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

TESIS
DRAFT BAB I - III

Oleh
JESSIE ANDIKA SETIADY
NIM: 23516064
(Program Studi Magister Informatika)

INSTITUT TEKNOLOGI BANDUNG
Februari 2018

DAFTAR ISI

DAFTAR GAMBAR DAN ILUSTRASI

DAFTAR TABEL

Bab I PENDAHULUAN

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

I.1 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, 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 dan Santhanam, 2002). Penyebab utama kegagalan perangkat lunak berdasarkan (Hailpern dan 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 berdasarkan (IEEE, 2002) adalah proses untuk menentukan apakah *requirement* untuk sistem atau komponen lengkap dan benar, produk yang dihasilkan oleh setiap fase pengembangan memenuhi *requirements* atau kondisi yang ditetapkan, dan sistem atau komponen akhir sesuai dengan *requirements* yang dispesifikasikan. Proses V&V meliputi banyak pendekatan dan kakas. Berdasarkan (Berztiss dan Ardis, 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*.

Objektif umum dari pendekatan V&V pada perangkat lunak adalah memastikan bahwa produk bebas dari kesalahan, dan memenuhi ekspektasi pengguna. Objektif spesifik dari V&V harus ditentukan berbasis proyek, dan dipengaruhi oleh level kritis produk, dan kompleksitas produk (Berztiss dan Ardis, 1988). Aktivitas V&V berdasarkan (Berztiss dan Ardis, 1988) (IEEE-SA Standards Board dkk., 2013) umumnya fokus pada pengujian perangkat lunak. Pengujian perangkat lunak bertujuan untuk mencari kesalahan, namun belum mampu menjamin perangkat lunak bebas dari kesalahan (Dijkstra dkk., 1972) (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 menyelaraskan kasus uji dengan perubahan spesifikasi yang cepat. Penyelarasan kasus uji ini sering kali diabaikan, dan hanya dilakukan pada tahap akhir menjelang tenggat waktu (Wilcox, 2014). Pada kasus ekstrim, pengujian bahkan sangat minim dilakukan karena keterbatasan waktu atau anggaran (Wilcox, 2014).

Proses pengembangan perangkat lunak dengan *Behavior-Driven Development* (BDD) menggabungkan prinsip *Test-Driven Development* (TDD) dengan *Domain-Driven Design* (Rose dkk., 2015). BDD difasilitasi dengan *Domain-Specific Language* (DSL) untuk mendefinisikan spesifikasi perangkat lunak, sekaligus untuk melakukan *acceptance testing* yang mampu dikonversikan menjadi kode pengujian (Solis dan Wang, 2011). Dengan memanfaatkan DSL, pengembang dan *stakeholder* dapat berkolaborasi untuk mendefinisikan perilaku perangkat lunak dan hasil yang diharapkan. Dengan demikian, pengembang dan *stakeholder* akan mengacu ke daftar spesifikasi yang sama.

Pengujian pada proses pengembangan perangkat lunak dengan BDD belum cukup untuk menjamin bahwa perangkat lunak sudah bebas dari kesalahan (*correctness*) (Diepenbeck dan Drechsler, 2015). Berdasarkan (Berztiss dan Ardis, 1988), hal tersebut disebabkan oleh dua hal: tidak mungkin untuk menguji semua kemungkinan input, dan tidak mungkin untuk menguji semua jalur eksekusi.

Verifikasi formal, yang telah menjadi metode yang diterima secara luas untuk memeriksa kesesuaian rancangan atau model dengan spesifikasi (Narisawa dkk., 2015), dapat melengkapi pengujian untuk menjamin kesesuaian perangkat lunak dengan spesifikasi. 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. Model tersebut kemudian diverifikasi terhadap properti yang juga didefinisikan secara formal.

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 melengkapi proses V&V pada proses pengembangan perangkat lunak *Behavior-Driven Development*, dengan menambahkan proses verifikasi formal, yang diterapkan pada *existing source code* perangkat lunak. Dengan demikian, manfaat dari pengujian dan verifikasi formal 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*. *Acceptance testing* didefinisikan dalam bentuk skenario yang dituliskan dalam DSL, dan dapat dibangkitkan menjadi kode pengujian. Hasil penelitian ini adalah sebuah DSL yang mencakup spesifikasi dan skenario pengujian, serta mekanisme untuk melakukan verifikasi formal implementasi perangkat lunak terhadap spesifikasi formal yang dianotasikan pada *source code*. Dengan demikian, verifikasi formal yang relevan terhadap implementasi akan melengkapi pengujian pada proses pengembangan dengan BDD.

I.2 Rumusan Masalah

Dari uraian latar belakang, dapat dirumuskan beberapa masalah yang menjadi fokus penelitian ini:

1. Pengujian yang dilakukan pada proses pengembangan perangkat lunak dengan BDD, tidak cukup untuk menjamin perangkat lunak dari kesalahan (*correctness*). Verifikasi formal dapat melengkapi proses V&V pada BDD. Diperlukan strategi untuk membuat spesifikasi perangkat lunak yang mencakup kebutuhan untuk pengujian dan verifikasi formal.
2. Model perangkat lunak yang telah terverifikasi secara formal belum dapat menjamin *correctness* perangkat lunak, karena sangat tergantung dari implementasinya. Verifikasi formal harus dikenakan pada *source code*, agar hasil verifikasi selalu relevan dan mampu menjadi parameter evaluasi *correctness* perangkat lunak.

I.3 Tujuan Penelitian

Tujuan dari penelitian ini adalah:

1. Menghasilkan sebuah *Domain-Specific Language* yang mampu menjadi spesifikasi perangkat lunak, dan mencakup kebutuhan untuk pengujian dan verifikasi formal
2. Menghasilkan kerangka kerja untuk melengkapi proses V&V pada pengembangan perangkat lunak *Behavior-Driven Development*, dengan verifikasi formal yang dikenakan pada level *source code*

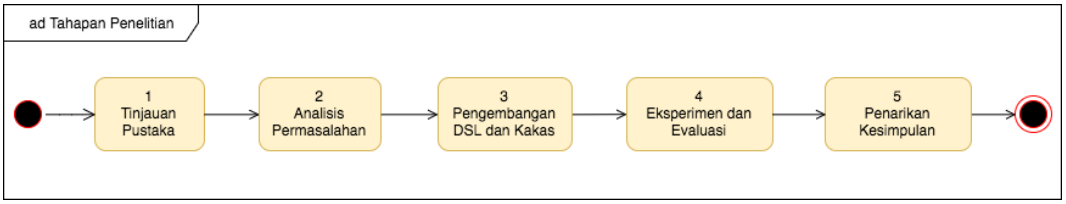
I.4 Batasan Masalah

Batasan masalah pada penelitian ini adalah:

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

I.5 Metodologi Penelitian

Penelitian ini meliputi 5 tahap utama, yaitu studi literatur, analisis permasalahan, pengembangan DSL dan kakas, eksperimen dan evaluasi, serta penarikan kesimpulan. Berikut pada Gambar I-1 adalah tahapan utama pada penelitian ini.



Gambar I-1 Tahapan Utama Penelitian

1. Tinjauan Pustaka

   Pada tahap ini dilakukan tinjauan pustaka terkait V&V perangkat lunak, verifikasi formal, ekstraksi model dari *source code*, pengujian perangkat lunak pada proses pengembangan *Behavior-driven Development*, rekayasa *Domain-Specific Language*, dan eksplorasi kakas yang dapat dimanfaatkan.

2. Analisis

   Pada tahap ini dilakukan analisis terhadap permasalahan yang diangkat pada penelitian. Tahap ini mencakup analisis mengenai V&V pada proses pengembangan BDD yang ada, identifikasi masalah-masalah pada proses pengembangan yang ada, dan analisis solusi yang diusulkan pada penelitian. Hasil dari tahap ini adalah perbedaan dari proses V&V pada BDD yang ada, dengan proses V&V pada BDD dengan penggabungan pengujian dan verifikasi yang dilakukan penelitian ini.

3. Pengembangan DSL dan Kakas

   Pada tahap ini dilakukan pengembangan DSL dan pengembangan kakas. Berikut adalah rincian tahap-tahap yang dilakukan pada pengembangan kakas

   2.1 Pengembangan DSL

   Pada tahap ini dilakukan rekayasa DSL yang meliputi 5 fase berdasarkan (Mernik dkk., 2005) yaitu: *decision*, *analysis*, *design*, *implementation*, dan *deployment*. Tahap ini menghasilkan DSL yang

mampu dijadikan spesifikasi perangkat lunak, dan mencakup kebutuhan verifikasi formal dan pengujian.

2.2 Pengembangan Kakas

Pada tahap ini dilakukan pengembangan kakas yang mampu mengekstraksi properti formal yang dianotasikan pada *source code*, melakukan transformasi *source code* menjadi model yang mampu diverifikasi, dan melakukan verifikasi formal. Tahap ini meliputi 4 fase: analisis, desain, implementasi, dan pengujian. Tahap ini akan menghasilkan daftar kemampuan kakas, rancangan kakas yang diimplementasikan, dan kakas hasil implementasi.

4. Eksperimen dan evaluasi

   Pada tahap ini dilakukan eksperimen dan evaluasi terhadap DSL dan kakas yang dihasilkan pada tahap 3. Objek eksperimen adalah sebuah perangkat lunak. Eksperimen dilakukan dengan cara mengembangkan perangkat lunak dengan spesifikasi yang dituliskan dengan DSL yang dihasilkan pada tahap 3.1, dan mengeksekusi kakas yang dihasilkan tahap 3.2 untuk melakukan pengujian dan verifikasi formal. Adapun desain eksperimen dari penelitian yang dilakukan ditunjukkan pada Lampiran 1.

5. Penarikan Kesimpulan

   Pada tahap ini akan dilakukan evaluasi kinerja dari mekanisme pengujian dan verifikasi formal yang dihasilkan pada penelitian, kemudian ditarik kesimpulan terkait apakah mekanisme yang diusulkan pada penelitian ini mampu memberikan hasil yang dapat dijadikan parameter untuk mengukur kesesuaian implementasi perangkat lunak terhadap spesifikasinya (*correctness*).

I.6 Asumsi

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

I.7 Hipotesis

Gabungan dari verifikasi formal pada *source code* dan pengujian pada proses pengembangan perangkat lunak dengan *Behavior-driven Development*, mampu memberikan hasil yang dapat dijadikan parameter untuk mengukur kesesuaian implementasi perangkat lunak terhadap spesifikasinya (*correctness*).

I.8 Sistematika Penulisan

Sistematika penulisan laporan pada penelitian ini adalah sebagai berikut:

Bab I Pendahuluan. Bab ini memaparkan latar belakang, rumusan masalah, tujuan, batasan masalah, tahapan penelitian, dan sistematika penulisan laporan.

Bab II Tinjauan Pustaka. Bab kedua berisi uraian hasil-hasil penelitian terdahulu yang melandasi penelitian ini, beserta teori-teori pendukung yang bersumber dari buku teks dan jurnal.

Bab III Analisis. Bab ketiga berisi analisis permasalahan, dan analisis solusi yang ditawarkan penelitian ini.

Bab IV Perancangan dan Implementasi. Bab keempat berisi uraian tentang rancangan serta implementasi dari solusi yang digunakan.

Bab V Eksperimen dan Evaluasi. Bab kelima memaparkan tentang skenario-skenario, lingkungan, hasil, dan evaluasi dari eksperimen yang dilakukan

Bab VI Kesimpulan dan Saran. Bab keenam berisi kesimpulan hasil penelitian, dan saran untuk penelitian selanjutnya.

Bab II TINJAUAN PUSTAKA DAN EKSPLORASI KAKAS

Bab ini menyajikan hasil dari kajian literatur yang relevan dengan penelitian mengenai V&V perangkat lunak, pengujian pada proses pengembangan BDD, verifikasi formal, dan ekstraksi model dari *source code*. Bab ini juga menyajikan tinjauan mengenai kakas yang digunakan untuk memenuhi tujuan dari tesis.

II.1 Verifikasi dan Validasi (V&V) Perangkat Lunak

Berdasarkan (IEEE Std 1012-2016, 2016) proses verifikasi dan validasi adalah proses teknikal pada rekayasa sistem, perangkat lunak, dan perangkat keras. Tujuan dari V&V adalah untuk membantu organisasi membangun kualitas pada sistem selama siklus hidup sistem. Proses V&V menyediakan penilaian objektif dari produk dan proses selama siklus hidup. Penilaian ini mendemonstrasikan apakah *requirements* benar, lengkap, akurat, konsisten, dan mampu diuji. Proses V&V menentukan apakah produk yang dikembangkan sesuai dengan *requirement* dan produk memenuhi kebutuhan penggunanya. Penentuan dilakukan dengan penilaian, analisis, evaluasi, *review*, inspeksi, dan pengujian dari produk dan proses.

