

NAIST-IS-MT1551112

修士論文

高信頼性ソフトウェア検証における不具合の分類結果に
基づいた単体テストケース自動生成の適用及び改善

山崎 雅也

2017 年 2 月 2 日

奈良先端科学技術大学院大学
情報科学研究科 情報科学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
修士(工学) 授与の要件として提出した修士論文である。

山崎 雅也

審査委員：

飯田 元 教授	(主指導教員)
松本 健一 教授	(副指導教員)
片平 眞史 教授	(副指導教員)
石濱 直樹 准教授	(副指導教員)
川口 真司 准教授	(副指導教員)

高信頼性ソフトウェア検証における不具合の分類結果に基づいた単体テストケース自動生成の適用及び改善*

山崎 雅也

内容梗概

宇宙機ソフトウェアと呼ばれる人工衛星等に搭載されるソフトウェアは、宇宙空間で使用するという特性上、大きな修正を加えることができないため、非常に高信頼かつ安全であることが求められる。このような高信頼性や安全性を実現するために全てのソフトウェア開発プロセスにおいて様々な検証が行われるが、ソフトウェアより不具合を完全になくすことは現実的な時間では難しい。このことより、過去の開発過程で発生したソフトウェア不具合を分析することは、運用や未来の開発に対して、より高信頼かつ安全なものにするために有効な手段である。

本研究では、過去の宇宙機ソフトウェア開発の不具合を定性的及び定量的に評価することによって、高信頼ソフトウェアの開発が抱える問題を明らかにする。その結果を元に、多く不具合が発生している原因事象に対して、既存の検証では行われていなかった点に関して改善を図ることを研究目的とする。はじめに7つの宇宙機プロジェクト計50個の不具合に対して直行欠陥分類法の適用を試みた。その過程で宇宙機ソフトウェア開発に適した直行欠陥分類法を提案した。また分類されたそれぞれの属性に対して因果分析を行い、問題を洗い出した。改良した直行欠陥分類法での分類と因果分析を行ったところ、全体の不具合のうち28%が設計で混入した不具合がシステムテストで見つかることがわかった。また、割り込み等のタイミングの問題は様々な検証の観点で見つかることなどが分かった。

この問題を解決すべく、単体テストケースの自動生成技術である Concolic Testing の適用を行うことで解決を試みた。Concolic Testing とは記号実行と具体的な値を

* 奈良先端科学技術大学院大学 情報科学研究科 情報科学専攻 修士論文, NAIST-IS-MT1551112, 2017年2月2日.

用いた動的解析を行う単体テスト手法であり、単体テストケース入力を自動で生成することができる。この手法を用いることによって、不具合分析の結果で明らかになった単体テストが十分にできていなかったという問題に対して、自動化によるカバレッジの向上ができる。しかしながら、Concolic Testing ツールである CREST を小型衛星の姿勢制御系モジュールに適用した結果、特定の条件文でテストケースを生成できない場合が存在することが分かった。その中でも最も多く存在した配列の要素数が定数でない場合に対して、最適化コンパイラで用いられることが多いループ展開を応用し、性能の向上を図った。ループ展開はループのない形にプログラムを書き換えることによって、プログラムの実行速度を上げる手法である。この手法を今回はループの削除を行い、CREST が適用可能なプログラムに書き換えるために用いた。CREST が配列の要素数が定数でないものに対してもテストケース生成ができるようなループ展開プログラムを構築し、小型衛星の姿勢制御系モジュールに適用することによってテストカバレッジの向上ができることを確認した。結果として単体テストが不十分であったものに対してはテストケース生成を通じて、改善を行うことができた。

キーワード

宇宙機ソフトウェア, 直交欠陥分類法, ソフトウェアテスト, 自動単体テストケース生成, Concolic Testing

*

Masaya Yamazaki

Abstract

Abstract

Keywords:

Spacecraft Software, Orthogonal Defect Classification, Software Test, Automated Unit Test Case Generation, Concolic Test

* Master's Thesis, Department of Information Science, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT1551112, February 2, 2017.

目次

1.	はじめに	1
2.	関連技術・関連研究	3
2.1.	不具合分析の関連技術	3
2.2.	不具合分析の関連研究	4
2.3.	テストケース自動生成の関連技術	7
2.4.	テストケース自動生成の関連研究	9
3.	提案分類方法	11
3.1.	分析目的	11
3.2.	予備実験	11
3.3.	分類	12
4.	分類結果	14
4.1.	Insertion Activity	14
4.2.	Detection Activity	15
4.2.1.	Supplier Testing	16
4.3.	Effect	17
4.4.	Trigger	18
4.5.	Type	22
4.6.	Mode	25
4.7.	考察及び課題	27
5.	提案手法	28
5.1.	Concolic Testing の宇宙機ソフトウェアへの適用	28
5.1.1.	対象宇宙機ソフトウェア	28
5.1.2.	適用結果	29
5.1.3.	考察及び解決策	30
5.2.	ループ展開	31

6.	評価実験	33
6.1.	実験目的と内容	33
6.2.	実験結果	34
6.3.	考察	34
7.	おわりに	36
	謝辞	37
	参考文献	38

図目次

1	Insertion Activity の分布	15
2	Detection Activity の分布	16
3	Supplier Testing の分布	17
4	Effect の分布	18
5	Trigger の分布	18
6	Trigger-Effect の分布	21
7	Type の分布	22
8	Type-Supplier Testing の分布	24
9	Mode の分布	25
10	Mode-Trigger の分布	26

表目次

1	Lutz らの運用上の不具合分類の属性 (文献 [20] より引用)	5
2	Lutz らの運用上の不具合分類の要素 (文献 [20] より引用)	6
3	Concolic Test ツールの一覧 (文献 [34] より引用)	8
4	テスト改善のための欠陥分類の属性	12
5	テスト改善のための欠陥分類の要素	13
6	Trigger の定義	19
7	Trigger の実例	20
8	Trigger-Type の分布	23
9	CREST 実行環境	29
10	Concolic Testing 適用結果	30
11	実験対象対象モジュール	33
12	ループ展開適用前	35
13	ループ展開適用後	35

1. はじめに

現在、世界中で宇宙開発が盛んにおこなわれている。日本も例外ではなく、人工衛星をはじめとする様々な宇宙開発プロジェクトが行われている。しかしながら、Ariane 5 [18] や Mars Climate Orbiter [7] に代表されるようにソフトウェアの些細な不具合、複合的なエラーや開発者の想定不足によって引き起こされる不具合によってミッションが失敗となる事が多々ある。原因として、レガシーシステムの再利用に伴うリグレッションテストの不足や、ソフトウェア開発プロセスの問題などが挙げられる。そういった不具合をなくすためソフトウェアを開発する組織から技術面、組織面、及び資金面で独立している組織が実施する IV&V (Independent Verification and Validation) [16] などの様々な取り組みがなされている。一方でこれらは人に頼った検証が多くを占め、システム間の不整合、工数増大、品質のエンジニア依存、ノウハウの継承の問題など様々な問題を抱えている。Brian らの Mars Climate Orbiter のミッション失敗要因の検証によると、これらの問題を解決するためには一定の水準を保ちながら、できることはソフトウェアによって検証を行うことが 1 つの解決案であると報告されている [7]。

本研究では宇宙機ソフトウェア開発で蓄積された開発に関する不具合情報を基に効率的なソフトウェア検証方法の明確化及び改善を目指す。不具合分類の手法として Ram Chillarege らによって開発された直交欠陥分類法 [28] がある。直交欠陥分類法は定性的分析 (e.g. HAZOP) と定量的分析 (e.g. FTA) の両方の性質を併せ持つ技法であり、欠陥の性質及びその量から傾向を分類し、ソフトウェア全体の問題を洗い出すことができる。また、開発プロセスや特定の技術に依存することがなく、幅広い分野で用いられている。特に本研究では、宇宙機における特殊な開発プロセスや性質をより明確に明らかにし、有効なテスト手法を確立するために宇宙機ソフトウェアに特化した欠陥分類法を提案する。また、実際に 5 つの宇宙機プロジェクト計 50 個の不具合について分類を行った。またその結果に対して Deep Dive [8] と呼ばれる手法を用い、不具合の根本的原因や傾向を導き出し、適切なテスト手法を明確にすることを本研究の目的とする。

以降、2 章では関連技術及び関連研究を紹介する。3 章では本研究の目的及び分類手法について述べる。4 章では分類及び分析結果を記載する。5 章では **Concolic Test** を実際に適用した結果を述べる。6 章では提案手法の実験及び検証結果を示す。7 章では本研究のまとめと今後の課題について述べる。

