD berdasarkan referensi (Mishra, 2017):

1. *Subject under Test* / *System under Test* (SUT)

Bagian kode yang akan diuji. Umumnya *Subject under Test* adalah sebuah method tunggal dari sebuah kelas. Namun *Subject under Test* dapat pula berupa skenario dimana kelompok method atau kelas diuji bersama-sama. Pada beberapa kasus, *subject under test* merepresentasikan fungsionalitas lengkap / *end-to-end* proses bisnis.

1. Unit Test

Unit Test adalah bagian kode yang melakukan pengujian terhadap SUT. Sebuah unit test juga dikenal dengan nama *test case* (kasus uji). Unit test bekerja dengan memanggil subject under test pada kondisi terkontrol, dan memverifikasi perilaku yang diharapkan dihasilkan dari kode dalam kondisi tersebut.

1. *State Verification* *Test*

*State verification test* adalah tipe dari unit test yang memanggil method dari objek (SUT) dan memverifikasi state dari objek setelah memanggil method. State Verification Test biasanya tergantung pada *assertion* untuk melakukan verifikasi.

1. *Interaction Test*

*Interaction Test* adalah tipe dari unit test yang mencoba untuk memverifikasi urutan dari interaksi antar objek ketika sebuah method dipanggil. Pengujian inii dikenal juga dengan nama behavior verification tests. Interaction test test tidak selalu harus melibatkan beberapa objek.

1. Negative Test

*Negative unit test* adalah pengujian yang memverifikasi bahwa sesuatu tidak terjadi. Negative test dapat bermanfaat untuk kasus-kasus tertentu, namun pengujian tidak boleh tergantung pada negative test. Hal ini karena ketika suatu negative test dapat memverifikasi bahwa sesuatu tidak terjadi, pengujian tersebut menjadi kebal terhadap jumlah berapapun hal yang mungkin terjadi. Negative unit test umumnya adalah state verification test.

1. *Test Suite*

*Test suite* adalah sebuah koleksi dari berkas skrip pengujian.

1. *Assertions*

*Assertions* digunakan pada *state verification* dan *interaction tests*. Sebuah *assertion* merepresentasikan sebuah kesalahan (*failure*) pada unit test.

## Behavior-driven Development (BDD)

Berdasarkan referensi (Alliance, 2017), Behavior-Driven adalah sebuah pendekatan untuk pengembangan perangkat lunak yang dibangun dengan tujuan untuk memformalkan *best practices* yang diikuti oleh praktisi *Test-Driven Development*. Pada pengembangan perangkat lunak, *Behavior-driven Development* (BDD) testing mengkombinasikan teknik umum dan prinsip-prinsip TDD dengan gagasan dari *domain-driven design*. BDD juga dirujuk sebagai *specification by example*.

Salah satu masalah yang dihadapi oleh orang yang baru menerapkan pendekatan TDD adalah memutuskan apa yang akan diuji. *Behavior-driven Development* memberikan pendekatan pengujian yang berbeda, yaitu dengan menguji perilaku (*behavior*) dari sistem (Mishra, 2017).

### Perbedaan BDD dan TDD

Berdasarkan referensi (Mishra, 2017), perbedaaan utama antara behavior-driven development dan test-driven development adalah level kedetilannya. BDD melakukan pengujian dimana perilaku yang diterima (*accepted behavior*) dari sistem didefinisikan dengan himpunan skenario yang dapat diturunkan dari *business requirement*. Pengujian dengan BDD sering kali lebih deskriptif dan bermakna untuk bisnis (Mishra, 2017). BDD dideskripsikan dalam *Domain Specific Language* (DSL) yang mengandung *terms* dan konsep yang dikenal oleh domain bisnis.

## Kakas pada Behavior-driven Development

### Cucumber

Cucumber adalah kakas untuk melakukan eksekusi pengujian secara otomatis. Kakas ini menggabungkan spesifikasi dan dokumentasi pengujian dalam satu bagian utuh. Cucumber diimplementasikan spesifik untuk perangkat lunak yang menerapkan *Behavior-Driver Development*.

### Gherkin

Gherkin adalah Bahasa yang dipahami oleh Cucumber. Gherkin adalah “DSL *Business Readable*” yang mengizinkan pengembang untuk mendeskripsikan perilaku dari perangkat lunak tanpa memperinci bagaimana perangkat lunak tersebut diimplementasikan. Gherkin menyediakan dua tujuan, yaitu: dokumentasi dan otomasi pengujian.