Proses V&V terdiri dari proses verifikasi dan proses validasi. Proses verifikasi menyediakan bukti objektif yang menentukan apakah produk (IEEE Std 1012-2016, 2016):

1. Sesuai dengan *requirement* (lewat atribut *correctness*, *completeness*, *consistency*, dan *accuracy*) untuk semua aktivitas pada setiap proses
2. Memenuhi standar, praktis, dan konvensi selama proses siklus hidup
3. Secara sukses melengkapi semua aktivitas siklus hidup dan memenuhi semua kriteria untuk memulai aktivitas siklus hidup berikutnya (*builds the product correctly*)

Proses validasi menyediakan bukti untuk menentukan apakah produk (IEEE Std 1012-2016, 2016):

1. Memenuhi daftar kebutuhan sistem yang dialokasikan ke produk
2. Menyelesaikan masalah yang tepat (misalnya: mengimplementasikan aturan bisnis dengan benar, menggunakan asumsi sistem yang tepat)

3. Memenuhi tujuan yang diinginkan dan kebutuhan sistem pada lingkungan operasional (*builds the correct product*)

Berdasarkan (IEEE Std 1012-2016, 2016), hasil dari V&V menyediakan manfaat-manfaat berikut:

1. Memfasilitasi deteksi awal dan koreksi dari aomali
2. Mendukung proses siklus hidup untuk menjami kesesuaian pada performansi program, jadwal, dan anggaran
3. Menyediakan penilaian awal dari performansi sistem
4. Menyediakan bukti objektif terkait kesesuaian untuk mendukung proses sertifikasi
5. Mendukung aktivitas peningkatan proses

Aktivitas V&V berdasarkan (Berztiss dan Ardis, 1988) (IEEE-SA Standards Board dkk., 2013) umumnya fokus pada pengujian perangkat lunak. (Berztiss dan Ardis, 1988) mengkategorisasikan pendekatan V&V dalam 5 kategori, yaitu:

1. *Technical reviews*
2. Pengujian perangkat lunak
3. *Proof of correctness* (verifikasi program)
4. Simulasi dan prototyping
5. *Requirement tracing*

II.1.1 Limitasi Verifikasi dan Validasi (V&V)

Tujuan umum dari pendekatan-pendekatan V&V adalah memastikan produk bebas dari kesalahan dan memenuhi ekspektasi pengguna (Berztiss dan Ardis, 1988). Namun, ada beberapa limitasi yang membuat tujuan ini tidak mungkin dicapai di banyak produk:

1. Teoritis

   (Howden, 1987) menyatakan bahwa tidak ada prosedur pengujian dan analisis yang dapat digunakan untuk membuktikan *correctness* dari program

2. Tidak mungkin melakukan pengujian pada semua kemungkinan data
   Pada kebanyakan program, tidak mungkin untuk mencoba menguji program dengan segala kemungkinan input, karena *combinatorial explosion* (Howden, 1987).
3. Tidak mungkin melakukan pengujian pada semua kemungkinan jalur eksekusi
   Untuk kebanyakan program, tidak mungkin untuk menguji semua kemungkinan jalur eksekusi, karena *combinatorial explosion* (Beizer, 1990). Tidak mungkin juga untuk mengembangkan sebuah algoritma untuk membangkitkan data uji untuk semua jalur, karena ketidakmampuan untuk menentukan jumlah jalur yang layak (Adrion dkk., 1982).
4. Tidak ada *proof of correctness* yang absolut
   (Howden, 1987) menyatakan bahwa tidak ada *proof of correctness* yang absolut. (Howden, 1987) menyarankan *proof of equivalency* yaitu bukti bahwa suatu deskripsi produk ekuivalen dengan deskripsi lainnya. (Beizer, 1990) dan (Howden, 1987) menyatakan, tidak ada klaim proof of *correctness* yang dapat dibuat, kecuali spesifikasi formal dapat dibuktikan benar, dan benar-benar merefleksikan ekspektasi dari pengguna.

II.1.2 Teknik-Teknik pada Verifikasi dan Validasi (V&V)

Aktivitas V&V terjadi selama perangkat lunak berevolusi. Ada banyak teknik dan kakas yang dapat digunakan secara terisolasi atau dikombinasikan dengan teknik lain. Berikut adalah teknik-teknik yang pernah digunakan untuk memverifikasi perangkat lunak berdasarkan literatur (Berztiss dan Ardis, 1988), (Kukimoto, 1996), (Harrison, 2008), (Wiegers, 2002), (Parnas dan Lawford, 2003), (Ciolkowski dkk., 2002), dan (Liu, 2016):

1. *Formal Proof / Proof of Correctness /* Verifikasi Formal
   Formal proof (Harrison, 2008), atau *proof of correctness* (Berztiss dan Ardis, 1988), adalah teknik matematis yang digunakan untuk verifikasi perangkat lunak (Harrison, 2008). Ide utamanya adalah untuk memformalisasi properti dari perangkat lunak kemudian membuktikan

validitasnya dengan menggunakan *rules* dan *axioms*. Verifikasi formal adalah tindakan untuk membuktikan kebenaran suatu sistem berkenaan dengan spesifikasi atau properti formal tertentu (Kukimoto, 1996). Ada beberapa logic yang pernah digunakan dalam literatur, yaitu *propositional logic*, *first-order predicate logic*, *high-order predicate logic*, *Hoare logic* (Liu, 2016). Setiap logic punya kegunaan yang berbeda-beda. Kebanyakan sistem program menggunakan *first-order logic* (Liu, 2016).

Teknik *proof of correctness* umumnya disajikan dalam konteks memverifikasi implementasi terhadap spesifikasi. Teknik ini juga dapat diaplikasikan untuk memverifikasikan produk lain, selama produk tersebut memiliki spesifikasi formal (Berztiss dan Ardis, 1988).

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):

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

Berdasarkan (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).

Menurut (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.

2. Verifikasi Formal : *Model Checking*

Salah satu metode untuk melakukan verifikasi formal adalah *model checking* (Kukimoto, 1996). *Model checking* adalah teknik yang mampu secara otomatis memverifikasi apakah model pada finite automata memenuhi properti yang diekspresikan dalam *temporal logic* (Liu, 2016). 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.

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 (Liu, 2016). Tantangan *model checking* pada verifikasi perangkat lunak adalah sulitnya mengabstraksikan implementasi dari perangkat lunak kedalam *finite automata*, dan penanganan struktur data yang kompleks (Liu, 2016).

3. *Review* Perangkat Lunak

   Review Perangkat Lunak adalah proses atau aktivitas untuk memeriksa dokumentasi perangkat lunak, atau kode perangkat lunak oleh pengembang yang relevan, atau pengguna. 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, 2002), *inspection* (Parnas dan Lawford, 2003), dan *walkthrough* (Ciolkowski dkk., 2002). *Peer review* adalah evaluasi perangkat lunak oleh satu atau lebih developer dengan kompetensi yang sama atau lebih tinggi dari developer yang di-*review* (Wiegers, 2002). Inspeksi adalah proses pemeriksaan apakah perangkat lunak memenuhi standar yang ditentukan, atau *checklist* yang telah dibuat sebelumnya (Parnas dan Lawford, 2003). *Walkthrough* adalah proses membaca algoritma dari perangkat lunak untuk mendeteksi *bugs* dengan cara mencoba memahami perilaku yang diharapkan dari tujuan program (Ciolkowski dkk., 2002).

   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.

4. 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):

a. Pengujian *White-box*

   Berdasarkan (Liu, 2016), 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 penguji, dan hanya mampu memeriksa fungsionalitas yang sudah diimplementasikan pada program, dan tidak mampu memeriksa apakah program telah mengimplementasikan semua fungsi atau konstrain yang dispesifikasikan.

b. Pengujian Black-box

   Pada pengujian *Black-box*, program yang diujikan (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).

c. Pengujian *Grey-box*

   Pengujian *grey-box* adalah kombinasi dari pengujian *white-box* dan pengujian *black-box* (Liu, 2016). Pembuatan kasus uji dilakukan

berdasarkan struktur program dan spesifikasi kebutuhan. Namun, umumnya hanya sebagian kode yang dapat dipelajari oleh penguji. 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 spesifikasi kebutuhan untuk membuat kasus uji.

5. *Testing-Based Formal Verification* (TBFV) (Liu, 2016)

(Liu, 2016) mengusulkan teknik TBFV (*Testing-based Formal Verification*) untuk memeriksa teorema fungsi algoritmik (*algorithmic function theorems*) yang merepresentasikan properti formal. *Algorithmic function theorems* adalah teorema *first-order logic* yang mana fungsi matematis didefinisikan dengan sebuah algoritma, bukan ekspresi matematika. TBFV digunakan untuk memeriksa validitas dari teorema dengan menggunakan *predicate-based testing*.

(Liu, 2016) juga mengusulkan pendekatan *scenario-based* untuk mengaplikasikan teknik TBFV saat melakukan V&V program.

Pendekatan ini melibatkan dua konsep fundamental: *algorithmic function theorem* (AFT) dan *testing-based formal verification* (TBFV):

a. AFT: teorema *first-order logic* yang mana fungsi matematika didefinisikan dengan sebuah algoritma daripada ekspresi matematika.
b. TBFV: Teknik yang dihasilkan dari mengkombinasikan *predicate-based testing* dengan *formal proof* untuk memverifikasi propoerti dari artifak perangkat lunak atau kode. Prinsip utamanya adalah untuk memformalisasi properti kedalam teorema, kemudian memvalidasi teorema untuk tujuan pengujian.

Ekspresi dari properti melibatkan sebagian atau keseluruhan kode sebagai fungsi matematis, dimana fungsi didefinisikan dengan algoritma yang bisa dieksekusi di mesin, tapi sulit diabstraksikan kedalam ekspresi matematis. Manfaat signifikan dari TBFV untuk AFT berdasarkan (Liu, 2016) adalah fakta bahwa kombinasi tersebut menyatukan *formal proof*, *software testing*, dan *software review*, dengan cara: Pengetahuan utama digunakan untuk membentuk teorema fungsi algoritmik yang terverifikasi untuk mendeteksi error dan membangun *confidence* menggunakan prinsip fundamental dari *software testing* dan *software review*.

Pada (Liu, 2016), teknik verifikasi yang digunakan adalah *formal proof*, yang digabungkan dengan *software review*. *Formal proof* telah terbukti efektivitasnya pada *hardware verification*, namun pada punya tantangan yaitu: sulitnya menangani fungsi logika dengan definisi matematis yang minim, dan hanya sedikit praktisi yang mampu melakukan *formal proof*. Teknik verifikasi lain yang bisa digunakan adalah *model checking*, yang dapat dilakukan otomatis dengan memodelkan kode program kemudian melakukan verifikasi terhadap properti.

6. Formal Unit Test / Formal Testing

(Bentes dkk., 2016) mendeskripsikan hasil *preliminary* dari pekerjaan yang menyajikan sebuah metode untuk mengintegrasikan teknik formal verifikasi yang mengadaptasi tools ESC/Java2 dan JCute dengan *unit testing* dengan framework TestNG untuk memverifikasi program Java. Metode ini akan mengekstraksi properti *safety* yang dibangkitkan oleh ESC/Java2, untuk secara otomatis membangkitkan *test cases* menggunakan himpunan *assertion* yang disediakan oleh framework TestNG dan JCute untuk memvalidasi *test case* tersebut.

Berdasarkan (Ghosh dkk., 2013), secara tradisional, kualitas perangkat lunak dijamin lewat *manual testing*, yang sulit dan memberikan *coverage* yang buruk. Salah satu teknik formal adalah *symbolic execution* yang dapat

digunakan untuk secara otomatis membangkitkan *test input* dengan *coverage* struktural yang tinggi. *Symbolic execution* digunakan untuk mencapai *coverage* kode yang tinggi dengan melakukan *reasoning* pada semua nilai input yang mungkin. Hal itu dicirikan dengan pada setiap path dari program, *Symbolic execution* akan melakukan eksplorasi pada kondisi *path* yang dikodekan sebagai sebuah konjungsi dari Boolean clauses. Sebuah path condition mendenotasikan sebuah himpulan dari branching decisions. Ketika ekskusi selesai, beberapa kondisi path mungkin dibangkitkan. Solusi untuk kondisi path adalah input test yang akan menjamin bahwa program yang akan di test.

(Ghosh dkk., 2013) berkontribusi sebuah kakas JST (*Java String Testing*), yaitu sebuah kakas pengujian Java yang komprehensif yang mengalamatkan isu-isu tersebut dalam *traditional symbolic execution engines*. JST berbasis pada *Java PathFinder* (JPF) *model checker* dan *symbolic execution extension*-nya. JPF mengimplementasikan *Java Virtual Machine* milik sendiri untuk mengeksekusi *Java bytecode* yang mana pada penelitian ini diperluas untuk menangani semua *Java primitive type* dan *String*.