2. 関連技術・関連研究

本章では、関連技術とそれらについて調査を行った既存研究を紹介する。

2.1. 不具合分析の関連技術

不具合分析には大きく分けて、統計的手法と原因解析が存在する。統計的手法ではその不具合の数のみを不具合分析の要素とするため定量的分析手法である。具体的な例としては FTA(Fault Tree Analysis) の定量的分析などがある。一方で原因解析は、1 つの不具合について原因や結果などを用いて詳細に分析するため、定性的分析手法である。具体的な例として、HAZOP(Hazards and Operability Analysis) などがある。一般的にソフトウェア開発プロセス全体の見直しをする場合には統計的手法が用いられ、ソフトウェア開発の大きな不具合が発生した時には原因解析が行われる [15]。この 2 つの問題点として、両方の性質を同時に扱うことは非常に手間がかかることがある。

直交欠陥分類法 (ODC: Orthogonal Defect Classification) は IBM の Ram Chillarege らによって開発されたソフトウェアのための欠陥分類法である [28]。観点の異なる属性から構成されていることから直交と呼ばれる。また、定性的分析と定量的分析の両方の性質を併せ持つことから、欠陥の数と性質から改善方法を考えることができる。まず、欠陥報告時と欠陥修正時に大きく分類され、その後より小さな分類である属性 (e.g. Trigger, Activity, Target 等) に分類される。属性のうち、欠陥報告書に含まれる情報で活用できるものを選択し、実際に不具合分類を行う。現在は version 5.2 [11] がリリースされており、これはソースコードおよび設計に焦点を絞ったものである。大きくオープナーセクションとクローザーセクションに分かれており、これらは欠陥発見時と欠陥修正時という意味がある。またそれぞれに属性が存在し、合計 8 つの属性からなる。属性のうち欠陥報告書に含まれる情報を活用できるものを選択し、実際に不具合分類を行う。また、同様の規格として IEEE が提唱している IEEE 1044-2009 がある [31]。これは IEEE Standard Association によって定められた不具合分類法に関する規格である。属性が ODC に比べても多いことが特徴である。

2.2. 不具合分析の関連研究

Lutz らは宇宙機ソフトウェアの不具合分析結果より、要求の問題をソフトウェアテストから発見する方法や、運用の不具合を明確化している [19, 20]。テストは通常要求が満たされているかを確認するための手順であるが、ソフトウェアの欠陥以外の要求等の問題が抽出できるということが明らかになっている。また、運用中のソフトウェアの不具合を解析することによって、ソフトウェアの操作文書と通信の問題など運用特有の問題が存在していることや、通常行わない以上回復などの操作は不具合の要因になりやすいことを明らかにしている。表 1 及び表 2 に Lutz らの運用上の不具合分析の属性と要素を示す。以上の 2 つの事例に関しては両方とも直交欠陥分類法と呼ばれるものを利用している。しかしながら、どちらもソフトウェアテストを改善するための分類方法ではなく本研究に当てはめることはできない。また、どちらも直交欠陥分類法自体はそのまま使っており、属性の中の要素の変更のみである。このことより、宇宙機ソフトウェアのテスト手法を検討するためには改善が必要である。

Gortte らは宇宙機ソフトウェアにおける不具合を「小さいバグ」という単位で分析している [23]。彼らの分析では、ソフトウェアのバグを”Bohrbugs”, “Mandelbugs”, “aging-related bugs” に分類し、ミッションごとの特徴を明らかにしている。しかしながら、改善策としてはモデル検査によるコードの質の改善やレジリエントにする方法しか示されておらず、ソフトウェア開発プロセス全体の問題にはフォーカス出来ていない。

Huang らは ODC の自動化のツールを公開している [17]。これは機械学習を基に分類を行うものだが、今回は学習データが少ないので利用できない。

宇宙機ソフトウェア以外を対象に直交欠陥分類法を変化させた例として、Freimut らの BOSCH 社でのガソリンエンジンシステムを例として欠陥を抽出した研究がある [6]。彼らの研究では、実際に開発している社員に対してインタビューを適正に行い、ある分野に特化した欠陥分類法へと変化させてさせることにより精度の高いものにすることにできることを示している。このことより、直交欠陥分類法は改良を加えることにより、より適切に分析することが可能になることが示されている。

表 1: Lutz らの運用上の不具合分類の属性 (文献 [20] より引用)

タイミング	属性	意味
欠陥報告時	Activity	どこの運用で発見したか
	Trigger	どのような状況で発生したか
欠陥修正時	Target	不具合存在箇所
	Type	何を間違ったのか

表 2: Lutz らの運用上の不具合分類の要素 (文献 [20] より引用)

Activity	Trigger	Target	Type
System Test Flight Operations Unknown	Software Configuration	Ground Software	Function/Algorithm
	Hardware Configuration	Flight Software	Interfaces
	Start/Restart, Shutdown	Build/Package	Assignment/Initialization
	Command Sequence Test	Ground Resource	Timing
	Inspection/Review	Information Development	Flight Rule
	Recovery	Hardware	Install Dependency
	Nomal Activity	None/Unknown	Packaging Scripts
	Special Procedure		Resource Conflict
	Hardware Failure		Documentation
	Unknown		Procedures
			Hardware
			Nothing Fixed
			Unknown

2.3. テストケース自動生成の関連技術

テストケース自動生成技術はシステムテスト、統合テスト、単体テストの各フェーズにて存在する [10]。

本章では研究対象である単体テストケース生成自動生成を中心に述べる。

Concolic Testing [1] とは Koushik らによって提唱された SMT ソルバを用いた形式手法によるテストケース生成とランダムテストを合わせて現実的な時間で動的に単体テストケース生成を行う手法である。Concolic とは Concrete と Symbolic を合わせた造語であり、形式手法を用いながら動的な探索も行うことを意味する。SMT ソルバは不定要素を含む論理式があるときに、その不定要素に対して、適切な値を割り当てることで、対象論理式全体を成立できるようにできるかという充足可能性判定問題と呼ばれるものを解くのに用いられるものである [35]。プログラムを論理式に変換することによって単体テストケース自動生成を実現している。Concolic Test を実行するツールは様々存在し、Xiao らによってまとめられている [34]。また、現状の限界として (1) float, double 型が扱えないこと、(2) ポインタ扱えないこと、(3) ネイティブコールが使えないこと、(4) 非線形演算ができないこと、(5) ビット演算ができないこと、(6) 配列のオフセットが扱えないこと、(7) ファンクションポインタが使えないことが挙げられる。なお (1) に関しては、Softfloat というシミュレーションソフトウェアを用いることで回避できることが示されている [27]。また、(4) に関しては SMT ソルバとして Yices [3] の代わりに Microsoft Research で開発されている Z3 [4] を用いることによって解決されている [2]。実際に、Microsoft ではこの技術を .NET Framework 用の単体テストケース生成に利用した機能を Visual Studio に組み込んでいる [26]。インターフェースにおいてコマンドライン上でしか使えない CREST [12] と比較すると有用であるが、今回対象に選定した宇宙機ソフトウェアで用いられている C を扱うことができない。

表 3: Concolic Test ツールの一覧 (文献 [34] より引用)

Tool	Language	Platform	Oracle	Constraint Solver
DART	C	NA	UE	lp_solver
SMART	C	NA	UE	lp_solve r
CUTE	C	Linux	UE	lp_solver
jCUTE	Java	Linux	UE	
CREST	C	Linux	UE	Yices
EXE	C	Linux	UE	STP
KLEE	C	Linux	UE	STP
Rwset	C	Linux	UE	STP
JPuzz	Java	Linux	UE	built on JPF
PathCrawler	C	NA	NA	NA
Pex	.NET	Windows	UE	Z3
SAGE	machine code	Windows	UE	Dislover
NA	PHP	NA	NA	NA
Apollo	PHP	NA	NA	NA

2.4. テストケース自動生成の関連研究

自動テストケース生成の研究としては遺伝的アルゴリズム、山登り法、焼きなまし法などのサーチ技法を用いた **Search-based Testing** を用いた研究がある [22]。これらは網羅的にソフトウェアを検証するよりも、特定の目的に対して適合関数を設定し、それを解決するアルゴリズムを提供するものである。したがって、解決すべき適合関数が明白である場合には有効なテストケース自動生成を行える。