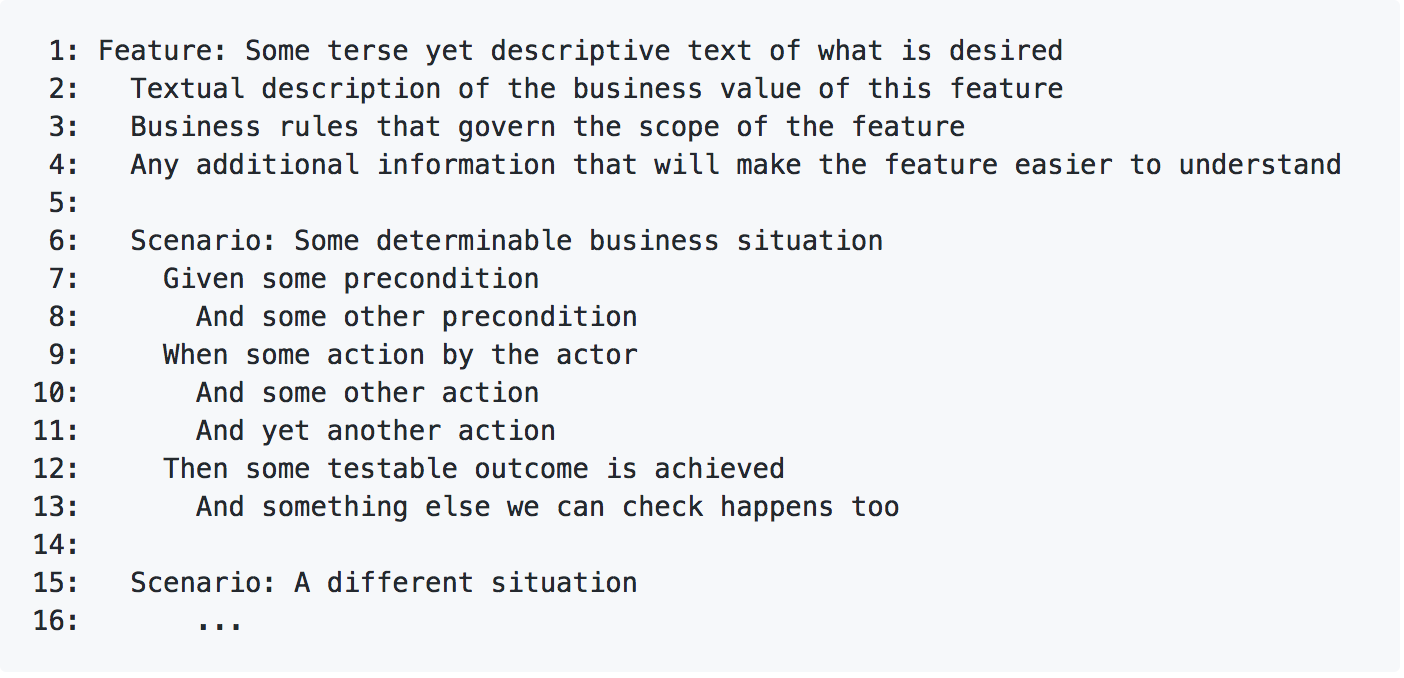
Grammar pada Gherkin didefinisikan dalam *Treetop Grammar* yang adalah bagian dari *codebase* Cucumber. Ada dua konvensi pada Gherkin:

1. Berkas tunggal Gherkin mendeskripsikan satu fitur tunggal
2. Berkas dibuat dalam ekstensi .feature

#### **Sintaks**

Gherkin adalah Bahasa berorientasi baris, seperti Python dan YAML, yang menggunakan indentasi untuk mendefinisikan struktur. Akhir baris akan mengakhiri sebuah statement (misalnya langkah pengujian pada skenario). Spasi atau tabulasi dapat digunakan sebagai indentasi. Penulisan komentar diawali dengan tanda pagar (#).

Input pada berkas Gherkin terbagi menjadi *features*, *scenarios*, dan *steps*. Gambar berikut adalah salah satu contoh berkas Gherkin.