7. Testing pada *Model-Driven Development*

   (Marín dkk., 2017) menyajikan T4MDD (*Testing for Model-Driven Development*), yaitu pendekatan testing yang secara otomatis membangkitkan *executable test cases* untuk perangkat lunak yang dikembangkan dengan teknologi MDD. Kontribusi utama dari (Marín dkk., 2017) dibagi ke dalam 2 bagian:

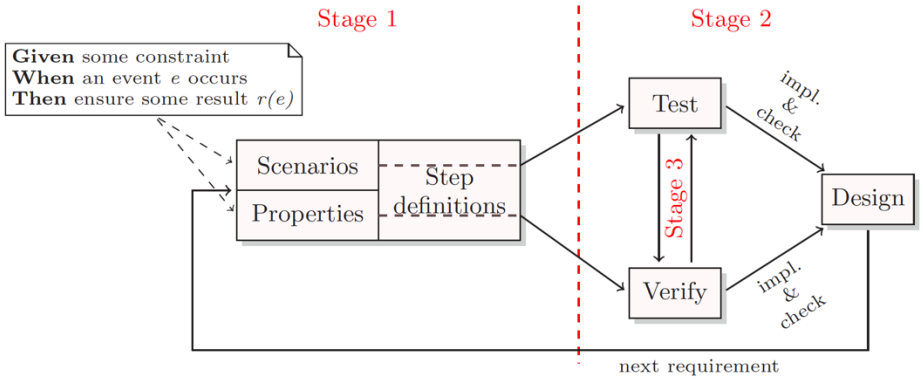
   a. Sebuah teknik *model-based testing* yang secara otomatis mampu membangkitkan *abstract test case* dari model konseptual yang digunakan pada *environment* MDD
   b. Teknik *model-based testing* yang secara otomatis membangkitkan *test case* konkrit dalam bahasa Java dan C#, mereduksi *effort testing* pada MDD *project*

(Mashkoor, 2016) menyajikan hasil awal dan *lesson learned* dari sebuah *ongoing project* yang berfokus pada pengembangan dari sebuah model formal pada *subsystem* dari *software-controlled safety-critical active medical device* (AMD) yang bertanggungjawab pada *renal replacement therapy*. Penggunaan pendekatan formal pada pengembangan AMD sangat direkomendasikan oleh standar dan regulasi, dan memotivasi *advancement* terkini pada metode dan tools terkait seperti Event-B dan Rodin, yang juga diaplikasikan oleh (Mashkoor, 2016).

Kontribusi utama dari (Mashkoor, 2016) adalah menghasilkan model formal yang mendemonstrasikan sebuah contoh bagaimana *requirement* dari software pada AMD modern dapat secara ketat dispesifikasikan melalui rantai *refinement* untuk merepresentasikan *requirement* pada level abstraksi yang berbeda-beda. Selain itu, (Mashkoor, 2016) juga mengarah pada *software safety requirement* yang menjamin *correctness* dari aspek yang dialamatkan pada *behavior*, *support verification* dari spesifikasi berdasarkan analisis. Menghindari *fault* akibat spesifikasi, dan mereduksi ambiguitas pada proses penulisan spesifikasi dengan melibatkan *customer* pada tahap awal *development*.

8. Pengujian dan Verifikasi pada *Behavior-Driven Development* untuk Perancangan Perangkat Keras (Diepenbeck dan Drechsler, 2015)

Penelitian (Diepenbeck dan Drechsler, 2015) menyajikan *flow* pendekatan BDD yang menggabungkan pengujian dan verifikasi untuk rancangan perangkat keras. Menurut penelitian (Diepenbeck dan Drechsler, 2015), skenario pada BDD kurang mencakup seluruh kemungkinan input dan *state*. Beberapa bugs mungkin luput, dan hanya mampu dideteksi oleh metode formal seperti *model checking*. Berikut pada adalah *flow* perancangan perangkat keras yang diusulkan oleh (Diepenbeck dan Drechsler, 2015).



Gambar- II.1 Usulan Pengujian dan Verifikasi pada BDD untuk Perancangan Perangkat Keras oleh Penelitian (Diepenbeck dan Drechsler, 2015)

Tiga tahap utama untuk perancangan perangkat keras dengan BDD pada penelitian (Diepenbeck dan Drechsler, 2015) adalah berikut:

Tahap 1: *Acceptance test* dan *properties*

Pada tahap 1, fitur-fitur dari komponen individual pada komponen perangkat keras dideskripsikan dalam *scenario* atau *properties* dalam struktur kalimat *Given-When-Then*, setiap kalimat disebut *step*.

Tahap 2: BDD untuk pengujian dan verifikasi

Desain diimplementasikan pada tahap 2. Implementasi dikembangkan secara *iterative* dengan menspesifikasikan *step definition* untuk setiap *step* pada tahap 1, kemudian mengimplementasikan rancangan yang didefinisikan pada *step*. *Step definition* mengandung kode pengujian yang mendeskripsikan *behavior* setiap *step* tunggal.

Tahap 3: Membangkitkan *properties* dan pengujian

Skenario BDD hanya mempertimbangkan beberapa input data, dan tidak mencakup seluruh *scenario* secara menyeluruh. *Property* dapat diverifikasi menggunakan *model checker* yang ada. *Property* diperoleh menangkap *verification intent* dan *scenario*, dan dipetakan pada kode uji.

II.1.3 Teknik Verifikasi Formal

Penelitian (Halpern dan Vardi, 1991) memaparkan dibandingkan menyajikan pengetahuan sebagai kumpulan formula, dan melakukan *theorem proving* untuk memverifikasi bahwa formula memenuhi pengetahuan dasar, akan lebih baik juga pengetahuan disajikan sebagai model, dan melakukan *model checking* untuk

memverifikasi bahwa model mewakili pengetahuan. Kesulitan menggunakan teknik *model checking* adalah bagaimana menemukan model, dan bagaimana melakukan *model checking* pada struktur yang besar. Pada penelitian (Halpern dan Vardi, 1991) dijelaskan bahwa *model checking* dapat menangkap *theorem proving*. Kedua teknik punya filosofi yang berbeda dan domain aplikasi yang berbeda. *Theorem proving* lebih cocok digunakan ketika model tidak diketahui dengan pasti, dan lebih mudah mendeskripsikan pengetahuan dalam *axioms*. *Model checking* lebih cocok jika pengetahuan mudah untuk dimodelkan. *Model checking* memungkinkan untuk secara efektif untuk membuktikan deskripsi properti yang harus dipenuhi pada model.

II.1.4 Pengukuran pada Verifikasi dan Validasi (V&V)

Pengelolaan aktivitas V&V menggunakan ukuran untuk menyediakan *feedback* untuk peningkatkan berkelanjutan dari proses V&V dan untuk mengevaluasi proses pengembangan sistem dan produk. Tren dapat diidentifikasi dan dialamatkan dengan menghitung ukuran evaluasi selama periode waktu tertentu. Nilai ambang batas dari sebuah ukuran harus ditentukan, dan tren harus dievaluasi sebagai indikator apakah proses, produk, atau kegiatan V&V telah dilaksanakan dengan sesuai. Berdasarkan (IEEE Std 1012-2016, 2016), ada tiga kategori ukuran yang diasosiasikan dengan usaha V&V: 1) pengukuran untuk mengevaluasi kepadatan anomali, 2) pengukuran untuk mengevaluasi efektivitas V&V, 3) pengukuran untuk mengevaluasi efisiensi V&V.

II.1.4.1 Pengukuran untuk Mengevaluasi Kepadatan Anomali

Pengukuran kepadatan anomali (IEEE Std 1012-2016, 2016) dapat menyediakan informasi yang bermanfaaat terkait kualitas produk, kualitas dari proses pengembangan sistem, dan kualitas dari usaha V&V untuk menemukan anomali pada sistem/perangkat lunak/perangkat keras, dan untuk memfasilitasi koreksi dari anomali. Pengukuran kepadatan anomali dipengaruhi oleh banyak variable, misalnya kompleksitas perangkat lunak, tipe domain, dan fase waktu pada proses V&V. Konsekuensinya, pengukuran dianalisis untuk mendapatkan informasi mengenai interdependensi antara usaha pengembangan dan usaha V&V.

Jika nilai hasil pengukuran kepadatan anomali V&V rendah, maka hal tersebut menunjukkan bahwa kualitas pengembangan program tinggi, atau proses V&V perlu ditingkatkan, atau kombinasi keduanya. Jika nilai hasil pengukuran kepadatan anomali V&V tinggi, maka hal tersebut menunjukkan bahwa kualitas pengembangan program rendah, atau proses V&V efektif, atau kombinasi keduanya

Pengukuran anomali dan tren dapat digunakan untuk meningkatkan kualitas dari proyek, dan dapat digunakan untuk meningkatkan perencanaan dan eksekusi proses V&V untuk proyek berikutnya dengan karakteristik yang sama. Pengukuran didefinisikan dengan empat persamaan berikut, yang diaplikasikan pada empat fase siklus hidup:

$$\text{Kepadatan anomali } requirement = \frac{\#\text{ Anomali } requirement \text{ yang ditemukan V\&V}}{\#\ Requirement \text{ yang direview oleh V\&V}} \quad \text{(II.1)}$$

$$\text{Kepadatan anomali } desain = \frac{\#\text{ Anomali } desain \text{ yang ditemukan V\&V}}{\#\ Statement\ desain \text{ yang direview oleh V\&V}} \quad \text{(II.2)}$$

$$\text{Kepadatan anomali } implementasi = \frac{\#\text{ Anomali } implementasi \text{ yang ditemukan V\&V}}{\#\ Volume\ implementasi \text{ yang direview oleh V\&V}} \quad \text{(II.3)}$$

$$\text{Kepadatan anomali } pengujian = \frac{\#\text{ Anomali } pengujian \text{ yang ditemukan V\&V}}{\#\ Pengujian \text{ yang direview oleh V\&V}} \quad \text{(II.4)}$$

Pada penelitian ini, pengukuran anomali dikenakan pada fase implementasi, sehingga persamaan yang digunakan untuk evaluasi adalah persamaan II.3.

II.1.4.2 Pengukuran untuk Mengevaluasi Efektivitas V&V

Pengukuran efektivitas V&V (IEEE Std 1012-2016, 2016) sangat dipengaruhi oleh usaha pengembangan perangkat lunak, dan usaha V&V. Jika diasumsikan usaha tersebut dilakukan secara pararel, nilai efektivitas V&V yang rendah menunjukkan usaha pengembangan perangkat lunak efektif, atau proses V&V perlu peningkatan, atau kombinasi dari keduanya. Jika nilai efektivitas V&V tinggi, menunjukkan usaha pengembangan perangkat lunak perlu peningkatan, atau proses V&V efektif,

atau kombinasi dari keduanya. Pengukuran didefinisikan dalan empat persamaan berikut:

$$\text{Efektivitas V\&V pada fase } requirement = \frac{\#\text{ Anomali } requirement \text{ yang ditemukan V\&V}}{\#\text{ Anomali } requirement \text{ yang ditemukan semua sumber}} \quad \text{(II.5)}$$

$$\text{Efektivitas V\&V pada fase } desain = \frac{\#\text{ Anomali } desain \text{ yang ditemukan V\&V}}{\#\text{ Anomali } desain \text{ yang ditemukan semua sumber}} \quad \text{(II.6)}$$

$$\text{Efektivitas V\&V pada fase } implementasi = \frac{\#\text{ Anomali } implementasi \text{ yang ditemukan V\&V}}{\#\text{ Anomali } implementasi \text{ yang ditemukan semua sumber}} \quad \text{(II.7)}$$

$$\text{Efektivitas V\&V pada fase } pengujian = \frac{\#\text{ Anomali } pengujian \text{ yang ditemukan V\&V}}{\#\text{ Anomali } pengujian \text{ yang ditemukan semua sumber}} \quad \text{(II.8)}$$

Pada penelitian ini, nilai efektivitas V&V yang pengukuran anomali dikenakan pada fase implementasi, sehingga persamaan yang digunakan untuk evaluasi adalah persamaan II.7.

II.1.4.3 Pengukuran untuk Mengevaluasi Efisiensi V&V

Pengukuran yang diasosiasikan dengan efisiensi usaha V&V (IEEE Std 1012-2016, 2016) menyediakan data yang mengkarakteristikan kapabilitas dari usaha V&V untuk menemukan anomali pada perangkat lunak dan proses dalam aktivitas pengembangan. Manfaat maksimum dirasakan ketika anomali ditemukan sedini mungkin pada siklus pengembangan perangkat lunak, sehingga biaya pengembangan dapat diminimalisir.