網羅的にソフトウェアを検証できているかの基準としてコードカバレッジがある。具体的な基準として命令網羅 (C0)、分岐網羅 (C1)、条件網羅 (C2) などが存在する [9]。しかしながら、最もテストケース数が多くなる条件網羅は条件文 n が存在した場合に、そのテストケース数は n^2 となり、全ての機能に対してテストケースを作成することは難しい。そこで、航空機ソフトウェアをテストするカバレッジの基準として、RTCA によって策定された DO-178B に記述されている MC/DC(Modified Condition/Decision Coverage) がある [13]。この基準を用いると条件文 n が存在した場合に、そのテストケース数は $n + 1$ となり C2 に比べてテストケース数を削減することができ、実用的な基準となっている。

網羅的にソフトウェアを検証するためのテストケース自動生成を行う研究としては 2.3 章で述べた、記号実行を具体的な値をテストケースとして生成しながら動的にテストを行う、**Concolic Testing** がある。Sagharatna らは組み込み分野で用いられることが多い、Matlab/Simulink によって生成された簡単な C コードに対して MC/DC を満たすテストケースを生成することができることを実証するとともに [29]、循環的複雑度が高くなるほど、MC/DC が低くなることを明らかにしている [30]。また、**Concolic Testing** を実用ソフトウェアに対して適用した研究として Moonzoo らの Sumsung の Linux プラットフォームやセキュリティライブラリに適用した研究 [24] や Braione らの原子力プラントのマニピレータの制御ソフトウェアに適用した研究がある [27]。この研究では Baluda らのカバレッジを優先させた ARC と呼ばれる **Concolic Testing** のアルゴリズムも比較対象となっており、カバレッジを向上させることによって通常の **Concolic Testing** では発見できない不具合を発見できている [21]。

宇宙機ソフトウェアに対するテストケース自動生成を試みた研究としては

Shankar らの SILS(Software in Loop Simulation) と呼ばれる制御システム全体をソフトウェアでシミュレーションする際のテストケース生成の自動生成 [32] や Barltrop らの宇宙機のシステムテストの自動化を実現する研究 [14] などがある。

3. 提案分類方法

本章では、分析の目的及び本研究での分類方法について説明する。

3.1. 分析目的

ここでは宇宙機ソフトウェアを開発する過程で発生した不具合について分類、分析することにより最も不具合が発生する要因を明らかにし、その不具合に対して効率的にソフトウェアを検証する方法を明らかにすることを目的としている。またそれだけではなく、開発者は本研究の分析結果を考慮した上でソフトウェアテスト等の検証を行うことで、効率的にソフトウェアの不具合を発見することが可能になると考えられる。本研究ではこの分析結果を元に自動テストケース生成の指針とした。そのためには不具合の分析方法をテストとの関連を明確化できるようなものにする必要がある。

3.2. 予備実験

まず、着眼点として ODC および IEEE 1044-2009 を用いることを検討した。また、不具合分析に関する先行研究は全て ODC であったが、IEEE の規格に関しても適用が必要であると考え、姿勢制御系 (AOCS: Attitude and Orbit Control System) のソフトウェア不具合 10 件に対して適用を行った。なお、ODC の分類で利用したのは宇宙機ソフトウェアに対応した [20] であり、IEEE 1044-2009 は規格に準じて行った。その結果、先行研究の改良型 ODC に存在していた属性の項目のほとんどは使われることはなかった。理由としては運用フェーズにターゲットを絞ったものであり、ソフトウェア開発にはうまく適合できなかったことが挙げられる。また、対象としているのが、宇宙機ソフトウェア自体なので Target は分類する必要がなかった。一方で IEEE 1044-2009 の方は半分の属性が” Unknown” であったものの、” Effect” や” Mode” などは欠陥の特徴を大きくとらえて有効に分けられるものも存在し、いくつかの属性は ODC と重複していながらも、項目自体は ODC よりも適切に分類できているものも存在した。添付資料にその結果を示す。

3.3. 分類

上記の結果により、表 4 の ODC を改良したソフトウェア分類方法を構築した。まず、先行研究 [20] をより効率的なソフトウェアテスト手法の検討のために変化させた点としては、まず運用に関する欠陥分類を削除し、IEEE の属性である Mode 及び Effect を組み込んだことである。次に ODCv5.2 を組み込んだことである。これは前述のとおりソフトウェアのコーディングと設計における欠陥について詳細を記述したものである。これを利用することにより、AOCS に関しても不具合を洗い出すことができると考えた。また、テストフェースと不具合の特徴を関連付けるために [1] で提案されているようにテストを行った段階を項目として追加した。また、属性 Detection Activity の要素である Supplier Testing は Unit・Function Test、System Test、Integration Test、Other に細分化した。

表 4: テスト改善のための欠陥分類の属性

タイミング	属性	意味
欠陥報告時	Insertion Activity	開発のどの段階で不具合が混入したのか
	Detection Activity	開発のどの段階で不具合が発見されたのか
	Effect	不具合が与える影響
	Trigger	どのような状況で発生したか
欠陥修正時	Type	何を間違ったのか
	Mode	機能が抜けていたのか、間違った振舞いになっていたのか

表 5: テスト改善のための欠陥分類の要素

Insertion Activity	Detection Activity	Effect	Trigger	Type	Mode
Requirement Design Coding Unknown	Design Review	Functionality Performance	Design Conformance	Function/Algorithm Interface Assignment/Initialization Timing	Missing Wrong
	Coding		Logic/Flow		
	IV&V		Lateral Compatibility		
	Customer Testing		Rare Situation		
	Production		Workload/Stress		
	Unknown		Blocked Test		
	Supplier Testing		Test Coverage		
			Software Configuration		
			Hardware Configuration		
			Simple Path		
			Normal Activity		
			Complex Path		

4. 分類結果

この章では実際の宇宙機プロジェクト計 5 個 50 個の不具合に対して分析を行った結果を示す。今回対象としたデータはロケットや衛星からの分離後に姿勢軌道を制御する装置である姿勢軌道制御系 (ACOS: Attitude and Orbit Control Subsystem) や人工衛星の機器の不具合データである。この不具合情報は 1 つの不具合に対して、プロジェクト名、不具合が発見されたタイミング (e.g. ソフトウェアテスト等)、不具合の詳細情報、原因、発生要因/流出要因、要因に関する分析が含まれている。また、Deep Dive と呼ばれる属性同士を掛け合わせて関連性を調べる手法を用いることによって宇宙機ソフトウェア開発が抱える問題について明らかにする。

4.1. Insertion Activity

図 1 はソフトウェア開発工程のどこで欠陥が混入したかという属性である Insertion Activity の分布である。Design に起因する不具合が最も多く 33 個 (66%) 存在した。また、分類不能という意味の Unknown が 5 つ存在するが、これは Design または Coding で混入したことはわかるが、不具合報告表からはどちらなのか判別できなかったものである。そのため、Coding 及び Design の実際の個数は図 1 よりも多くなる。いずれにしろ、通常のソフトウェア開発と同様に設計段階で不具合が混入することが多いことがわかる。

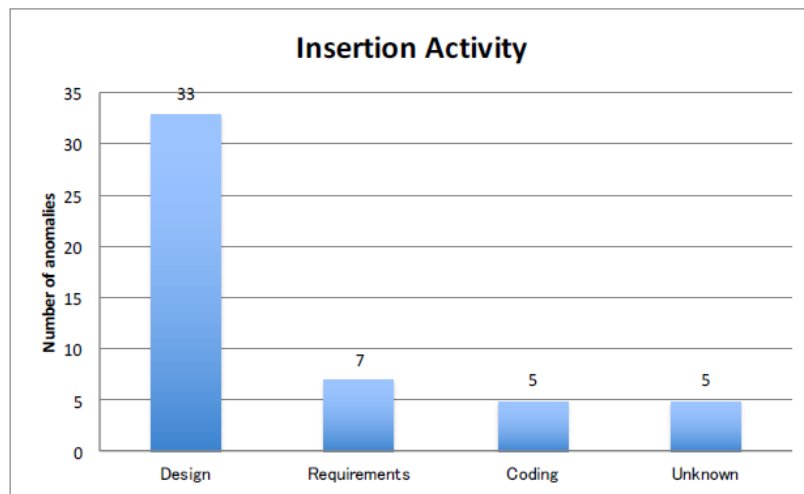


図 1: Insertion Activity の分布

4.2. Detection Activity

図 2 はソフトウェア開発におけるどのフェーズで不具合が見つかったかという属性の Detection Activity の分布である。本研究で対象とするデータが開発しているメーカーより提供を受けたものであるため、ほとんどは Supplier Testing 見ついている (45 件 or 90%)。一方で、JAXA で実施するソフトウェアの独立検証と有効性確認 (IV&V) でも 3 件 (6%) の不具合が発見されていることから有効性が確認される。また Production に分類された 1 件 (2%) は JAXA での通電試験まで不具合が発見されなかった例である。これは試験が最終段階まで抜けてしまったことに起因する。また、Design Review で見つかった 1 件 (2%) は複数ドキュメントの整合性が取れていなかったことで発見された不具合である。