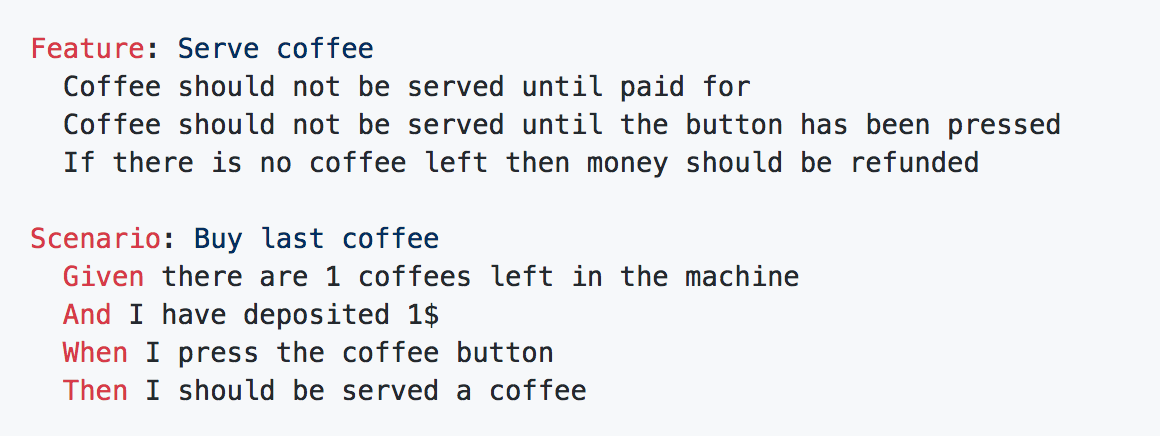


Gambar- II.1 Contoh Berkas dengan Bahasa Gherkin

Baris pertama mendeskripsikan fitur, Baris 2-4 adalah unparsed text, sehingga diekspektasikan mendefinisikan lebih rinci mengenai fitur. Skenario dimulai pada baris 6, dan baris 7-13 adalah langkah-langkah pada skenario tersebut.

#### **Berkas .*feature***

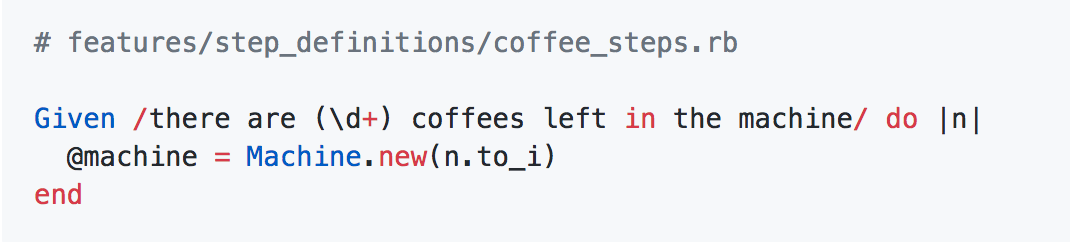
Setiap berkas .feature secara konvensional terdiri dari fitur tunggal dari sebuah perangkat lunak. Baris yang dimulai dengan kata kunci **Feature** dan diikuti oleh teks adalah deskripsi dari fitur. Sebuah fitur biasanya mengandung daftar skenario. Skenario dimulai dengan kata kunci Scenario. Setiap skenario terdiri dari daftar langkah-langkah (*steps*), yang harus dimulai dengan salah satu kata kunci berikut: **Given**, **When**, **Then**, **But**, atau **And**. Berikut adalah contoh berkas **Feature** untuk fungsi menyediakan kopi.



Gambar- II.2 Contoh Berkas *Feature*

#### ***Step Definition***

Untuk setiap langkah (*steps*), Cucumber akan mencari **step definition** yang cocok. *Step definition* dituliskan dalam Bahasa Ruby. Setiap *step definition* terdiri dari string atau regular expression. Gambar berikut adalah contoh *step definition*.



Gambar- II.3 Contoh *Step Definition* pada Gherkin

Step definition dapat dianalogikan dengan definisi method atau fungsi pada Bahasa pemrograman. Step, jika dianalogikan pada method, adalah invokasi dari fungsi.

# DAFTAR PUSTAKA

Richards, J., 2003. *CREATIVE ASSERTION AND CONSTRAINT METHODS FOR FORMAL DESIGN VERIFICATION.* California: DVCon.

Dill, D. L., 1998. *What’s Between Simulation and Formal Verification? (Extended Abstract).* California: Stanford University.

Liu, S., 2016. *Testing-Based Formal Verification for Algorithmic Function Theorems and Its Application to Software Verification and Validation.* Tokyo, IEEE.

Cánovas Izquierdo J. L., G. M. J. .., 2014. *Extracting models from source code in software modernization..* s.l., IEEE, p. 713–734.

Deepti Mishra, A. M., 2007. *Adapting Test-Driven Development for Innovative Software Development Project.* Berlin, Springer-Verlag , pp. 171-173.

Mishra, A., 2017. *iOS Code Testing - Test-Driven Development and Behavior-Driven Development with Swift.* London: Springer Science.

Mukund, M., 1996. Linear Temporal Logic and Buchi Automata. In: Madras: ISI Calcutta.

Sebastiani, R., 2017. *Linear Temporal Logic - LTL,* Trento: Università di Trento.