Nilai efisiensi V&V yang rendah menunjukkan bahwa usaha V&V tidak berhasil menemukan anomali sedini mungkin dalam suatu fase aktivitas, atau produk yang dikembangkan belum matang, atau kombinasi dari keduanya. Jika pengukuran efisiensi tinggi, maka nilai tersebut menunjukkan usaha V&V berhasil menemukan anomali sedini mungkin, atau pengembangan produk sudah matang, atau kombinasi dari keduanya. Pengukuran didefinisikan dalam persamaan II.9 sampai II.12.

$$\text{Efisiensi V\&V pada fase } requirement = \frac{\#\text{ Anomali } requirement \text{ yang ditemukan V\&V di aktivitas requirement}}{\#\text{ Anomali } requirement \text{ yang ditemukan semua aktivitas}} \; x\ 100\% \quad \text{(II.9)}$$

$$\text{Efisiensi V\&V pada fase } desain = \frac{\#\text{ Anomali } desain \text{ yang ditemukan V\&V di aktivitas desain}}{\#\text{ Anomali } desain \text{ yang ditemukan semua aktivitas}} \; x\ 100\% \quad \text{(II.10)}$$

$$\text{Efisiensi V\&V pada fase } implementasi = \frac{\text{Anomali } implementasi \text{ yang ditemukan VV di aktivitas implementasi}}{\text{Anomali } implementasi \text{ yang ditemukan semua aktivitas}} \text{ x } 100\% \quad \text{(II.11)}$$

$$\text{Efisiensi V\&V pada fase } pengujian = \frac{\text{\# Anomali } pengujian \text{ yang ditemukan V\&V di aktivitas pengujian}}{\text{\#Anomali } pengujian \text{ yang ditemukan semua aktivitas}} \; x \; 100\% \quad \text{(II.12)}$$

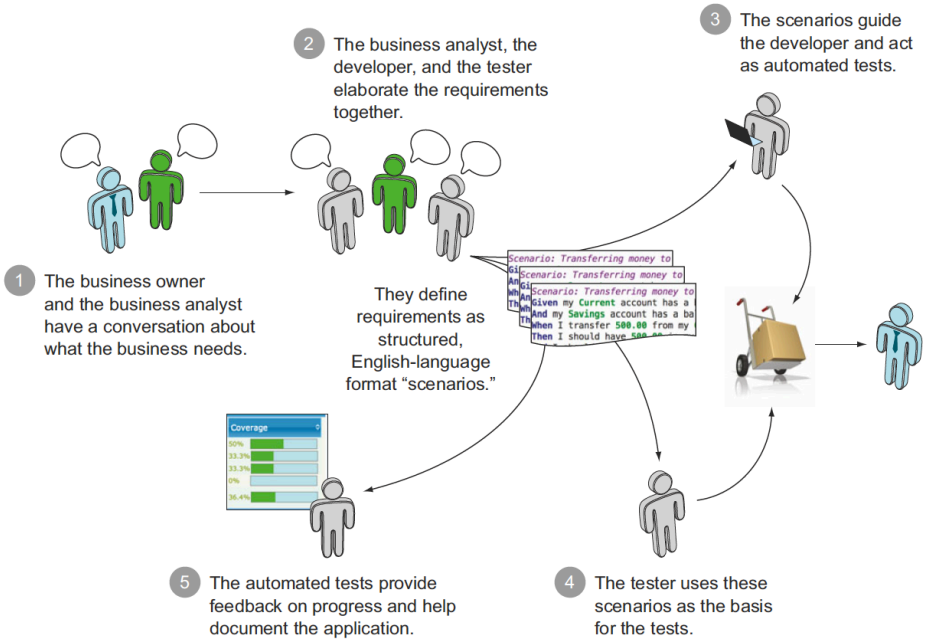
Pada penelitian ini, nilai efisiensi usaha V&V yang pengukuran anomali dikenakan pada fase implementasi, sehingga persamaan yang digunakan untuk evaluasi adalah persamaan II.11.

II.2 *Behavior-Driven Development*

Behavior-Driven Development (BDD) adalah sebuah proses pengembangan perangkat lunak muncul dari pendekatan *Test-Driven Development* (TDD). BDD dikembangkan oleh Dan North sebagai tanggapan dari masalah-masalah yang ada pada TDD (Solis dan Wang, 2011). BDD menggabungkan teknik-teknik dan prinsip TDD dengan *Domain-Driven Design*.

Berdasarkan referensi (Agile Alliance, 2015), 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).

Gambar II-1 menjelaskan proses pengembangan BDD berdasarkan (Ferguson, 2015).



Gambar II-1 Proses Pengembangan dengan BDD (Ferguson, 2015)

Pada tim yang menggunakan proses pengembangan BDD, *business analyst*, *tester*, dan *developer* berkolaborasi dan mendefinisikan *requirement* bersama-sama. Mereka menggunakan sebuah bahasa (DSL) yang mampu digunakan untuk mendefinisikan *requirement* dan sebagai basis untuk *automated test*. Pengujian tersebut mendefinisikan bagaimana perangkat lunak harus berperilaku dan membimbing *developer* dalam membangun perangkat lunak (Ferguson, 2015). Berikut adalah langkah-langkah pada proses pengembangan BDD seperti pada Gambar II-1 berdasarkan (Ferguson, 2015):

1. *Stakeholder* berdiskusi dengan *business analyst* terkait fitur yang diinginkan. Untuk mengurangi ambiguitas, diskusi dilakukan dengan menyediakan contoh konkrit mengenai apa yang sebuah fitur harus lakukan.
2. *Business analyst*, *developer*, dan *tester* berdiskusi mengenai *feature*, dan mentranslasikan *feature* menjadi *requirement* atau *scenario* yang dituliskan dalam DSL, misalnya Gherkin (Subbab II.4.1).
3. Developer menggunakan kakas BDD untuk mengubah *requirement* menjadi *automated test* yang dijalankan terhadap kode, dan membantu menentukan secara objektif apakah sebuah fitur selesai.

4. *Tester* menggunakan hasil dari pengujian sebagai basis dari *manual test* dan *exploratory test*.
5. *Automated test* bertindak sebagai *low-level technical documentation*, dan menyediakan contoh bagaimana sistem bekerja.

Berdasarkan penelitian (Solis dan Wang, 2011), 7 kakas yang umum digunakan untuk V&V pada proses pengembangan dengan BDD, yaitu: `Cucumber`[1], `Specflow`[2], *xBehave family* (`JBehave`[3], dan `NBehave`[4]), *xSpec family* (`RSpec`[5], dan `MSpec`[6]), dan `StoryQ`[7]. Penelitian (Solis dan Wang, 2011) menyajikan 6 karakteristik utama dari proses pengembangan BDD. Karakteristik tersebut adalah:

1. *Ubiquitous Language*

   Ubiquitous Language adalah sebuah bahasa yang strukturnya berasal dari *domain model*. Bahasa ini mengandung *term* yang digunakan untuk mendefinisikan *behavior* dari sistem. Pada fase implementasi, pengembang akan menggunakan bahasa tersebut untuk menamakan kelas-kelas dan *method*.
2. *Iterative Decomposition Process*

   Pada BDD, analisis dimulai dengan identifikasi dari *behavior* yang diharapkan dari sistem. *Behavior* sistem diturunkan dari *business outcomes*, yang digali menjadi *feature sets*. Sebuah *feature* kemudian direalisasikan dengan *user story*. *Scenarios* pada BDD harus mendeskripsikan konteks spesifik dan *outcome* dari *user story*. Skenario pada BDD digunakan sebagai *acceptance criteria*.

[1] https://cucumber.io/
[2] http://specflow.org
[3] http://jbehave.org
[4] https://github.com/nbehave
[5] http://rspec.info
[6] http://nspec.org
[7] https://storyq.codeplex.com

3. Deskripsi *User Story* dan *Scenario* dengan *Plain Text* dan *Template*

   Pada BDD, deskripsi dari *features*, *user stories*, dan *scenario* dispesifikasikan dalam *template*. *User stories* umumnya dispesifikasikan dengan *template* berikut:

   [*Story title*] Satu kalimat yang mendeskripsikan *story*
   As a [Role]
   I want a [Feature]
   So that I can get [Benefit]

   Template untuk menulis *scenario* adalah sebagai berikut:

   Scenario 1: [Judul Skenario]
   Given [context]
   And [context]…
   When [event]
   Then [outcome]
   And [outcome]…
   Scenario 2: [Judul Skenario]
   …

   Scenario mendeskripsikan bagaimana sistem yang mengimplementasi *feature* harus berperilaku ketika ada *state* atau *event* tertentu. *Outcome* dari *scenario* adalah sebuah aksi yang mengganti *state* dari sistem, atau menghasilkan *output*.

4. *Automated Acceptance Testing* dengan *Mapping Rules*

   BDD mewarisi karakteristik dari *Automated Acceptance Testing* dari ATDD. Sebuah *acceptance test* pada BDD adalah sebuah spesifikasi dari *behavior* pada sistem. Sebuah *acceptance test* juga adalah *executable specification*.

   Developer akan memulai dari *scenario* yang dihasilkan dari proses *iterative decomposition*. *Scenario* yang mencakup *steps* akan ditranslasikan menjadi kode pengujian. BDD memungkinkan skenario *plain text* yang *executable*.

   Mapping rules menyediakan sebuah standar untuk memetakan *scenario* menjadi kode pengujian. Pada kakas `JBehave`, *user story* adalah sebuah *file* yang mengandung himpunan dari *scenario*. Nama dari *file* dipetakan

pada sebuah kelas *user story*. Setiap *scenario step* dipetakan pada *test method* yang ditempatkan menggunakan sebuah anotasi yang mendeskripsikan *step*, dan biasanya *test method* punya nama yang sama dengan *annotation text*. Berbeda dengan `JBehave`, kakas `Cucumber` menggunakan *regular expression* untuk melakukan pemetaan.

5. *Readable Behavior Oriented Specification Code*

   BDD menyarankan bahwa kode harus menjadi bagian dari dokumentasi sistem, yang juga sejalan dengan prinsip Agile. Kode harus *readable*, dan spesifikasi harus menjadi bagian dari kode. Nama dari *method* harus mengindikasikan apa yang *method* harus lakukan. Nama dari kelas dan *method* dituliskan dalam kalimat. Kode harus mendeskripsikan *behavior* dari objek. Aplikasi dari mapping rules membantu menghasilkan kode yang *readable*. Hal ini memastikan bahwa nama kelas dan nama *method* sama dengan judul *user story* dan judul *scenario*.

   `StoryQ`, `RSpec`, dan `MSpec` menyediakan API yang mengizinkan developer untuk menspesifikasikan *user stories* dan *scenario* sebagai *behavior-driven code*. `JBehave` dan `NBehave` juga membantu menulis *scenario* sebagai kode dan membuat kode *readable* dengan memanfaatkan anotasi. Sedangkan `Cucumber` tidak berfokus pada level implementasi dan tidak mendukung karakteristik ini.

6. *Behavior-driven* pada fase yang berbeda-beda

   Behavior-driven terjadi pada fase yang berbeda-beda pada pengembangan perangkat lunak menggunakan pendekatan BDD. *Automated acceptance testing* bagian penting pada implementasi dengan pendekatan BDD. Kelas-kelas pengujian diturunkan dari *scenario* dan nama mereka mengikuti aturan dari *mapping rules*. Dengan cara ini, nama kelas akan menspesifikasikan apa yang kelas harus lakukan atau apa *behavior* dari kelas.

Tabel II-1 berikut merangkum dukungan dari kakas BDD pada 6 karakteristik BDD.

Tabel II-1 Dukungan Karakteristik BDD pada Kakas (Solis dan Wang, 2011)

Karakteristik BDD		**JBehave**	**NBehave**	**RSpec**	**MSpec**	**StoryQ**	**Cucumber**	**SpecFlow**
Ubiquitous Language		x	x	x	x	x	x	x
Proses *Iterative Decomposition*		x	x	x	x	x	x	x
Plain Text berbasis	*User Story Template*	√	√	x	x	x	√	√
	Scenario Template	√	√	x	x	x	√	√
Automated Acceptance Testing dengan *Mapping Rules*		√	√	√	x	x	√	√
Readable Behavior Oriented Specification Code		√	√	√	√	√	x	√
Behavior driven pada fase	*Planning*	x	x	x	x	x	x	x
	Analysis	√	√	x	x	x	√	√
	Implementation	√	√	√	√	√	x	√

BDD berdasarkan (JBehave) berkisar antara **Story** yang mewakili fungsionalitas bisnis, dan didefinisikan dalam file yang mampu dieksekusi. Pada intinya Story meliputi satu atau lebih ***scenario***, yang mana setiap *scenario* merepresentasikan contoh konkrit dari *behavior* sistem. Setiap *scenario* meliputi sejumlah ***executable steps***. Steps dapat terdiri, tapi tidak terbatas pada, tiga jenis: **Given**, **When**, dan **Then**. Given, When, dan Then juga disebut dengan kata kunci BDD. Gambar II-2

```
A story is a collection of scenarios

Narrative:
In order to communicate effectively to the business some functionality
As a development team
I want to use Behaviour-Driven Development

Scenario:  A scenario is a collection of executable steps of different type

Given step represents a precondition to an event
When step represents the occurrence of the event
Then step represents the outcome of the event

Scenario:  Another scenario exploring different combination of events

Given a precondition
When a negative event occurs
Then a the outcome should be captured
```

Gambar II-2 Contoh Konten Berkas **Story**

II.2.1 *Test-Driven Development*

Berdasarkan (Mishra, 2017), 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):

a. *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.

b. 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.

c. *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.

d. 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* tidak selalu harus melibatkan beberapa objek.

e. 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*.

f. Test Suite

Test suite adalah sebuah koleksi dari berkas skrip pengujian.

g. Assertions

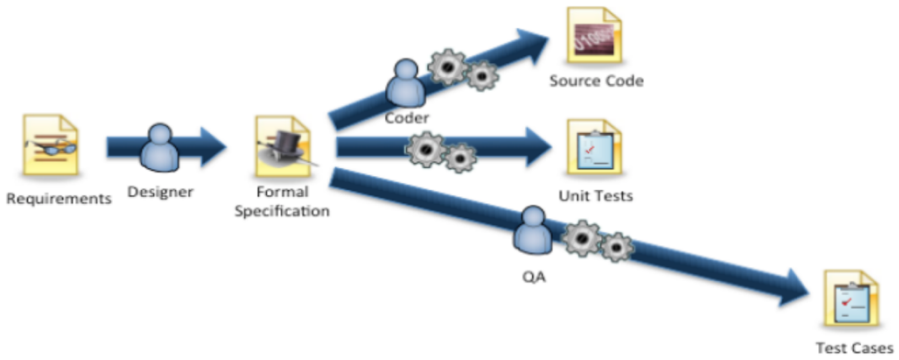
Assertions digunakan pada *state verification* dan *interaction tests*. Sebuah *assertion* merepresentasikan sebuah kesalahan (*failure*) pada unit test.