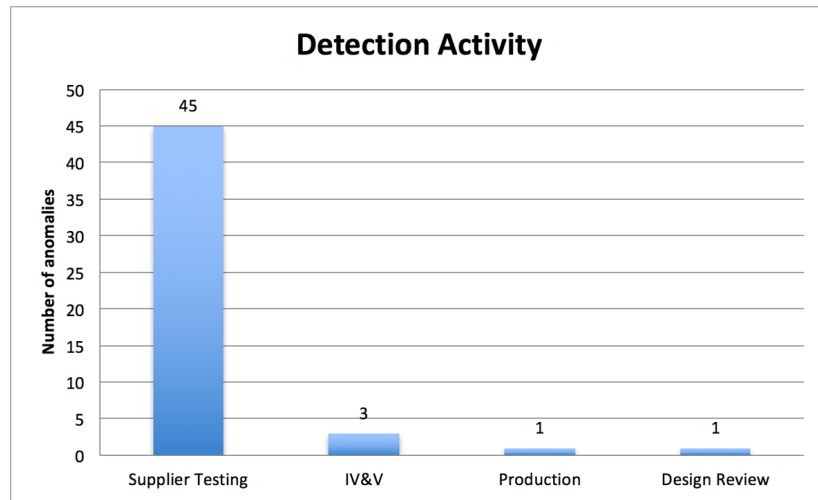


図 2: Detection Activity の分布

4.2.1. Supplier Testing

図 3 はソフトウェアテストのフェーズ別で不具合を分類した、Supplier Testing の分布である。これは元々特徴としてはシステムテストにおいて最も多くの不具合が存在していることにある。通常のソフトウェア開発プロセスでは unit・function テストで多く発生するが [7]、今回の不具合分類では unit・function テストが多く発生していることに特徴がある。なお、ここで言われている unit・function テストは CSCI(Computer Software Configuration Item) と呼ばれるもので分類されており、これは通常のインテグレーションテストのようなものであることに注意したい。この場合のシステムテストはコンポーネント同士のつながりではなく、大きなシステム同士の結合である (e.g. ground system - spacecraft) したがって、この 2 つの事象に対して対策を行うことが重要であることがわかる。

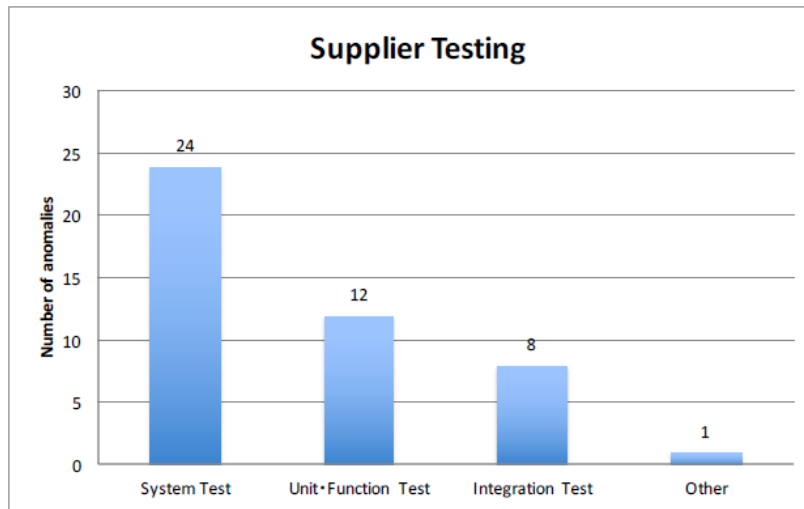


図 3: Supplier Testing の分布

4.3. Effect

図 4 は発生した不具合がシステムに与える影響の属性である Effect の分布である。Effect は Functionally と Performance に分けることができる。Functionally は機能の実装忘れなど機能が抜けているまたは要求されていない機能が実装されているなどがある。また、Performance は要求される Performance を出せないこと (e.g., capacity, computational accuracy, response time, throughput, or availability) ことによる不具合である。今回の分類では Functionality が 35 個 (70%) 存在し、Performance が 15 個 (30%) 存在した。

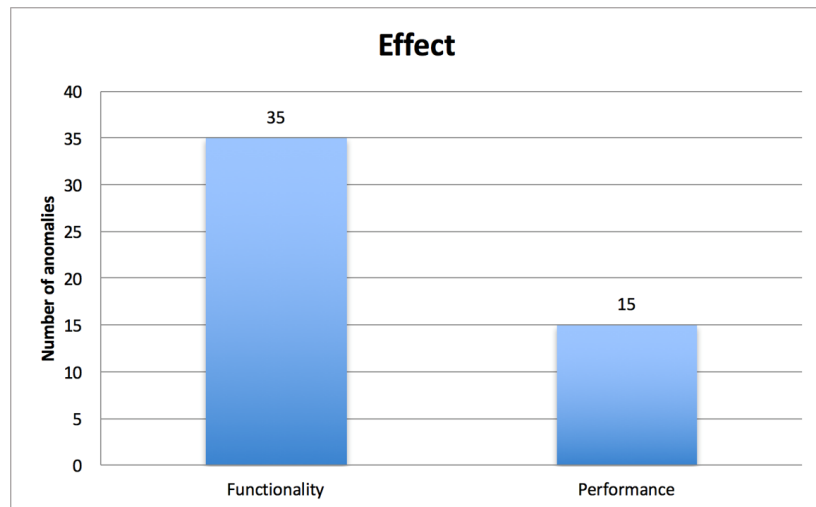


図 4: Effect の分布

4.4. Trigger

図 5 は欠陥が発生した要因の属性である Trigger の分布である。これからもわかるように、開発工程やシステムの矛盾によって発生する不具合である Design Conformance が約 20% を占め、大きな問題となっていることがわかる。

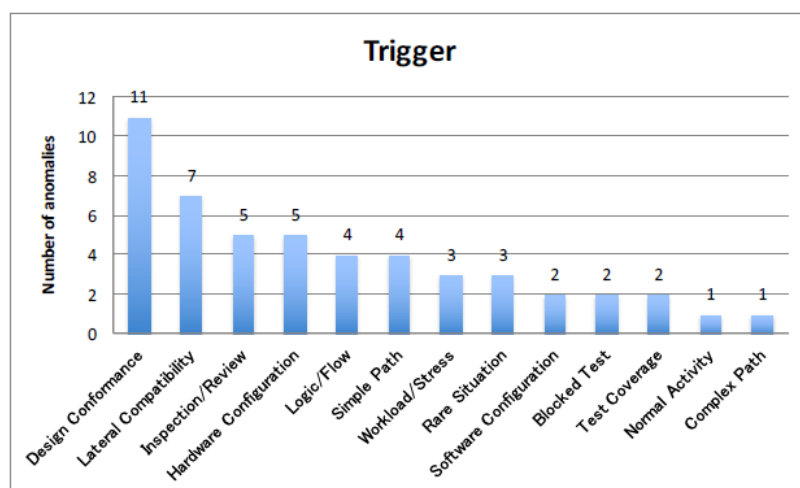


図 5: Trigger の分布

表 6: Trigger の定義

要素	定義
Design Conformance	設計とコードなど前後の開発での問題の混入
Lateral Compatibility	単純なシステム間の矛盾等
Inspection/Review	Inspection や Review の行程に問題があったことに起因する不具合
Hardware configuration	特定のハードウェア条件においてソフトウェアが期待通りに動作しない不具合
Logic/Flow	言語知識から理論・データに矛盾が生じることによって発生する不具合
Simple Path	ホワイトボックステストで発生した不具合
Workload/Stress	システムを限界条件付近で使用した時に発生した不具合
Rare Situation	想定外または言及されていないシステムの振る舞いによって生成されるソフトウェアの不具合
Software Configuration	特定のソフトウェアの設定下で動作しない不具合
Blocked Test	何らかの原因でテストが実行できなかったことに起因する不具合
Test Coverage	ブラックボックステストで一つの機能をパラメータまたは単一のパラメータでの呼出時に発生する不具合
Complex Path	ホワイトボックステストで幾つかの分岐やテスト条件が重なった時に見つかる不具合