Wolper, M. V. a. P., 1986. An automata theoretic approach to automatic program verification. pp. 332-345.

Kukimoto, Y., 1996. [Online]   
Available at: https://embedded.eecs.berkeley.edu/research/vis/doc/VisUser/vis\_user/node4.htm  
[Accessed 31 July 2017].

Harisson, J., 2008. Formal Proof - Theory and Practices. *Notices of the American Mathematical Society,* December.55(11).

F. Nieson, H. N. C. H., 2005. Principles of Program Analysis.

Wiegers, K. E., 2001. *Peer Review in Software: A Practical Guide.* s.l.:Addison-Wesley.

D. L. Parnas, M. L., 2003. The Role of Inspection in Software Quality Assurance. *IEEE Transaction of Software Engineering,* August, 29(8), pp. 674-676.

Anon., 2012. IEEE Standard for System and Software Verification and Validation. pp. 1-223.

Hailpern, B. & Santhanam, P., 2002. Software debugging, testing, and verification. *IBM Systems Journal,* January, 41(1), pp. 4-12.

Collofello, J. S., 1988. Introduction to Software Verification and Validation. *SEI Curriculum Module SEI-CM-13-1.1,* December.

Dijkstra, E., n.d. *Notes on Structured Programming,* s.l.: s.n.

Wilcox, R., n.d. *Your Boss Won't Appreciate TDD: Try This Behavior-Driven Development Example.* [Online]   
Available at: https://www.toptal.com/freelance/your-boss-won-t-appreciate-tdd-try-bdd  
[Accessed 20 09 2017].

Sharma, L., 2014. *Behavior-Driven Development.* [Online]   
Available at: http://toolsqa.com/cucumber/behavior-driven-development/  
[Accessed 20 09 2017].

Wynne, M. & Hellesoy, A., 2012. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers.* s.l.:Pragmatic Bookshelf.

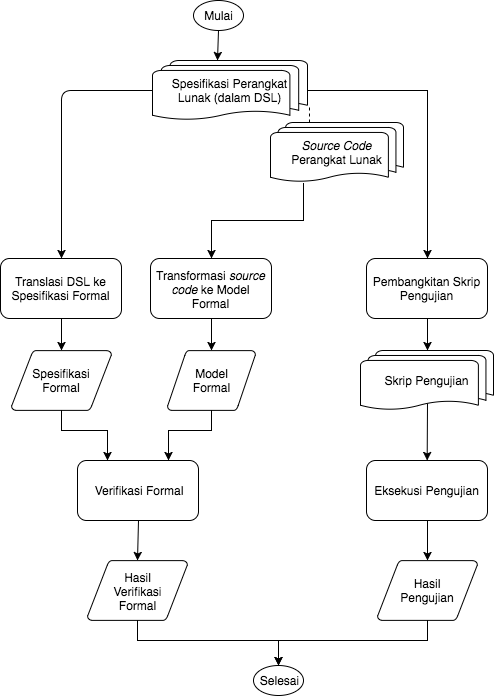
Ciolkowski, M. et al., 2002. *Software Inspections, Reviews, and Walkthroughs.* s.l., IEEE Computer Society Press, pp. 641-642.

M. Ciolkowski, O. L. D. R. F. S. D. P., 2002. *Software Inspections, Reviews, and Walkthroughs.* s.l., IEEE Computer Society Press, pp. 641-642.

Alliance, A., 2017. *Behavior-Driven Development.* [Online]   
Available at: https://www.agilealliance.org/glossary/bdd/#q=~(filters~(postType~(~'page~'post~'aa\_book~'aa\_event\_session~'aa\_experience\_report~'aa\_glossary~'aa\_research\_paper~'aa\_video)~tags~(~'bdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1)  
[Accessed 11 September 2017].

# LAMPIRAN 1: *Workflow* Usulan Penelitian

Hasil dari penelitian ini yaitu DSL untuk menspesifikasikan perangkat lunak, dan kakas untuk melakukan verifikasi formal dan pengujian. Kakas untuk verifikasi formal dan pengujian menerima 2 *input*, yaitu spesifikasi perangkat lunak yang dituliskan dalam DSL, dan *source code* dari perangkat lunak. Spesifikasi perangkat lunak akan ditranslasikan menjadi spesifikasi formal dan skrip pengujian, sedangkan *source code* akan ditransformasikan menjadi model formal. Spesifikasi formal dan model formal menjadi *input* untuk melakukan verifikasi formal. Kemudian skrip pengujian dieksekusi sehingga diperoleh hasil pengujian perangkat lunak. *Workflow* usulan penelitian ditunjukkan pada Gambar berikut:



Gambar-Lampiran 1.1 *Workflow* usulan penelitian