II.2.2 Perbedaan BDD dan TDD

Berdasarkan referensi (Mishra, 2017), perbedaaan utama antara BDD dan TDD yaitu 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.

II.3 *Formal Specification-Driven Development (FSDD)*

Penelitian (Rutledge dkk., 2014) menyajikan pendekatan Formal Specification-Driven Development yang mengadaptasi teknik TDD dan BDD lewat tambahan artifak, yaitu spesifikasi formal didefinisikan dalam bahasa spesifikasi *behavioral*. Teknik FSDD tidak mempengaruhi model proses perangkat lunak, tetapi hanya modifikasi prinsip perancangan artifak perangkat lunak. Dengan pendekatan ini, spesifikasi formal dihasilkan dalam bahasa spesifikasi *behavioral* (misalnya: JML, Dafny, Spec#), yang menjadi basis bagi implementasi dan pengujian. Gambar II-3 adalah rancangan artifak FSDD yang disajikan pada penelitian (Rutledge dkk., 2014).



Gambar II-3 Artifak FSDD (Rutledge dkk., 2014)

Berikut adalah alur kerja FSDD (Rutledge dkk., 2014):

1. Designer mentranslasikan requirements *behavioral* menjadi *formal specification* yang dituliskan dalam bahasa spesifikasi *behavioral* (misalnya: JML)
2. *Coder* menuliskan kode perangkat lunak secara manual. *Coder* dapat memulai lewat *stub function* yang dibangkitkan dari *formal specification*
3. Pengembang dibantu lewat himpunan pengujian unit yang dibangkitkan secara otomatis lewat FSDD unit testing framework.
4. *Quality Assurance* menganalisis dan menguji sistem menggunakan kasus uji yang dibuat secara manual

Penelitian (Fofung dan Duggins, 2015) menyajikan metrik kuantitatif terhadap pendekatan FSDD. Berdasarkan penelitian ini, pengembang perangkat lunak yang menerapkan pendekatan FSDD cenderung meningkatkan beberapa aspek kualitas perangkat lunak, jika dibandingkan dengan SDLC tradisional. FSDD unggul pada metric berikut: *Cyclomatic Complexity* (CCP), *Depth of Inheritance* (DOI), dan *Maintence Index* (MI).

Penelitian (Carter, 2017) menyajikan BHive, yaitu menambahkan formal model dalam pengujian BDD, yang akan menyediakan wawasan tambahan terkait validitas pengujian, dan meningkatkan visibilitas dari problem domain. BHive memetakan skenario BDD "*Given-When-Then*" menjadi "*Precondition-Command-Postcondition*" yang dibangun dengan Floyd-Hoare *Logic*. Pemetaan ini mengizinkan representasi B-Method untuk dibuat, dan model ini bermanfaat untuk

mengeksplorasi *behavior* sistem dan menunjukkan sela yang ada antara *requirement* dan kasus uji (Rutledge dkk., 2014).

II.4 *Domain-Specific Language* (DSL)

Domain-specific languages (DSLs) adalah bahasa yang disesuaikan untuk domain aplikasi yang spesifik (Hudak, 1997; Karsai dkk., 2014; Mernik dkk., 2005). DSL dikenal juga dengan nama *application-oriented*, *special purposes*, *specialized*, *task-specific*, atau *application language*. DSL 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 II-2 adalah beberapa DSL yang digunakan secara luar beserta domain aplikasinya.

Tabel II-2 Beberapa DSL yang digunakan secara luas (Mernik dkk., 2005)

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

Fitur dari DSL adalah sebagai berikut (Mernik dkk., 2005):

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

2. DSL dibangun diatas kosakata pengguna dari domain
3. Sintaks yang disediakan DSL menyembunyikan aspek inheren pemrograman aplikasi dari *client*

(Hooman, 2016) menyajikan penggunaan teknik formal untuk meningkatkan *confidence* tentang *correctness* dari sebuah model dengan DSL. Metode ini menyembunyikan detail matematis dari pengguna. (Hooman, 2016) menyatakan DSL menyediakan pendekatan yang *lightweight* untuk *incorporate formal technique* pada *industrial workflow*. Dari DSL *instances*, *formal models* dan artefak lain dapat dibangkitkan, misalnya simulasi model dan kode. Banyak perusahaan mengalami kesulitan dengan fase integrasi dan pengujian yang panjang, alasannya adalah:

a. Banyak masalah yang terdeteksi pada fase ini
b. Perbaikan dari masalah yang timbul sering kali tidak mudah, dan mungkin akan mengakibatkan masalah baru

Akibatnya, sulit untuk mengelola fase ini dan memprediksi kapan fase ini akan selesai. Pendekatan yang diusulkan pada (Hooman, 2016) bertujuan untuk mendeteksi masalah lebih awal pada proses pengembangan.

II.4.1 Gherkin Language

Gherkin Language berdasarkan (Rose dkk., 2015) adalah bahasa yang dipahami oleh Cucumber. Gherkin adalah sebuah DSL yang mengizinkan pengembang untuk mendeskripsikan perilaku dari perangkat lunak tanpa memperinci bagaimana perangkat lunak tersebut diimplementasikan (Rose dkk., 2015). 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

II.4.1.1 Sintaks

Gherkin adalah Bahasa berorientasi baris, seperti Python dan YAML, yang menggunakan indentasi untuk mendefinisikan struktur (Rose dkk., 2015). 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.

```
 1: Feature: Some terse yet descriptive text of what is desired
 2:   Textual description of the business value of this feature
 3:   Business rules that govern the scope of the feature
 4:   Any additional information that will make the feature easier to understand
 5:
 6:   Scenario: Some determinable business situation
 7:     Given some precondition
 8:       And some other precondition
 9:     When some action by the actor
10:       And some other action
11:       And yet another action
12:     Then some testable outcome is achieved
13:       And something else we can check happens too
14:
15:   Scenario: A different situation
16:       ...
```

Gambar II-4 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.

II.4.1.2 Berkas *Feature*

Setiap berkas .feature secara konvensional terdiri dari fitur tunggal dari sebuah perangkat lunak (Rose dkk., 2015). 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 skenario. 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.

```
Feature: Serve coffee
  Coffee should not be served until paid for
  Coffee should not be served until the button has been pressed
  If there is no coffee left then money should be refunded

Scenario: Buy last coffee
  Given there are 1 coffees left in the machine
  And I have deposited 1$
  When I press the coffee button
  Then I should be served a coffee
```

Gambar II-5 Contoh Berkas *Feature*

II.4.1.3 Step Definition

Untuk setiap langkah (*steps*), Cucumber akan mencari *step definition* yang cocok (Rose dkk., 2015). *Step definition* dituliskan dalam Bahasa Ruby. Setiap *step definition* terdiri dari string atau *regular expression*. Gambar berikut adalah contoh *step definition*.

```
# features/step_definitions/coffee_steps.rb

Given /there are (\d+) coffees left in the machine/ do |n|
  @machine = Machine.new(n.to_i)
end
```

Gambar II-6 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.

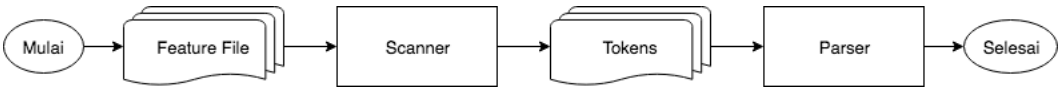
II.4.2 Gherkin

Gherkin adalah *parser* dan *compiler* untuk Gherkin Language (Cucumber, 2018). Gherkin saat ini diimplementasikan untuk platform berikut:

1. NET
2. Java
3. JavaScript
4. Ruby
5. Go

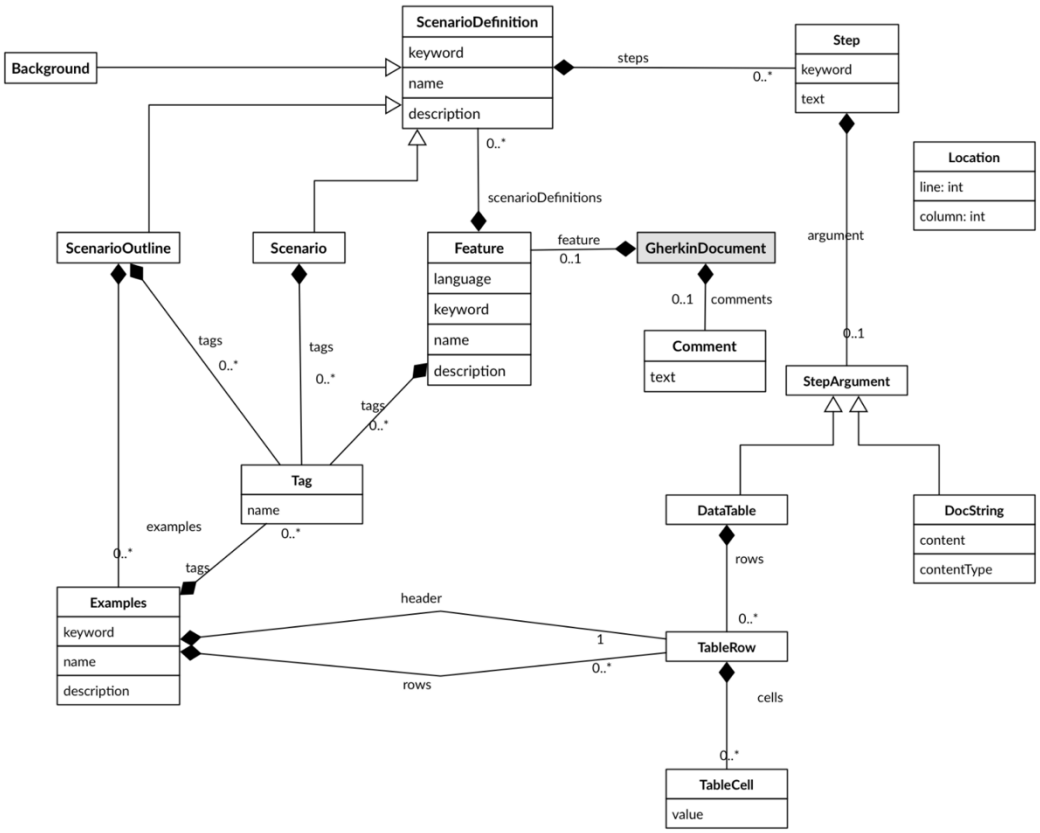
6. Python
7. Objective-C
8. Perl

Gherkin dapat digunakan sebagai *Command Line Interface* (CLI) atau sebagai *library*. Gherkin CLI `gherkin` membaca berkas Gherkin (berkas .features) dan mengeluarkan hasil berupa AST dan Pickles. Program `gherkin` mengambil sejumlah berkas sebagai argument dan menampilkan hasilnya sebagai *Newline Delimited JSON* (setiap line adalah dokumen JSON). Gambar berikut adalah arsitektur Gherkin dalam diagram alur berdasarkan (Cucumber, 2018).



Gambar II-7 Asitektur Gherkin (Cucumber, 2018)

Scanner membaca dokumentasi Gherkin yang berupa *Feature File*, kemudian membuat *token* untuk setiap barisnya. Setiap *token* dilewatkan pada *parser*, yang akan menghasilkan AST (*Abstract Syntax Tree*). AST yang dihasilkan oleh *parser* dapat dideskripsikan lewat *class diagram* pada Gambar II.8. Setiap kelas merepresentasikan sebuah node pada AST. Setiap node memiliki location yang mendeskripsikan nomor baris dan nomor kolom pada input file.



Gambar II-8 AST yang dihasilkan oleh *parser* (Cucumber, 2018)

II.5 Specification Pattern

TBD

II.6 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 (Li Tan dkk., 2004). 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 dan Vardi, 1986), LTL diperluas penggunaannya ke *model checking*. Persoalan yang ditangani oleh *model checking* adalah verifikasi: apakah *finite-state program P* memenuhi spesifikasi $\propto$.