表 7: Trigger の実例

要素	実例
Design Conformance	設計基準を元に設計書が書かれていなかったことによる仕様もれ
Lateral Compatibility	非同期と同期でのシステムの矛盾
Inspection/Review	A and B の時の異常停止が定義されていない
Hardware configuration	EEPROM の書き込み制限がなされていない
Logic/Flow	制御コマンドシーケンスの誤り
Simple Path	大域変数の扱い間違い
Workload/Stress	不可試験によるロールオーバー時の不具合の負荷試験による発見
Rare Situation	オフノミナルに対しての考慮不足
Software Configuration	信号への設計マージンの考慮不足
Blocked Test	割り込み処理有効、無効化のタイミングの検証を実機が無かったことによる検証不足に起因する不具合
Test Coverage	算術アルゴリズムの誤り
Complex Path	タイミング設計がうまくいっていないことに起因して、用いる変数の誤り

ここで、前述のソフトウェアの不具合が機能的なものなのかパフォーマンス的なものなのかの属性である Effect を用いて Deep Dive で分析を行う。この 2 つの属性で分析することによって要因となった不具合が機能的なものなのかパフォーマンス的なものなのかを見ることができる。実際に分析を行った結果が図 6 である。顕著に表れている結果として、ホワイトボックステストで発生した不具合である Simple Path、Inspection や Review の不備が不具合の原因となる Inspection/Review、単純なシステム間の矛盾である Lateral Compatibility は Function の問題が多いことがわかる。一方で Hardware Configuration に関しては Performance の不具合のみである。これはハードウェアにある特定の値を与えるとパフォーマンスが下がることが多く、それをソフトウェア側で設定できていないことに起因する。

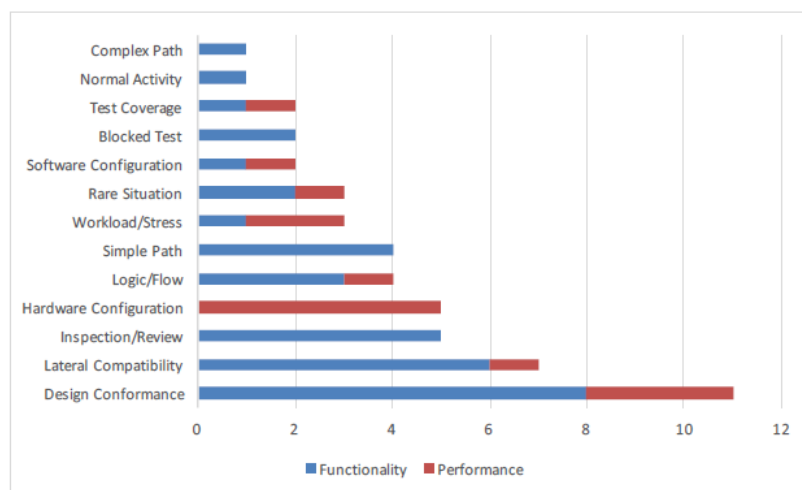


図 6: Trigger-Effect の分布

4.5. Type

図7は修正した欠陥の種類の属性である Type の分布である。最も多いのは機能またはアルゴリズムの修正を行ったときに分類をされる Function/Algorithm となっている。しかしながら、データの到着タイミングや同期の問題で発生する Timing, 値の割り当て、初期化の問題である Assignment/Initialization, S/W 間 (例えばモジュール)、H/W-S/W 間の問題も Interface との差は小さい。一方で Documentation の修正がないのは、開発段階のエラーがほとんどであり、文書の修正よりも機能の修正に力を入れたからであると考えられる。

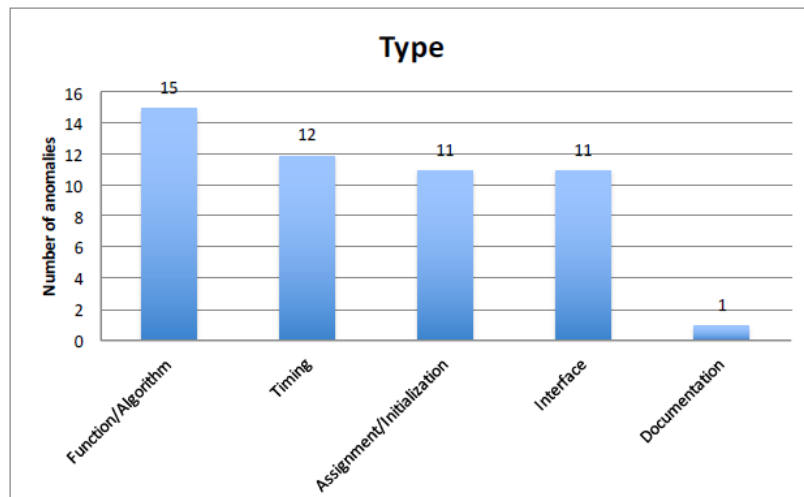


図 7: Type の分布

ここで、Type と前述のソフトウェアの不具合の原因である Trigger に関して Deep Dive で分析を行った結果を表 8 に示す。特徴的な点として Interface が修正箇所では不具合原因が単純なシステム間の矛盾等である Lateral Compatibility となっているものがある。これは、異なるモジュールを作成する技術者間で十分な合意をすることができなかったことに起因する。

表 8: Trigger-Type の分布

Trigger	Type					
	Func/Alg	Timing	Ass/Init	Interface	Doc	Total
Des Conf	3	2	4	1	1	11
Late Comp	0	0	1	6	0	7
Ins/Rev	2	0	2	1	0	5
HW Conf	0	2	1	2	0	5
Logic/Flow	2	1	1	0	0	4
Simple Path	2	1	0	1	0	4
Work/Str	1	1	1	0	0	3
Rare Situ	2	1	0	0	0	3
SW conf	1	1	0	0	0	2
Blo Test	0	2	0	0	0	2
Test Cov	1	1	0	0	0	2
Nor Act	1	0	0	0	0	1
Comp Path	0	0	1	0	0	1
Total	15	12	11	11	1	50

また、Type と前述のソフトウェアの不具合が発見されたテスト段階の属性である Supplier Testing を用いて、修正された欠陥との関係を Deep Dive で分析した結果を図 8 に示す。なお、比較の為にすべてのソフトウェアテストが終わった後に行われる IV&V も表に挿入している。最も多いのは Timing に関する修正が System Test で発見される場合である。Unit・Function Test ではほとんど発見することはできていない点に問題が存在する。また一方で、Timing、Assignment/Initialization の修正に関する不具合は Unit・Function Test と System Test で個数の違いがあまりないことから十分なテストができていないと考えられる。

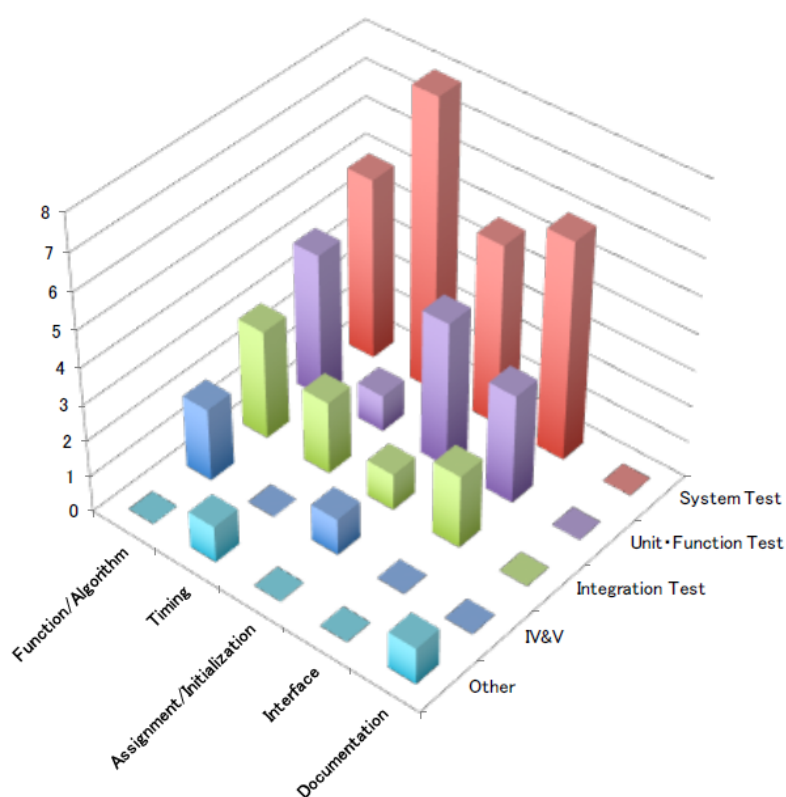


図 8: Type-Supplier Testing の分布

4.6. Mode

Mode は発生した不具合が間違っただけのふりまをするようになっていた (Wrong) のかそれとも機能が抜けていた (Missing) のかを表す属性である。Mode の分布を図 9 に示す。それぞれ Wrong が 29 個 (20%)、Missing が 20 個 (14%)、分類ができなかった Extra が 1 個 (0.7%) 存在した。