II.6.1 Sintaks LTL

Berikut adalah notasi dasar dan definisi pada LTL berdasarkan referensi (Kröger dan Merz, 2008):

1. Proposisi atomic $P = \{p_o, p_1, \dots\}$, adalah formula LTL
2. Jika φ_1 dan φ_2 adalah LTL formula, maka $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$ adalah LTL formula
3. Jika φ_1 dan φ_2 adalah LTL formula, maka Xφ_1, φ_1Uφ_2, Gφ_1, Fφ_1 adalah LTL formula, dimana X, G, F, U merepresentasikan "next", "globally", "eventually", dan "until"
4. Operator lain yang digunakan pada LTL adalah R (Release) φ_1Rφ_2

II.6.2 Semantik LTL

Berikut adalah semantik dari LTL berdasarkan referensi (Kröger dan Merz, 2008):

1. *Next* (X), X_φ adalah *true* pada s_t jika dan hanya jika φ true pada s_{t+1}
2. *Finally* (F), F_φ adalah *true* pada s_t jika dan hanya jika φ true pada beberapa $s_{t'}$ dimana t' > t
3. *Globally* (G), X_φ adalah *true* pada s_t jika dan hanya jika φ true pada semua $s_{t'}$ dimana t' > t
4. *Until* (U), $\varphi U \phi$ adalah *true* pada s_t jika dan hanya jika pada beberapa state $s_{t'}$ dimana t' ≥t:
   - ϕ *true* pada $s_{t'}$ dan
   - φ *true* pada semua *state* $s_{t''}$, t < t'' < t'
5. *Release* (R), $\varphi R \phi$ adalah *true* pada s_t jika dan hanya jika untuk semua state $s_{t'}$ dimana t' ≥t:
   - ϕ *true* pada $s_{t'}$ dan
   - φ *true* pada beberapa *state* $s_{t''}$, t < t'' < t'

   ϕ akan *false* jika φ bernilai *true* lebih dulu

II.7 *Correctness* pada program

Berdasarkan (Berztiss dan Ardis, 1988; Collofello, 1988), dalam konteks program komputer, *proof* dari *correctness* sebuah program adalah demonstrasi bahwa program tersebut konsisten dengan spesifikasi. *Functional correctness* berdasarkan (Dunlop dan Basili, 1982) merujuk pada *behavior* input-output dari program, misalnya untuk setiap input akan menghasilkan output yang diharapkan. Pada *formal proof*, sebuah program dianggap sebagai *formal object*, misalnya sebuah string pada sebuah bahasa, yang memiliki *formal syntax* dan *formal semantics* (Berztiss dan Ardis, 1988). Demikian pula spesifikasi, harus dituliskan dalam *formal language*. Dua level pembuktian dibedakan menjadi:

1. *Total proof*: *total proof* dari sebuah program menunjukkan bahwa program konsisten dengan spesifikasi
2. *Partial* atau *conditional proof*: menunjukkan bahwa program konsisten dengan spesifikasi jika eksekusi dari program mencapai terminasi

II.8 Kakas BDD

Pada bagian ini dijelaskan kakas-kakas yang digunakan pada proses pengembangan perangkat lunak dengan *Behavior-Driven Development* (BDD). Sesuai dengan batasan masalah pada subbab I.4, pada bagian ini hanya dijelaskan kakas-kakas yang mendukung bahasa pemrograman Java, yaitu: `Cucumber` dan `JBehave`.

II.8.1 Cucumber

Cucumber adalah sebuah kakas yang mendukung BDD. Cucumber mengeksekusi *executable specification* yang dituliskan dalam DSL yang bernama Gherkin, dan menghasilkan laporan yang mengindikasikan apakah perangkat lunak berperilaku sesuai dengan yang dispesifikasikan atau tidak (Cucumber.io, 2018). Cucumber dituliskan dalam bahasa Ruby, tetapi dapat digunakan untuk menguji kode yang dituliskan dalam bahasa lain. Berikut adalah bahasa pemrograman yang didukung Cucumber:

1. Ruby
2. JRuby
3. Java (menggunakan Cucumber-JVM)
4. Groovy

5. Javascript
6. Clojure
7. Gosu
8. Lua
9. .NET (menggunakan Specflow)
10. PHP (menggunakan Behat)
11. C++
12. TCL

Cucumber juga mampu diintegrasikan dengan *framework* untuk pengujian terotomasi lain. Tabel berikut adalah daftar *framework* yang mampu diintegrasikan dengan Cucumber (Cucumber.io, 2018).

Tabel II-3 Daftar Framework yang Mampu Diintegrasikan dengan Cucumber

No	*Framework*	Deskripsi
1	Ruby on Rails	Dengan menggunakan Cucumber-Rails membangkitkan dan memodifikasi files pada *project* Rails sehingga dapat digunakan dengan Cucumber
2	Selenium	*Browser Automation*
3	Pico Container	*Dependency Injection*
4	Spring Framework	*Dependency Injection*
5	Watir	Pengujian Aplikasi web di Ruby
6	Capybara	*Framework* untuk menguji aplikasi web dengan mensimulasikan bagaimana pengguna berinteraksi dengan aplikasi
7	Serenity	Pustaka untuk *reporting*

II.8.2 JBehave

JBehave adalah sebuah *framework* untuk BDD. JBehave secara kuat terintegrasi pada JVM, dan telah digunakan secara luas oleh tim pengembang berbasis Java yang mengimplementasikan BDD (JBehave, 2018). JBehave dirancang untuk implementasi *scenario* di bahasa pemrograman Java. Pengujian JBehave dijalankan lewat `JUnit Runner.`

Pada JBehave, *acceptance criteria* mampu diotomasi dengan menuliskan *test stories* dan *scenario* menggunakan notasi BDD "Given-When-Then". Skenario

dituliskan dalam berkas .`story`: berkas `story` dirancang untuk mencakup semua *scenario* (*acceptance criteria*) dari *user story*. Sebuah berkas `story` juga dapat mengandung deskripsi naratif yang menjelaskan latar belakang dan konteks dari `story` yang akan diuji.

JBehave juga mampu diintegrasikan dengan *framework* untuk pengujian terotomasi lain. Tabel berikut adalah daftar *framework* yang mampu diintegrasikan dengan JBehave (JBehave, 2018).

Tabel II-4 Daftar Framework yang Mampu Diintegrasikan dengan JBehave (JBehave, 2018)

No	***Framework***	**Deskripsi**
1	Selenium	*Browser Automation*
2	Pico Container	*Dependency Injection*
3	Spring Framework	*Dependency Injection*
4	Guice	*Dependency Injection*
5	Needle	*Dependency Injection*
6	Weld	*Dependency Injection*
7	Thucydides	Pustaka untuk *reporting*

II.8.3 Perbandingan JBehave dan Cucumber

(Kolesnik, 2013) membandingkan kakas JBehave dan Cucumber-JVM, yang kemudian diperbarui oleh (Gamage, 2017). (Kolesnik, 2013) membandingkan dan dalam aspek-aspek yang yang dijelaskan pada (Gamage, 2017; Kolesnik, 2013). Digunakan 4 skala nilai berikut untuk yang digunakan untuk mengevaluasi:

0: Fitur tidak didukung

1: Fitur tersedia dengan batasan

2: Fitur tersedia namun tanpa opsi tambahan

3: Dukungan fitur penuh

Tabel II-5 Perbandingan JBehave dan Cucumber (Gamage, 2017; Kolesnik, 2013)

Aspek	**JBehave**	**Cucumber-JVM**
Dokumentasi	3	3
Fleksibilitas dalam melewatkan *parameter*	2	3
Auto-complete	3	3
Scoping	3	2
Composite steps	3	0
Background and Hooks	0	3
Binding to code (annotations)	3	3
Formatting flexibility	2	3
Built-in reports	2	3
Comformity to Gherkin standard	3	3
Input data sources	3	0
Total	27	26

Berikut adalah penjelasan dari masing-masing aspek penilaian yang dibandingkan.

1. *Documentation*: dinilai berdasarkan ada atau tidaknya *website* resmi, dokumentasi API, contoh kode, dan forum. Dokumentasi Cucumber tersebar pada beberapa *platform*, yang terdokumetasi secara baik. Sedangkan JBehave lebih mengkhususkan ke bahasa pemrograman Java,
2. Fleksibilitas dalam melewatkan *parameter*: dinilai berdasarkan tersedia atau tidaknya fitur berikut: *parameter variant*, *parameter injection*, *tabular parameter*, *multi-line input*
3. *Auto-complete*: JBehave untuk Eclipse mendukung *auto-complete* pada *feature file*. Cucumber untuk Eclipse mendukung *auto-complete* dan *text-highlight* pada *feature file*
4. *Scoping*: *Scoping* dinilai berdasarkan ada atau tidaknya *test scope* (mengidentifikasi *sub-sets test* untuk dijalankan: *feature scope*, atau *scenario scope*), dan *step-scope*
5. *Composite steps*: fitur ini bermanfaat jika dibutuhkan *re-use steps*. Composite steps pada JBehave adalah built-in *feature*, sedangkan pada

Cucumber secara *default* tidak tersedia, dan dapat tersedia dengan melakukan *workaround*.

6. *Background and Hooks*: Aspek ini mencakup kemampuan untuk melakukan aksi tambahan setelah atau sebelum event utama (contoh: Before/After run)
7. *Binding to code (annotations)*: JBehave dan Cucumber menggunakan pendekatan yang sama mengikat instruksi teks pada kode, yaitu dengan menggunakan anotasi.
8. *Formatting flexibility*: JBehave mendukung formatting dengan mengimplementasikan method loadStoryAsText pada interface StoryLoader. Sedangkan Cucumber secara default mendukung *formatting flexibility*
9. *Built-in reports*: JBehave dan Cucumber mendukung *console output*, *structured file* (misalnya XML), format HTML, dan Usage Report. Namun, JBehave tidak menyediakan *pretty console output*.
10. *Comformity to Gherkin standard*: Kedua kakas mendukung standar Gherkin
11. *Input data sources*: JBehave punya interface StoryLoader, dan dapat diimplementasikan method loadStoryAsText. Pada implementasinya dapat didefinisikan untuk membaca data dari *source* manapun.

II.9 Kakas Anotasi di Java

Berikut dijelaskan kakas-kakas yang digunakan untuk menganotasikan properti formal pada *source code* Java.

II.9.1 Java Annotation

TBD

II.9.2 Java Modelling Language (JML)

JML → Hoare
Tapi ada yang extend JML dengan *temporal logic* → (Trentelman dan Huisman, 2002)
TBD

II.10 Kakas *Model Checking*

Berikut dijelaskan kakas-kakas yang digunakan untuk mengeksekusi model checking pada program atau *source code* Java.

II.10.1 Java Path Finder (JPF)

JPF → *model checker*. Input = Java Bytecode
Verifikasi: *concurrency defect* seperti *deadlocks*, dan *unhandled exception* seperti NullPointerException dan AssertionError
TBD

II.10.2 Bandera

Kakas Bandera adalah koleksi terintegrasi dari komponen *program analysis*, *transformation*, *visualization* yang dirancang untuk mengizinkan eksperimen dengan properti *model-checking* pada *source code* Java. Bandera mengambil Java *source code* sebagai input, dan mengaplikasikan optimisasi pada pada model seperti *slicing* dan *environment generation*, dan setelah itu memverifikasi model yang dihasilkan, dengan menggunakan *framework model checking* yaitu Bogor.

Kekuatan dari *model-checking* adalah kemampuannya untuk mendeteksi *defects* perangkat lunak pada *concurrently executing threads*. *Model checking* juga dapat digunakan untuk memeriksa *assertion*, *pre/post conditions*, dan *data invariants*. Namun, berdasarkan penelitian X, *model checking* lebih baik dalam memverifikasi properti-properti terkait kontrol, dibandingkan dengan properti-properti terkait data. *Model checking* telah digunakan untuk memverifikasi properti data pada objek "*container*" seperti *stacks* dan *queue*. Misalnya, memverifikasi bahwa kode pada *stacks* akan menjaga urutan LIFO. Namun, tidak mungkin untuk memverifikasi algoritma *sorting*, karena *sorting correctness* adalah *data-oriented property*. Untuk memverifikasi properti ini, metode formal lain seperti *theorem proving* dapat digunakan. Masalah verifikasi lain seperti memeriksa *array-bounds errors*, *buffer overflow*, dan *null-pointer dereference* akan lebih baik menggunakan *static dataflow analysis*.

Untuk membangun model, Bandera secara otomatis mengkompilasi program Java agar dapat digunakan pada kakas *model checking* yang ada, seperti: Spin, SMV, dan JPF. Bandera menyediakan *temporal specification language* yang mengizinkan pengguna untuk mengekspresikan *properties* pada level *source code* dengan menggunakan *temporal specification pattern*. Spesifikasi ini kemudian dikompilasi secara otomatis menjadi *input language* untuk *model checker* yang dipilih. Bandera juga mampu memetakan *counterexample* yang dihasilkan ke *source code*.

II.10.3 Bogor: Model Checker

Bandera → ekstraksi / konstruksi model dari *source code* dan menyediakan BSL (*Bandera Specification Language*) dengan *temporal specification language*
Model yang dikonstruksi didefinisikan dalam *guarded command language* yang disebut BIR (*Bandera Intermediate Representation*) . BIR language dapat ditranslasikan ke dalam *input language* pada kakas *model checking* lain.

Model checker
TBD

Bab III ANALISIS DAN PERANCANGAN

Bab ini menyajikan hasil tahap analisis dan perancangan dari DSL dan kakas yang akan dikembangkan. Bab ini meliputi deskripsi dan analisis DSL yang dikembangkan, analisis transformasi *source code* menjadi model yang mampu diverifikasi, serta mekanisme untuk menggabungkan verifikasi formal dan pengujian pada BDD.

III.1 Analisis

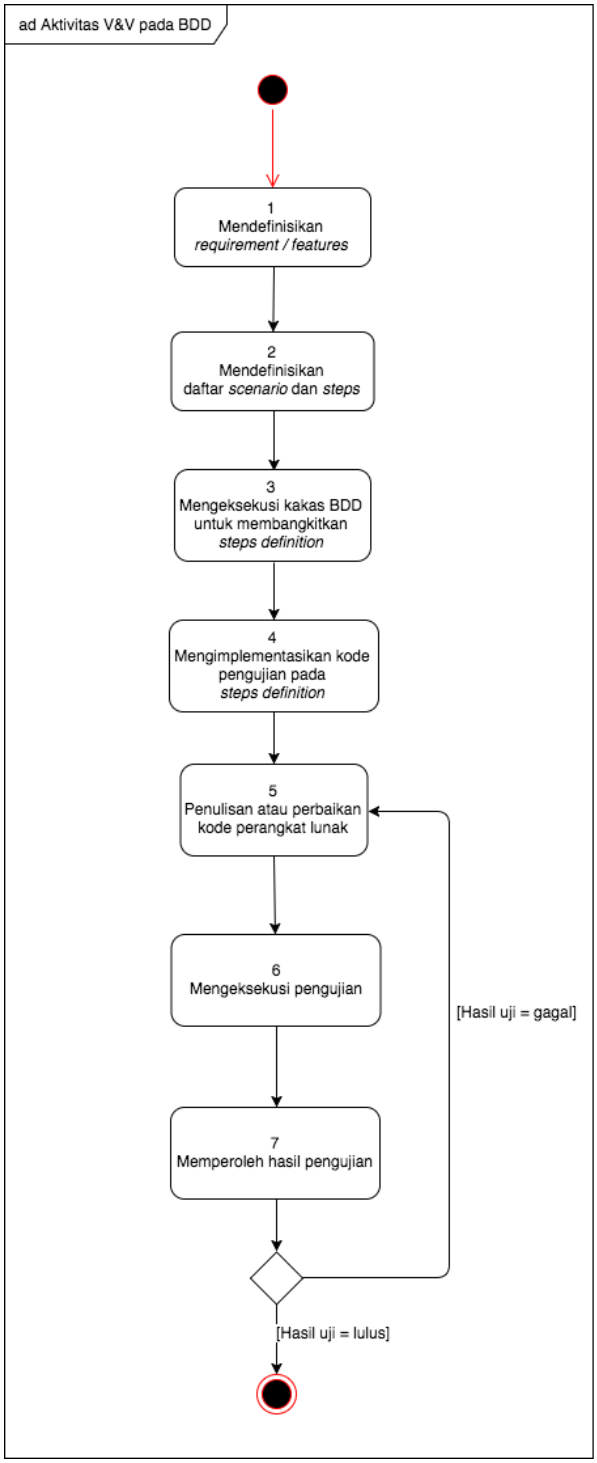
Analisis dimulai dengan mengidentifikasi masalah-masalah pada proses V&V di pengembangan perangkat lunak menggunakan BDD, analisis proses pengembangan yang diusulkan, analisis pemilihan properti yang harus diverifikasi pada *source code*, serta analisis kebutuhan dari DSL dan kakas yang dikembangkan. Hasil dari analisis adalah daftar kebutuhan dari DSL dan kakas yang akan dikembangkan.

III.1.1 Analisis V&V pada Proses Pengembangan BDD

Pada proses pengembangan *Behavior-Driven Development* (BDD) yang ada, pengembang dan *stakeholder* bersama-sama mendefinisikan spesifikasi perangkat lunak, dapat berupa *user story*, yang akan dikembangkan dalam sebuah berkas, sekaligus mendefinisikan *scenario* yang bertindak *acceptance test* yang mendeskripsikan *behavior* dari perangkat lunak. Setelah itu, *scenario* yang mencakup *steps* akan dipetakan menjadi definisi kode pengujian (*steps definition*) yang siap diimplementasikan. Ilustrasi proses pengembangan BDD berdasarkan (Ferguson, 2015) dijelaskan Gambar II-1, sedangkan Gambar III-1 menjelaskan hasil identifikasi V&V pada proses pengembangan BDD yang ada, lewat diagram aktivitas.

V&V pada BDD dimulai pada aktivitas 2 (Gambar III-1), yaitu pada pendefinisian *acceptance test* dalam bentuk *scenario* dan *steps*, dan dilanjutkan dengan mengeksekusi kakas BDD untuk membangkitkan *steps definition*, dan mengimplementasikan *steps definition*. Aktivitas V&V berikutnya adalah pada

yang dipetakan dari mengeksekusi pengujian. Pengujian dapat terdiri dari 2: *manual test* yang berbasis pada *scenario* yang didefinisikan pada aktivitas 2, dan eksekusi kode pengujian. Jika hasil pengujian gagal atau ditemukan *defect* pada perangkat lunak, maka kode perangkat lunak akan diperbaiki, hingga hasil pengujian lulus.



Gambar III-1 Aktivitas V&V pada BDD

Masalah yang dapat diidentifikasi pada proses V&V pengembangan perangkat lunak dengan BDD yaitu verifikasi dan validasi dilakukan lewat *acceptance test* yang meliputi contoh konkrit input yang diujikan, langkah-langkah (*steps*), dan . Seperti yang dipaparkan oleh (Beizer, 1990; Howden, 1987), masalah dari pengujian adalah *incomplete coverage*. Pengujian tidak mampu dilakukan untuk semua kemungkinan input dan jalur eksekusi, sehingga belum mampu menjamin *correctness* dari perangkat lunak. Pengujian hanya mampu mendeteksi kesalahan, bukan menjamin perangkat lunak bebas dari kesalahan.

III.1.2 Integrasi Verifikasi Formal pada V&V BDD

Mekanisme yang diusulkan pada penelitian ini adalah mengintegrasikan teknik verifikasi formal pada proses V&V perangkat lunak dengan pengembangan BDD, sebagai penjaminan *correctness* perangkat lunak (PL). Teknik verifikasi formal yang dipilih untuk diintegrasikan pada BDD adalah *model checking*, yaitu teknik untuk memverifikasi properti *correctness* pada model *finite-state system* yang mendeskripsikan *behavior* dari perangkat lunak. Alasan yang mendasari pemilihan teknik *model checking* untuk verifikasi formal dijelaskan pada subbab II.1.3. Selain itu, tersedianya kakas *model checking* juga menjadi dasar *model checking* menjadi teknik yang dipilih sebagai teknik verifikasi formal pada penelitian ini.

Pada kerangka kerja yang ada, BDD difasilitasi dengan *executable specification*, yaitu DSL `Gherkin` atau DSL `JBehave Story`. Saat ini DSL `Gherkin` atau `JBehave Story` hanya mampu mencakup *feature* (Cucumber) atau *story* (JBehave) dan skenario pengujian. DSL belum mencakup kebutuhan untuk melakukan verifikasi formal. Selain itu, spesifikasi formal perlu ditransformasikan kedalam representasi lain agar dapat di

Berdasarkan hasil analisis, ditentukan kebutuhan sebagai berikut:

1. DSL perlu dikembangkan agar mampu mencakup spesifikasi perangkat lunak, skenario pengujian, dan properti formal yang harus diverifikasi
2. Diperlukan DSL *parser* untuk mentransformasikan spesifikasi formal yang dituliskan pada DSL

3. Dibutuhkan aktivitas tambahan pada alur kerja BDD yaitu:
    a. Pendefinisian properti formal
    b. Penganotasian properti formal pada source code yang akan diverifikasi
    c. Eksekusi kakas yang mampu mengekstraksi model formal dari *source code*
    d. Eksekusi kakas *model checking*
    e. Evaluasi report hasil verifikasi dengan *model checking*

III.1.3 Pemilihan Properti yang Harus Diverifikasi pada *Source Code*

[DRAFT]

Properti yang harus diverifikasi sangat tergantung dari *domain* dan fungsi dari perangkat lunak.

- Safety Property

Contoh:
Paper tentang JPF → dari Java Bytecode
memeriksa *concurrency defect* seperti *deadlocks*, dan *unhandled exception* seperti NullPointerException dan AssertionError

- Liveness Property

Paper tentang JADE

- Fairness Property

Security Properties (di luar cakupan?)
Property templates for checking source code security
https://dl.acm.org/citation.cfm?id=3127063

III.1.4 Specification Pattern

III.1.5 Analisis DSL pada BDD

[DRAFT]

- Kekurangan DSL pada BDD (Gherkin / JBehave Story)
- Kebutuhan harus tambahin apa
- Berikut adalah daftar kebutuhan dari DSL;

DSL perlu dikembangkan agar mampu menjadi spesifikasi perangkat lunak, mampu ditranslasikan ke spesifikasi formal, serta mampu dijadikan input untuk pembangkitan skrip pengujian. Dengan demikian, DSL harus mencakup elemen berikut:

1. Nama fitur
2. Properti yang harus dijamin oleh perangkat lunak
3. Skenario yang dibutuhkan untuk pembangkitan pengujian.

Output dari tahap ini adalah DSL yang mencakup properti formal dan perilaku dari perangkat lunak.

III.1.6 Analisis Kakas pada Proses Pengembangan BDD

[DRAFT]

- Kemampuan kakasnya (parser / compiler)

III.2 Perancangan

Perancangan meliputi dua: perancangan DSL yang akan dikembangkan, dan perancangan kakas yang akan dikembangkan untuk memenuhi tujuan tesis.

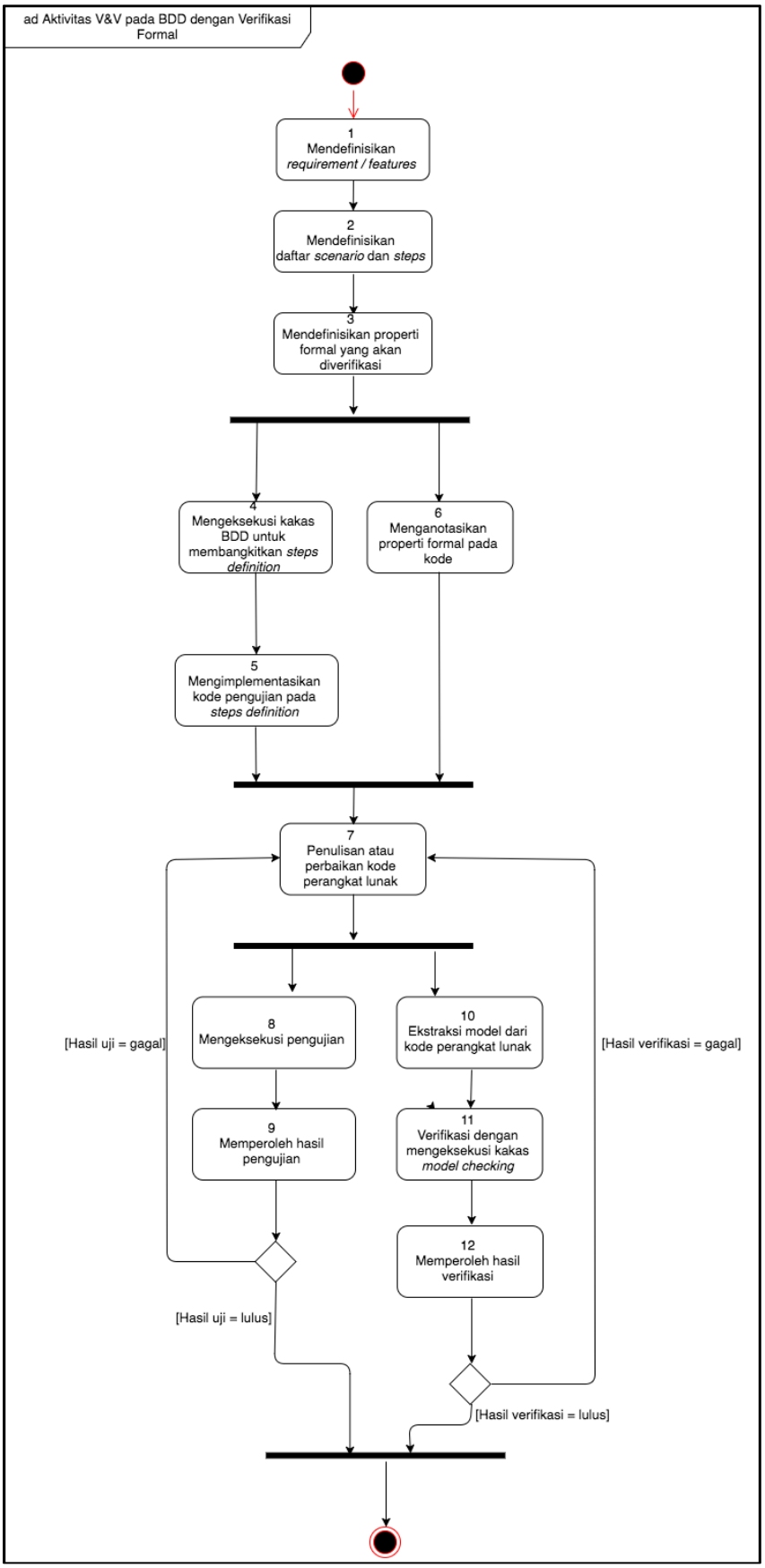
III.3 Rancangan Kerangka Kerja BDD yang Diusulkan

Gambar III-2 menjelaskan aktivitas V&V pada proses pengembangan BDD yang diintegrasikan dengan verifikasi formal. Pada penelitian ini, model tidak dibuat pada tahap perancangan perangkat lunak, melainkan diekstraksi dari *source code*. Properti *correctness* yang harus dipenuhi perangkat lunak didefinisikan pada berkas yang juga berisi definisi *story*, dan *scenario*, sedangkan properti-properti yang harus dibuktikan pada model dianotasikan pada *source code* perangkat lunak. Dengan demikian, aktivitas yang ditambahkan adalah:

1. Aktivitas 3: Mendefinisikan properti formal yang harus diverifikasi. Properti formal yang perlu diverifikasi perlu didefinisikan bersama

pendefinisian *feature*, *scenario* dan *steps*. Properti formal didefinisikan dalam DSL yang akan dikembangkan pada penelitian ini.