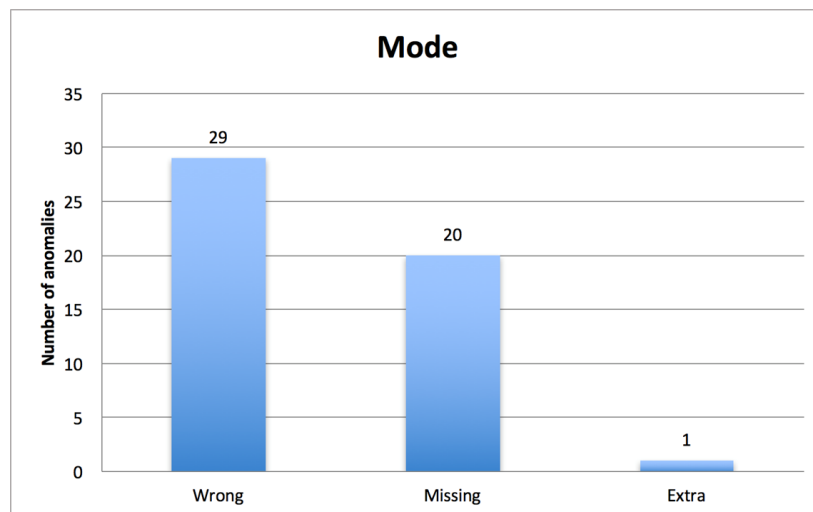


図 9: Mode の分布

ここで、Mode と前述のソフトウェア不具合が発見された Trigger を用いて、Deep Dive で分析した結果を図 10 に示す。大きな特徴としては設計とコードの間など前後の開発で不具合が生じたことを表す属性である Design Conformance では機能が抜ける Missing が多いのに対して、単純なシステム間の矛盾を表す Lateral Conformance や特定のハードウェア条件においてソフトウェアが期待通りに動作しない不具合である Hardware Configuration では Wrong が多いことがある。Design Conformance であれば機能自体は実装されているが、要求や設計の不具合により機能が誤って実装されていたことが考えられる。一方で Hardware Configuration であれば、特定の値によって不具合が発生することが主な要因となっているため、機能自体は存在しているが、エラー処理や SW/HW 間のインターフェースの問題によって不具合が発生していることが考えられる。

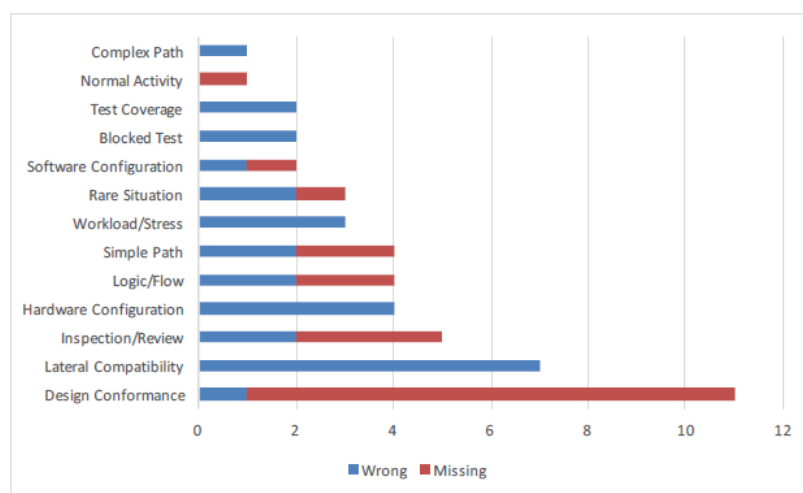


図 10: Mode-Trigger の分布

4.7. 考察及び課題

Insertion Activity より設計に起因する問題が多くを占めていることがわかる。また、Detection Activity の要素をさらに詳細に分類した Supplier Testing からわかるように、システムテストで最も多くの不具合が発見されている。早い工程で不具合が混入し、最終的なテストで発見されることがわかる。このような後の工程で見つかる不具合は修正するのに非常にコストがかかる点で問題である。Trigger に関する分類では開発工程やシステムの矛盾によって発生する不具合が最も多いことがわかる。これは高信頼システムにおいては、検証が繰り返し行われるため、プログラムの問題よりも作成しているシステムが意図に沿っていない不具合の方が多いことに起因する。Deep Dive を用いた分析においては、ハードウェアの不具合が存在すると、パフォーマンスに影響を及ぼすことがわかる。Type に関しては Trigger と合わせて Deep Dive を行うことにより他モジュールとのインターフェースの問題は開発者同士が認識を共有出来ていなかったことに起因していた。また、タイミングに関する不具合はシステムテストで最も多いにも関わらず、単体テストではほとんど発見できないことが明らかになった。

以上の事より、特にテストの検討事項として以下の3つが挙げられる。

- システムテストにて発見される不具合をより早い段階で発見すること
- 異なるコンポーネントを作成する開発者間の齟齬によって発生する不具合を防ぐこと
- ソフトウェアとハードウェアの不整合による不具合の発生を防ぐこと

現状において Spectrum [25] やモデルベースによる設計、モデル検査などの取り組みを JAXA では行っているため、現状において上流工程を改善することは難しいと考えられる。実際に Keijo らは人工衛星の姿勢制御系のソフトウェアをモデル検査で行った研究を行っているが、有効な手段ではあるが非常に時間がかかることを示している [33]。また、ハードウェアとソフトウェアの不整合による不具合の発生に関しては SILS(Software in Loop Simulation) を用いたテストの効率化などで取り組まれている。したがってコードカバレッジを向上させることによって、オフノミナルを自動単体テストケース生成で発見することを以降の目標とする。

5. 提案手法

本章では 4 章の結果に基づき、自動単体テストケース生成によって問題解決を行った結果を示す。以降では Concolic Testing の宇宙機ソフトウェアへの適用のアプローチ、対象とした宇宙機ソフトウェアの概要、適用結果及びその考察、問題点に関して述べる。

5.1. Concolic Testing の宇宙機ソフトウェアへの適用

2.3 章で述べた通り、Concolic Testing は SMT ソルバを用いた形式手法によるテストケース生成とランダムテストを合わせて現実的な時間で動的に単体テストケース生成を行う手法である。宇宙機ソフトウェアの単体テストカバレッジとして現在は、自動車の組み込みシステムや航空機システムと同様の明確な基準は存在せず、分岐網羅 (C1) 程度を実現することが一般的である。ここで、MC/DC を実現することができる Concolic Testing を宇宙機ソフトウェアに適用することによって、今までよりも厳しいカバレッジを実現できる。自動化を実現することによって単体テストの時間を削減するとともに、より厳しい網羅度によって複雑な条件でのみ発現する不具合を発見することができることが考えられる。特に宇宙機ソフトウェアでは条件によって分岐する実行する文による初期化や計算が多いため、適用することは有用であると考えた。

5.1.1. 対象宇宙機ソフトウェア

本研究では実験対象ソフトウェアとして小型実証衛星 4 型 (SDS-4: Small Demonstration Satellite 4) の姿勢軌道制御系 (ACFS: Attitude and Flight Control System) 搭載ソフトウェアを選択した [36]。この衛星は機器や部品などの新規技術を事前に宇宙で実証し、成熟度の高い技術を利用衛星や科学衛星に提供することを目的としている。小型であることから、大型衛星にくらべて様々な実験をタイムリーに進めることができる。また、メーカーによって作られるのではなく、人材育成の観点より設計から運用までの一連業務を JAXA 職員によって行われることに特徴がある。そのため、要求仕様、設計書などのドキュメントも存在するため、人手でプログラム

をトレースすることが可能である。特に今回対象とした ACFS は衛星の姿勢を決定やアクチュエータの制御を行う衛星の基本的な機能を担うものである。このプログラムはC言語によって約3万行(ヘッダファイルも合わせると4万行以上)で構成されている。