2. Aktivitas 6: Menganotasikan properti formal pada kode. Properti formal yang harus dipenuhi oleh implementasi kode dianotasikan pada *source code* yang terkait.
3. Aktivitas 10: Ekstraksi model dari *source code*. Setelah dilakukan implementasi perangkat lunak, model formal yang mampu diverifikasi diekstraksi dari *source code*.
4. Aktivitas 11: Verifikasi dengan mengeksekusi kakas *model checking*. Kakas model checking dieksekusi untuk memverifikasi model terhadap properti yang dianotasikan di *source code*.
5. Aktivitas 12: Memperoleh hasil verifikasi formal. Hasil verifikasi diperoleh, dan jika ditemukan ketidaksesuaian, ditunjukkan *counter example* dan aktivitas dilanjutkan ke perbaikan kode.



Gambar III-2 Integrasi Verifikasi Formal pada Pengembangan PL dengan BDD

III.4 Rancangan DSL

Kemampuan pengujian → acceptance test → scenario
Verifikasi → Specification pattern:

Contoh: **(draft)**
Formal Properties: <*deskripsi dari properti formal*>
Parameters:
* [A..Za..z0..9]*
* [A..Za..z0..9]*
Formula:
[Globally | Finally | Next | Until | Release] + <Param> + [is absent] + <param>

Tabel Contoh Spesifikasi dalam DSL

Usulan DSL	LTL Formula
Formal Properties: **Parameters P** **Formula:** **Globally Not** *P*	
Formal Properties: **Parameters P** **Formula:** **Globally Not** *P* **Before** *R*	
Formal Properties: **Parameters P** **Formula:** **Globally** Not P **After** *R*	

[DRAFT]
BDD terkait dengan bagaimana *behavior* yang diinginkan harus dispesifikasikan. Spesifikasi *behavioral*, diambil dari *user story* pada *object-oriented analysis and design*. Setiap *user story* harus mengikuti struktur berikut:

Judul: *story* harus punya judul yang jelas dan eksplisit

Naratif

Penjelasan singkat yang menspesifikasikan :

Siapa: yang menjadi stakeholder utama dari story (aktor yang mendapatkan manfaat dari story)

Apa: Efek yang stakeholder inginkan dari *story*

Mengapa: nilai bisnis yang stakeholder peroleh

Acceptance Criteria atau Skenario

Deskripsi dari setiap kasus spesifik dari naratif. Skenarip punya struktur berikut:

- Dimulai dengan menspesifikasikan kondisi awal yang diasumsikan benar pada awal skenario
- Status yang mentrigger suatu *event* sebagai awal dari skenario
- Dampak / *outcome* yang diharapkan

Rancangan DSL

Proof:
A = a

DAFTAR PUSTAKA

Adrion, W. R., Branstad, M. A., dan Cherniavsky, J. C. (1982): Validation, Verification, and Testing of Computer Software, *ACM Computing Surveys*, **14**(2), 159–192, diperoleh melalui situs internet: https://doi.org/10.1145/356876.356879.

Agile Alliance (2015, Desember 16): BDD: Learn about Behavior Driven Development, diperoleh melalui situs internet: https://www.agilealliance.org/glossary/bdd/.

Beizer, B. (1990): *Software testing techniques*, 2nd ed, New York: Van Nostrand Reinhold.

Bentes, L., Rocha, H., Valentin, E., dan Barreto, R. (2016): JFORTES: Java Formal Unit TESt Generation, 16–23, IEEE, diperoleh melalui situs internet: https://doi.org/10.1109/SBESC.2016.012.

Berztiss, A., dan Ardis, M. (1988): Formal Verification of Program, *Software Engineering Institute*, (CMU/SEI-88-CM-020), diperoleh melalui situs internet: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=10775.

Carter, J. D. (2017): *BHive: Behaviour-Driven Development Meets B-Method*, The University of Guelph, Ontario, Canada.

Ciolkowski, M., Laitenberger, O., Rombach, D., Shull, F., dan Perry, D. (2002): Software inspections, reviews & walkthroughs, 641, ACM Press, diperoleh melalui situs internet: https://doi.org/10.1145/581339.581422.

Collofello, J. (1988): Introduction to Software Verification and Validation, *Software Engineering Institute*, (CMU/SEI-88-CM-013), diperoleh

melalui situs internet: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=10775.

Cucumber (2018): *Gherkin - cucumber: Cucumber monorepo - polyglot home for Cucumber building blocks*, Ruby, Cucumber, diperoleh melalui situs internet: https://github.com/cucumber/cucumber.

Cucumber.io (2018): Documentation·Cucumber, diperoleh 9 Maret 2018, melalui situs internet: https://cucumber.io/docs.

Diepenbeck, M., dan Drechsler, R. (2015): Behavior Driven Development for Tests and Verification, R. Drechsler dan U. Kühne (Ed.), *Formal Modeling and Verification of Cyber-Physical Systems*, 275–277, Wiesbaden: Springer Fachmedien Wiesbaden, diperoleh melalui situs internet: https://doi.org/10.1007/978-3-658-09994-7_11.

Dijkstra, E. W., Dahl, O.-J., dan Hoare, C. A. R. (1972): *Structured programming*, London, New York: Academic Press.

Dunlop, D. D., dan Basili, V. R. (1982): A Comparative Analysis of Functional Correctness, *ACM Computing Surveys*, **14**(2), 229–244, diperoleh melalui situs internet: https://doi.org/10.1145/356876.356881.

Ferguson, J. (2015): *BDD in action: Behavior-Driven Development for the whole software lifecycle*, Shelter Island, NY: Manning Publications.

Fofung, T. D., dan Duggins, S. (2015): *Quantitative Analysis of Formal Specification Driven Development*, Southern Polytechnic State University, Marietta, Georgia, diperoleh melalui https://digitalcommons.kennesaw.edu/cgi/viewcontent.cgi?article=1687&context=etd.

Gamage, T. A. (2017, Juli 26): JBehave Vs Cucumber JVM: Comparison and Experience Sharing, diperoleh melalui situs internet: https://medium.com/agile-vision/jbehave-vs-cucumber-jvm-comparison-and-experience-sharing-439dfdf5922d.

Ghosh, I., Shafiei, N., Li, G., dan Chiang, W.-F. (2013): JST: An automatic test generation tool for industrial Java applications with strings, 992–1001, IEEE, diperoleh melalui situs internet: https://doi.org/10.1109/ICSE.2013.6606649.

Hailpern, B., dan Santhanam, P. (2002): Software debugging, testing, and verification, *IBM Systems Journal*, **41**(1), 4–12, diperoleh melalui situs internet: https://doi.org/10.1147/sj.411.0004.

Halpern, J. Y., dan Vardi, M. Y. (1991): Model checking vs. theorem proving: a manifesto, V. Lifschitz (Ed.), *Artificial Intelligence and Mathematical Theory of Computation*, 151–176, San Diego, CA, USA: Academic Press Professional, Inc., diperoleh melalui situs internet: http://dl.acm.org/citation.cfm?id=132218.132228.

Harrison, J. (2008): *Formal Proof—Theory and Practice*, **55**, American Mathematical Society.

Hooman, J. (2016): Industrial Application of Formal Models Generated from Domain Specific Languages, E. Ábrahám, M. Bonsangue, dan E. B. Johnsen (Ed.), *Theory and Practice of Formal Methods*, **9660**, 277–293, Cham: Springer International Publishing, diperoleh melalui situs internet: https://doi.org/10.1007/978-3-319-30734-3_19.

Howden, W. E. (1987): *Functional program testing and analysis*, New York: McGraw-Hill.

Hudak, P. (1997): Domain-specific languages, *Handbook of programming languages*, **3**(39–60), 21.

IEEE (2002): *610.12-1990 IEEE Standard Glossary of Software Engineering Terminology.*, IEEE, diperoleh melalui situs internet: http://ieeexplore.ieee.org/servlet/opac?punumber=2238.

IEEE Std 1012-2016 (2016): *IEEE Std 1012-2016 (Revision of IEEE Std 1012-2012/ Incorporates IEEE Std 1012-2016/Cor1-2017): IEEE Standard for System, Software, and Hardware Verification and Validation*, IEEE.

IEEE-SA Standards Board, International Organization for Standardization, International Electrotechnical Commission, dan Institute of Electrical and Electronics Engineers (2013): *Software and systems engineering: software testing. Part 1, Part 1,* diperoleh melalui situs internet: http://ieeexplore.ieee.org/servlet/opac?punumber=6588535.

JBehave (2018): What is JBehave?, diperoleh 11 Maret 2018, melalui situs internet: http://jbehave.org/.

Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., dan Völkel, S. (2014): Design Guidelines for Domain Specific Languages, *arXiv:1409.2378 [cs]*, diperoleh melalui situs internet: http://arxiv.org/abs/1409.2378.

Kolesnik, N. (2013): JBehave vs Cucumber JVM comparison, diperoleh melalui situs internet: http://mkolisnyk.blogspot.com/2013/03/jbehave-vs-cucumber-jvm-comparison.html.

Kröger, F., dan Merz, S. (2008): First-Order Linear Temporal Logic, *Temporal Logic and State Systems*, 153–179, Berlin, Heidelberg: Springer Berlin Heidelberg, diperoleh melalui situs internet: https://doi.org/10.1007/978-3-540-68635-4_5.

Kukimoto, Y. (1996, Februari 6): Introduction to Formal Verification, diperoleh 27 Februari 2018, melalui situs internet: https://embedded.eecs.berkeley.edu/research/vis/doc/VisUser/vis_user/node4.html.

Li Tan, Sokolsky, O., dan Insup Lee (2004): Specification-based testing with linear temporal logic, 493–498, IEEE, diperoleh melalui situs internet: https://doi.org/10.1109/IRI.2004.1431509.

Liu, S. (2016): Testing-Based Formal Verification for Algorithmic Function Theorems and Its Application to Software Verification and Validation, 1–6, IEEE, diperoleh melalui situs internet: https://doi.org/10.1109/ISSSR.2016.010.

Marín, B., Gallardo, C., Quiroga, D., Giachetti, G., dan Serral, E. (2017): Testing of model-driven development applications, *Software Quality Journal*, **25**(2), 407–435, diperoleh melalui situs internet: https://doi.org/10.1007/s11219-016-9308-8.

Mashkoor, A. (2016): Model-driven development of high-assurance active medical devices, *Software Quality Journal*, **24**(3), 571–596, diperoleh melalui situs internet: https://doi.org/10.1007/s11219-015-9288-0.

Mernik, M., Heering, J., dan Sloane, A. M. (2005): When and how to develop domain-specific languages, *ACM Computing Surveys*, **37**(4), 316–344, diperoleh melalui situs internet: https://doi.org/10.1145/1118890.1118892.

Mishra, A. (2017): *iOS code testing: test-driven development and behavior-driven development with Swift*, Berkeley, California: Apress.

Mukund, M. (1996): *Linear Temporal Logic and Buchi Automata*, Madras: ISI Calcutta.

Narisawa, F., Matsubara, M., Nishi, M., dan Ebina, T. (2015): Formal Verification Method for Safety Diagnosis Software, diperoleh melalui situs internet: https://doi.org/10.4271/2015-01-0279.

Parnas, D. L., dan Lawford, M. (2003): The role of inspection in software quality assurance, *IEEE Transactions on Software Engineering*, **29**(8), 674–676, diperoleh melalui situs internet: https://doi.org/10.1109/TSE.2003.1223642.

Rose, S., Wynne, M., dan Hellesøy, A. (2015): *The cucumber for Java book: behaviour-driven development for testers and developers*, Dallas, Texas: The Pragmatic Bookshelf.

Rutledge, R., Duggins, S., Lo, D., dan Tsui, F. (2014): *Formal Specification-Driven Development*, Georgia Institute of Technology, Atlanta, GA, diperoleh melalui http://ksuweb.kennesaw.edu/~ftsui/images/Paper_FSDD%20SERP%202014%20Final.pdf.

Solis, C., dan Wang, X. (2011): A Study of the Characteristics of Behaviour Driven Development, 383–387, IEEE, diperoleh melalui situs internet: https://doi.org/10.1109/SEAA.2011.76.

Trentelman, K., dan Huisman, M. (2002): Extending JML Specifications with Temporal Logic, *Algebraic Methodology and Software Technology*, 334–348, Springer, Berlin, Heidelberg, diperoleh melalui situs internet: https://doi.org/10.1007/3-540-45719-4_23.

Wiegers, K. E. (2002): *Peer reviews in software: a practical guide*, Boston, MA: Addison-Wesley.

Wilcox, R. (2014, Februari 14): BDD vs TDD: A Behavior-Driven Development Example | Toptal, diperoleh 23 Februari 2017, melalui situs internet: https://www.toptal.com/freelance/your-boss-won-t-appreciate-tdd-try-bdd.

Wolper, P., dan Vardi, M. Y. (1986): An Automata-Theoretic Approach to Automatic Program Verification, *Proceedings of the First Symposium on Logic in Computer Science*, 322–331, IEEE Computer Society.