5.1.2. 適用結果

Concolic Testing ツールである CREST を実際に適用した。実際に CREST を適用する計算環境を表に示す。予め CREST 実行環境を構築した VM を Cent OS 上で動作させている。また、今回用いた CREST のバージョンは 0.1.2 であり、内部で用いている SMT ソルバは Yices-1.0.40 である。非線形演算に関しては事前調査で見当たらなかったため、安定した動作をする Yices を用いた CREST を本研究では利用した。

調査の手順としてはまず、実験的に地上コマンド処理モジュールに対して CREST 変数の宣言などのモジュールの改変を行い、実行した。実行の際には、プログラムの解析オプションとしてヒューリスティック及びソルバの組み合わせを選択できるが、今回はこの両方を用いる-hybrid オプションを利用した。次に分岐条件の調査として、その他のモジュールに対して地上コマンド処理モジュールで見つかった CREST では実行不可能な条件文をカウントした。地上コマンド処理モジュールに存在しなかったパターンの条件文は実行した。結果として表 10 の4つのパターンが実行不可能な条件文として抽出された。

表 9: CREST 実行環境

ホスト OS	CentOS 6.7 (32bit)
ゲスト OS	Ubuntu 12.04 (32bit)
仮想マシン環境	Virtual Box 5.14
CPU	Intel Core i7 2640M(2.80GHz)
メモリ	8GB

表 10: Concolic Testing 適用結果

パターン	コード例	行数 (条件文数)
配列の添え字が変数	<code>(p_acsf_input->gumd[i].f_update)</code>	98
ビット演算	<code>(data_buf_u >> 20) & 0x01) != 1</code>	65
ポインタ変数	<code>* mode == * pre_mod</code>	11
unsigned long 型	<code>int prg_ret(unsigned long a){switch(a){</code>	2

5.1.3. 考察及び解決策

以下では表 10 の結果に対して考察を述べる。まず、unsigned long 型の変数に関しては unsigned int 型にプログラムを改変することで対応可能である。対象とする人工衛星の環境においてはオーバーフロー等は発生しないため問題はない。また、個数としても 2 個と少ないので、変数表などのドキュメントを参照しながら人手で書き換えることは容易である。ポインタ変数に関しては、現在明確な対策手法がないため、検討が必要であり、この点は今後の課題とする。ビット演算に関しては、SMT ソルバでは対策されているので、今後 CREST でも対策がされることが予想される。最後に最も多いパターンである配列の添え字が変数なものは、SMT ソルバが対応していないため、CREST の書き換えでは対応できない。これは、配列がポインタの役割を果たしているためである。一方で、ループ内の分岐は MC/DC よりも高いカバレッジのみでしかテストされない。理由として、ループで繰り返される回数はプログラム中の変数により決定される事が多く、何回テストすればよいかという明確な基準がないためである。しかしながら、今回対象とした宇宙機ソフトウェアでは変数に応じてループの実行回数が増えるのではなく、プログラム中で予め定義されている記号定数を用いて定数回ループを実行するため、基準を設けることは容易である。よって、ループの中に分岐があるこのループの実行回数を基にループ内の分岐を実行コンパイラの最適化手法として用いられるループ展開によって解決可能である。本研究ではこの手法を実装し、評価することを試みた。

5.2. ループ展開

一般的にループ展開とは for 文などで定数回同じ処理を繰り返す場合に、繰り返しの構文を削除することである。この手法はコンパイラなどの最適化の際に使われることが多い。コンパイラではコードサイズが増大するというデメリットが存在するが、パフォーマンスを向上させることができる。これはループカウンタの更新頻度が低くなり、実行される分岐分が少なくなるためである [5]。以下のリスト 1、2 にループ展開前後のプログラム例を示す。

リスト 1: ループ展開前のプログラム例

```
1 Ground.Command_3(void){
2     for( i=0 ; i<NUM ;i++){
3         if(p_input->g[i].f.update ){
4             event_buf  ++;}
5         else{
6             event buf = 0;
7         }
8     }
9 }
```

リスト 2: ループ展開後のプログラム例

```
1 Ground.Command_3(void){
2     if(p_input->g[0].f.update ){
3         event_buf  ++;}
4     else{
5         event buf = 0;
6     }
7     if(p_input->g[1].f.update ){
8         event_buf  ++;}
9     else{
10        event buf = 0;
11    }
12    if(p_input->g[2].f.update ){
13        event_buf  ++;}
14    else{
15        event buf = 0;
16    }
17    /*以降同様の処理*/
18 }
```

本研究では配列の for 文を削除することによって解析ができなかった配列の添え字が変数である条件文に対して適用可能な形に展開することでカバレッジを向上させた。今回構築した CREST 適用のためのプログラムは以下の要件を満たすようなものとした。

- 3次元までの配列に対応すること
- 繰返文及び分岐がネストされている場合にも対応できること
- 繰返回数がプログラム中に記述されている場合はそれを利用し、ない場合はテスト者が任意に繰返しをしていけること

その他に、ループ展開を行うことによってテストデータを生成する変数を CREST に伝える処理が膨大になるため、自動的にこの処理を加える機能も追加した。

6. 評価実験

本章では CREST 適用のためのループ展開プログラムに対して、その有用性を検証するための評価実験を行う。対象とするのは 5.1.1 章で使用したものと同じ SDS-4 に搭載されている ACFS である。

6.1. 実験目的と内容

本実験の目的は作成したループ展開プログラムを用いることによって、(1)CREST が展開後プログラムに対して解析およびテストケース生成ができることを確認するとともに、(2) ループ内の条件分岐をテストできるようになったことでカバレッジが向上したことを確認することである。表 11 は今回対象としたモジュールの詳細である。

表 11: 実験対象対象モジュール

モジュール名	LOC	機能
モジュール A	1004	地上コマンド処理
モジュール B	1613	データ入力処理

まず、実験を行うにあたっては作成したループ展開プログラムを適用し、展開が必要なループを検出及び展開を行う。次に展開されたプログラムに対して、自動生成されたテストデータを生成する変数を CREST に伝える処理を追加するとともに単体テストを実行するためのテストドライバを追加する。このプログラムに対して CREST を実行した。カバレッジの測定には CREST を実行した際に出力される、パス数及び実際にテストケース生成が実行されたパス数を用いた。

6.2. 実験結果

表 13 および表に実験結果を示す。比較対象として、ループ展開前のループを含んだ場合のカバレッジと含まない場合のカバレッジ及び適用後のループを含まないカバレッジを示している。ループ展開前はループを無視するか、それともその部分が解析できていないとするかで基準が変化する。一方で、ループ展開後はカバレッジの測定対象となるループは存在しないため、ループを含まないカバレッジを基準としている。したがって、ループ展開によってカバレッジが向上しているかを確認するためには、ループ展開前のループを含むカバレッジとループ展開後のループを含まないカバレッジを比較すればよい。結果ループ展開前後で両モジュールともにカバレッジが向上していることを確認できた。

6.3. 考察

まず、モジュール A に関しては分岐数が多いものの、簡単な分岐条件がいくつも重なっているものであり、ループ展開前でもある程度のカバレッジを確認できている。しかしながら、ループ展開前後で約 12% カバレッジが向上していることがわかる。また、モジュール B のカバレッジに関してはループ展開前はループを無視した場合のカバレッジでも 63%、ループを含んだ場合のカバレッジは 18.4% とモジュール A と比較すると低いカバレッジとなっている。これには理由が 2 つあり、一つ目は CREST が対応できないビット演算を含む代入が入っていることに起因する。もう一つは セキュアプログラミングが施されている分岐が多数存在し、解析できたとしても値を生成できない分岐が存在するためである。特にこの場合はループ展開前後で 37% カバレッジが向上している。以上の結果より、ループ展開は宇宙機ソフトウェアのより複雑な単体テストの自動化に有用であることが示されている。

表 12: ループ展開適用前

	テストケースを生成した分岐数	解析可能な分岐数	ループを含まないカバレッジ	ループを含むカバレッジ
モジュール A	427	430	99.3%	81.8%
モジュール B	14	22	63.6%	18.4%

表 13: ループ展開適用後

	テストケースを生成した分岐数	解析可能な分岐数	ループを含まないカバレッジ
モジュール A	486	522	93.1%
モジュール B	42	76	55.2%

7. おわりに

本研究では、まずより効率的なソフトウェアテスト手法を検討するために不具合の分類を行った。合わせてテスト手法を検討するのに有効な不具合分類項目を提案し、実際に適用した。また、複数の項目を掛け合わせ検証する **Deep Dive** を用いて分析を行った。その結果として様々な傾向が見受けられたが、より効率的なソフトウェアテストの為に検討すべき点として、(1) システムテストにて発見される不具合をより早い段階で発見すること、(2) 異なるコンポーネントを作成する開発者間の齟齬によって発生する不具合を防ぐこと、(3) ソフトウェアとハードウェアの不整合による不具合の発生を防ぐことの3つが抽出された。本研究では単体テスト自動ケース生成の技術を用いて (1) の問題を解決すべく研究課題と設定した。

次に **Concolic Testing** と呼ばれる記号実行と具体的な値を用いた動的解析を行う単体テスト手法を小型衛星の姿勢制御系プログラムに対して用いることで効率化を図った。実際に **Concolic Testing** の実行ツールである **CREST** を適用した結果、**CREST** では解析できない条件文をその中でも最も多く存在した、ループの中に条件文が存在し、配列の添え字がループのインクリメントによって変化するものに対して、宇宙機ソフトウェアでは定数回のみループを実行する性質を利用して、コンパイラで利用されるループ展開の手法を応用することによって解決を図った。結果としてループ展開を用いることによってカバレッジが向上することが確認された。

今後の課題として、不具合分析の結果のうちで対処できていない問題に対して対策を立て、実行することがある。また、**Concolic Testing** の宇宙機ソフトウェア適用という点では、実際に開発段階のソフトウェアに対して適用するような検証が必要であると考えられる。また、宇宙機ソフトウェア以外に対して本手法を適用した場合についても検討が必要であると考えられる。

謝辞

本研究を進めるにあたり、多くの方々に御指導、御協力、御支援を頂きました。ここに謝意を添えてお名前を記させていただきます。本当にありがとうございました。

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学研究室 飯田 元教授には、本研究の全課程において熱心な御指導を賜りました。研究方針だけではなく、研究に対する姿勢、研究者としての心構え、論文執筆、発表方法についても多くの御助言を頂きました。心より厚く御礼を申し上げます。

参考文献

- [1] CREST Concolic Test generation tool for C. <http://www.burn.im/crest/>.
- [2] CREST-z3. <https://z3.codeple.c/>.
- [3] The Yices SMT Solver. <http://yices.csl.sri.com/>.
- [4] Z3. <https://z3.codeplex.com/>.
- [5] ARM. コンパイラ ソフトウェア開発ガイド. http://infocenter.arm.com/help/topic/com.arm.doc.dui0773aj/DUI0773AJ_software_development_guide.pdf.
- [6] Christian Denger Bernd Freimut and Markus Ketterer. An Industrial Case Study of Implementing and Validating Defect Classification for Process Improvement and Quality Management. In *Proceedings of 11th IEEE International Software Metrics Symposium (METRICS'05)*, pp. 10–19, 2005.
- [7] Aaron J. Shenhar [Brian J. Sauser, Richard R. Reilly. Why projects fail? how contingency theory can provide new insights – a comparative analysis of nasa’s mars climate orbiter loss. *International Journal of Project Management*, Vol. 27, No. 7, p. 665–679, 2009.
- [8] Ram Chillarege. Using ODC to diagnose an Agile Enterprise Application Development. In *Proceedings of the 24th International Symposium on Software Reliability Engineering (ISSRE 2013)*, p. 45, 2013.
- [9] Lee Copeland. *A Practitioner’s Guide to Software Test Design*.
- [10] Rick D. Craig and Stefan P. Jaskiel. *Systematic Software Testing*. Artech House, 2002.
- [11] IBM. *Orthogonal Defect Classification v 5.2 for Software Design and Code*. , 2013.
- [12] Koushik Sen Jacob Burnim. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, pp. 443–446, 2008.

- [13] John J. Chilenski Leanna K. Rierson Kelly J. Hayhurst, Dan S. Veerhusen. *A practical Tutorial on modified condition/decision coverage*. National Aeronautics and Space Administration, 2001.
- [14] Kenneth H. Friberg Kevin J. Barltrop and Gregory A. Horvath. Automated Generation and Assessment of Autonomous Systems Test Cases. In *Proceedings of IEEE Aerospace Conference (AeroConf 2008)*, pp. 1–8, 2008.
- [15] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Professional, 1995.
- [16] [R.O. Lewis. *Independent verification and validation: A life cycle engineering process for quality software*. Wiley, 1992.
- [17] Isaac Persing Ruili Geng Xu Bai LiGuo Huang, Vincent Ng and Jeff Tian. AutoODC: Automated Generation of Orthogonal Defect Classifications. In *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, pp. 412–415, 2015.
- [18] J.L. Lions. *Ariane 5 flight 501 failure*. European Space Agency(ESA), 1996.
- [19] Robyn R. Lutz and Ines Carmen Mikulsk. Requirements Discovery During the Testing of Safety-Critical Software. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pp. 578–583, 2003.
- [20] Robyn R. Lutz and Ines Carmen Mikulsk. Empirical analysis of safety-critical anomalies during operations. *IEEE Transaction on Software Engineering*, Vol. 30, No. 3, pp. 172–180, 2004.
- [21] Giovanni Denaro Mauro Pezz'e Mauro Baluda, Pietro Braione. Enhancing structural software coverage by incrementally computing branch executability. *Software Quality Journal*, Vol. 19, No. 4, pp. 725–751, 2011.
- [22] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, Vol. 14, No. 2, pp. 105–156, 2004.
- [23] Kishor S. Trivedi Micheal Grotte, Allen P. Nikora. An Empirical Investigation of Fault Types in Space Mission System Software. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, pp. 447–456, 2010.

- [24] Yoonkyu Jang Moonzoo Kim, Yunho Kim. Industrial Application of Concolic Testing on Embedded Software: Case Studies. In *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST 2012)*, pp. 390–399, 2012.
- [25] Jon D. Reese Nancy G. Leveson and Mats P.E. Heimdahl. SpecTRM: a CAD system for digital automation. In *Proceedings of 17th DASC. AIAA/IEEE/SAE Digital Avionics Systems Conference*, pp. 1–8, 1998.
- [26] Jonathan de Halleux Nikolai Tillmann. Pex—White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs (TAP 2008)*, pp. 134–153, 2008.
- [27] Andrea Mattavelli Mattia Vivanti Ali Muhammad Pietro Braione, Giovanni Denaro. An Industrial Case Study of the Effectiveness of Test Generators. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST 2012)*, pp. 50–56, 2012.
- [28] Jarir K. Micheal J. Chaar Halliday Diane S. Moebus Ram Chillarege, Inderpal S. Bhandari. Orthogonal defect classification—a concept for in-process measurements. *IEEE Transaction on Software Engineering*, Vol. 18, No. 11, pp. 943–956, 1992.
- [29] Bhupendra kharpuse Durga Prasad Mohapatra Banshidhar Majhi Sangharatna Godbole, Adepu Sridhar. Generation of branch coverage test data for simulink/stateflow models using crest tool. *International Journal of Advanced Computer Research*, Vol. 3, No. 4, pp. 222–229, 2013.
- [30] Durga Prasad Mohapatra Sangharatna Godbole. Analysis of mc/dc coverage percentage and cyclometric complexity for structured c programs. *International Journal of Computer & Mathematical Sciences*, Vol. 2, No. 2, pp. 16–20, 2014.
- [31] IEEE Computer Society. *1044-2009 - IEEE Standard Classification for Software Anomalies*. , 2010.
- [32] Rajiv R Chetwani M. Ravindra Sumith Shankar S, Kiran Desai and K.M.Bharadwaj. Mission Critical Software Test Philosophy A SILS based approach In Indian Mars Orbiter Mission. In *Proceedings of the International Con-*

- ference on Contemporary Computing and Informatics (IC3I)*, pp. 414–419, 2014.
- [33] Keijo Heljanko Xiang Gan, Jori Dubrovin. A symbolic model checking approach to verifying satellite onboard software. *Science of Computer Programming*, Vol. 82, pp. 44–55, 2014.
- [34] Brian Robinson Xiao Qu. A Case Study of Concolic Testing Tools and their Limitations. In *Proceedings of the 8th ACM international conference on Embedded software (EMSOFT'11)*, pp. 117–125, 2011.
- [35] 梅村晃広. S A T ソルバ・S M T ソルバの技術と応用. コンピュータソフトウェア, Vol. 27, No. 3, pp. 24–35, 2010.
- [36] 国立研究開発法人 宇宙航空研究開発機構. 小型実証衛星 4 型「SDS-4」. http://www.jaxa.jp/projects/sat/sds4/index_j.html.